JUnit 5 Tutorial: Writing Our First Test Class

Petri Kainulainen September 8, 2017

9 comments

JUnit, JUnit 5, Testing

This blog post describes how we can write our first test class with JUnit 5. After we have finished this blog post, we:

- Can create test classes with JUnit 5.
- Know how we can use setup and teardown methods.
- Understand how we can write simple test methods with JUnit 5.

Let's start by creating our first test class.

This blog post assumes that:

- You can run your unit tests with Maven **OR**
- You can run your unit tests with Gradle

Creating Our First Test Class

A test class is just a normal Java class that is eitherpublic or package private. We can create our first test class by following these steps:

1. Create a new package private class.
2. Configure the display name of our test class by annotating it with the@DisplayName annotation. This is an optional step, but I think that we should always use the @DisplayName annotation because it allows us to specify a custom display name that's used by our IDE and other build tools when they create or show test reports. In other words, we can replace a technical name with a sentence that describes the purpose of our test class.

After we have created our first test class, its source code looks as follows:

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

@DisplayName("JUnit 5 Example")
class JUnit5ExampleTest {

}
```

We can add the @DisplayName annotation to a test class or to a test method.

Additional Reading:

- JUnit 5 User Guide: 2.3. Display Names
- JUnit 5 User Guide: 4.4.2. Display Names vs. Technical Names
- The Javadoc of the @DisplayName annotation

We have now created our first test class. Next, we will find out how we can use setup and teardown methods.

Using Setup and Teardown Methods

A test class can have four setup and teardown methods that must fulfill these two conditions:

- These methods must not return anything. In other words, their return type must be void.
- Setup and teardown methods cannot be private.

The supported setup and teardown methods are described in the following:

- The method that is annotated with the @BeforeAll annotation must be static, and it's run once before any test method is run.
- The method that is annotated with the @BeforeEach is invoked before each test method.
- The method that is annotated with the @AfterEach annotation is invoked after each test method.
- The method that is annotated with the @AfterAll annotation must be static, and it's run once after all test methods have been run.

Let's add these methods to our test class. After we have added these setup and teardown methods to our test class, its source looks as follows:

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

@DisplayName("JUnit 5 Example")
class JUnit5ExampleTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("Before all test methods");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("Before each test method");
    }

    @AfterEach
    void afterEach() {
        System.out.println("After each test method");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("After all test methods");
    }
}
```

There are three things I want to point out:

- We write information to System.out only because it's the easiest way to demonstrate the invocation order of setup, teardown, and test methods.
- All setup and teardown methods are inherited as long as they are not overridden. Also, the setup and teardown methods found from the superclasses are invoked before the setup and teardown methods which are found from the subclasses.
- The methods which are annotated with the @BeforeAll and @AfterAll annotations must be static only if you create a new test instance for each test method (this is the default behavior of JUnit 5). However, it is also possible to use a setting that creates a new test instance for each test class. If you use the latter option, the methods annotated with the @BeforeAll and @AfterAll annotations don't have to be static.

Additional Reading:

- JUnit 5 User Guide: 2.1. Annotations
- JUnit 5 User Guide: 2.10. Test Instance Lifecycle
- The Javadoc of the @BeforeAll annotation
- The Javadoc of the @BeforeEach annotation
- The Javadoc of the @AfterEach annotation
- The Javadoc of the @AfterAll annotation

After we have added setup and teardown methods to our test class, we can finally write our first test methods. Let's find out how we can do it.

Writing Our First Test Methods

A test method is a method that fulfills these three requirements:

- A test method isn't private or static
- A test method must not return anything. In other words, its return type must be void.
- A test method must be annotated with the @Test annotation.

If we want to override the display name of a test method, we have to annotate the test method with the @DisplayName annotation.

Let's add two test methods to our test class:

1. The firstTest() method has a custom display name and it writes a unique string to System.out.
2. The secondTest() method has a custom display name and it writes a unique string to System.out.

After we have written these two test methods, the source code of our test class looks as follows:

```
1  import org.junit.jupiter.api.*;
2
3  @DisplayName("JUnit 5 Example")
4  class JUnit5ExampleTest {
5
6      @BeforeAll
7      static void beforeAll() {
8          System.out.println("Before all test methods");
9      }
10
11     @BeforeEach
12     void beforeEach() {
13         System.out.println("Before each test method");
14     }
15
16     @AfterEach
17     void afterEach() {
18         System.out.println("After each test method");
19     }
20
21     @AfterAll
22     static void afterAll() {
23         System.out.println("After all test methods");
24     }
25
26     @Test
27     @DisplayName("First test")
28     void firstTest() {
29         System.out.println("First test method");
30     }
31
32     @Test
33     @DisplayName("Second test")
34     void secondTest() {
35         System.out.println("Second test method");
36     }
37 }
```

Again, I think that we should specify custom display names to our test methods because we can use real sentences instead of technical names.

Additional Reading:

- JUnit 5 User Guide: 2.1. Annotations

- The Javadoc of the @Test annotation

We have just written our first test methods. Let's see what happens when we run our unit tests.

Running Our Unit Tests

When we run our unit tests, we should see the following output:

```
Before all test methods
Before each test method
First test method
After each test method
Before each test method
Second test method
After each test method
After all test methods
```

This output proves that the setup, teardown, and test methods are run in the correct order. Let's summarize what we learned from this blog post.

Summary

This blog post has taught us four things:

- A test class is a normal Java class that is either public or package private.
- Setup and teardown methods must not be private and they must not return anything.
- A test method is a method that isn't private and doesn't return anything.
- We should specify the display name of a test class and a test method because this allows us to replace technical names with sentences that make sense.

P.S. You can get the example application of this blog post from Github.

JUnit 5 Tutorial: Writing Nested Tests

Petri Kainulainen October 11, 2017

6 comments

Clean Code, JUnit, JUnit 5

This blog post describes how we can write nested tests with JUnit 5. After we have finished this blog post, we:

- Can create nested test classes.
- Know how we can add setup, teardown, and test methods to nested test classes.
- Understand the invocation order of setup, teardown, and test methods.

Let's start by taking a look at our test class.

This blog post assumes that:

- You can create test classes with JUnit 5

Introduction to Our Test Class

The previous part of this tutorial described how we can use setup and teardown methods, and add test methods to our test classes. Also, we wrote a simple test class and added all setup and teardown methods to the created class.

The source code of our test class looks as follows:

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

@DisplayName("JUnit 5 Nested Example")
class JUnit5NestedExampleTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("Before all test methods");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("Before each test method");
    }

    @AfterEach
    void afterEach() {
        System.out.println("After each test method");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("After all test methods");
    }
}
```

Next, we will add nested setup, teardown, and test methods to our test class.

Writing Nested Tests

When we write nested tests with JUnit 5, we have to create a nested test class hierarchy that contains our setup, teardown, and test methods. When we add nested test classes to our test class, we have to follow these rules:

- All nested test classes must be non-static inner classes.
- We have annotate our nested test classes with the@Nested annotation. This annotation ensures that JUnit 5 recognizes our nested test classes.
- There is no limit for the depth of the class hierarchy.
- By default, a nested test class can contain test methods, one@BeforeEach method, and one @AfterEach method.
- Because Java doesn't allow static members in inner classes, the @BeforeAll and @AfterAll methods don't work by default.

> There are two things I want to point out:
>
> - We should annotate our nested test classes with the@DisplayName annotation because it allows us to replace technical names with a sentence that describes the purpose of the nested test class.
> - If you want to use the @BeforeAll and @AfterAll methods, you should take a look at the JUnit 5 User Guide.

Let's add a few inner classes to our test class. The idea of this exercise is to demonstrate the invocation order of setup, teardown, and test methods. We can add the required inner classes to our test class by following these steps:

First, we have to add a new inner class calledA to our test class and annotate the inner class with the @Nested annotation. After we have created theA class, we have to add one setup, teardown, and test method to the created inner class.

After we have added this inner class to theJUnit5NestedExampleTest class, the source code of our test class looks as follows:

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

@DisplayName("JUnit 5 Nested Example")
class JUnit5NestedExampleTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("Before all test methods");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("Before each test method");
    }

    @AfterEach
    void afterEach() {
        System.out.println("After each test method");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("After all test methods");
    }

    @Nested
    @DisplayName("Tests for the method A")
    class A {

        @BeforeEach
        void beforeEach() {
            System.out.println("Before each test method of the A class");
        }

        @AfterEach
        void afterEach() {
            System.out.println("After each test method of the A class");
        }

        @Test
        @DisplayName("Example test for method A")
        void sampleTestForMethodA() {
            System.out.println("Example test for method A");
        }
    }
}
```

Second, we have to add a new inner class calledWhenX to the A class and annotate the inner class with the @Nested annotation. After we have created theWhenX class, we have to add one setup, teardown, and test method to the created inner class.

After we have added this inner class to theA class, the source code of our test class looks as follows:

```java
import org.junit.jupiter.api.*;

@DisplayName("JUnit 5 Nested Example")
class JUnit5NestedExampleTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("Before all test methods");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("Before each test method");
    }

    @AfterEach
    void afterEach() {
        System.out.println("After each test method");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("After all test methods");
    }

    @Nested
    @DisplayName("Tests for the method A")
    class A {

        @BeforeEach
        void beforeEach() {
            System.out.println("Before each test method of the A class");
        }

        @AfterEach
        void afterEach() {
            System.out.println("After each test method of the A class");
        }

        @Test
        @DisplayName("Example test for method A")
        void sampleTestForMethodA() {
            System.out.println("Example test for method A");
        }

        @Nested
        @DisplayName("When X is true")
        class WhenX {

            @BeforeEach
            void beforeEach() {
                System.out.println("Before each test method of the WhenX class");
            }

            @AfterEach
            void afterEach() {
                System.out.println("After each test method of the WhenX class");
            }

            @Test
            @DisplayName("Example test for method A when X is true")
            void sampleTestForMethodAWhenX() {
                System.out.println("Example test for method A when X is true");
            }
        }
    }
}
```

When I write nested tests for my code, I use the following class hierarchy:

- The “root” test class contains all test methods of the system under test. Typically I name this class by appending the string: Test to a string that identifies the system under test.
    - An inner class that contains the all tests of a feature or a method. I use the name of the tested feature or method as the name of this inner class.
        - The inner classes which which verify that the system under test is working as expected when a specific condition is true. I name these inner classes by prepending the string: When to a string that describes the condition. For example, if we are writing tests for a finder method, we could have two inner classes: WhenXIsFound and WhenXIsNotFound.

Additional Reading:

- JUnit 5 User Guide: 2.11. Nested Tests
- The Javadoc of the @Nested annotation

We have now written a test class that contains nested tests. Let's see what happens when we run our tests.

Running Our Tests

When we run our tests, we should see the following output:

```
Before all test methods
Before each test method
Before each test method of the A class
Example test for method A
After each test method of the A class
After each test method
Before each test method
Before each test method of the A class
Before each test method of the WhenX class
Example test for method A when X is true
After each test method of the WhenX class
After each test method of the A class
After each test method
After all test methods
```

In other words, JUnit 5 invokes the setup and teardown methods by following the context hierarchy of the invoked test method. This means that we can eliminate duplicate code by putting our code to the correct place.

If you want to learn how to put your code to the correct place, you should take a look at this sample lesson of my Test With Spring course: The "Best Practices" of Nested Unit Tests.

We have now written a test class that contains nested setup, teardown, and test methods. Also, we understand the invocation order of these methods. Let's summarize what we learned from this blog post.

Summary

This blog post has taught us five things:

- All nested test classes must be non-static.
- All nested test classes must be annotated with the@Nested annotation.
- The depth of the test class hierarchy is not limited in any way.
- A nested test class can contain test methods, one @BeforeEach method, and one @AfterEach method.

JUnit 5 Tutorial: Writing Assertions With JUnit 5 Assertion API

Petri Kainulainen March 17, 2018

6 comments

JUnit, JUnit 5

This blog post describes how we can write assertions by using the JUnit 5 assertion API. After we have finished this blog post, we:

- Can write basic assertions with JUnit 5.
- Know how we can customize the error message shown when an assertion fails.
- Understand how we can run multiple assertions as an assertion group.

Let's begin.

This blog post assumes that:

- You can create test classes with JUnit 5
- You can write nested tests with JUnit 5

Writing Assertions With JUnit 5

If we want to write assertions by using the "standard" JUnit 5 API, we must use the org.junit.jupiter.api.Assertions class. It provides static factory methods that we can use for writing assertions.

Before we will take a closer look at these methods, we have to know a few basic rules:

- If we want to specify a custom error message that is shown when our assertion fails, we have to pass this message as the last method parameter of the invoked assertion method.
- If we want to compare two values (or objects), we have to pass these values (or objects) to

the invoked assertion method in this order: the expected value (or object) and the actual value (or object).

Next, we will find out how we can write assertions with the Assertions class. Let's start by finding out how we can write assertions for boolean values.

Asserting Boolean Values

If we want to verify that a boolean value is true, we have to use the assertTrue() method of the Assertions class. In order words, we have to use this assertion:

```
1  import org.junit.jupiter.api.DisplayName;
2  import org.junit.jupiter.api.Nested;
3  import org.junit.jupiter.api.Test;
4
5  import static org.junit.jupiter.api.Assertions.assertTrue;
6
7  @DisplayName("Write assertions for booleans")
8  class BooleanAssertionTest {
9
10     @Nested
11     @DisplayName("When boolean is true")
12     class WhenBooleanIsTrue {
13
14         @Test
15         @DisplayName("Should be true")
16         void shouldBeTrue() {
17             assertTrue(true);
18         }
19     }
20 }
```

If we want to verify that a boolean value is false, we have to use the assertFalse() method of the Assertions class. In order words, we have to use this assertion:

```
1  import org.junit.jupiter.api.DisplayName;
2  import org.junit.jupiter.api.Nested;
3  import org.junit.jupiter.api.Test;
4
5  import static org.junit.jupiter.api.Assertions.assertFalse;
6
7  @DisplayName("Write assertions for booleans")
8  class BooleanAssertionTest {
9
10     @Nested
11     @DisplayName("When boolean is false")
12     class WhenBooleanIsFalse {
13
14         @Test
15         @DisplayName("Should be false")
16         void shouldBeFalse() {
17             assertFalse(false);
18         }
19     }
20 }
```

Next, we will find out how we can verify that an object is null or isn't null.

Asserting That an Object Is Null or Isn't Null

If we want to verify that an object is null, we have to use the assertNull() method of the Assertions class. In other words, we have to use this assertion:

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

import static org.junit.jupiter.api.Assertions.assertNull;

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When object is null")
    class WhenObjectIsNull {

        @Test
        @DisplayName("Should be null")
        void shouldBeNull() {
            assertNull(null);
        }
    }
}
```

If we want to verify that an object isn't null, we have to use the assertNotNull() method of the Assertions class. In other words, we have to use this assertion:

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

import static org.junit.jupiter.api.Assertions.assertNotNull;

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When object is not null")
    class WhenObjectIsNotNotNull {

        @Test
        @DisplayName("Should not be null")
        void shouldNotBeNull() {
            assertNotNull(new Object());
        }
    }
}
```

Let's move on and find out how we can verify that two objects (or values) are equal or aren't equal.

Asserting That Two Objects or Values Are Equal

If we want to verify that the expected value (or object) is equal to the actual value (or object), we have to use the assertEquals() method of the Assertions class. For example, if we want to compare two Integer objects, we have to use this assertion:

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

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

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When two objects are equal")
    class WhenTwoObjectsAreEqual {

        @Nested
        @DisplayName("When objects are integers")
        class WhenObjectsAreIntegers {

            private final Integer ACTUAL = 9;
            private final Integer EXPECTED = 9;

            @Test
            @DisplayName("Should be equal")
            void shouldBeEqual() {
                assertEquals(EXPECTED, ACTUAL);
            }
        }
    }
}
```

If we want to verify that the expected value (or object) isn't equal to the actual value (or object), we have to use the assertNotEquals() method of the Assertions class. For example, if we want

to compare two Integer objects, we have to use this assertion:

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

import static org.junit.jupiter.api.Assertions.assertNotEquals;

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When two objects aren't equal")
    class WhenTwoObjectsAreNotEqual {

        @Nested
        @DisplayName("When objects are integers")
        class WhenObjectsAreIntegers {

            private final Integer ACTUAL = 9;
            private final Integer EXPECTED = 4;

            @Test
            @DisplayName("Should not be equal")
            void shouldNotBeEqual() {
                assertNotEquals(EXPECTED, ACTUAL);
            }
        }
    }
}
```

Next, we will find out how we can write assertions for object references.

Asserting Object References

If we want to ensure that two objects refer to the same object, we have to use the assertSame() method of the Assertions class. In other words, we have to use this assertion:

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

import static org.junit.jupiter.api.Assertions.assertSame;

@DisplayName("Writing assertions for objects")
class ObjectAssertionTest {

    @Nested
    @DisplayName("When two objects refer to the same object")
    class WhenTwoObjectsReferToSameObject {

        private final Object ACTUAL = new Object();
        private final Object EXPECTED = ACTUAL;

        @Test
        @DisplayName("Should refer to the same object")
        void shouldReferToSameObject() {
            assertSame(EXPECTED, ACTUAL);
        }
    }
}
```

If we want to ensure that two objects don't refer to the same object, we have to use the assertNotSame() method of the Assertions class. In other words, we have to use this assertion:

```
1  import org.junit.jupiter.api.DisplayName;
2  import org.junit.jupiter.api.Nested;
3  import org.junit.jupiter.api.Test;
4
5  import static org.junit.jupiter.api.Assertions.assertNotSame;
6
7  @DisplayName("Writing assertions for objects")
8  class ObjectAssertionTest {
9
10     @Nested
11     @DisplayName("When two objects don't refer to the same object")
12     class WhenTwoObjectsDoNotReferToSameObject {
13
14        private final Object ACTUAL = new Object();
15        private final Object EXPECTED = new Object();
16
17        @Test
18        @DisplayName("Should not refer to the same object")
19        void shouldNotReferToSameObject() {
20           assertNotSame(EXPECTED, ACTUAL);
21        }
22     }
23 }
```

Let's move on and find out how we can verify that two arrays are equal.

Asserting That Two Arrays Are Equal

If we want to verify that two arrays are equal, we have to use theassertArrayEquals() method of the Assertions class. For example, if we want verify that twoint arrays are equal, we have to use this assertion:

```
1  import org.junit.jupiter.api.DisplayName;
2  import org.junit.jupiter.api.Nested;
3  import org.junit.jupiter.api.Test;
4
5  import static org.junit.jupiter.api.Assertions.assertArrayEquals;
6
7  @DisplayName("Write assertions for arrays")
8  class ArrayAssertionTest {
9
10     @Nested
11     @DisplayName("When arrays contain integers")
12     class WhenArraysContainIntegers {
13
14        final int[] ACTUAL = new int[]{2, 5, 7};
15        final int[] EXPECTED = new int[]{2, 5, 7};
16
17        @Test
18        @DisplayName("Should contain the same integers")
19        void shouldContainSameIntegers() {
20           assertArrayEquals(EXPECTED, ACTUAL);
21        }
22     }
23 }
```

Two arrays are considered as equal if:

- They are both null or empty.
- Both arrays contain the "same" objects or values. To be more specific, JUnit 5 iterates both arrays one element at a time and ensures that the elements found from the given index are equal.

Next, we will find out how wen can verify that two iterables are equal.

Asserting That Two Iterables Are Equal

If we want to verify that two iterables are deeply equal, we have to use the assertIterableEquals() method of the Assertions class. For example, if we want verify that two Integer lists are deeply equal, we have to use this assertion:

```
1  import org.junit.jupiter.api.DisplayName;
2  import org.junit.jupiter.api.Nested;
3  import org.junit.jupiter.api.Test;
4
5  import java.util.Arrays;
6  import java.util.List;
7
8  import static org.junit.jupiter.api.Assertions.assertIterableEquals;
9
10 @DisplayName("Writing assertions for lists")
11 class ListAssertionTest {
12
13     @Nested
14     @DisplayName("When we compare two lists")
15     class WhenWeCompareTwoLists {
16
17         private final List<Integer> FIRST = Arrays.asList(1, 2, 3);
18         private final List<Integer> SECOND = Arrays.asList(1, 2, 3);
19
20         @Test
21         @DisplayName("Should contain the same elements")
22         void shouldContainSameElements() {
23             assertIterableEquals(FIRST, SECOND);
24         }
25     }
26 }
```

Two iterables are considered as equal if:

- They are both null or empty.
- Both iterables contain the “same” objects or values. To be more specific, JUnit 5 iterates both iterables one element at a time and ensures that the elements found from the given index are equal.

Let’s move on and find out how we can write assertions for the exception thrown by the system under test.

Writing Assertions for Exceptions

If we want to write assertions for the exceptions thrown by the system under test, we have to use the assertThrows() method of the Assertions class. This method takes the following method parameters:

- A Class object that specifies the type of the expected exception.
- An Executable object that invokes the system under test.
- An optional error message.

For example, if we want to verify that the system under test throws a NullPointerException, our assertion looks as follows:

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

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

@DisplayName("Writing assertions for exceptions")
class ExceptionAssertionTest {

    @Test
    @DisplayName("Should throw the correct exception")
    void shouldThrowCorrectException() {
        assertThrows(
                NullPointerException.class,
                () -> { throw new NullPointerException(); }
        );
    }
}
```

Because the assertThrows() method returns the thrown exception object, we can also write additional assertions for the thrown exception. For example, if we want to verify that the thrown exception has the correct message, we can use the following assertions:

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

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

@DisplayName("Writing assertions for exceptions")
class ExceptionAssertionTest {

    @Test
    @DisplayName("Should throw an exception that has the correct message")
    void shouldThrowAnExceptionWithCorrectMessage() {
        final NullPointerException thrown = assertThrows(
                NullPointerException.class,
                () -> { throw new NullPointerException("Hello World!"); }
        );
        assertEquals("Hello World!", thrown.getMessage());
    }
}
```

On the other hand, even though a test method fails if the system under test throws an exception, sometimes we want to explicitly assert that no exception is thrown by the tested code. If this is the case, we have to use the assertDoesNotThrow() method of the Assertions class. When we want to verify that no exception is thrown by the system under test, we can use one of these two options:

First, if we don't want to write assertions for the value returned by the system under test, we have to pass the following method parameters to the assertDoesNotThrow() method:

- An Executable object that invokes the system under test.
- An optional error message.

For example, if we want to verify that the system under test doesn't throw an exception, our assertion looks as follows:

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

import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;

@DisplayName("Writing assertions for exceptions")
class ExceptionAssertionTest {

    @Test
    @DisplayName("Should not throw an exception")
    void shouldNotThrowException() {
        assertDoesNotThrow(() -> {});
    }
}
```

Second, if we want to write assertions for the value returned by the system under test, we have to pass the following method parameters to the assertDoesNotThrow() method:

- A ThrowingSupplier<T> object that invokes the system under test (and returns the return value).
- An optional error message.

For example, if we want to verify that the system under test doesn't throw an exception AND we want to verify that the system under test returns the correct message, our assertion looks as follows:

```
1  import org.junit.jupiter.api.DisplayName;
2  import org.junit.jupiter.api.Test;
3
4  import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;
5  import static org.junit.jupiter.api.Assertions.assertEquals;
6
7  @DisplayName("Writing assertions for exceptions")
8  class ExceptionAssertionTest {
9
10     @Test
11     @DisplayName("Should not throw an exception")
12     void shouldNotThrowException() {
13         String message = assertDoesNotThrow(() -> { return "Hello World!"; } );
14         assertEquals("Hello World!", message);
15     }
16 }
```

Next, we will find out how we can write assertions for the execution time of the system under test.

Writing Assertions for the Execution Time of the System Under Test

If we want to ensure that the execution of the system under test is completed before a specified timeout is exceeded, we can use the assertTimeout() and assertTimeoutPreemptively() methods of the Assertions class. Both of these methods take the following method parameters:

- A Duration object that specifies the timeout.
- An Executable or a ThrowingSupplier object that invokes the system under test.
- An optional error message that is shown if the specified timeout is exceeded.

Even though these two methods are quite similar, they have one crucial difference. This difference is explained in the following:

- If we use the assertTimeout() method, the provided Executable or ThrowingSupplier will be executed in the same thread as the code that calls it. Also, this method doesn't abort the execution if the timeout is exceeded.
- If we use the assertTimeoutPreemptively() method, the provided Executable or ThrowingSupplier will be executed in a different thread than the code that calls it. Also, this method aborts the execution if the timeout is exceeded.

As we see, we can verify that the execution of the system under test is completed before a specified timeout is exceeded by using one of these two options:

First, if we want that the execution is not aborted if the timeout is exceeded, we have to use the assertTimeout() method of the Assertions class. For example, if we want to verify that the system under test returns the message: 'Hello world!' before the specified timeout (50ms) is exceeded, we have to write assertions that look as follows:

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

import java.time.Duration;

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

@DisplayName("Writing assertions for the execution time of the system under test")
class TimeoutAssertionTest {

    @Test
    @DisplayName("Should return the correct message before timeout is exceeded")
    void shouldReturnCorrectMessageBeforeTimeoutIsExceeded() {
        final String message = assertTimeout(Duration.ofMillis(50), () -> {
            Thread.sleep(20);
            return "Hello World!";
        });
        assertEquals("Hello World!", message);
    }
}
```

Second, if we want that the execution is aborted if the timeout is exceeded, we have to use the assertTimeoutPreemptively() method of the Assertions class. For example, if we want to verify that the system under test returns the message: 'Hello world!' before the specified timeout (50ms) is exceeded, we have to write assertions that look as follows:

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

import java.time.Duration;

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

@DisplayName("Writing assertions for the execution time of the system under test")
class TimeoutAssertionTest {

    @Test
    @DisplayName("Should return the correct message before timeout is exceeded")
    void shouldReturnCorrectMessageBeforeTimeoutIsExceeded() {
        final String message = assertTimeoutPreemptively(Duration.ofMillis(50), () -> {
            Thread.sleep(20);
            return "Hello World!";
        });
        assertEquals("Hello World!", message);
    }
}
```

Additional Reading:

- The Javadoc of the Duration class

We can now write basic assertions with JUnit 5. Let's move on and find out how we can customize the error messages which are shown by JUnit 5 if an assertion fails.

Providing a Custom Error Message

As we remember, if we want to specify a custom error message that is shown when our assertion fails, we have to pass this message as the last method parameter of the invoked assertion method. We can create this message by using one of these two options:

First, we can create a newString object and pass this object as the last method parameter of the invoked assertion assertion method. This is a good choice if our error message has no parameters. For example, if we want to provide a custom error message for an assertion which verifies that a boolean value is false, we have to write an assertion that looks as follows:

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

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

@DisplayName("Write assertions for booleans")
class BooleanAssertionTest {

    @Nested
    @DisplayName("When boolean is false")
    class WhenBooleanIsFalse {

        @Test
        @DisplayName("Should be false")
        void shouldBeFalse() {
            assertFalse(false, "The boolean is not false");
        }
    }
}
```

Second, we can create a message supplier (Supplier<String>) and pass this supplier as the last method parameter of the invoked assertion method. If we use this approach, JUnit 5 creates the actual error messages **only** if our assertion fails. That's why this is a good choice if we want to create a "complex" error message that has parameters.

For example, if we want to provide a custom error message for an assertion which verifies that a map contains the given key, we have to write an assertion that looks as follows:

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

import java.util.HashMap;
import java.util.Map;

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

@DisplayName("Writing assertions for maps")
class MapAssertionTest {

    private static final String KEY = "key";
    private static final String VALUE = "value";

    private Map<String, String> map;

    @BeforeEach
    void createAndInitializeMap() {
        map = new HashMap<>();
        map.put(KEY, VALUE);
    }

    @Test
    @DisplayName("Should contain the correct key")
    void shouldContainCorrectKey() {
        assertTrue(
                map.containsKey(KEY),
                () -> String.format("The map doesn't contain the key: %s", KEY)
        );
    }
}
```

> It's good to understand that the custom error message doesn't override the default error message shown if an assertion fails. It is simply a prefix that is prepended to the default error message of the used assertion object. At first, this feels a bit weird, but it's actually quite useful after you get used to it.

Next, we will find out how we can group assertions with JUnit 5.

Grouping Assertions

If we have to write an assertion for a state that requires multiple assertions, we can group our assertions by using the assertAll() method of the Assertions class. This method takes the following method parameters:

- An optional heading that identifies the asserted state.
- An array, a Collection, or a Stream of Executable objects which invoke our assertions.

When we invoke the assertAll() method, it invokes all assertions given as a method parameter and reports all assertion failures after all assertions have been run.

Let's assume that we have to write an assertion which verifies that a Person object has the correct name. The source code of the Person class looks as follows:

```
public class Person {

    private String firstName;
    private String lastName;

    public Person() {}

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

As we can see, if we want to verify that a person has the correct name, we have to verify that the asserted Person object has the correct first and last name. In other words, we have to write an assertion that looks as follows:

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

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

@DisplayName("Group multiple assertions")
class GroupAssertionsTest {

    private static final String FIRST_NAME = "Jane";
    private static final String LAST_NAME = "Doe";

    private Person person;

    @BeforeEach
    void createPerson() {
        person = new Person();
        person.setFirstName(FIRST_NAME);
        person.setLastName(LAST_NAME);
    }

    @Test
    @DisplayName("Should have the correct name")
    void shouldHaveCorrectName() {
        assertAll("name",
                () -> assertEquals(FIRST_NAME,
                        person.getFirstName(),
                        "The first name is incorrect"
                ),
                () -> assertEquals(LAST_NAME,
                        person.getLastName(),
                        "The last name is incorrect"
                )
        );
    }
}
```

We can now write basic assertions with JUnit 5, provide a custom error message that is shown when an assertion fail, and group assertions with JUnit 5.

Additional Reading:

- JUnit 5 User Guide: 2.4 Assertions
- The Javadoc of the Assertions class

Let's summarize what we learned from this blog post.

Summary

This blog post has taught us four things:

- If we want to write assertions by using the "standard" JUnit 5 API, we must use the org.junit.jupiter.api.Assertions class.
- If we want to specify a custom error message that has no parameters, we have to create a new String object and pass this object as the last method parameter of the invoked assertion method.
- If we want to specify a "complex" error message that has parameters, we have to create a message supplier (Supplier<String>) and pass this supplier as the last method parameter of the invoked assertion method.
- If we have to write an assertion for a state that requires multiple assertions, we can group our assertions by using the assertAll() method of the Assertions class.

P.S. You can get the example application of this blog post from Github

RELATED POSTS

WEEKLY /

Guide to JUnit 5 Parameterized Tests

Last modified: January 7, 2020

by Ali Dehghani

Testing

JUnit 5

1. Overview

JUnit 5, the next generation of JUnit, facilitates writing developer tests with new and shiny features.

One such feature is *parameterized tests*. This feature enables us to **execute a single test method multiple times with different parameters.**

In this tutorial, we're going to explore parameterized tests in depth, so let's get started!

That means when using Maven, we'll add the following to our *pom.xml*:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.4.2</version>
    <scope>test</scope>
</dependency>
```

Also, when using Gradle, we'll specify it a little differently:

```
testCompile("org.junit.jupiter:junit-jupiter-params:5.4.2")
```

3. First Impression

Let's say we have an existing utility function and we'd like to be confident about its behavior:

```
public class Numbers {
    public static boolean isOdd(int number) {
        return number % 2 != 0;
    }
}
```

Parameterized tests are like other tests except that we add the *@ParameterizedTest* annotation:

```
@ParameterizedTest
@ValueSource(ints = {1, 3, 5, -3, 15, Integer.MAX_VALUE}) // six numbers
void isOdd_ShouldReturnTrueForOddNumbers(int number) {
    assertTrue(Numbers.isOdd(number));
}
```

JUnit 5 test runner executes this above test – and consequently, the *isOdd* method – six times. And each time, it assigns a different value from the *@ValueSource* array to the *number* method parameter.

So, this example shows us two things we need for a parameterized test:

- **a source of arguments**, an *int* array, in this case
- **a way to access them**, in this case, the *number* parameter

There is also one more thing not evident with this example, so stay tuned.

4. Argument Sources

As we should know by now, a parameterized test executes the same test multiple times with different arguments.

And, hopefully, we can do more than just numbers – so, let's explore!

4.1. Simple Values

With the *@ValueSource* annotation, we can pass an array of literal values to the test method.

For example, suppose we're going to test our simple *isBlank* method:

```
public class Strings {
    public static boolean isBlank(String input) {
        return input == null || input.trim().isEmpty();
    }
}
```

We expect from this method to return *true* for *null* for blank strings. So, we can write a parameterized test like the following to assert this behavior:

```
@ParameterizedTest
@ValueSource(strings = {"", "  "})
void isBlank_ShouldReturnTrueForNullOrBlankStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

As we can see, JUnit will run this test two times and each time assigns one argument from the array to the method parameter.

One of the limitations of value sources is that they only support the following types:

- *short* (with the *shorts* attribute)
- *byte* (with the *bytes* attribute)
- *int* (with the *ints* attribute)
- *long* (with the *longs* attribute)
- *float* (with the *floats* attribute)
- *double* (with the *doubles* attribute)
- *char* (with the *chars* attribute)
- *java.lang.String* (with the *strings* attribute)
- *java.lang.Class* (with the *classes* attribute)

Also, **we can only pass one argument to the test method each time**.

And before going any further, did anyone notice we didn't pass *null* as an argument? That's another limitation: **We can't pass *null* through a *@ValueSource*, even for *String* and *Class*!**

4.2. Null and Empty Values

As of JUnit 5.4, we can pass a single *null* value to a parameterized test method using *@NullSource*:

```
@ParameterizedTest
@NullSource
void isBlank_ShouldReturnTrueForNullInputs(String input) {
    assertTrue(Strings.isBlank(input));
}
```

Since primitive data types can't accept *null* values, we can't use the *@NullSource* for primitive arguments.

Quite similarly, we can pass empty values using the *@EmptySource* annotation:

```
@ParameterizedTest
@EmptySource
void isBlank_ShouldReturnTrueForEmptyStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

***@EmptySource* passes a single empty argument to the annotated method**

For *String* arguments, the passed value would be as simple as an empty *String*. **Moreover, this parameter source can provide empty values for *Collection* types and arrays**.

In order to pass both *null* and empty values, we can use the composed *@NullAndEmptySource* annotation:

```
@ParameterizedTest
@NullAndEmptySource
void isBlank_ShouldReturnTrueForNullAndEmptyStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

As with the *@EmptySource*, the composed annotation works for *String*s, *Collection*s, and arrays.

In order to pass a few more empty string variations to the parameterized test, **we can combine *@ValueSource, @NullSource, and @EmptySource* together:**

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = {"  ", "\t", "\n"})
void isBlank_ShouldReturnTrueForAllTypesOfBlankStrings(String input) {
    assertTrue(Strings.isBlank(input));
}
```

4.3. Enum

In order to run a test with different values from an enumeration, we can use the*@EnumSource* annotation.

For example, we can assert that all month numbers are between 1 and 12:

```
@ParameterizedTest
@EnumSource(Month.class) // passing all 12 months
void getValueForAMonth_IsAlwaysBetweenOneAndTwelve(Month month) {
    int monthNumber = month.getValue();
    assertTrue(monthNumber >= 1 && monthNumber <= 12);
}
```

Or, we can filter out a few months by using the *names* attribute.

How about asserting the fact that April, September, June, and November are 30 days long:

```
@ParameterizedTest
@EnumSource(value = Month.class, names = {"APRIL", "JUNE", "SEPTEMBER", "NOVEMBER"})
void someMonths_Are30DaysLong(Month month) {
    final boolean isALeapYear = false;
    assertEquals(30, month.length(isALeapYear));
}
```

By default, the *names* will only keep the matched enum values. We can turn this around by setting the *mode* attribute to *EXCLUDE*:

```
@ParameterizedTest
@EnumSource(
  value = Month.class,
  names = {"APRIL", "JUNE", "SEPTEMBER", "NOVEMBER", "FEBRUARY"},
  mode = EnumSource.Mode.EXCLUDE)
void exceptFourMonths_OthersAre31DaysLong(Month month) {
    final boolean isALeapYear = false;
    assertEquals(31, month.length(isALeapYear));
}
```

In addition to literal strings, we can pass a regular expression to the *names* attribute:

```
@ParameterizedTest
@EnumSource(value = Month.class, names = ".+BER", mode = EnumSource.Mode.MATCH_ANY)
void fourMonths_AreEndingWithBer(Month month) {
    EnumSet<Month> months =
      EnumSet.of(Month.SEPTEMBER, Month.OCTOBER, Month.NOVEMBER, Month.DECEMBER);
    assertTrue(months.contains(month));
}
```

Quite similar to *@ValueSource*, *@EnumSource* is only applicable when we're going to pass just one argument per test execution.

4.4. CSV Literals

Suppose we're going to make sure that the *toUpperCase()* method from *String* generates the expected uppercase value. *@ValueSource* won't be enough.

In order to write a parameterized test for such scenarios, we have to:

- Pass an **input value** *and* an **expected value** to the test method
- Compute the **actual result with those input values**
- **Assert the actual value with the expected value**

So, we need argument sources capable of passing multiple arguments. The *@CsvSource* is one of those sources:

```
@ParameterizedTest
@CsvSource({"test,TEST", "tEst,TEST", "Java,JAVA"})
void toUpperCase_ShouldGenerateTheExpectedUppercaseValue(String input, String expected) {
    String actualValue = input.toUpperCase();
    assertEquals(expected, actualValue);
}
```

The *@CsvSource* accepts an array of comma-separated values and each array entry corresponds to a line in a CSV file.

This source takes one array entry each time, splits it by comma and passes each array to the annotated test method as separate parameters. By default, the comma is the column separator but we can customize it using the *delimiter* attribute:

```
@ParameterizedTest
@CsvSource(value = {"test:test", "tEst:test", "Java:java"}, delimiter = ':')
void toLowerCase_ShouldGenerateTheExpectedLowercaseValue(String input, String expected) {
    String actualValue = input.toLowerCase();
    assertEquals(expected, actualValue);
}
```

```
}
```

Now it's a *colon*-separated value, still a CSV!

4.5. CSV Files

Instead of passing the CSV values inside the code, we can refer to an actual CSV file.

For example, we could use a CSV file like:

```
input,expected
test,TEST
tEst,TEST
Java,JAVA
```

We can load the CSV file and **ignore the header column** with *@CsvFileSource*:

```
@ParameterizedTest
@CsvFileSource(resources = "/data.csv", numLinesToSkip = 1)
void toUpperCase_ShouldGenerateTheExpectedUppercaseValueCSVFile(
  String input, String expected) {
    String actualValue = input.toUpperCase();
    assertEquals(expected, actualValue);
}
```

The *resources* attribute represents the CSV file resources on the classpath to read. And, we can pass multiple files to it.

The *numLinesToSkip* attribute represents the number of lines to skip when reading the CSV files. **By default, *@CsvFileSource* does not skip any lines, but this feature is usually useful for skipping the header lines** like we did here.

Just like the simple *@CsvSource*, the delimiter is customizable with the *delimiter* attribute.

In addition to the column separator:

- The line separator can be customized using the *lineSeparator* attribute – a newline is the default value
- The file encoding is customizable using the *encoding* attribute – UTF-8 is the default value

4.6. Method

The argument sources we've covered so far are somewhat simple and share one limitation: It's hard or impossible to pass complex objects using them!

One approach to **providing more complex arguments is to use a method as an argument source.**

Let's test the *isBlank* method with a *@MethodSource*:

```
@ParameterizedTest
@MethodSource("provideStringsForIsBlank")
void isBlank_ShouldReturnTrueForNullOrBlankStrings(String input, boolean expected) {
    assertEquals(expected, Strings.isBlank(input));
}
```

The name we supply to *@MethodSource* needs to match an existing method.

So let's next write *provideStringsForIsBlank*, **a *static* method that returns a *Stream* of *Argument*s**:

```
private static Stream<Arguments> provideStringsForIsBlank() {
    return Stream.of(
      Arguments.of(null, true),
      Arguments.of("", true),
      Arguments.of("  ", true),
      Arguments.of("not blank", false)
    );
}
```

Here we're literally returning a stream of arguments, but it's not a strict requirement. For example, **we can return any other collection-like interfaces like *List*.**

If we're going to provide just one argument per test invocation, then it's not necessary to use the *Arguments* abstraction:

```
@ParameterizedTest
@MethodSource // hmm, no method name ...
void isBlank_ShouldReturnTrueForNullOrBlankStringsOneArgument(String input) {
    assertTrue(Strings.isBlank(input));
}

private static Stream<String> isBlank_ShouldReturnTrueForNullOrBlankStringsOneArgument() {
    return Stream.of(null, "", "  ");
}
```

When we don't provide a name for the *@MethodSource*, JUnit will search for a source method with the same

name as the test method.

Sometimes it's useful to share arguments between different test classes. In these cases, we can refer to a source method outside of the current class by its fully-qualified name:

```
class StringsUnitTest {

    @ParameterizedTest
    @MethodSource("com.baeldung.parameterized.StringParams#blankStrings")
    void isBlank_ShouldReturnTrueForNullOrBlankStringsExternalSource(String input) {
        assertTrue(Strings.isBlank(input));
    }
}

public class StringParams {

    static Stream<String> blankStrings() {
        return Stream.of(null, "", "  ");
    }
}
```

Using the *FQN#methodName* format we can refer to an external static method.

4.7. Custom Argument Provider

Another advanced approach to pass test arguments is to use a custom implementation of an interface

called *ArgumentsProvider*:

```
class BlankStringsArgumentsProvider implements ArgumentsProvider {

    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return Stream.of(
          Arguments.of((String) null),
          Arguments.of(""),
          Arguments.of("   ")
        );
    }
}
```

Then we can annotate our test with the *@ArgumentsSource* annotation to use this custom provider:

```
@ParameterizedTest
@ArgumentsSource(BlankStringsArgumentsProvider.class)
void isBlank_ShouldReturnTrueForNullOrBlankStringsArgProvider(String input) {
    assertTrue(Strings.isBlank(input));
}
```

Let's make the custom provider a more pleasant API to use with a custom annotation!

4.8. Custom Annotation

How about loading the test arguments from a static variable? Something like:

```
static Stream<Arguments> arguments = Stream.of(
  Arguments.of(null, true), // null strings should be considered blank
  Arguments.of("", true),
  Arguments.of("  ", true),
  Arguments.of("not blank", false)
);

@ParameterizedTest
@VariableSource("arguments")
void isBlank_ShouldReturnTrueForNullOrBlankStringsVariableSource(
  String input, boolean expected) {
    assertEquals(expected, Strings.isBlank(input));
}
```

Actually, **JUnit 5 does not provide this!** However, we can roll our own solution.

First off, we can create an annotation:

```
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@ArgumentsSource(VariableArgumentsProvider.class)
public @interface VariableSource {

    /**
     * The name of the static variable
     */
    String value();
}
```

Then we need to somehow **consume the annotation** details and **provide test arguments**. JUnit 5 provides two abstractions to achieve those two things:

- *AnnotationConsumer* to consume the annotation details
- *ArgumentsProvider* to provide test arguments

So, we next need to make the *VariableArgumentsProvider* class read from the specified static variable and return its

value as test arguments:

```
class VariableArgumentsProvider
  implements ArgumentsProvider, AnnotationConsumer<VariableSource> {

    private String variableName;

    @Override
    public Stream<? extends Arguments> provideArguments(ExtensionContext context) {
        return context.getTestClass()
                .map(this::getField)
                .map(this::getValue)
                .orElseThrow(() ->
                  new IllegalArgumentException("Failed to load test arguments"));
    }

    @Override
    public void accept(VariableSource variableSource) {
        variableName = variableSource.value();
    }

    private Field getField(Class<?> clazz) {
        try {
            return clazz.getDeclaredField(variableName);
        } catch (Exception e) {
            return null;
        }
    }

    @SuppressWarnings("unchecked")
    private Stream<Arguments> getValue(Field field) {
        Object value = null;
        try {
            value = field.get(null);
        } catch (Exception ignored) {}

        return value == null ? null : (Stream<Arguments>) value;
    }
}
```

And it works like a charm!

5. Argument Conversion

5.1. Implicit Conversion

Let's re-write one of those *@EnumTest*s with a *@CsvSource*:

```
@ParameterizedTest
@CsvSource({"APRIL", "JUNE", "SEPTEMBER", "NOVEMBER"}) // Pssing strings
void someMonths_Are30DaysLongCsv(Month month) {
    final boolean isALeapYear = false;
    assertEquals(30, month.length(isALeapYear));
}
```

This shouldn't work, right? But, somehow it does!

So, JUnit 5 converts the *String* arguments to the specified enum type. To support use cases like this, JUnit Jupiter provides a number of built-in implicit type converters.

The conversion process depends on the declared type of each method parameter. The implicit conversion can convert the *String* instances to types like:

- *UUID*

- *Locale*
- *LocalDate, LocalTime, LocalDateTime, Year, Month, etc.*
- *File* and *Path*
- *URL* and *URI*
- *Enum* subclasses

5.2. Explicit Conversion

Sometimes we need to provide a custom and explicit converter for arguments.

Suppose we want to convert strings with the *yyyy/mm/dd* format to *LocalDate* instances. First off, we need to implement the *ArgumentConverter* interface:

```
class SlashyDateConverter implements ArgumentConverter {

    @Override
    public Object convert(Object source, ParameterContext context)
      throws ArgumentConversionException {
        if (!(source instanceof String)) {
            throw new IllegalArgumentException(
              "The argument should be a string: " + source);
        }
        try {
            String[] parts = ((String) source).split("/");
            int year = Integer.parseInt(parts[0]);
            int month = Integer.parseInt(parts[1]);
            int day = Integer.parseInt(parts[2]);

            return LocalDate.of(year, month, day);
        } catch (Exception e) {
            throw new IllegalArgumentException("Failed to convert", e);
        }
    }
}
```

Then we should refer to the converter via the *@ConvertWith* annotation:

```
@ParameterizedTest
@CsvSource({"2018/12/25,2018", "2019/02/11,2019"})
void getYear_ShouldWorkAsExpected(
  @ConvertWith(SlashyDateConverter.class) LocalDate date, int expected) {
    assertEquals(expected, date.getYear());
}
```

6. Argument Accessor

By default, each argument provided to a parameterized test corresponds to a single method parameter. Consequently, when passing a handful of arguments via an argument source, the test method signature gets very large and messy.

One approach to address this issue is to encapsulate all passed arguments into an instance of *ArgumentsAccessor* and retrieve arguments by index and type.

For example, let's consider our *Person* class:

```
class Person {

    String firstName;
    String middleName;
    String lastName;

    // constructor

    public String fullName() {
        if (middleName == null || middleName.trim().isEmpty()) {
            return String.format("%s %s", firstName, lastName);
        }

        return String.format("%s %s %s", firstName, middleName, lastName);
    }
}
```

Then, in order to test the *fullName()* method, we'll pass four arguments: *firstName*, *middleName*, *lastName*, and the *expected fullName*. We can use the *ArgumentsAccessor* to retrieve the test arguments instead of declaring them as method parameters:

```
@ParameterizedTest
@CsvSource({"Isaac,,Newton,Isaac Newton", "Charles,Robert,Darwin,Charles Robert Darwin"})
void fullName_ShouldGenerateTheExpectedFullName(ArgumentsAccessor argumentsAccessor) {
    String firstName = argumentsAccessor.getString(0);
    String middleName = (String) argumentsAccessor.get(1);
    String lastName = argumentsAccessor.get(2, String.class);
    String expectedFullName = argumentsAccessor.getString(3);

    Person person = new Person(firstName, middleName, lastName);
    assertEquals(expectedFullName, person.fullName());
}
```

Here, we're encapsulating all passed arguments into an *ArgumentsAccessor* instance and then, in the test method body, retrieving each passed argument with its index. In addition to just being an accessor, type conversion is supported through *get** methods:

- *getString(index)* retrieves an element at a specific index and converts it to *String* – the same is true for primitive types
- *get(index)* simply retrieves an element at a specific index as an *Object*
- *get(index, type)* retrieves an element at a specific index and converts it to the given *type*

7. Argument Aggregator

Using the *ArgumentsAccessor* abstraction directly may make the test code less readable or reusable. In order to address these issues, we can write a custom and reusable aggregator.

To do that, we implement the *ArgumentsAggregator* interface:

```
class PersonAggregator implements ArgumentsAggregator {

    @Override
    public Object aggregateArguments(ArgumentsAccessor accessor, ParameterContext context)
      throws ArgumentsAggregationException {
        return new Person(
          accessor.getString(1), accessor.getString(2), accessor.getString(3));
    }
}
```

And then we reference it via the *@AggregateWith* annotation:

```
@ParameterizedTest
@CsvSource({"Isaac Newton,Isaac,,Newton", "Charles Robert Darwin,Charles,Robert,Darwin"})
void fullName_ShouldGenerateTheExpectedFullName(
  String expectedFullName,
  @AggregateWith(PersonAggregator.class) Person person) {

    assertEquals(expectedFullName, person.fullName());
}
```

The *PersonAggregator* takes the last three arguments and instantiates a *Person* class out of them.

8. Customizing Display Names

By default, the display name for a parameterized test contains an invocation index along with a *String* representation of all passed arguments, something like:

```
├─ someMonths_Are30DaysLongCsv(Month)
|     |  ├─ [1] APRIL
|     |  ├─ [2] JUNE
|     |  ├─ [3] SEPTEMBER
|     |  └─ [4] NOVEMBER
```

However, we can customize this display via the *name* attribute of the *@ParameterizedTest* annotation:

```
@ParameterizedTest(name = "{index} {0} is 30 days long")
@EnumSource(value = Month.class, names = {"APRIL", "JUNE", "SEPTEMBER", "NOVEMBER"})
void someMonths_Are30DaysLong(Month month) {
    final boolean isALeapYear = false;
    assertEquals(30, month.length(isALeapYear));
}
```

April is 30 days long surely is a more readable display name*:*

```
├─ someMonths_Are30DaysLong(Month)
|     |  ├─ 1 APRIL is 30 days long
|     |  ├─ 2 JUNE is 30 days long
|     |  ├─ 3 SEPTEMBER is 30 days long
|     |  └─ 4 NOVEMBER is 30 days long
```

The following placeholders are available when customizing the display name:

- **{index}** will be replaced with the invocation index – simply put, the invocation index for the first execution is 1, for the second is 2, and so on
- **{arguments}** is a placeholder for the complete, comma-separated list of arguments
- **{0}, {1}, …** are placeholders for individual arguments

9. Conclusion

In this article, we've explored the nuts and bolts of parameterized tests in JUnit 5.

We learned that parameterized tests are different from normal tests in two aspects: they're annotated with the *@ParameterizedTest*, and they need a source for their declared arguments.

Also, by now, we should now that JUnit provides some facilities to convert the arguments to custom target types or to customize the test names.

As usual, the sample codes are available on our GitHub project, so make sure to check it out!