ABOUT RAFAL BOROWIEC

Software developer, Team Leader, Agile practitioner, occasional blogger, lecturer. Open Source enthusiast, quality oriented and open-minded.

Parameterized JUnit tests with JUnitParams

Posted by: Rafal Borowiec in Core Java December 22nd, 2013

Parameterized unit tests are used to to test the same code under different conditions. Thanks to parameterized unit tests we can set up a test method that retrieves data from some data source. This data source can be a collection of test data objects, external file or maybe even a database. The general idea is to make it easy to test different conditions with the same unit test method, which will limit the source code we need to write and makes the test code more robust. We can call these tests data-driven unit tests.

The best way to achieve data-driven unit tests in JUnit is to use a JUnit's custom runner –

```
Parameterized
```

or JUnitParams'

```
JUnitParamsRunner
```

. Using JUnit's approach may work in many cases, but the latter seems to be more easy to use and more powerfull.

Basic example

In our example, a poker dice, we need to calculate the score of a full house. As in card poker, a Full House is a roll where you have both a 3 of a kind, and a pair. For the sake of simplicity, the score is a sum of all dice in a roll. So let's see the code:

```
class FullHouse implements Scoreable {

    @Override
    public Score getScore(Collection dice) {
        Score pairScore = Scorables.pair().getScore(dice);
        Score threeOfAKindScore = Scorables.threeOfAKind().getScore(pairScore.getReminder());
        if (bothAreGreaterThanZero(pairScore.getValue(), threeOfAKindScore.getValue())) {
            return new Score(pairScore.getValue() + threeOfAKindScore.getValue()); // no reminder
        }
        return new Score(0, dice);
    }

    private boolean bothAreGreaterThanZero(int value1, int value2) {
        return value1 > 0 && value2 > 0;
    }
}
```

I would like to be sure that the roll is properly scored (of course I have unit tests for both Pair and ThreeOfAKind already). So I would like to test the following conditions:

- Score is 11 for: 1, 1, 3, 3, 3
- Score is 8 for: 2, 2, 2, 1, 1
- Score is 0 for: 2, 3, 4, 1, 1

```
@RunWith(Parameterized.class)
public class FullHouseTest {

    private Collection rolled;
    private int score;

    public FullHouseTest(Collection rolled, int score) {
        this.rolled = rolled;
        this.score = score;
    }

    @Test
    public void fullHouse() {
        assertThat(new FullHouse().getScore(rolled).getValue()).isEqualTo(score);
    }

    @Parameterized.Parameters
    public static Iterable data() {
        return Arrays.asList(
                new Object[][]{
                        {roll(1, 1, 3, 3, 3), score(11)},
                        {roll(2, 2, 2, 1, 1), score(8)},
                        {roll(2, 3, 4, 1, 1), score(0)},
                        {roll(5, 5, 5, 5, 5), score(25)}
                }
        );
    }

    private static int score(int score) {
        return score;
    }
}
```

And the other one, with **JUnitParams**:

```
@RunWith(JUnitParamsRunner.class)
public class FullHouseTest {

    @Test
    @Parameters
    public void fullHouse(Collection rolled, int score) {
        assertThat(new FullHouse().getScore(rolled).getValue()).isEqualTo(score);
    }

    public Object[] parametersForFullHouse() {
        return $(
                $(roll(1, 1, 3, 3, 3), score(11)),
                $(roll(2, 2, 2, 1, 1), score(8)),
                $(roll(2, 3, 4, 1, 1), score(0)),
                $(roll(5, 5, 5, 5, 5), score(25))
        );
    }

    private static int score(int score) {
        return score;
    }
}
```

At first glance, both look very similar. And that's true. So what are the differences between JUnit

```
Parameterized
```

(1) and JUnitParams (2)? The most important one is the way of passing the parameters, so in fact the architecture of the solution. In (1) parameters are passed in constructor whereas in (2) parameters are passed directly to test method. Should I care? Yes. One of the reason is, in (2), we can have multiple parameterized tests methods with different data for each of the method, like in the below example:

```
@RunWith(JUnitParamsRunner.class)
public class NumberOfAKindTest {

    @Test
    @Parameters
    public void pair(Collection rolled, int[] expected, int score) {
        NumberOfAKind sut = new NumberOfAKind(2);
        doTest(rolled, expected, score, sut);
    }

    @Test
    @Parameters
    public void threeOfAKind(Collection rolled, int[] expected, int score) {
        NumberOfAKind sut = new NumberOfAKind(3);
        doTest(rolled, expected, score, sut);
    }

    public Object[] parametersForPair() {
        return $(
                $(roll(1, 1, 1, 2, 3), hand(1, 1), score(2)),
                $(roll(2, 1, 1, 1, 1), hand(1, 1), score(2)),
                $(roll(2, 3, 4, 1, 1), hand(1, 1), score(2)),
                $(roll(2, 3, 5, 5, 5), hand(5, 5), score(10)),
                $(roll(2, 1, 5, 4, 3), null, score(0))
        );
    }

    public Object[] parametersForThreeOfAKind() {
```

179,260 insiders are already er
weekly updates and complimentary
whitepapers!

Join them now to gain exc
access to the latest news in the J
as well as insights about Android, Sc
Groovy and other related technologi

Email address:

Your email address

☑ Receive Java & Developer job al
your Area from our partners over at
ZipRecruiter

Sign up

JOIN US

With **1,240,6**
unique visitors
500 authors
placed among
related sites a
Constantly bei
lookout for pa
encourage you
So If you have
unique and interesting content then yo
check out our **JCG** partners program. Y
be a **guest writer** for Java Code Geek
your writing skills!

```
                $(roll(2, 3, 5, 5, 5), hand(5, 5, 5), score(15)),
                $(roll(2, 5, 5, 5, 6), hand(5, 5, 5), score(15)),
                $(roll(2, 2, 5, 5, 3), null, score(0))
        );
    }

    private static int[] hand(int... dice) {
        return dice;
    }

    private static int score(int score) {
        return score;
    }

}
```

In simpler examples, parameters can be defined as a String array directly in @Parameters annotation via value method. We can also extract the data to an external class and have our tests more clean and readable. The complete test for

```
NumberOfAKind
```

reads as following:

```
@RunWith(JUnitParamsRunner.class)
public class NumberOfAKindTest {

    @Test
    @Parameters(source = NumberOfAKindProvider.class, method = "providePair")
    public void pair(Collection rolled, int[] expected, int score) {
        NumberOfAKind sut = new NumberOfAKind(2);
        doTest(rolled, expected, score, sut);
    }

    @Test
    @Parameters(source = NumberOfAKindProvider.class, method = "provideThreeOfAKind")
    public void threeOfAKind(Collection rolled, int[] expected, int score) {
        NumberOfAKind sut = new NumberOfAKind(3);
        doTest(rolled, expected, score, sut);
    }

    @Test
    @Parameters(source = NumberOfAKindProvider.class, method = "provideFourOfAKind")
    public void fourOfAKind(Collection rolled, int[] expected, int score) {
        NumberOfAKind sut = new NumberOfAKind(4);
        doTest(rolled, expected, score, sut);
    }

    @Test
    @Parameters(source = NumberOfAKindProvider.class, method = "provideFiveOfAKind")
    public void fiveOfAKind(Collection rolled, int[] expected, int score) {
        NumberOfAKind sut = new NumberOfAKind(5);
        doTest(rolled, expected, score, sut);
    }

    private void doTest(Collection rolled, int[] expected, int score, NumberOfAKind sut) {
        Collection consecutiveDice = sut.getConsecutiveDice(rolled);

        assertDiceContainsValues(consecutiveDice, expected);
        assertThat(sut.getScore(rolled).getValue()).isEqualTo(score);
    }

    private void assertDiceContainsValues(Collection dice, int[] expected) {
        Collection values = toInts(dice);
        if (expected == null) {
            assertThat(values).isEmpty();
            return;
        }
        for (int i = 0; i < expected.length; i++) {
            assertThat(values).hasSize(expected.length).contains(expected[i]);
        }
    }

    private Collection toInts(Collection dice) {
        return Collections2.transform(dice, new Function() {
            @Override
            public Integer apply(Dice input) {
                return input.getValue();
            }
        });
    }

}
```

Each method specifies a provider class and the provider's method name. Look at the provider below:

```
public class NumberOfAKindProvider {

    public static Object[] providePair() {
        return $(
                $(roll(1, 1, 1, 2, 3), hand(1, 1), score(2)),
                $(roll(2, 1, 1, 1, 1), hand(1, 1), score(2)),
                $(roll(2, 3, 4, 1, 1), hand(1, 1), score(2)),
                $(roll(2, 3, 5, 5, 5), hand(5, 5), score(10)),
                $(roll(2, 1, 5, 4, 3), null, score(0))
        );
    }

```

```
                $(roll(2, 3, 1, 1, 1), hand(1, 1, 1), score(3)),
                $(roll(2, 3, 5, 5, 5), hand(5, 5, 5), score(15)),
                $(roll(2, 5, 5, 5, 6), hand(5, 5, 5), score(15)),
                $(roll(2, 2, 5, 5, 3), null, score(0))
        );
    }

    public static Object[] provideFourOfAKind() {
        return $(
                $(roll(1, 1, 1, 1, 3), hand(1, 1, 1, 1), score(4)),
                $(roll(2, 1, 1, 1, 1), hand(1, 1, 1, 1), score(4)),
                $(roll(2, 5, 5, 5, 5), hand(5, 5, 5, 5), score(20)),
                $(roll(2, 3, 4, 5, 5), null, score(0))
        );
    }

    public static Object[] provideFiveOfAKind() {
        return $(
                $(roll(1, 1, 1, 1, 1), hand(1, 1, 1, 1, 1), score(5)),
                $(roll(6, 6, 6, 6, 6), hand(6, 6, 6, 6, 6), score(30)),
                $(roll(6, 6, 6, 6), null, score(0)),
                $(roll(2, 3, 4, 6, 6), null, score(0))
        );
    }

    private static int[] hand(int... dice) {
        return dice;
    }

    private static int score(int score) {
        return score;
    }
}
```

Summary

To me

```
JUnitParams
```

is a better solution for writing good data-driven unit tests. But what is presented above, is not everything the library has to offer to a developer. There are more features. Params can be passed as a CSV string, we can mix both parameterized and non-parameterized tests just to mention a few.

Please visit the project's website to find out more about this library: https://code.google.com/p/junitparams

Reference: Parameterized JUnit tests with JUnitParams from our JCG partner Rafal Borowiec at the Codeleak.pl blog.

Tagged with: JUNIT TESTING

2 COMMENTS

Denilson
December 26th, 2013 at 6:15 pm

Have you tried DDTUnit? At first the separate XML for the data may be intimidating, byt the Java code 3is much simpler.

Reply

piotrek
November 22nd, 2014 at 3:22 pm

try zohhak.googlecode.com. It lets you write parameters in your java code in a really clean manner, without any providers or external xml files.

Reply

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

[✓] Sign me up for the newsletter!

Receive Email Notifications?
no, do not subscribe
instantly
Or, you can subscribe without commenting.

Post Comment

- .NET Code Geeks
- Java Code Geeks
- System Code Geeks
- Web Code Geeks

- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet