3

JUnit 5 Standard Tests

Talk is cheap. Show me the code.
- Linus Torvalds

JUnit 5 provides a brand-new programming model called Jupiter. We can see this programming model as an API for software engineers and testers which allow to create JUnit 5 tests. These tests are later executed on the JUnit Platform. As we will discover, the Jupiter programming model allows to create many different types of tests. This chapter tackles the basics of Jupiter. To that aim, this chapter is structured as follows:

- **Test lifecycle**: In this section, we analyze the structure of the Jupiter tests, describing the annotations involved in the management of the test life cycle in the JUnit 5 programming model. Then, we discover how to skip tests, and also how to annotate tests with a custom display name.
- **Assertions**: In this section, first we present a brief overview of the verification assets, called assertions (also known as predicates). Second, we study how the assertions have been implemented in Jupiter. Finally, we present several third-party libraries about assertions, providing some examples for Hamcrest.
- **Tagging and filtering tests**: In this section, first we will learn how to label Jupiter tests, that is, how to create tags in JUnit 5. Then, we will learn how to filter our tests using Maven and Gradle. Finally, we are going to analyze how to create meta-annotations using Jupiter.
- **Conditional test execution**: In this section, we will learn how to disable tests based on a given condition. After that, we make a review of the so-called assumptions in Jupiter, which are a mechanism provided out of the box by Jupiter to run tests only if certain conditions are as expected.
- **Nested tests**: This section presents how Jupiter allows to express the relationship among a group of tests, called nested tests.

- **Repeated tests**: This section reviews how Jupiter provides the ability to repeat a test a specified number of times.
- **Migration from JUnit 4 to JUnit 5**: This section provides a set of hints about the main differences between JUnit 5 and its immediate antecessor, that is, JUnit 4. Then, this section presents the support for several JUnit 4 rules within Jupiter tests.

Test lifecycle

As we saw in `Chapter 1`, *Retrospective on software quality and Java testing*, a unit test case is composed of four stages:

1. **Setup** (optional): First, the test initializes the test fixture (before the picture of the SUT).
2. **Exercise**: Second, the test interacts with the SUT, getting some outcome from it as a result.
3. **Verify**: Third, the outcome from the system under test is compared to the expected value using one or several assertions (also known as predicates). As a result, a test verdict is created.
4. **Teardown** (optional): Finally, the test releases the test fixture to put the SUT back into the initial state.

In JUnit 4, there were different annotations to control these test phases. JUnit 5 follows the same approach, that is, Java annotations are used to identify different methods within Java classes, implementing the test life cycle. In Jupiter, all these annotations are contained in the package `org.junit.jupiter.api`.

The most basic JUnit annotation is `@Test`, which identifies the methods that have to be executed as tests. Therefore, a Java method annotated with `org.junit.jupiter.api.Test` will be treated as a test. The difference of this annotation with respect to JUnit 4's `@Test` is two folded. On the one hand, the Jupiter `@Test` annotation does not declare any attributes. In JUnit 4, `@Test` can declare the test timeout (as long attribute with the timeout in milliseconds), on the other hand, in JUnit 5, neither test classes nor test methods need to be public (this was a requirement in JUnit 4).

Take a look at the following Java class. Possibly, it is the simplest test case we can create with Jupiter. It has simply a method with the `@Test` annotation. The test logic (that is the exercise and verify stages as described before) would be contained inside the method `myTest`.

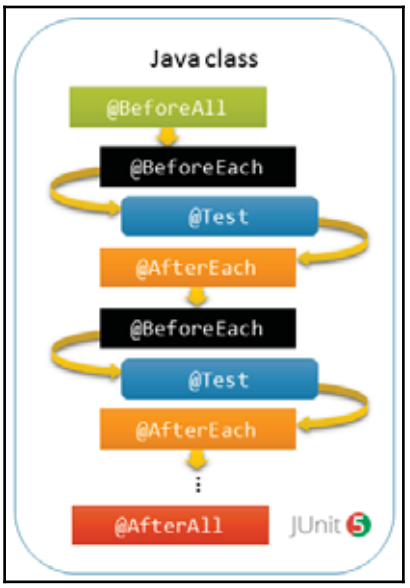
```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;

class SimpleJUnit5Test {

    @Test
    void mySimpleTest() {
          // My test logic here
    }

}
```

The Jupiter annotations (also located in the package `org.junit.jupiter.api`) aimed to control the setup and tear down stages in JUnit 5 tests are described in the following table:

JUnit 5 annotation	Description	JUnit 4's equivalence
`@BeforeEach`	Method executed before each `@Test` in the current class	`@Before`
`@AfterEach`	Method executed after each `@Test` in the current class	`@After`
`@BeforeAll`	Method executed before all `@Test` in the current class	`@BeforeClass`
`@AfterAll`	Method executed after all `@Test` in the current class	`@AfterClass`

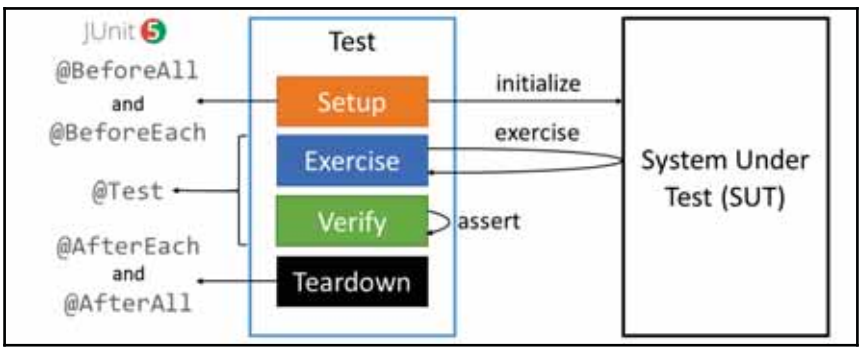
Methods annotated with these annotations (`@BeforeEach`, `@AfterEach`, `@AfterAll`, and `@BeforeAll`) are always inherited.

The following picture depicts the order of execution of these annotations in a Java class:



Jupiter annotations to control the test lyfecycle

Let's go back to the generic structure for tests we saw at the beginning of this section. Now, we are able to map the Jupiter annotations to control the test lifecycle with the different parts of a test case. As illustrated in the following picture, we carry out the setup stage by annotating methods with `@BeforeAll` and `@BeforeEach`. Then, we carry out the exercise and verify stages in methods annotated with `@Test`. Finally, we carry out the tear down process in the methods with `@AfterEach` and `@AfterAll`.



Relationship among the unit test cases stages and the Jupiter annotations

Let's see a simple example, which uses all these annotations in a single Java class. This example defines two tests (that is, two methods annotated with `@Test`), and we define additional methods for the rest of the test life cycle with the annotations `@BeforeAll`, `@BeforeEach`, `@AfterEach`, and `@AfterAll`:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class LifecycleJUnit5Test {

      @BeforeAll
      static void setupAll() {
            System.out.println("Setup ALL TESTS in the class");
      }

      @BeforeEach
      void setup() {
            System.out.println("Setup EACH TEST in the class");
      }

      @Test
      void testOne() {
            System.out.println("TEST 1");
      }

      @Test
      void testTwo() {
            System.out.println("TEST 2");
      }

      @AfterEach
      void teardown() {
            System.out.println("Teardown EACH TEST in the class");
      }

      @AfterAll
      static void teardownAll() {
            System.out.println("Teardown ALL TESTS in the class");
      }

}
```

If we run this test class, first `@BeforeAll` will be executed. Then, the two test methods will be executed sequentially, that is, the first one and then the other. In each execution, the setup method annotated with `@BeforeEach` will be executed before the test, and then the `@AfterEach` method. The following screenshot shows an execution of the tests using Maven and the command line:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.LifecycleJUnit5Test
Setup ALL TESTS in the class
Setup EACH TEST in the class
TEST 1
Teardown EACH TEST in the class
Setup EACH TEST in the class
TEST 2
Teardown EACH TEST in the class
Teardown ALL TESTS in the class
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.168 sec
- in io.github.bonigarcia.LifecycleJUnit5Test

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

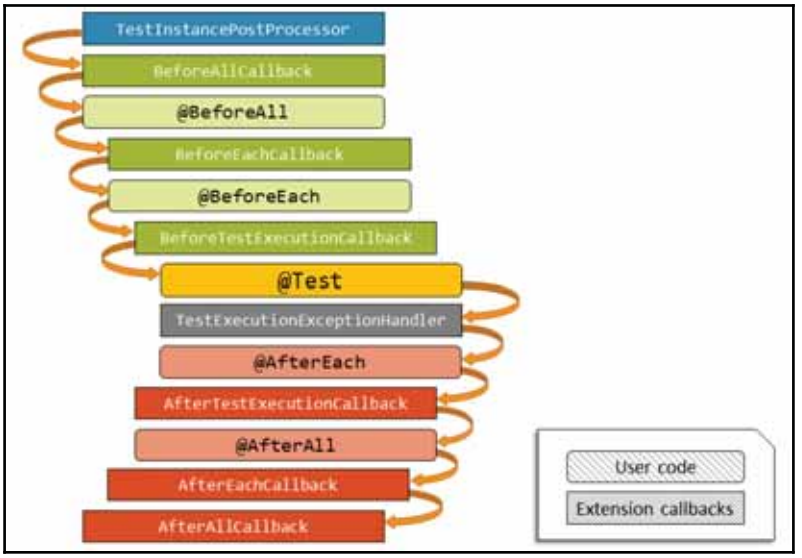
Execution of a Jupiter test which controls its lifecycle

Test instance lifecycle

In order to provide execution in isolation, the JUnit 5 framework creates a new test instance before executing the actual test (that is, the method annotated with `@Test`). This *per-method* test instance life cycle is the behavior in the Jupiter test and also in its antecessors (JUnit 3 and 4). As a novelty, this default behavior can be changed in JUnit 5, simply by annotating a `test` class with `@TestInstance(Lifecycle.PER_CLASS)`. Using this mode, the test instance will be created once per class, instead of once per test method.

This *per-class* behavior implies that it is possible to declare the `@BeforeAll` and `@AfterAll` methods as non-static. This is beneficial to be used in conjunction with some advanced capabilities, such as nested test or default test interfaces (explained in the next chapter).

All in all, and taking into account the extension callback (as explained in the *The extension model of JUnit 5* section of Chapter 2, *What's new in JUnit 5*), the relative execution order of user code and extensions is depicted in the following picture:



Relative execution order of user code and extensions

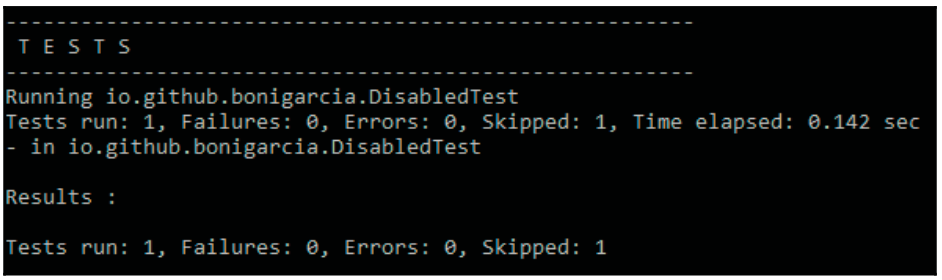
Skipping tests

The Jupiter annotation `@Disabled` (located in the package `org.junit.jupiter.api`) can be used to skip tests. It can be used at class level or method level. The following example uses the annotation `@Disabled` at method level and therefore it forces to skip the test:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

class DisabledTest {

    @Disabled
    @Test
```

```
    void skippedTest() {
    }

}
```

As shown in the following screenshot, when we execute this example, the test will be counted as skipped:

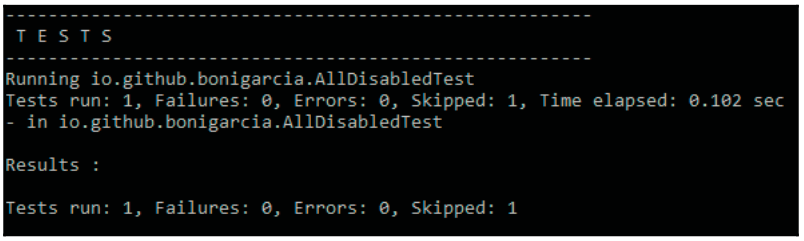


Disabled test method console output

In this other example, the annotation `@Disabled` is placed at the class level and therefore all the tests contained in the class will be skipped. Note that a custom message, typically with the reason of the disabling, can be specified within the annotation:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;

@Disabled("All test in this class will be skipped")
class AllDisabledTest {

    @Test
    void skippedTestOne() {
    }

    @Test
    void skippedTestTwo() {
    }

}
```

The following screenshot shows how the test case is skipped when it is executed (in this example using Maven and the command line):



Disabled test class console output

Display names

JUnit 4 identified tests basically with the name of the method annotated with `@Test`. This imposes a limitation on name tests, since these names are constrained by the way of declaring methods in Java.

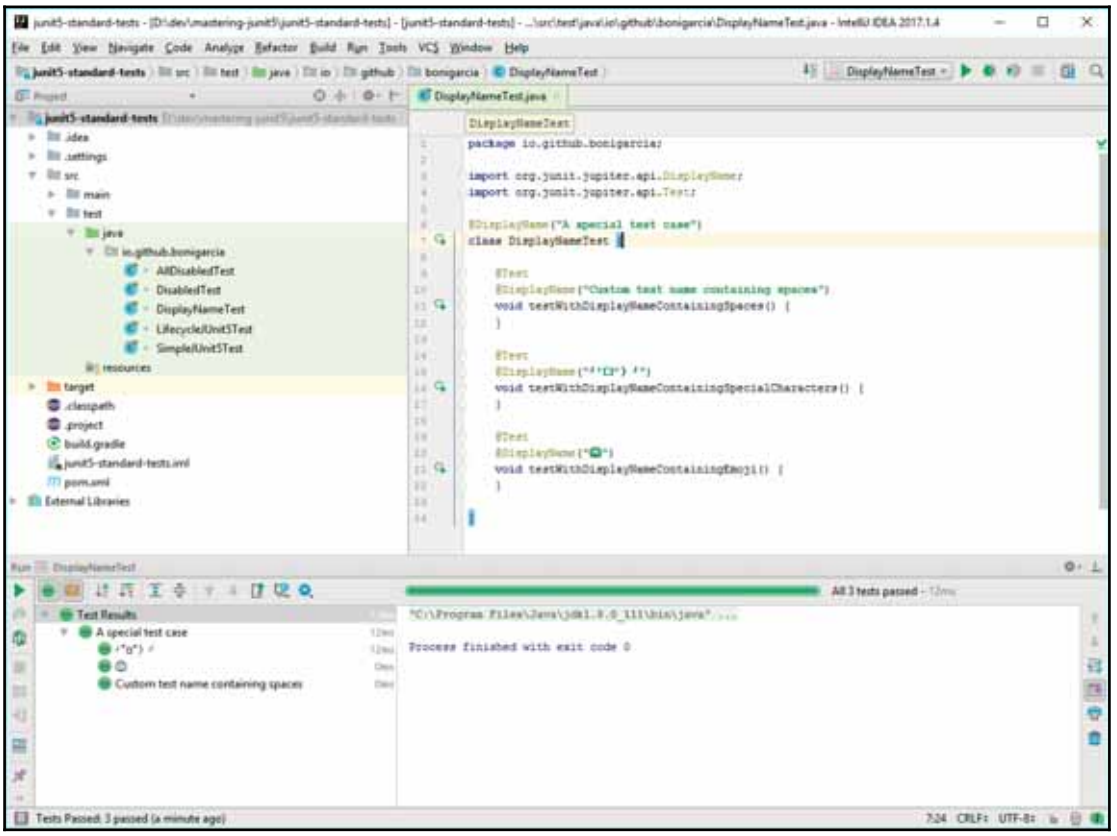
To overcome this problem, Jupiter provides the ability of declaring a custom display name (different to the test name) for tests. This is done with the annotation `@DisplayName`. This annotation declares a custom display name for a test class or a test method. This name will be displayed by test runners and reporting tools, and it can contain spaces, special characters, and even emojis.

Take a look at the following example. We are annotating the test class, and also the three test methods declared inside the class with a custom test name using `@DisplayName`:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class DisplayNameTest {

    @Test
    @DisplayName("Custom test name containing spaces")
    void testWithDisplayNameContainingSpaces() {
    }

    @Test
    @DisplayName("(ノ°Д°)ノ")
```

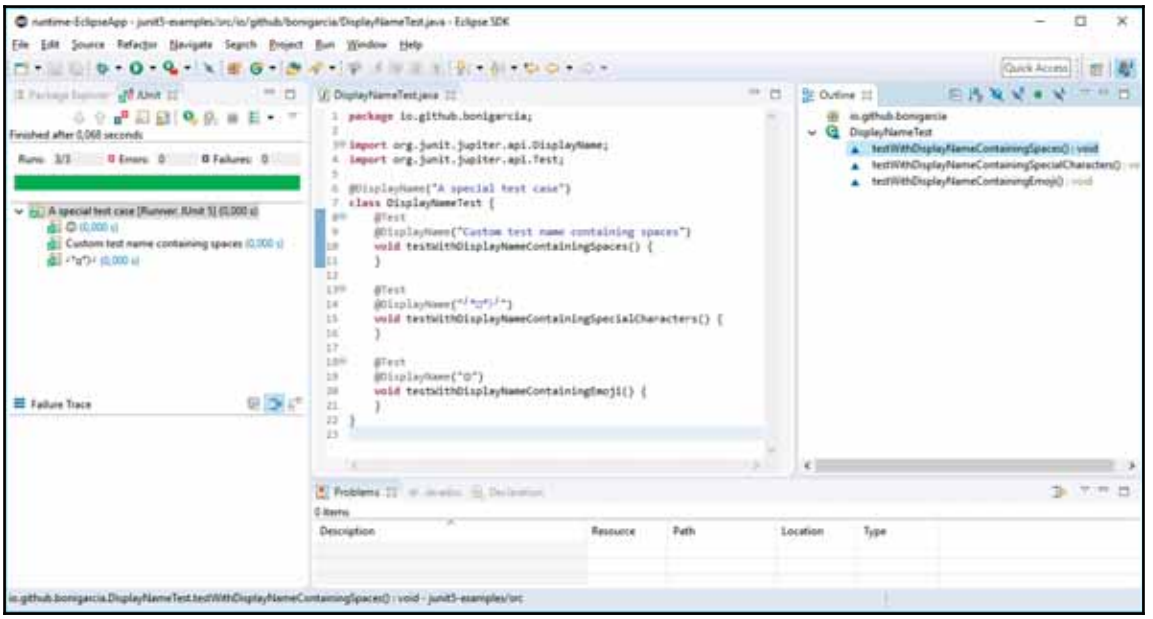
```
        void testWithDisplayNameContainingSpecialCharacters() {
        }

        @Test
        @DisplayName("😱")
        void testWithDisplayNameContainingEmoji() {
        }

    }
```

As a result, we see these labels when executing this test in a JUnit 5 compliant IDE. The following picture shows the execution of the example on IntelliJ 2016.2+:



Execution of a test case using *@DisplayName* in IntelliJ

On the other hand, the display name can be also seen in Eclipse 4.7 (Oxygen) or newer:



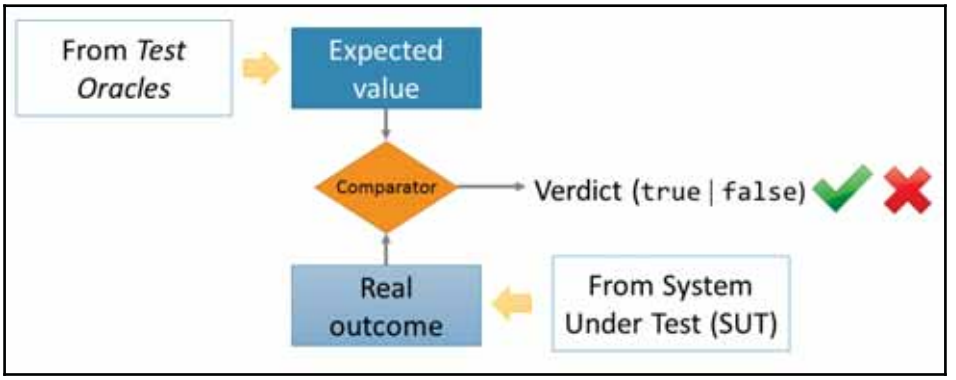
Execution of a test case using *@DisplayName* in Eclipse

Assertions

As we know, the general structure of a test case is composed of four stages: setup, exercise, verify, and tear down. The actual test happens during the second and third stage, when the test logic interacts with the system under test, getting some kind of outcome from it. This outcome is compared with the expected result in the verify stage. In this stage, we find what we call assertions. In this section, we take a closer look at them.

An assertion (also known as a predicate) is a `boolean` statement typically used to reason about software correctness. From a technical point of view, an assertion is composed of three parts (see the image after the list):

1. First, we find the expected value, which comes from what we call test oracles. A test oracle is a reliable source of expected outputs, for example, the system specification.
2. Second, we find the real outcome, which comes from the exercise stage made by the test against the SUT.
3. Finally, these two values are compared using some logic comparator. This comparison can be done in many different ways, for example, we can compare the object identity (equals or not), the magnitude (higher or lower value), and so on. As a result, we obtain a test verdict, which, in the end, is going to define if the test has succeeded or failed.

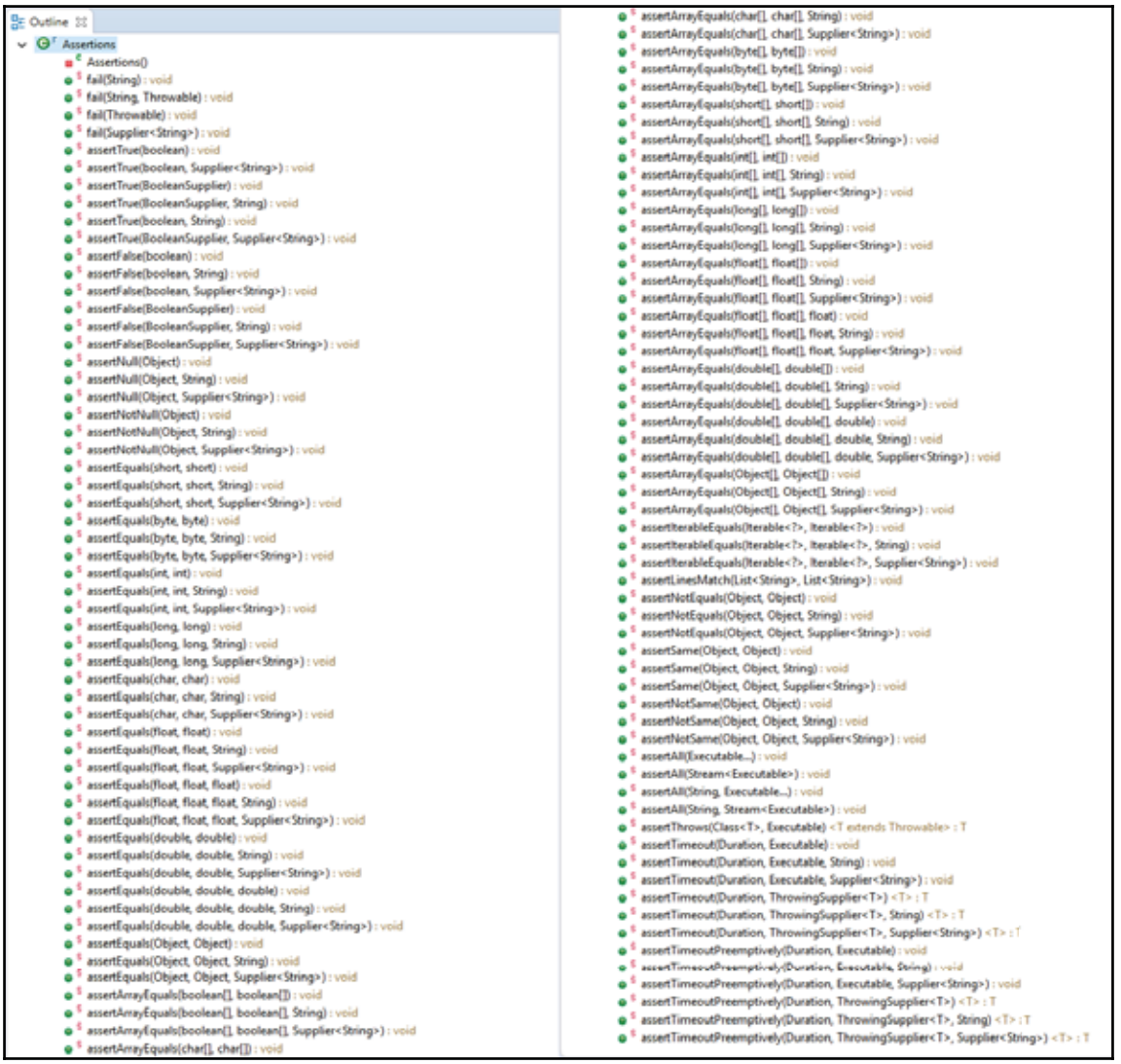


Schematic view of an assertion

Jupiter assertions

Let's move on to the JUnit 5 programming model. Jupiter comes with many of the assertion methods such as the ones in JUnit 4, and also adds several that can be used with Java 8 lambdas. All JUnit Jupiter assertions are static methods in the `Assertions` class located in `org.junit.jupiter package`.

The following picture shows the complete list of these methods:



Complete list of Jupiter assertions (class *org.junit.jupiter.Assertions*)

The following table reviews the different types of basic assertions in Jupiter:

Assertion	Description
`fail`	Fails a test with a given message and/or exception
`assertTrue`	Asserts that a supplied condition is true
`assertFalse`	Asserts that a supplied condition is false
`assertNull`	Asserts that a supplied object is `null`
`assertNotNull`	Asserts that a supplied object is not `null`
`assertEquals`	Asserts that two supplied objects are equal
`assertArrayEquals`	Asserts that two supplied arrays are equal
`assertIterableEquals`	Asserts that two iterable objects are deeply equal
`assertLinesMatch`	Asserts that two lists of Strings are equals
`assertNotEquals`	Asserts that two supplied objects are not equal
`assertSame`	Asserts that two objects are the same, compared with `==`
`assertNotSame`	Asserts that two objects are different, compared with `!=`

For each of the assertions contained in the table, an optional failure message (String) can be provided. This message is always the last parameter in the assertion method. This is a small difference with respect to JUnit 4, in which this message was the first parameter in the method invocation.

The following example shows a test using the `assertEquals`, `assertTrue`, and `assertFalse` assertion. Note that we are importing the static assertion methods at the beginning of the class in order to improve the readability of the test logic. In the example, we find the `assertEquals` method, in this case comparing two primitive types (it could also be used for objects). Second, the method `assertTrue` evaluates if a `boolean` expression is true. Third, the method `assertFalse` evaluates if a Boolean expression is false. In this case, notice that the message is created as a Lamdba expression. This way, assertion messages are lazily evaluated to avoid constructing complex messages unnecessarily:

```
package io.github.bonigarcia;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
```

```
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.Test;

class StandardAssertionsTest {

    @Test
    void standardAssertions() {
          assertEquals(2, 2);
          assertTrue(true,
          "The optional assertion message is now the last parameter");
          assertFalse(false, () -> "Really " + "expensive " + "message"
            + ".");
    }

}
```

The following parts of this section review the advance assertions provided by Jupiter: `assertAll`, `assertThrows`, `assertTimeout`, and `assertTimeoutPreemptively`.

Group of assertions

An important Jupiter assertion is `assertAll`. This method allows to group different assertions at the same time. In a grouped assertion, all assertions are always executed, and any failures will be reported together.

The method `assertAll` accepts a vargargs of lambda expressions (`Executable...`) or a stream of those (`Stream<Executable>`). Optionally, the first parameter of `assertAll` can be a String message aimed to label the assertion group.

Let's see an example. In the following test, we are grouping a couple of `assertEquals` using lambda expressions:

```
package io.github.bonigarcia;

import static org.junit.jupiter.api.Assertions.assertAll;
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class GroupedAssertionsTest {

    @Test
    void groupedAssertions() {
          Address address = new Address("John", "Smith");
          // In a grouped assertion all assertions are executed, and any
```

```
            // failures will be reported together.
            assertAll("address", () -> assertEquals("John",
            address.getFirstName()),
                () -> assertEquals("User", address.getLastName()));
        }

    }
```

When executing this test, all assertions of the group will be evaluated. Since the second assertion fails (`lastname` does not match), one failure is reported in the final verdict, as can be seen in the following screenshot:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.GroupedAssertionsTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.173 sec <<< FAILURE!
- in io.github.bonigarcia.GroupedAssertionsTest
groupedAssertions()  Time elapsed: 0.037 sec  <<< FAILURE!
org.opentest4j.MultipleFailuresError:
address (1 failure)
        expected: <User> but was: <Smith>
        at io.github.bonigarcia.GroupedAssertionsTest.groupedAssertions(GroupedAssertio
nsTest.java:32)


Results :

Failed tests:
  GroupedAssertionsTest.groupedAssertions:32 address (1 failure)
        expected: <User> but was: <Smith>

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
```

Console output of grouped assertions example

Asserting exceptions

Another important Jupiter assertion is `assertThrows`. This assertion allows to verify if a given exception is raised in a piece of code. To that aim, the method `assertThrows` accepts two arguments. First, the exception class expected, and second, an executable object (lambda expression), in which the exception is supposed to happen:

```
package io.github.bonigarcia;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
```

```
import org.junit.jupiter.api.Test;

class ExceptionTest {

    @Test
    void exceptionTesting() {
          Throwable exception =
            assertThrows(IllegalArgumentException.class,
            () -> {
               throw new IllegalArgumentException("a message");});
          assertEquals("a message", exception.getMessage());
    }

}
```

The is expecting `IllegalArgumentException` to be thrown, and this is actually happening inside this lambda expression. The following screenshot shows that the test actually succeeds:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.ExceptionTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.153 sec
- in io.github.bonigarcia.ExceptionTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Console output of *assertThrows* example

Asserting timeouts

To assess timeouts in JUnit 5 tests, Jupiter provides two assertions: `assertTimeout` and `assertTimeoutPreemptively`. On the one hand, `assertTimeout`, allows us to verify the timeout of a given operation. In this assertion, the expected time is defined using the class `Duration` of the standard Java package `java.time`.

We are going to see several running examples to clarify the use of this assertion method. In the following class, we find two tests using `assertTimeout`. The first test is designed to be succeeded, due to the fact that we are expecting that a given operation takes less than 2 minutes, and we are doing nothing there. On the other side, the second test will fail, since we are expecting that a given operation takes a maximum of 10 milliseconds, and we are forcing it to last 100 milliseconds.

```
package io.github.bonigarcia;

import static java.time.Duration.ofMillis;
import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertTimeout;

import org.junit.jupiter.api.Test;

class TimeoutExceededTest {

    @Test
    void timeoutNotExceeded() {
          assertTimeout(ofMinutes(2), () -> {
              // Perform task that takes less than 2 minutes
          });
    }

    @Test
    void timeoutExceeded() {
          assertTimeout(ofMillis(10), () -> {
              Thread.sleep(100);
          });
    }
}
```

When we execute this test, the second test is declared as failed because the timeout has been exceeded in 90 milliseconds:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.TimeoutExceededTest
Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.177 sec <<< FAILURE!
 - in io.github.bonigarcia.TimeoutExceededTest
timeoutExceeded()  Time elapsed: 0.13 sec  <<< FAILURE!
org.opentest4j.AssertionFailedError: execution exceeded timeout of 10 ms by 90 ms
        at io.github.bonigarcia.TimeoutExceededTest.timeoutExceeded(TimeoutExceededTes
t.java:36)


Results :

Failed tests:
  TimeoutExceededTest.timeoutExceeded:36 execution exceeded timeout of 10 ms by 90 ms

Tests run: 2, Failures: 1, Errors: 0, Skipped: 0
```

Console output of *assertTimeout* first example

Let's see a couple more tests using `assertTimeout`. In the first test, `assertTimeout` evaluates a piece of code as a lambda expression in a given timeout, obtaining its result. In the second test, `assertTimeout` evaluates a method in a given timeout, obtaining its result:

```
package io.github.bonigarcia;

import static java.time.Duration.ofMinutes;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTimeout;

import org.junit.jupiter.api.Test;

class TimeoutWithResultOrMethodTest {

    @Test
    void timeoutNotExceededWithResult() {
          String actualResult = assertTimeout(ofMinutes(1), () -> {
              return "hi there";
          });
          assertEquals("hi there", actualResult);
    }

    @Test
    void timeoutNotExceededWithMethod() {
```

```
            String actualGreeting = assertTimeout(ofMinutes(1),
                TimeoutWithResultOrMethodTest::greeting);
            assertEquals("hello world!", actualGreeting);
      }

      private static String greeting() {
            return "hello world!";
      }

}
```

In both cases, the tests take less time than expected and therefore both of them are succeeded:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.TimeoutWithResultOrMethodTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.12 sec
- in io.github.bonigarcia.TimeoutWithResultOrMethodTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Console output of *assertTimeout* second example

The other Jupiter assertion for timeouts is called `assertTimeoutPreemptively`. The difference with `assertTimeoutPreemptively` with respect to `assertTimeout` is that `assertTimeoutPreemptively` does not wait until the end of the operation, and the execution is aborted when the expected timeout is exceeded.

In this example, the test will fail since we are simulating an operation which lasts 100 milliseconds, and we have defined a timeout of 10 milliseconds:

```
package io.github.bonigarcia;

import static java.time.Duration.ofMillis;
import static org.junit.jupiter.api.Assertions.assertTimeoutPreemptively;

import org.junit.jupiter.api.Test;

class TimeoutWithPreemptiveTerminationTest {

      @Test
```

```
        void timeoutExceededWithPreemptiveTermination() {
              assertTimeoutPreemptively(ofMillis(10), () -> {
                    Thread.sleep(100);
              });
        }

}
```

In this example, when the timeout of 10 ms is reached, instantly the test is declared as a failure:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.TimeoutWithPreemptiveTerminationTest
Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.174 sec <<< FAILURE!
- in io.github.bonigarcia.TimeoutWithPreemptiveTerminationTest
timeoutExceededWithPreemptiveTermination()  Time elapsed: 0.051 sec  <<< FAILURE!
org.opentest4j.AssertionFailedError: execution timed out after 10 ms
        at io.github.bonigarcia.TimeoutWithPreemptiveTerminationTest.timeoutExceededWit
hPreemptiveTermination(TimeoutWithPreemptiveTerminationTest.java:28)


Results :

Failed tests:
  TimeoutWithPreemptiveTerminationTest.timeoutExceededWithPreemptiveTermination:28 exec
ution timed out after 10 ms

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0
```

Console output of *assertTimeoutPreemptively* example

Third-party assertion libraries

As we have seen, the built-in assertions provided out of the box for Jupiter are sufficient for many testing scenarios. Nevertheless, there are times when more additional functionality, such as matchers, can be desired or required. In such situations, the JUnit team recommends the use of the following third-party assertion libraries:

- Hamcrest (`http://hamcrest.org/`): an assertion framework to write matcher objects allowing rules to be defined declaratively.

- AssertJ (`http://joel-costigliola.github.io/assertj/`): fluent assertions for Java.
- Truth (`https://google.github.io/truth/`): an assertions Java library designed to make test assertions and failure messages more readable.

In this section, we are going to make a brief review of Hamcrest. This library provided the assertion `assertThat`, which allows to create readable highly configurable assertions. The method `assertThat` accepts two arguments: first the actual object, and second a `Matcher` object. This matcher implements the interface `org.hamcrest.Matcher`, and enables a partial or an exact match for an expectation. Hamcrest provides different matcher utilities, such as `is`, `either`, `or`, `not`, and `hasItem`. The Matcher methods use the builder pattern, allowing to combine one or more matchers to build a matcher chain.

In order to use Hamcrest, first we need to import the dependency in our project. In a Maven project, this means that we have to include the following dependency in our `pom.xml` file:

```
<dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
      <version>${hamcrest.version}</version>
      <scope>test</scope>
</dependency>
```

If we are using Gradle, we need to add the equivalent configuration within the `build.gradle` file:

```
dependencies {
      testCompile("org.hamcrest:hamcrest-core:${hamcrest}")
}
```

As usual, it is recommended using the latest version of Hamcrest. We can check it on the Maven central web (`http://search.maven.org/`).

The following example demonstrates how to use Hamcrest inside a Jupiter test. Concretely, this test uses the assertion `assertThat` together with the matchers `containsString`, `equalTo`, and `notNullValue`:

```
package io.github.bonigarcia;

import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.notNullValue;
```

```
import static org.hamcrest.MatcherAssert.assertThat;

import org.junit.jupiter.api.Test;

class HamcrestTest {

    @Test
    void assertWithHamcrestMatcher() {
          assertThat(2 + 1, equalTo(3));
          assertThat("Foo", notNullValue());
          assertThat("Hello world", containsString("world"));
    }

}
```

As shown in the following screenshot, this test is executed with no failure:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.HamcrestTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.138 sec
- in io.github.bonigarcia.HamcrestTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Console output of example using the Hamcrest assertion library

Tagging and filtering tests

Test classes and methods can be tagged in the JUnit 5 programming model by means of the annotation `@Tag` (package `org.junit.jupiter.api`). Those tags can later be used to filter test discovery and execution. In the following example, we see the use of `@Tag` at class level and also at method level:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("simple")
class SimpleTaggingTest {
```

```
        @Test
        @Tag("taxes")
        void testingTaxCalculation() {
        }

}
```

As of JUnit 5 M6, the label for tagging tests should meet the following syntax rules:

- A tag must not be null or blank.
- A trimmed tag (that is, tags in which leading and trailing whitespace have been removed) must not contain a white space.
- A trimmed tag must not contain ISO control characters nor the following reserved characters: `,`, `(`, `)`, `&`, `|`, and `!`.

Filtering tests with Maven

As we already know, we need to use `maven-surefire-plugin` in a Maven project to execute Jupiter test. Moreover, this plugin allows us to filter the test execution in several ways: filtering by JUnit 5 tags and also using the regular inclusion/exclusion support of `maven-surefire-plugin`.

In order to filter by tags, the properties `includeTags` and `excludeTags` of the `maven-surefire-plugin` configuration should be used. Let's see an example to demonstrate how. Consider the following tests contained in the same Maven project. On the one hand, all tests in this class are tagged with the `functional` word.

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("functional")
class FunctionalTest {

    @Test
    void testOne() {
        System.out.println("Functional Test 1");
    }

    @Test
    void testTwo() {
        System.out.println("Functional Test 2");
    }
```

```
}
```

On the other hand, all tests in the second class are tagged as `non-functional` and each individual test is also labeled with more tags (`performance`, `security`, `usability`, and so on):

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

@Tag("non-functional")
class NonFunctionalTest {

    @Test
    @Tag("performance")
    @Tag("load")
    void testOne() {
        System.out.println("Non-Functional Test 1 (Performance/Load)");
    }

    @Test
    @Tag("performance")
    @Tag("stress")
    void testTwo() {
        System.out.println("Non-Functional Test 2 (Performance/Stress)");
    }

    @Test
    @Tag("security")
    void testThree() {
        System.out.println("Non-Functional Test 3 (Security)");
    }

    @Test
    @Tag("usability")
    void testFour() {
        System.out.println("Non-Functional Test 4 (Usability)");
    }

}
```

As described before, we use the configuration keywords `includeTags` and `excludeTags` in the Maven `pom.xml` file. In this example, we include the test with the tag `functional` and exclude `non-functional`:

```
        <build>
            <plugins>
                <plugin>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <version>${maven-surefire-plugin.version}</version>
                    <configuration>
                        <properties>
                            <includeTags>functional</includeTags>
                            <excludeTags>non-functional</excludeTags>
                        </properties>
                    </configuration>
                    <dependencies>
                        <dependency>
                            <groupId>org.junit.platform</groupId>
                            <artifactId>junit-platform-surefire-
    provider</artifactId>
                            <version>${junit.platform.version}</version>
                        </dependency>
                        <dependency>
                            <groupId>org.junit.jupiter</groupId>
                            <artifactId>junit-jupiter-engine</artifactId>
                            <version>${junit.jupiter.version}</version>
                        </dependency>
                    </dependencies>
                </plugin>
            </plugins>
        </build>
```

As a result, when we try to execute all the tests within the project, only two will be executed (those with the tag `functional`), and the rest are not recognized as tests:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.FunctionalTest
Functional Test 1
Functional Test 2
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.09 sec
- in io.github.bonigarcia.FunctionalTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Maven execution of test filtering by tags

Maven regular support

The regular inclusion/exclusion support of the Maven plugin can still be used to select which tests are going to be executed by `maven-surefire-plugin`. To that aim, we use the keywords `includes` and `excludes` to configure the test name pattern used to filter the execution by the plugin. Notice that for both inclusions and exclusions, regular expressions can be used to specify a pattern of the test filenames:

```
<configuration>
   <includes>
      <include>**/Test*.java</include>
      <include>**/*Test.java</include>
      <include>**/*TestCase.java</include>
   </includes>
</configuration>
<configuration>
   <excludes>
      <exclude>**/TestCircle.java</exclude>
      <exclude>**/TestSquare.java</exclude>
   </excludes>
</configuration>
```

These three patterns, that is, the Java files containing the word *Test* or ending with *TestCase*, are included by default by a *maven-surefire plugin*.

Filtering tests with Gradle

Let's move now to Gradle. As we already know, we can also use Gradle to run JUnit 5 tests. Regarding the filtering process, we can select the test to be executed based on:

- The test engine: Using the keyword engines we can include or exclude the test engine to be used (that is `junit-jupiter` or `junit-vintage`).
- The Jupiter tags: Using the keyword `tags`.
- The Java packages: Using the keyword `packages`.
- The class name patterns: Using the keyword `includeClassNamePattern`.

By default, all engines and tags are included in the test plan. Only the classname containing the word `Tests` is applied. Let's see a working example. We reuse the same tests presented in the former Maven project, but this time in a Gradle project:

```
junitPlatform {
      filters {
            engines {
                  include 'junit-jupiter'
                  exclude 'junit-vintage'
            }
            tags {
                  include 'non-functional'
                  exclude 'functional'
            }
            packages {
                  include 'io.github.bonigarcia'
                  exclude 'com.others', 'org.others'
            }
            includeClassNamePattern '.*Spec'
            includeClassNamePatterns '.*Test', '.*Tests'
      }
}
```

Notice that we are including the tags `non-functional` and excluding `functional`, and therefore we execute four tests:

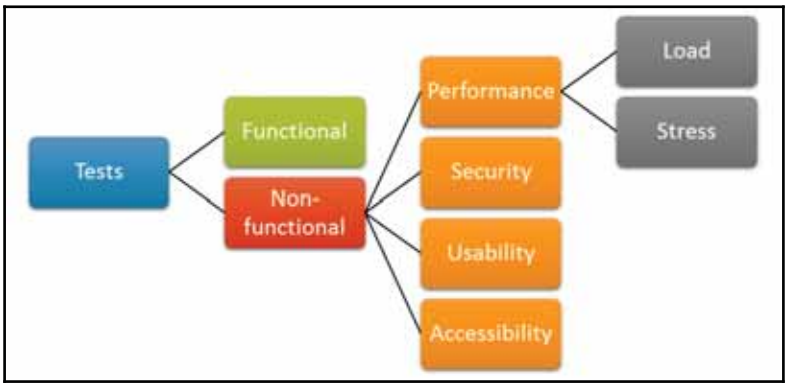
```
D:\dev\mastering-junit5\junit5-tagging-filtering>gradle test
:compileJava NO-SOURCE
:processResources NO-SOURCE
:classes UP-TO-DATE
:compileTestJava
:processTestResources NO-SOURCE
:testClasses
:junitPlatformTest
Non-Functional Test 1 (Performance/Load)
Non-Functional Test 2 (Performance/Stress)
Non-Functional Test 3 (Security)
Non-Functional Test 4 (Usability)

Test run finished after 81 ms
[         2 containers found      ]
[         0 containers skipped    ]
[         2 containers started    ]
[         0 containers aborted    ]
[         2 containers successful ]
[         0 containers failed     ]
[         4 tests found           ]
[         0 tests skipped         ]
[         4 tests started         ]
[         0 tests aborted         ]
[         4 tests successful      ]
[         0 tests failed          ]

:test SKIPPED

BUILD SUCCESSFUL
```

Gradle execution of test filtering by tags

Meta-annotations

The final part of this section is about the definition of meta-annotations. The JUnit Jupiter annotations can be used in the definition of other annotations (that is, can be used as meta-annotations). That means that we can define our own composed annotation that will automatically inherit the semantics of its meta-annotations. This feature is very convenient to create our custom test taxonomy by reusing the JUnit 5 annotation `@Tag`.

Let's see an example. Consider the following classification for test cases, in which we classify all tests as functional and non-functional, and then we make another level under the non-functional tests:



Example taxonomy for tests (functional and non-functional)

With that scheme in mind, we are going to create our custom meta-annotations for leaves of that tree structure: `@Functional`, `@Security`, `@Usability`, `@Accessiblity`, `@Load`, and `@Stress`. Notice that in each annotation we are using one or more `@Tag` annotations, depending on the structure previously defined. First, we can see the declaration of `@Functional`:

```
package io.github.bonigarcia;

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("functional")
public @interface Functional {
}
```

Then, we define the annotation `@Security` with tags `non-functional` and `security`:

```
package io.github.bonigarcia;

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("non-functional")
@Tag("security")
public @interface Security {
}
```

Similarly, we define the annotation `@Load`, but this time tagging with `non-functional`, `performance`, and `load`:

```
package io.github.bonigarcia;

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("non-functional")
@Tag("performance")
@Tag("load")
public @interface Load {
}
```

Finally we create the annotation `@Stress` (with tags `non-functional`, `performance`, and `stress`):

```
package io.github.bonigarcia;

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("non-functional")
@Tag("performance")
@Tag("stress")
public @interface Stress {
}
```

Now, we can use our annotations to tag (and later filter) tests. For instance, in the following example we are using the annotation `@Functional` at class level:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;

@Functional
class FunctionalTest {

      @Test
      void testOne() {
            System.out.println("Test 1");
      }

      @Test
      void testTwo() {
            System.out.println("Test 2");
      }

}
```

We can also out annotations at method level. In the following test, we annotate the different tests (methods) with different annotations (`@Load`, `@Stress`, `@Security`, and `@Accessibility`):

```
package io.github.bonigarcia;

import org.junit.jupiter.api.Test;

class NonFunctionalTest {

    @Test
    @Load
    void testOne() {
        System.out.println("Test 1");
    }

    @Test
    @Stress
    void testTwo() {
```

```
        System.out.println("Test 2");
    }

    @Test
    @Security
    void testThree() {
        System.out.println("Test 3");
    }

    @Test
    @Usability
    void testFour() {
        System.out.println("Test 4");
    }

}
```

All in all, we can filter the test by simply changing the included tags. On the one hand, we can filter by the tag `functional`. Notice that in this case, only two tests are executed. The following snippet shows the output of this kind of filtering using Maven:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.FunctionalTest
Functional Test 1
Functional Test 2
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.153 sec
 - in io.github.bonigarcia.FunctionalTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Filtering test by tags (functional) using Maven and the command line

On the other hand, we can also filter with different tags, such as `non-functional`. The following picture shows an example of this type of filtering, this time using Gradle. As usual, we can play with these examples by forking the GitHub repository (`https://github.com/bonigarcia/mastering-junit5`):

```
D:\dev\mastering-junit5\junit5-meta-annotations>gradle test
:compileJava
:processResources NO-SOURCE
:classes
:compileTestJava
:processTestResources NO-SOURCE
:testClasses
:junitPlatformTest
Non-Functional Test 1 (Performance/Load)
Non-Functional Test 2 (Performance/Stress)
Non-Functional Test 2 (Security)
Non-Functional Test 2 (Usability)

Test run finished after 97 ms
[         2 containers found      ]
[         0 containers skipped    ]
[         2 containers started    ]
[         0 containers aborted    ]
[         2 containers successful ]
[         0 containers failed     ]
[         4 tests found           ]
[         0 tests skipped         ]
[         4 tests started         ]
[         0 tests aborted         ]
[         4 tests successful      ]
[         0 tests failed          ]

:test SKIPPED

BUILD SUCCESSFUL

Total time: 2.073 secs
```

Filtering test by tags (non-functional) using Gradle and the command line

Conditional test execution

In order to establish custom conditions for test execution, we need to use the JUnit 5 extension model (introduced in `Chapter 2`, *What's new in JUnit 5*, in the section *The extension model of JUnit 5*). Concretely, we need to use the conditional extension point called `ExecutionCondition`. This extension can be used to deactivate either all tests in a class or individual tests.

We are going to see a working example in which we create a custom annotation to disable tests based on the operative system. First of all, we create a custom utility enumeration to select one operative system (`WINDOWS`, `MAC`, `LINUX`, and `OTHER`):

```
package io.github.bonigarcia;

public enum Os {
    WINDOWS, MAC, LINUX, OTHER;

    public static Os determine() {
        Os out = OTHER;
        String myOs = System.getProperty("os.name").toLowerCase();
        if (myOs.contains("win")) {
            out = WINDOWS;
        }
        else if (myOs.contains("mac")) {
            out = MAC;
        }
        else if (myOs.contains("nux")) {
            out = LINUX;
        }
        return out;
    }
}
```

Then, we create an extension of `ExecutionCondition`. In this example, the evaluation is done by checking whether or not the custom annotation `@DisabledOnOs` is present. When the annotation `@DisabledOnOs` is present, the value of the operative system is compared with the current platform. Depending on the result of that condition, the test is disabled or enabled.

```
package io.github.bonigarcia;

import java.lang.reflect.AnnotatedElement;
import java.util.Arrays;
import java.util.Optional;
import org.junit.jupiter.api.extension.ConditionEvaluationResult;
import org.junit.jupiter.api.extension.ExecutionCondition;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.platform.commons.util.AnnotationUtils;

public class OsCondition implements ExecutionCondition {

    @Override
    public ConditionEvaluationResult evaluateExecutionCondition(
            ExtensionContext context) {
          Optional<AnnotatedElement> element = context.getElement();
```

```
            ConditionEvaluationResult out = ConditionEvaluationResult
                  .enabled("@DisabledOnOs is not present");
            Optional<DisabledOnOs> disabledOnOs = AnnotationUtils
                  .findAnnotation(element, DisabledOnOs.class);
            if (disabledOnOs.isPresent()) {
               Os myOs = Os.determine();
               if(Arrays.asList(disabledOnOs.get().value())
                   .contains(myOs)) {
               out = ConditionEvaluationResult
                 .disabled("Test is disabled on " + myOs);
               }
               else {
                 out = ConditionEvaluationResult
                  .enabled("Test is not disabled on " + myOs);
               }
             }
             System.out.println("--> " + out.getReason().get());
             return out;
        }

    }
```

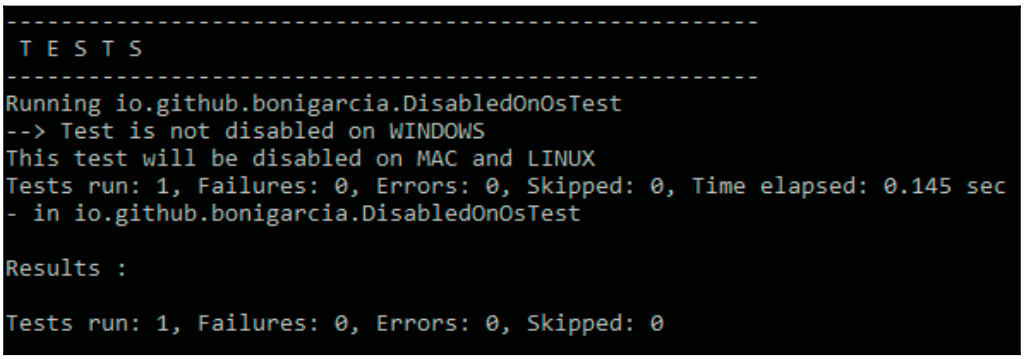
Moreover, we need to create our custom annotation `@DisabledOnOs`, which is also annotated with `@ExtendWith` pointing to our extension point.

```
package io.github.bonigarcia;

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.extension.ExtendWith;

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(OsCondition.class)
public @interface DisabledOnOs {
    Os[] value();
}
```

Finally, we use our annotation `@DisabledOnOs` in a Jupiter test.

```
import org.junit.jupiter.api.Test;

import static io.github.bonigarcia.Os.MAC;
import static io.github.bonigarcia.Os.LINUX;

class DisabledOnOsTest {

    @DisabledOnOs({ MAC, LINUX })
    @Test
    void conditionalTest() {
        System.out.println("This test will be disabled on MAC and LINUX");
    }

}
```

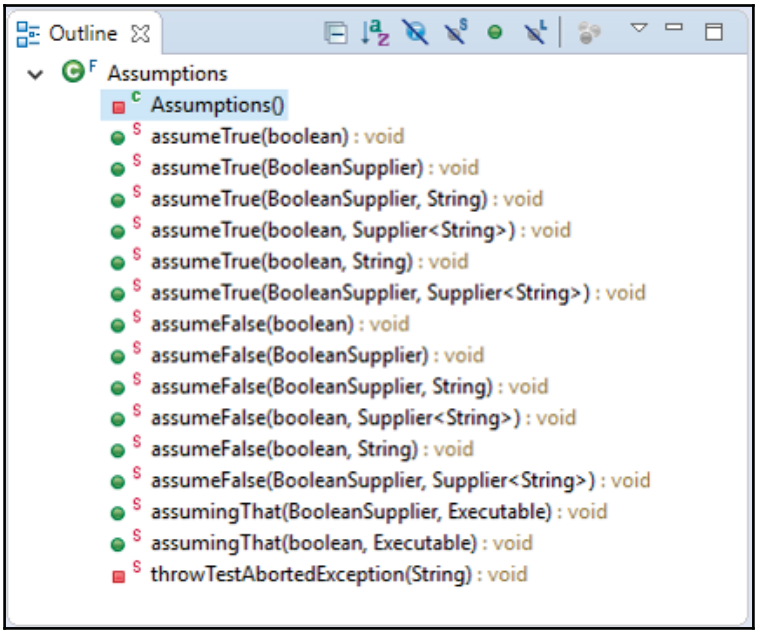
If we execute this test in a Windows machine, the test is not skipped, as we can see in this snapshot:



Execution of conditional test example

Assumptions

In this part of this section is about the so-called assumptions. Assumptions allow us to only run tests if certain conditions are as expected. All JUnit Jupiter assumptions are static methods in the class `Assumptions`, located inside the `org.junit.jupiter` package. The following screenshot shows all the methods of this class:



Methods of the class *org.junit.jupiter.Assumptions*

On the one hand, the methods `assumeTrue` and `assumeFalse` can be used to skip tests whose preconditions are not met. On the other hand, the method `assumingThat` is used to condition the execution of a part in a test:

```
package io.github.bonigarcia;

import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeFalse;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import static org.junit.jupiter.api.Assumptions.assumingThat;

import org.junit.jupiter.api.Test;

class AssumptionsTest {
```

```
    @Test
    void assumeTrueTest() {
        assumeTrue(false);
        fail("Test 1 failed");
    }

    @Test
    void assumeFalseTest() {
        assumeFalse(this::getTrue);
        fail("Test 2 failed");
    }

    private boolean getTrue() {
        return true;
    }

    @Test
    void assummingThatTest() {
        assumingThat(false, () -> fail("Test 3 failed"));
    }

}
```

Notice that in this example, the two first tests (`assumeTrueTest` and `assumeFalseTest`) are skipped since the assumptions are not met. Nevertheless, in the `assummingThatTest` test, only this part of the test (a lambda expression in this case) is not executed, but the whole test is not skipped:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.AssumptionsTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 2, Time elapsed: 0.183 sec
- in io.github.bonigarcia.AssumptionsTest

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 2
```

Execution of assumptions test example

Nested tests

Nested tests give the test writer more capabilities to express the relationship and order in a group of tests. JUnit 5 makes it effortless to nest test classes. We simply need to annotate inner classes with `@Nested` and all test methods in there will be executed as well, going from the regular tests (defined in the top-level class) to the tests defined in each of the inner classes.

The first thing we need to take into account is that only non-static nested classes (that is inner classes) can serve as `@Nested` tests. Nesting can be arbitrarily deep, and the setup and tear down for each test (that is, `@BeforeEach` and `@AfterEach` methods) are inherited in the nested tests. Nevertheless, inner classes cannot define the `@BeforeAll` and `@AfterAll` methods, due to the fact that Java does not allow static members in inner classes. However, this restriction can be avoided using the annotation `@TestInstance(Lifecycle.PER_CLASS)` in the test class. As described in the section *Test instance lifecycle* in this chapter, this annotation force to instance a test instance per class, instead of a test instance per method (default behavior). This way, the methods `@BeforeAll` and `@AfterAll` do not need to be static and therefore it can be used in nested tests.

Let's see a simple example composed by a Java class with two levels of inner classes, that is, the class contains two nested inner classes annotated with `@Nested`. As we can see, there are tests in the three levels of the class. Notice that the top class defined a setup method (`@BeforeEach`), and also the first nested class (called `InnerClass1` in the example). In the top-level class, we define a single test (called `topTest`), and in each nested class we find another test (called `innerTest1` and `innerTest2`, respectively):

```
package io.github.bonigarcia;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

class NestTest {

    @BeforeEach
    void setup1() {
        System.out.println("Setup 1");
    }

    @Test
    void topTest() {
       System.out.println("Test 1");
    }
```

```
        @Nested
        class InnerClass1 {

            @BeforeEach
            void setup2() {
                System.out.println("Setup 2");
            }

            @Test
            void innerTest1() {
                System.out.println("Test 2");
            }

            @Nested
            class InnerClass2 {

                @Test
                void innerTest2() {
                    System.out.println("Test 3");
                }
            }
        }

    }
```

If we execute this example, we can trace the execution of the nested tests by simply looking to the console traces. Note that the top `@BeforeEach` method (called `setup1`) is always executed before each test. Therefore, the trace `Setup 1` is always present in the console before the actual test execution. Each test also writes a line the console. As we can see, the first test logs `Test 1`. After that, the tests defined in the inner classes are executed. The first inner class executes the test `innerTest1`, but after that, the setup method of the top-level class and the first inner class are executed (logging `Setup 1` and `Setup 2`, respectively).

Finally, the test defined in the last inner class (`innerTest2`) is executed, but as usual, the cascade of setup methods is executed before the test:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.NestTest
Setup 1
Test 1
Setup 1
Setup 2
Test 2
Setup 1
Setup 2
Test 3
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.263 sec
- in io.github.bonigarcia.NestTest
Running io.github.bonigarcia.StackTest
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec
- in io.github.bonigarcia.StackTest

Results :

Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
```

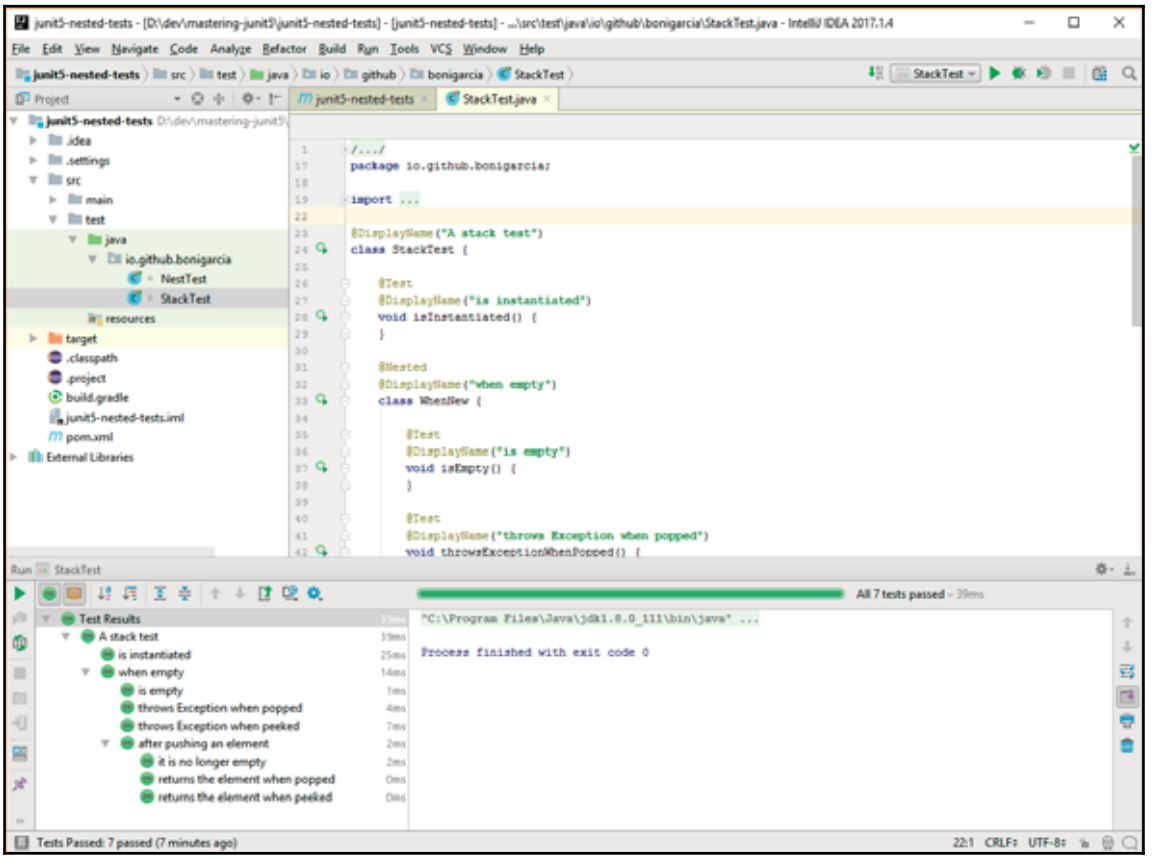
Console output of the execution of the nested test example

Nested tests can be used in conjunction with the display name (that is, the annotation `@DisplayName`) to help to produce a nicely readable test output. The following example demonstrates how. This class contains the structure to test the implementation of a stack, that is, a *last-in-first-out* (LIFO) collection. The class is designed to first test the stack when it is just instantiated (the method `isInstantiatedWithNew`). After that, the first inner class (`WhenNew`) is supposed to test the stack as an empty collection (methods `isEmpty`, `throwsExceptionWhenPopped` and `throwsExceptionWhenPeeked`). Finally, the second inner class is supposed to test when the stack is not empty (methods `isNotEmpty`, `returnElementWhenPopped`, and `returnElementWhenPeeked`):

```
package io.github.bonigarcia;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;

@DisplayName("A stack test")

 class StackTest {
```

```
        @Test
        @DisplayName("is instantiated")
        void isInstantiated() {
        }

        @Nested
        @DisplayName("when empty")
        class WhenNew {

            @Test
            @DisplayName("is empty")
            void isEmpty() {
            }

            @Test
            @DisplayName("throws Exception when popped")
            void throwsExceptionWhenPopped() {
            }

            @Test
            @DisplayName("throws Exception when peeked")
            void throwsExceptionWhenPeeked() {
            }

            @Nested
            @DisplayName("after pushing an element")
            class AfterPushing {

                @Test
                @DisplayName("it is no longer empty")
                void isNotEmpty() {
                }

                @Test
                @DisplayName("returns the element when popped")
                void returnElementWhenPopped() {
                }

                @Test
                @DisplayName("returns the element when peeked")
                void returnElementWhenPeeked() {
                }

            }
        }
    }
```

The objective of this type of test is two folded. On the one hand, the class structure provides an order for the execution of the tests. On the other hand, the use of `@DisplayName` improves the readability of the test execution. We can see that when the test is executed in an IDE, concretely in IntelliJ IDEA.



Execution of nested test using *@DisplayName* on Intellij IDEA

Repeated tests

JUnit Jupiter provides for the ability to repeat a test a specified number of times simply by annotating a method with `@RepeatedTest`, specifying the total number of repetitions desired. Each repeated test behaves exactly as a regular `@Test` method. Moreover, each repeated test preserves the same lifecycle callbacks (`@BeforeEach`, `@AfterEach`, and so on).

The following Java class contains a test that is going to be repeated five times:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.RepeatedTest;

class SimpleRepeatedTest {

    @RepeatedTest(5)
    void test() {
        System.out.println("Repeated test");
    }

}
```

Due to the fact that this test only writes a line (`Repeated test`) in the standard output, when executing this test in the console, we will see that trace five times:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.SimpleRepeatedTest
Repeated test
Repeated test
Repeated test
Repeated test
Repeated test
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.195 sec
- in io.github.bonigarcia.SimpleRepeatedTest

Results :

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0
```

Execution of repeated test in the console

In addition to specifying the number of repetitions, a custom display name can be configured for each repetition via the name attribute of the `@RepeatedTest` annotation. The display name can be a pattern composed of a combination of static text and dynamic placeholders. The following are currently supported:

- `{displayName}`: This is the name of the `@RepeatedTest` method.
- `{currentRepetition}`: This is the current repetition count.
- `{totalRepetitions}`: This is the total number of repetitions.

The following example shows a class with three repeated tests in which the display name is configured with the property name of `@RepeatedTest`:

```
package io.github.bonigarcia;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.TestInfo;

class TunningDisplayInRepeatedTest {

    @RepeatedTest(value = 2, name = "{displayName}
    {currentRepetition}/{totalRepetitions}")
    @DisplayName("Repeat!")
    void customDisplayName(TestInfo testInfo) {
        System.out.println(testInfo.getDisplayName());
    }

    @RepeatedTest(value = 2, name = RepeatedTest.LONG_DISPLAY_NAME)
    @DisplayName("Test using long display name")
    void customDisplayNameWithLongPattern(TestInfo testInfo) {
        System.out.println(testInfo.getDisplayName());
    }

    @RepeatedTest(value = 2, name = RepeatedTest.SHORT_DISPLAY_NAME)
    @DisplayName("Test using short display name")
    void customDisplayNameWithShortPattern(TestInfo testInfo) {
        System.out.println(testInfo.getDisplayName());
    }

}
```

In this test, the display name for these repeated tests will be as follows:

- For the test `customDisplayName`, the display name will follow the long display format:
  - `Repeat 1 out of 2.`
  - `Repeat 2 out of 2.`
- For the test `customDisplayNameWithLongPattern`, the display name will follow the long display format:
  - `Repeat! 1/2.`
  - `Repeat! 2/2.`
- For the test `customDisplayNameWithShortPattern`, the display name in this test will follow the short display format:
  - `Test using long display name :: repetition 1 of 2.`
  - `Test using long display name :: repetition 2 of 2.`

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.TunningDisplayInRepeatedTest
repetition 1 of 2
repetition 2 of 2
Test using long display name :: repetition 1 of 2
Test using long display name :: repetition 2 of 2
Repeat! 1/2
Repeat! 2/2
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.192 sec
- in io.github.bonigarcia.TunningDisplayInRepeatedTest

Results :

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0
```

Execution of repeated test example in conjunction with *@DisplayName*

Migration from JUnit 4 to JUnit 5

JUnit 5 does not support JUnit 4 features, such as Rules and Runners, natively. Nevertheless, JUnit 5 provides a gentle migration path via the JUnit Vintage test engine, which allows us to execute legacy test cases (including JUnit 4 but also JUnit 3) on the top of the JUnit Platform.

The following table can be used to summarize the main differences between JUnit 4 and 5:

Feature	JUnit 4	JUnit 5
Annotations package	`org.junit`	`org.junit.jupiter.api`
Declaring a test	`@Test`	`@Test`
Setup for all tests	`@BeforeClass`	`@BeforeAll`
Setup per test	`@Before`	`@BeforeEach`
Tear down per test	`@After`	`@AfterEach`
Tear down for all tests	`@AfterClass`	`@AfterAll`
Tagging and filtering	`@Category`	`@Tag`
Disable a test method or class	`@Ignore`	`@Disabled`
Nested tests	NA	`@Nested`
Repeated test	Using custom rule	`@Repeated`
Dynamic tests	NA	`@TestFactory`
Test templates	NA	`@TestTemaplate`
Runners	`@RunWith`	This feature is superseded by the extension model (`@ExtendWith`)
Rules	`@Rule` and `@ClassRule`	This feature is superseded by the extension model (`@ExtendWith`)

Rule support in Jupiter

As described before, Jupiter does not support JUnit 4 rules natively. Nevertheless, the JUnit 5 team realized that JUnit 4 rules are widely adopted in many test codebases nowadays. In order to provide a seamless migration from JUnit 4 to JUnit 5, the JUnit 5 team implemented the `junit-jupiter-migrationsupport` module. If this module is going to be used in a project, the module dependency should be imported. Examples for Maven are shown here:

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-migrationsupport</artifactId>
   <version>${junit.jupiter.version}</version>
   <scope>test</scope>
```

```
</dependency>
```

The Gradle declaration for this dependency is like this:

```
dependencies {
      testCompile("org.junit.jupiter:junit-jupiter-
      migrationsupport:${junitJupiterVersion}")
}
```

The rule support in JUnit 5 is limited to those rules semantically compatible with the Jupiter extension model, including the following rules:

- `junit.rules.ExternalResource` (including `org.junit.rules.TemporaryFolder`).
- `junit.rules.Verifier` (including `org.junit.rules.ErrorCollector`).
- `junit.rules.ExpectedException`.

In order to enable these rules in Jupiter tests, the test class should be annotated with the class-level annotation `@EnableRuleMigrationSupport` (located in the package `org.junit.jupiter.migrationsupport.rules`). Let us see several examples. First, the following test case defines and uses a `TemporaryFolder` JUnit 4 rule within a Jupiter test:

```
package io.github.bonigarcia;

import java.io.IOException;
import org.junit.Rule;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.migrationsupport.rules.EnableRuleMigrationSupport;
import org.junit.rules.TemporaryFolder;

@EnableRuleMigrationSupport
class TemporaryFolderRuleTest {

    @Rule
    TemporaryFolder temporaryFolder = new TemporaryFolder();

    @BeforeEach
    void setup() throws IOException {
        temporaryFolder.create();
    }

    @Test
    void test() {
        System.out.println("Temporary folder: " +
```

```
                temporaryFolder.getRoot());
        }

        @AfterEach
        void teardown() {
            temporaryFolder.delete();
        }

    }
```

When executing this test, the path of the temporary folder will be logged on the standard output:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.TemporaryFolderRuleTest
Temporary folder: C:\Users\boni\AppData\Local\Temp\junit3324174191725360154

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.445 sec -
 in io.github.bonigarcia.TemporaryFolderRuleTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Execution of Jupiter test using a JUnit 4 *TemporaryFolder* rule

The following test demonstrates the use of the `ErrorCollector` rule in a Jupiter test. Notice that the collector rule allows the execution of a test to continue after one or more problems are found:

```
package io.github.bonigarcia;

import static org.hamcrest.CoreMatchers.equalTo;

import org.junit.Rule;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.migrationsupport.rules.EnableRuleMigrationSupport;
import org.junit.rules.ErrorCollector;

@EnableRuleMigrationSupport
class ErrorCollectorRuleTest {

    @Rule
    public ErrorCollector collector = new ErrorCollector();
```

```
    @Test
    void test() {
        collector.checkThat("a", equalTo("b"));
        collector.checkThat(1, equalTo(2));
        collector.checkThat("c", equalTo("c"));
    }

}
```

These problems are reported together at the end of the test:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.ErrorCollectorRuleTest
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.237 sec
<<< FAILURE! - in io.github.bonigarcia.ErrorCollectorRuleTest
  test()  Time elapsed: 0.04 sec  <<< ERROR!
org.junit.internal.runners.model.MultipleFailureException:
There were 2 errors:
  java.lang.AssertionError(
Expected: "b"
     but: was "a")
  java.lang.AssertionError(
 Expected: <2>
     but: was <1>)


Results :

Tests in error:
  ErrorCollectorRuleTest.test » MultipleFailure There were 2 errors:
   java.lang...

Tests run: 1, Failures: 0, Errors: 1, Skipped: 0
```

Execution of Jupiter test using a JUnit 4 *ErrorCollector* rule

Finally, the `ExpectedException` rule allows us to configure a test to anticipate a given exception to be thrown within the test logic:

```
package io.github.bonigarcia;

import org.junit.Rule;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.migrationsupport.rules.EnableRuleMigrationSupport;
import org.junit.rules.ExpectedException;
```

```
@EnableRuleMigrationSupport
class ExpectedExceptionRuleTest {

    @Rule
    ExpectedException thrown = ExpectedException.none();

    @Test
    void throwsNothing() {
    }

    @Test
    void throwsNullPointerException() {
        thrown.expect(NullPointerException.class);
        throw new NullPointerException();
    }

}
```

In this example, even when the second test raises a `NullPointerException`, the test will be marked as having succeeded since that exception was expected.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running io.github.bonigarcia.ExpectedExceptionRuleTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.216 sec
- in io.github.bonigarcia.ExpectedExceptionRuleTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Execution of Jupiter test using a JUnit 4 *ExpectedException* rule

Summary

In this chapter, we introduced the basics of the brand-new programming model of the JUnit 5 framework, known as Jupiter. This programming model provides a rich API that can be used by practitioners to create test cases. The most basic element of Jupiter is the annotation `@Test`, which identifies the methods in Java classes treated as tests (that is logic which exercises and verifies a SUT). Moreover, there are different annotations that can be used to control the test life cycle, namely, `@BeforeAll`, `@BeforeEach`, `@AfterEach` , and `@AfterAll`. Other useful Jupiter annotations are `@Disabled` (to skip tests), `@DisplayName` (to provide a test name), `@Tag` (to label and filter tests).

Jupiter provides a rich set of assertions, which are static methods in the class `Assertions` used to verify if the outcome obtained from the SUT corresponds with some expected value. We can impose conditions for the test execution in several ways. On the one hand, we can use `Assumptions` to only run tests (or a part of those) if certain conditions are as expected.

We have learned how nested tests can be created simple annotating inner Java classes with `@Nested`. This can be used to create test executions following an order given the nested classes relationship. We have also studied how easy is to created repeated test using the JUnit 5 programming model. The annotation `@RepeatedTest` is used to that aim, providing the ability to repeat a test a specified number of times. Finally, we have seen how Jupiter provides support for several legacy JUnit 4 test rules, including `ExternalResource`, `Verifier`, and `ExpectedException`.

In the `chapter 4`, *Simplifying Testing With Advanced JUnit Features,* we continue discovering the JUnit programming model. Concretely, we review the advance features of JUnit 5, namely, dependency injection, dynamic tests, test interfaces, test templates, parameterized tests, compatibility of JUnit 5 and Java 9. Finally, we review some of the planned features in the backlog for JUnit 5.1, not implemented yet at the time of this writing.