Parameterized test

```
public class Calculator {

    public int maxOf(int a, int b) {
        if (a >= b) {
            return a;
        } else {
            return b;
        }
    }
}

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

class CalculatorTests {

    @Test
    void testMaxFirstArgGreaterThanSecondArg() {
        Calculator calculator = new Calculator();
        int result = calculator.maxOf(2, 1);
        int expected = 2;

        assertEquals(expected, result);
    }

    @Test
    void testMaxFirstArgLessThanSecondArg() {
        Calculator calculator = new Calculator();
        int result = calculator.maxOf(1, 2);
        int expected = 2;

        assertEquals(expected, result);
    }

    @Test
    void testMaxFirstArgEqualToSecondArg() {
        Calculator calculator = new Calculator();
        int result = calculator.maxOf(2, 2);
        int expected = 2;

        assertEquals(expected, result);
    }
}
```

Now, let's run these tests to be sure that our implementation of the max method successfully passes all the tests. Running the test using Gradle gives the following output:

```
CalculatorTests > testMaxFirstArgEqualToSecondArg() PASSED
CalculatorTests > testMaxFirstArgLessThanSecondArg() PASSED
CalculatorTests > testMaxFirstArgGreaterThanSecondArg() PASSED
```

However, if you look at these tests you will notice that they are nearly identical and the only difference is the values we use in their bodies. Do we have a way to write such tests in a cleaner manner? JUnit provides us such an option, which is called "parameterized tests".
First, let's add the following dependency to our project so that JUnit be able to work with parameterized tests.
Gradle:

```
dependencies {
   testImplementation "org.junit.jupiter:junit-jupiter-params:5.7.1"
}
```

Maven:

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-params</artifactId>
   <version>5.7.1</version>
   <scope>test</scope>
</dependency>
```

If you use JUnit Jupiter aggregator artifact dependency 'org.junit.jupiter:junit-jupiter:5.7.1', it automatically pulls in dependencies on junit-jupiter-api, junit-jupiter-params, and junit-jupiter-engine.

@ParameterizedTest

@ParameterizedTest allows us to invoke a single test method multiple times, passing different arguments to it.

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

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

class CalculatorTests {

   @ParameterizedTest
   @CsvSource({"2, 1, 2", "1, 2, 2", "1, 1, 1"})
```

```
    void testMax(int first, int second, int expected) {
        Calculator calculator = new Calculator();
        int result = calculator.maxOf(first, second);

        assertEquals(expected, result);
    }
}
```

In this example, we use the @ParameterizedTest annotation instead of @Test to specify that the corresponding test should be executed multiple times with different arguments.

We also use the @CsvSource annotation to provide an array of such arguments. JUnit has plenty of annotations for different sources of arguments, such as @ValueSource, @EnumSource, @MethodSource, @CsvSource, @CsvFileSource, and @ArgumentsSource

Note that the test method now has three parameters: int first, int second and int expected, which are used in the body of the test method, and respective arguments are supplied by JUnit at runtime based on the specified argument source. Let's run this test:

```
CalculatorTests > [1] 2, 1, 2 PASSED
CalculatorTests > [2] 1, 2, 2 PASSED
CalculatorTests > [3] 1, 1, 1 PASSED
```

The default output consists of the current invocation index and the list of the arguments. We can specify a custom message format for a test using attributes and placeholders, for example:

```
@ParameterizedTest(name = "{index} => maxOf({0}, {1}) == {2}")
```

The execution of the same test with such a custom display name looks as follows:

```
CalculatorTests > 1 => maxOf(2, 1) == 2 PASSED
CalculatorTests > 2 => maxOf(1, 2) == 2 PASSED
CalculatorTests > 3 => maxOf(1, 1) == 1 PASSED
```

With the help of custom display names, you can easily and conveniently provide pretty and informative test outputs.

Sources of arguments

JUnit provides a number of annotations to define a source of arguments. Such arguments may be a sequence of test method arguments of the same type.

They accept a single test method argument or a sequence of arguments of the same or different types, which in turn accept multiple arguments.

@ValueSource
@ValueSource is an argument source that supplies an array of literal values for test methods with a single parameter. Such literal values may be of any of the following
types: short, byte, int, long, float, double, char, boolean, java.lang.String, and java.lang.Class.

Let us add another method to our Calculator class, which will accept a single int argument and return boolean:

```
public boolean isEven(int a) {
   return a % 2 == 0;
}
```

After that, we will use the following test method that will be invoked multiple times with different integer arguments supplied by @ValueSource:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

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

class CalculatorTests {

   @ParameterizedTest
   @ValueSource(ints = { 0, 2, 4, 1000 })
   void testIsEven(int arg) {
      assertTrue(new Calculator().isEven(arg));
   }
}
```

For non-primitive types, you can use @EmptySource, @NullSource or @NullAndEmptySource annotations to pass null and empty values.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EmptySource;
import org.junit.jupiter.params.provider.NullAndEmptySource;

import java.util.List;

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

class CalculatorTests {

   @ParameterizedTest
   @EmptySource
   void testEmpty(int[] arg) {
       assertEquals(0, arg.length);
   }

   @ParameterizedTest
   @NullAndEmptySource
   void testNullAndEmpty(List<String> arg) {
       assertTrue(arg == null || arg.isEmpty());
   }
}
```

The testNullAndEmpty(List<String> arg) method is invoked twice: the first time arg is null, and the second time arg is empty.

You can even combine these annotations with @ValueSource values to check the whole range of test cases in a single test method.

@MethodSource

This annotation allows you to use a method of your test class or an external class as a source of arguments. Each such method must satisfy the following requirements: it must be static, it must not accept any arguments, and must return a stream (see this topic for details), an array, or a collection of arguments.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.List;

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

class CalculatorTests {

    @ParameterizedTest
    @MethodSource("stringFactory")
    void testStrings(String str) {
        assertFalse(str.isEmpty());
    }

    static List<String> stringFactory() {
        return List.of("apple", "banana", "lemon", "orange");
    }
}
```

You may use non-static methods as long as they are internal methods of the test class annotated with @TestInstance(Lifecycle.PER_CLASS), but if you use methods of external classes as a source of arguments, they must always be declared as static

If a parameterized test method has multiple parameters, your argument source method needs to return a collection, a stream, or an array of Arguments or an array of Object. In this case, Arguments can be generated by
the arguments static method:

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.params.provider.Arguments.arguments;

class CalculatorTests {

    @ParameterizedTest
    @MethodSource("argFactory")
    void testStringLength(String str, int length) {
        assertEquals(length, str.length());
    }

    static List<Arguments> argFactory() {
        return List.of(arguments("apple", 5), arguments("watermelon", 10));
    }
}
```

Working with CSV
In the first example of a parameterized test, we used the annotation @CsvSource. It allows you to supply a list of arguments as comma-separated values (CSV format), for example:

```
@CsvSource({ "apple, 5", "strawberry, 10", "cherry, 6" })
```

In this case, each value is represented by a String literal containing a list of arguments separated by a comma which serves as the default delimiter. @CsvSource also has a number of attributes to define the format of the arguments.

You can change the default delimiter to another character or even a String literal, as well as define the representation of empty and null values, however, all these attributes are optional and can be used when needed.

In addition to @CsvSource, JUnit has the @CsvFileSource annotation which is used to load a CSV file from the classpath or the local file system.

Each line from a CSV file serves as a source of arguments for one invocation of the parameterized test.

You may skip the desired number of lines in the file by setting the numLinesToSkip attribute. Also, if you want any lines in the CSV file to be ignored, you can use the symbol # at the beginning of the respective lines to comment them out.

ere is an example of a CSV file:
String, Length
apple, 5
strawberry, 10
# commented line
cherry, 6
And an example of the @CsvFileSourceannotation:
@CsvFileSource(resources = "/dataset.csv", numLinesToSkip = 1)