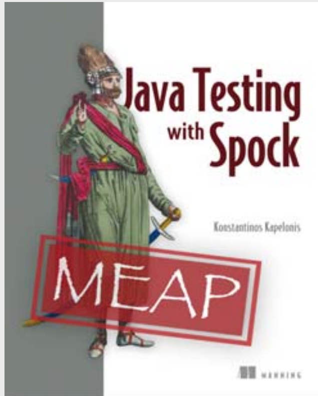

A first look at Spock in action

By Konstantinos Kapelonis

You know the theory behind a solid testing process and how Spock can test classes written in Java. Now it's time to delve into code! In this article, I'll show you what spock looks like in a minimal, but fully functioning, example.

This article is an excerpt from [Java Testing with Spock](http://www.manning.com/kapelonis) by Konstantinos Kapelonis. Save 39% on Java Testing with Spock with code 15dzamia at manning.com.

When introducing a new library/language/framework everybody expects a simple example for illustration purposes. In this article, I will show you how a Spock unit test works like in a minimal, but fully functional, example.

Here is the Java class we will test. For comparison purposes, a possible JUnit test is first shown, as JUnit is the defacto testing framework for Java.

Listing 1 Java class under test and JUnit test

```
public class Adder { #A
       public int add(int a, int b)
       {
         return a+b;
       }
}
public class AdderTest {     #B
       @Test
       public void simpleTest()
       {
         Adder adder = new Adder(); #C
         assertEquals("1 + 1 is 2",2,adder.add(1, 1)); #D
       }
       @Test
       public void orderTest() #E
       {
         Adder adder = new Adder();
         assertEquals("Order does not matter ",5,adder.add(2, 3)); #F
         assertEquals("Order does not matter ",5,adder.add(3, 2));
       }
}
```

#A A trivial class that will be tested (a.k.a. class under test)
#B Test case for the class in question
#C Initialization of class under test
#D JUnit assert statement that compares 2 and the result of add(1,1)
#E A second scenario for the test under class
#F Two assert statements that compare 5 with adding 2 and 3

We introduce two test methods, one that tests the core functionality of our Adder class, and one that tests the order of arguments in our "add" method.

Running this JUnit test in the Eclipse Development Environment (right click on the .java file and choose run as-> Junit test from the menu) gives the following:

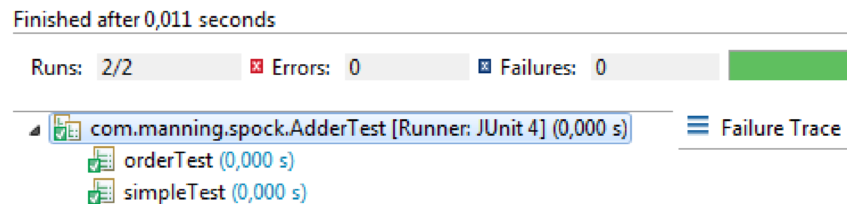

Figure 1 Running a JUnit test in Eclipse

A very simple test with Spock

And here is the same test in Groovy/Spock. Again this test examines the correctness of the Java class Adder that creates the sum of two numbers.

Listing 2 Spock test for the Adder Java class

```
class AdderSpec extends spock.lang.Specification{ #A
      def "Adding two numbers to return the sum"() { #B
        when: "a new Adder class is created" #C
        def adder = new Adder(); #D

        then: "1 plus 1 is 2" #E
        adder.add(1, 1) == 2 #F
      }
      def "Order of numbers does not matter"() { #G
        when: "a new Adder class is created"
        def adder = new Adder();

        then: "2 plus 3 is 5"
        adder.add(2, 3) == 5 #H

        and: "3 plus 2 is also 5" #I
        adder.add(3, 2) == 5
      }
}
```

#A All Spock tests extends the Specification class
#B A Groovy method with a human readable name that contains a test scenario

For source code, sample chapters, the Online Author Forum, and other resources, go to
http://www.manning.com/kapelonis

#C A "when" block that sets the scene
#D Initialization of Java class under test
#E A "then" block that will hold verification code
#F A Groovy assert statement
#G Another test scenario
#I A "and" block that accompanies the "then" block

If you have never seen Groovy code before, this Spock segment will indeed appear very strange to you. The code has mixed lines of things you know (like the first line with the *extends* keyword) and things completely alien to you like the *def* keyword. Details on Groovy syntax will be shown in Chapter 2.

On the other hand if you've already seen the concepts of Behavior-driven design (BDD) you'll already be familiar with the "when/then" pattern of feature testing.

We will explain the Spock syntax in detail in the coming chapters. For example the "def" keyword (comes from define) is how you declare things in Groovy without specifying explicitly their type (which is a strict requirement in Java). The Spock blocks (when, then, and) will be covered in chapter 4.

TAKEAWAYS FROM THESE CODE EXAMPLES

What you need take away from this code sample is:

- The almost English-like flow of the code. It is very easy to see what is being tested even if you are a business analyst or don't know Groovy.
- The lack of any assert statements. Spock has a declarative syntax, where you explain what you consider correct behavior.

So how do you run this test? You run it in the same way as a JUnit test! Again you right click on the Groovy class and select Run as -> Junit test from the popup menu.The result in Eclipse looks like this:

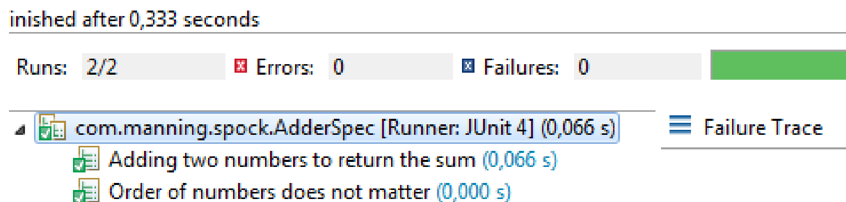

Figure 2 Running a Spock test in Eclipse

Other than the most descriptive method names, there isn't any big difference between JUnit and Spock results with this trivial example.

Some of the power of Spock will begin to appear in our next example.

Inspecting failed tests with Spock

One of the big highlights of Spock code is the lack of assert statements compared to JUnit. In the previous section you saw what happens when all tests pass and the happy green bar is shown in Eclipse. But how does Spock cope with test failures?

In order to demonstrate the advantage over JUnit we will add another (trivial) Java class that we wish to test

```
public class Multiplier {
     public int multiply(int a, int b)
     {
             return a * b;
     }
}
```

For this class we will also write the respective JUnit test. But as an additional twist (for demonstration purposes) we wish to test this class not only by itself, but also in relation with the Adder class shown in the previous section.

Listing 3 A JUnit test for two Java classes

```
public class MultiplierTest {
       @Test
       public void simpleMultiplicationTest()
       {
         Multiplier multi = new Multiplier();
         assertEquals("3 times 7 is 21",21,multi.multiply(3, 7));
       }
       @Test
       public void combinedOperationsTest() #A
       {
         Adder adder = new Adder(); #B
         Multiplier multi = new Multiplier(); #C

         assertEquals("4 times (2 plus 3) is 20", #D
                 20,multi.multiply(4, adder.add(2, 3)));
         assertEquals("(2 plus 3) times 4 is also 20",
         20,multi.multiply(adder.add(2, 3),4));
       }
}
```

#A A test scenario that will examine two Java classes at the same time
#B Creation of the first Java class
#C Creation of the second Java class
#D Verification of a mathematical result coming from both Java classes

Running this unit test will result in a green bar since both tests pass.

And now for the equivalent Spock test:

Listing 4 Spock test for two Java classes

```
class MultiplierSpec extends spock.lang.Specification{
      def "Multiply two numbers and return the result"() {
        when: "a new Multiplier class is created"
        def multi = new Multiplier();

        then: "3 times 7 is 21"
        multi.multiply(3, 7) == 21
      }
      def "Combine both multiplication and addition"() { #A
        when: "a new Multiplier and Adder classes are created"
        def adder = new Adder(); #B
        def multi = new Multiplier() #C

        then: "4 times (2 plus 3) is 20" #D
        multi.multiply(4, adder.add(2, 3)) == 20

        and: "(2 plus 3) times 4 is also 20"
        multi.multiply(adder.add(2, 3),4) == 20
      }
}
```

#A A test scenario that will examine two Java classes at the same time
#B Creation of the first Java class
#C Creation of the second Java class
#D Verification of a mathematical result coming from both Java classes

Again running this test will pass with flying colors. You might start to believe that we gain nothing from using Spock instead of JUnit. But wait!

Let's introduce an artificial bug in our code to see how both JUnit and Spock deal with failure. To mimic a real world bug, we'll introduce it in the Multiplier class, but only for a special case.

Listing 5 Introducing an artificial bug in the Java class under test

```
public class Multiplier {
      public int multiply(int a, int b)
      {
        if(a == 4)  #A
        {
                return 5 * b; //multiply an extra time.
        }
        return a *b;
      }
}
```

#A A dummy bug that happens only if the first argument is 4

Now let's run the all JUnit test and see what happens.

```
Failure Trace
java.lang.AssertionError: 4 times (2 plus 3) is 20 expected:<20> but was:<25>
  at com.manning.spock.MultiplierTest.combinedOperationsTest(MultiplierTest.java:22)
```

Figure 3 Failure of JUnit test in Eclipse

We have a test failure. Notice however anything strange here? Because the bug we introduced is very subtle, JUnit essentially says to us:

- Addition by itself works fine.
- Multiplication by itself works fine.
- When both of them run together we have problem.

But where is the problem? Is the bug on the addition code or the multiplication one? We cannot say just by looking at the test result (ok, ok the math here might give you a hint in this trivial example).

We need to insert a debugger in the unit test to find out what happened. This is an extra step that takes a lot of time because recreating the same context environment can be a lengthy process.

SPOCK KNOWS ALL THE DETAILS WHEN A TEST FAILS

Spock comes to the rescue! If we run the same bug against Spock we get the following:

```
Failure Trace
Condition not satisfied:

multi.multiply(4, adder.add(2, 3)) == 20
|     |           |     |          |
|     25          |     5          false
|                 com.manning.spock.Adder@691a0e79
com.manning.spock.Multiplier@38d9e447
```

Figure 4 Failure of a Spock test in Eclipse

Spock comes with a super-charged error message that not only says we have a failure, but also calculates intermediate results!

As you see it is clear by the test that the addition works correctly (2 + 3 is indeed 5) and the bug is on the multiplication code (4 times 5 does not equal 25)

Armed with this knowledge we can go directly to the Multiplier code and find the bug. This is one of the killer features of Spock, and may be enough to entice you to rewrite all your JUnit tests in Spock. But a complete rewrite is not necessary, as both Spock and JUnit tests can co-exist in the same codebase, which we'll explore next.