JUnit 5 tutorial - Learn how to write unit tests

Prerequisite:

- Eclipse
- Junit5

PART 1. OVERVIEW

1 – Configuration for using JUnit 5–

1. To use JUnit5 in an Maven project, you need to:
2. Configure to use Java 11 or higher, as this is required by JUnit5
3. Configure the maven-surefire-plugin and maven-failsafe-plugin to be at version 2.22.2 so that they can run JUnit5
4. Add dependencies to the JUnit5 API and engine for your test code
5. Therefore you need to adjust your pom file
*6. Once you have done this, you can start using JUnit5 in your Maven project for writing unit tests.Right-click your pom file, select **Maven Update Project** and select your project. This triggers an update of your project settings and dependencies.*

2 – How to define a test in JUnit?–

A JUnit test is a method contained in a class which is only used for testing. This is called a Test class. To mark a method as a test method, annotate it with the @Test annotation. This method executes the code under test..

```java
package com.vogella.junit.first;

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

import org.junit.jupiter.api.Test;

class AClassWithOneJUnitTest {

    @Test
    void demoTestMethod() {
        assertTrue(true);
    }
}
```

You can use assert methods, provided by JUnit or another assert framework, to check an expected result versus the actual result. Such statement are called asserts or assert statements.

3 – JUnit test class naming conventions.

Build tools like Maven use a pattern to decide if a class is a test classes or not. The following is the list of classes Maven considers automatically during its build:

```
**/Test*.java          1
**/*Test.java          2
**/*Tests.java         3
**/*TestCase.java      4
```

1. includes all of its subdirectories and all Java filenames that start with `Test`.
2. includes all of its subdirectories and all Java filenames that end with `Test`.
3. includes all of its subdirectories and all Java filenames that end with `Tests`.
4. includes all of its subdirectories and all Java filenames that end with `TestCase`.

4 – Where should the test be located?

Build tools like Maven use a pattern to decide if a class is a test classes or not. The following is the list of classes Maven considers automatically during its build:

Typical, unit tests are created in a separate source folder to keep the test code separate from the real code. The standard convention from the Maven and Gradle build tools is to use:

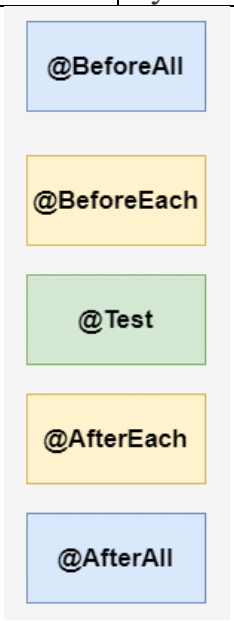
- src/main/java - for Java classes
- src/test/java - for test classes

5 – JUnit 5 Annotations

Below are some basic JUnit 5 annotations that we often use when we write tests.

JUnit 5 Annotations	Descriptions
@BeforeEach	The annotated method will be run before each test method(annotated with **@Test**) in the current class.
@AfterEach	The annotated method will be run after each test method (annotated with **@Test**) in the current class.
@BeforeAll	The annotated method will be run before all test methods in the current class.
@AfterAll	The annotated method will be run after all test methods in the current class.
@Test	Declares a test method
@DisplayName	Define custom display name for a test class or test method
@Disable	Is used to disable or ignore a test class or method.
@Nested	Used to declare nested test classes

@Tag	Declare tags for test discovering and filtering
@TestFactory	Denotes a method is a test factory for dynamic tests in JUnit 5



Below is an example of test class written by using JUnit 5 annotations:

```
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.Disabled;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;
@RunWith(JUnitPlatform.class)
public class JUnit5TestExample {
 @BeforeAll
 static void initAll() {
  System.out.println("@BeforeAll - Run before all methods once");
 }
 @BeforeEach
 void init() {
  System.out.println("  @BeforeEach - Run before each test methods ");
 }

 @DisplayName("First test")
 @Test
 void testMethod1() {
  System.out.println("    Test method 1");

 }
 @Test
 @Disabled
 void testMethod2() {
  System.out.println("  Test method 2");
 }

 @Test
 void testMethod3() {
  System.out.println("    Test method 3");

 }

 @AfterEach
 void tearDown() {
  System.out.println("  @AfterEach - Run after each test methods ");
 }

 @AfterAll
 static void tearDownAll() {
  System.out.println("@AfterAll - Run after all test methods once");
 }

}
```

PART 2. ASSERTIONS AND ASSUMPTIONS

Assume that we will write JUnit tests for below method which is used to convert a given string into Double

```
public final class StringUtils {

  public static Double convertToDouble(String str) {
    if (str == null) {
      return null;
    }
    return Double.valueOf(str);
  }
}
```

1. AssertAll, AssertNotNull, and AssertEquals

assertAll(Executable... executables)	Asserts that all supplied executables do not throw exceptions.
assertNotNull(Object actual)	Asserts that actual is not null.
assertEquals(expected, actual)	Asserts that expected and actual are equal.

In case we want to assert that two floats are equals, we can use the simple assertEquals assertion:

```
public final class StringUtils {

  public static Double convertToDouble(String str) {
    if (str == null) {
      return null;
    }
    return Double.valueOf(str);
  }
}
```

We will test the method **convertToDouble** with a test case that we will pass a parameter with a right Double value string. **AssertAll** is a new assertion in **JUnit 5** which allows us to execute a group of assertions in a bigger one.

```
@Test
 public void testConvertToDoubleOK() {
  // Test case with the age is a numeric string
  String age = "1990";
  Double expAge = Double.valueOf(age);
  Double actual = StringUtils.convertToDouble(age);

  assertAll("Do many assertions.", () -> {
   assertNotNull(actual);
   assertEquals(expAge, actual);
  });

  // Or Java 8 Lambdas style

  assertAll("Do many assertions.Java 8 Lambdas style", () -> {
   assertNotNull(actual, () -> "The actual is NULL");
   assertEquals(expAge, actual,
      () -> "The expected is: " + expAge + " while the actual is:" + actual);
  });

 }
```

2. AssertNull

assertNull(Object actual)	Asserts that actual is null.

In this example, we will test the method **convertToDouble** by passing a parameter as a NULL value.

```
@Test
 public void testConvertToDoubleWithNullArgument() {
  // Test case with the age is null
  String age = null;
  Double actual = StringUtils.convertToDouble(age);
  assertNull(actual, "The actual is not null");
  // Java 8 Style
  assertNull(actual, () -> "The actual is not null");
 }
```

3. AssertThrows

assertThrows(Class<T> expectedType, Executable executable)	*Asserts* that execution of the supplied executable throws an exception

	of the expectedType an d returns the exception.

In this example, we will test the method **convertToDouble** by passing a not numeric string as a parameter of the method. Note that in this case, the method will call: **Double.value(parameter)** to convert the string to Double and we will get the **NumberFormatException** exception.

```
@Test
 public void testConvertToDoubleThrowException() {
   // Test with the age is a non numeric string
   String age = "N/A";
   assertThrows(NumberFormatException.class, () -> {
     StringUtils.convertToDouble(age);
   });

   assertThrows(NumberFormatException.class, () -> {
     StringUtils.convertToDouble(age);
   });
```

4. AssertTrue and AssertFalse

assertTrue(boolean condition)	Asserts that the supplied condition is true.
assertFalse(boolean condition)	Asserts that the supplied condition is not true.

Assume that we will write JUnit tests for the following method:

```
public final class StringUtils {

  public static boolean isNullOrBlank(String st) {
    return st == null || st.trim().length() == 0;
  }

}
```

And we will write 3 test cases for this method by passing: NULL, empty and not empty string as parameters of the method.

```
@Test
 public void testIsNullOrBlankOK() {
   // Test the case that the input is NULL
   String input = null;

   assertTrue(StringUtils.isNullOrBlank(input));
   // Java 8 Lambdas Style
   assertTrue(StringUtils.isNullOrBlank(input), () -> "The string is not null or blank");

   // Test case with the input is empty
   input = " ";
   assertTrue(StringUtils.isNullOrBlank(input));

   // Test case with the input is not empty
   input = "abc";

   assertFalse(StringUtils.isNullOrBlank(input));

 }
```

5. AssertSame, AssertNotSame, and Fail

assertSame(Object expected, Object actual)	Asserts that expected and actual refer to the same object.
assertNotSame(Object unexpected, Object actual)	Asserts that expected and actual do not refer to the same object.
fail(String message)	Fails a test with the given failure message.

Assume that we will write JUnit tests for the following method which will test if a given string is null or not. If it is null then return the given default string.

```
public final class StringUtils {

  public static String getDefaultIfNull(final String st, final String defaultSt) {
    return st == null ? defaultSt : st;
  }

}
```

Here is our test method which using *JUnit 5 assertions*: **assertSame**, **assertNotSame**, and **fail**

```
public void testGetDefaultIfNull() {
    // Test case with input string is null
    String st = null;
    String defaultSt = "abc";

    String actual = StringUtils.getDefaultIfNull(st, defaultSt);
    assertSame(defaultSt, actual);
    // Java 8 Lambdas Style
    assertSame(defaultSt, actual, () -> "Expected ouput is not same with actual");

    // Test case with input string is not null
    st = "def";

    actual = StringUtils.getDefaultIfNull(st, defaultSt);
    assertNotSame(defaultSt, actual);
    // Java 8 Lambdas Style
    assertNotSame(defaultSt, actual, () -> "Expected ouput is same with actual");

    // Test case with input string is empty
    st = "";
    actual = StringUtils.getDefaultIfNull(st, defaultSt);
```

6. *ParameterizedTest*

You can also run parameterized tests by running a test multiple times and providing different set of inputs for each repetition. We have to add the ***@ParameterizedTest*** annotation to mark a test method as Parameterized Test.

In order to use JUnit 5 parameterized tests, we need to import the *junit-jupiter-params* artifact from JUnit Platform. That means, when using Maven, we'll add the following to our *pom.xml*:

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

6.1 Argument Sources

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

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

a. Simple Values

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

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 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 these types:

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

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

Before going any further, note that 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*.**

b. Null and Empty Values

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

```
@ParameterizedTest
@ValueSource(strings = {"", "  "})
void isBlank_ShouldReturnTrueForNullOrBlankStrings(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));
}
```

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));
}
```

c. 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.

We could also assert 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));
}
```

d. 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.

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, so still a CSV.

e. 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 this:

```
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 *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, we have these capabilities:

- 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.

f. 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.

6.2 **Repeatable Argument Source Annotations**

Advanced topics