1. Overview

In this article, we'll explore the *JUnitParams* library and its usages. Simply put, this library provides easy parameterization of test methods in *JUnit* tests.

There are situations where the only thing that changes between multiple tests are the parameters. *JUnit* itself has a parameterization support, and *JUnitParams* significantly improves on that functionality.

2. Maven Dependency

To use *JUnitParams* in our project, we need to add it to our *pom.xml*:

```
<dependency>
    <groupId>pl.pragmatists</groupId>
    <artifactId>JUnitParams</artifactId>
    <version>1.1.0</version>
</dependency>
```

The latest version of the library can be found here.

3. Test Scenario

Let's create a class which does the safe addition of two integers. This should return *Integer.MAX_VALUE* if it overflows, and *Integer.MIN_VALUE* if it underflows:

```
public class SafeAdditionUtil {

    public int safeAdd(int a, int b) {
        long result = ((long) a) + b;
        if (result > Integer.MAX_VALUE) {
            return Integer.MAX_VALUE;
        } else if (result < Integer.MIN_VALUE)
            return Integer.MIN_VALUE;
        }
        return (int) result;
    }
```

```
12  }
```

4. Constructing a Simple Test Method

We'll need to test method implementation for different combinations of input values, to make sure the implementation holds true for all possible scenarios. *JUnitParams* provides more than one way to achieve the parameterized test creation.

Let's take the basic approach with a minimal amount of coding and see how it is done. After that, we can see what the other possible ways of implementing the test scenarios using JUnitParams are:

```
1  @RunWith(JUnitParamsRunner.class)
2  public class SafeAdditionUtilTest {
3
4      private SafeAdditionUtil serviceUnderTest
5        = new SafeAdditionUtil();
6
7      @Test
8      @Parameters({
9        "1, 2, 3",
10       "-10, 30, 20",
11       "15, -5, 10",
12       "-5, -10, -15" })
13     public void whenWithAnnotationProvidedParam
14       int a, int b, int expectedValue) {
15
16         assertEquals(expectedValue, serviceUnde
17     }
18
19 }
```

Now let's see how this test class differs from a regular *JUnit* test class.

The first thing we notice is that **there is a different test runner** in the class annotation – *JUnitParamsRunner*.

Moving on to the test method, we see the test method is annotated with *@Parameters* annotation with an array of input parameters. It indicates different test scenarios which will be used for testing our service method.

If we run the test using Maven, we'll see that **we are running four test cases and not a single one**. The output would be similar to the following:

```
-------------------------------------------------
 T E S T S
-------------------------------------------------
Running com.baeldung.junitparams.SafeAdditionUt
Tests run: 4, Failures: 0, Errors: 0, Skipped:
  - in com.baeldung.junitparams.SafeAdditionUti

Results :

Tests run: 4, Failures: 0, Errors: 0, Skipped:
```

5. Different Types of Parameterization of Test Methods

Providing test parameters directly in the annotation is certainly not the most readable way if we have lots of possible scenarios that need to be tested. *JUnitParams* offers a set of different approaches we can utilize to create the parameterized tests:

- Directly in the *@Parameters* annotation (used in the example above)
- Using a named test method defined within the annotation
- Using a method mapped by test method name
- A named test class defined within the annotation
- Using a CSV file

Let's explore the approaches one by one.

5.1. Directly in the *@Parameters* Annotation

We have already used this approach in the example we tried. What we need to keep in mind is that we should be providing an array of parameter strings. Within the parameter string, each parameter is separated by a comma.

For example, the array would be in the form of *{ "1, 2, 3", "-10, 30, 20"}* and one set of parameters is represented as *"1, 2, 3"*.

The limitation of this approach is that we can only supply primitives and *String*s as test parameters. It is not possible to submit objects as test method parameters as well.

5.2. Parameter Method

We can provide the test method parameters using another method within the class. Let's see an example first:

```
@Test
@Parameters(method = "parametersToTestAdd")
public void whenWithNamedMethod_thenSafeAdd(
  int a, int b, int expectedValue) {

    assertEquals(expectedValue, serviceUnderTes
}

private Object[] parametersToTestAdd() {
    return new Object[] {
        new Object[] { 1, 2, 3 },
        new Object[] { -10, 30, 20 },
        new Object[] { Integer.MAX_VALUE, 2, In
        new Object[] { Integer.MIN_VALUE, -8, I
    };
}
```

The test method is annotated concerning the method *parametersToAdd(),* and it fetches the parameters by running the referenced method.

The specification of the provider method should return an array of *Object*s as a result. If a method with the given name is not available, the test case fails with the error:

```
java.lang.RuntimeException: Could not find metho
```

5.3. Method Mapped by Test Method Name

If we do not specify anything in the *@Parameters*annotation, *JUnitParams* tries to load a test data provider method based on the test method name. The method name is constructed as *"parametersFor"*+ *<test method name>:*

```
@Test
@Parameters
public void whenWithnoParam_thenLoadByNameSafeA
  int a, int b, int expectedValue) {

    assertEquals(expectedValue, serviceUnderTes
}

private Object[] parametersForWhenWithnoParam_t
    return new Object[] {
        new Object[] { 1, 2, 3 },
        new Object[] { -10, 30, 20 },
        new Object[] { Integer.MAX_VALUE, 2, In
        new Object[] { Integer.MIN_VALUE, -8, I
```

```
    };
}
```

In the above example the name of the test method is *whenWithnoParam_shouldLoadByNameAbdSafeAdd()*.

Therefore when the test method is being executed, it looks for a data provider method with the name *parametersForWhenWithnoParam_shouldLoadByNameAbdSafeAdd()*.

Since that method exists, it will load data from it and run the test. **If there is no such method matching the required name, the test fails** as in the above example.

5.4. Named Test Class Defined Within the Annotation

Similar to the way we referred to a data provider method in a previous example, we can refer to a separate class to provide the data for our test:

```
@Test
@Parameters(source = TestDataProvider.class)
public void whenWithNamedClass_thenSafeAdd(
  int a, int b, int expectedValue) {

    assertEquals(expectedValue, serviceUnderTes
}
public class TestDataProvider {

    public static Object[] provideBasicData() {
        return new Object[] {
            new Object[] { 1, 2, 3 },
            new Object[] { -10, 30, 20 },
            new Object[] { 15, -5, 10 },
            new Object[] { -5, -10, -15 }
        };
    }

    public static Object[] provideEdgeCaseData(
        return new Object[] {
            new Object[] {
              Integer.MAX_VALUE, 2, Integer.MAX
            new Object[] {
              Integer.MIN_VALUE, -2, Integer.MI
        };
    }
}
```

We can have any number of test data providers in a class given that the method name starts with “provide”. If so, the executor picks those methods and returns the data.

If no class methods are satisfying that requirement, even though those methods return an array of *Object*s, those methods will be ignored.

5.5. Using a CSV File

We can use an external CSV file to load the test data. This helps if the number of possible test cases is quite significant, or if test cases are frequently changed. The changes can be done without affecting the test code.

Let’s say that we have a CSV file with test parameters as *JunitParamsTestParameters.csv*:

```
1,2,3
-10, 30, 20
15, -5, 10
-5, -10, -15
```

Now let’s look how **this file can be used to load test parameters** in the test method:

```
@Test
@FileParameters("src/test/resources/JunitParamsT
public void whenWithCsvFile_thenSafeAdd(
  int a, int b, int expectedValue) {

    assertEquals(expectedValue, serviceUnderTest
}
```

One limitation of this approach is that it is not possible to pass complex objects. Only primitives and *String*s are valid.