ScalaCheck: The Definitive Guide

PrePrint™ Edition

Excerpt

artima

ARTIMA PRESS

MOUNTAIN VIEW, CALIFORNIA

Chapter 2

ScalaCheck vs. JUnit: A Complete Example

Now that you have a theoretical introduction to ScalaCheck concepts, let's explore a practical example. This chapter presents a small Java library that we'll test with JUnit and ScalaCheck. Although I won't explain everything in detail, you should get a rough understanding of the tasks involved when using ScalaCheck. By direct comparisons to JUnit, you will develop an understanding of the differences and similarities.

Don't worry if you get a little confused over the ScalaCheck syntax in this chapter, since I won't be going into much detail. Just try to visualize an overall picture of how ScalaCheck compares to traditional unit testing. The next chapter describes more closely how the different parts of ScalaCheck work together and what possibilities you have when you're designing your properties.

The class under test

The code we will unit test is a small library of string handling routines, written in Java. The complete source code is given in Listing 2.1.

Using JUnit

We will start off by writing and running JUnit tests for the library. I'll be using JUnit 4 in my examples.

```java
import java.util.StringTokenizer;

public class StringUtils {

  public static String truncate(String s, int n) {
    if(s.length() <= n) return s;
    else return s.substring(0, n) + "...";
  }

  public static String[] tokenize(
    String s, char delim
  ) {
    String delimStr = new Character(delim).toString();
    StringTokenizer st = new StringTokenizer(
      s, delimStr);
    String[] tokens = new String[st.countTokens()];
    int i = 0;
    while(st.hasMoreTokens()) {
      tokens[i] = st.nextToken();
      i++;
    }
    return tokens;
  }

  public static boolean contains(
    String s, String subString
  ) {
    return s.indexOf(subString) != -1;
  }

}
```

Listing 2.1 · `StringUtils.java`: the class under test.

We define a class that contains all the unit tests for our library. Look at the implementation below:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;

@RunWith(JUnit4.class)
public class StringUtilsTest {
  @Test public void testTruncateShortString() {
    String s = StringUtils.truncate("abc", 5);
    assertEquals("abc", s);
  }
  @Test public void testTruncateLongString() {
    String s = StringUtils.truncate("Hello World", 8);
    assertEquals("Hello Wo...", s);
  }
  @Test public void testTokenize() {
    String[] tokens = StringUtils.tokenize(
      "foo;bar;42", ';');
    String[] expected = { "foo", "bar", "42" };
    assertTrue(java.util.Arrays.equals(tokens, expected));
  }
  @Test public void testTokenizeSingle() {
    String[] tokens = StringUtils.tokenize(
      "Hello World", ',');
    String[] expected = { "Hello World" };
    assertTrue(java.util.Arrays.equals(tokens, expected));
  }
  @Test public void testContainsTrue() {
    assertTrue(StringUtils.contains("abc", "bc"));
  }
  @Test public void testContainsFalse() {
    assertFalse(StringUtils.contains("abc", "42"));
  }
}
```

As you can see, I've tried to include different kinds of test cases for each unit test. Let's now see whether the library passes the tests. We compile the library and its tests, and then use the console test runner in JUnit to run the tests.

```
$ javac -cp junit-4.11.jar \
  StringUtils.java StringUtilsTest.java
$ java -cp .:junit-4.11.jar:hamcrest-core-1.3.jar \
  org.junit.runner.JUnitCore StringUtilsTest
JUnit version 4.11
......
Time: 0.006

OK (6 tests)
```

Great! All six tests passed, which shows that our library behaved correctly. Now let's turn to ScalaCheck and look at how to define equivalent properties in it.

Using ScalaCheck

In ScalaCheck, you define *properties* instead of tests. To define a set of properties for our library under test, we extend the `org.scalacheck.Properties` class, which could be seen as corresponding to the `TestCase` class in JUnit. Consider the property definitions for our small string utility library in Listing 2.2.

The `Prop.forAll` method is a common way of creating properties in ScalaCheck. There are also other ways, which we'll describe in more detail in later chapters. The `forAll` method takes an anonymous function as its parameter, and that function in turn takes parameters that are used to express a boolean condition. Basically, the `forAll` method is equivalent to what in logic is called a *universal quantifier*. When ScalaCheck tests a property created with the `forAll` method, it tries to *falsify* it by assigning different values to the parameters of the provided function, and evaluating the boolean result. If it can't locate a set of arguments that makes the property false, then ScalaCheck will regard the property as *passed*. This testing process is described in detail in Chapter 4.

```
import org.scalacheck.Properties
import org.scalacheck.Prop
import org.scalacheck.Gen.{listOf, alphaStr, numChar}

object StringUtilsProps extends
  Properties("StringUtils")
{
  property("truncate") =
    Prop.forAll { (s: String, n: Int) =>
      val t = StringUtils.truncate(s, n)
      (s.length <= n && t == s) ||
      (s.length > n && t == s.take(n)+"...")
    }

  property("tokenize") =
    Prop.forAll(listOf(alphaStr), numChar) {
      (ts, d) =>
        val str = ts.mkString(d.toString)
        StringUtils.tokenize(str, d).toList == ts
    }

  property("contains") = Prop.forAll {
    (s1: String, s2: String, s3: String) =>
      StringUtils.contains(s1+s2+s3, s2)
  }
}
```

Listing 2.2 · ScalaCheck properties for `StringUtils`.

As you can see, the types of parameters vary. In the `truncate` property, we declare one string parameter `s` and one integer parameter `n`. That means that the property should hold for all possible pairs of strings and integers.

The second property, describing `tokenize`, differs a bit from what you have seen before. Instead of specifying the types of parameters, we tell ScalaCheck explicitly which data generators to use. In this case, we use `Gen.listOf` in combination with `Gen.alphaStr` to generate lists of alpha-only strings, and `Gen.numChar` to generate digit characters. We still define the property as a function literal, but now we don't need to specify the types of its parameters since they are given by the explicit generators.

Which types are available for use in a `forAll` property? ScalaCheck has built-in support for common Java and Scala types, so you can use ordinary types like integers, strings, dates, lists, arrays, and so on. However, you can also add support for any custom data type, by letting ScalaCheck know how to generate your type. I'll describe how this is done in Chapter 3.

Just as in JUnit, there's a console-based test runner in ScalaCheck:

```
$ javac StringUtils.java
$ scalac -cp .:scalacheck.jar StringUtilProps.scala
$ scala -cp .:scalacheck.jar StringUtilProps
! StringUtils.truncate: Exception raised on property
evaluation.
> ARG_0: ""
> ARG_1: -1
> ARG_1_ORIGINAL: -1110151355
> Exception: java.lang.StringIndexOutOfBoundsException:
String index out of range: -1
java.lang.String.substring(String.java:1911)
StringUtils.truncate(StringUtils.java:7)
StringUtilsProps$$anonfun$1.apply(StringUtilsProps.scala:9)
StringUtilsProps$$anonfun$1.apply(StringUtilsProps.scala:8)
org.scalacheck.Prop$$anonfun$forAll$10$$anonfun$apply$25
.apply(Prop.scala:759)
! StringUtils.tokenize: Falsified after 5 passed tests.
> ARG_0: List("")
> ARG_0_ORIGINAL: List("", "yHa", "vlez", "Oyex", "lhz")
> ARG_1: 2
+ StringUtils.contains: OK, passed 100 tests.
```

What happened here? It certainly doesn't look as if the test passed, does it? Let's try to break things up a bit first. ScalaCheck tested three properties: `StringUtils.truncate`, `StringUtils.tokenize`, and `StringUtils.contains`. For each property, ScalaCheck prints the test results, starting with an exclamation mark for failed properties and a plus sign for properties that passed the tests. Hence, we can conclude that the first two properties failed, and the third one succeeded. Let's investigate the failures in ScalaCheck more closely.

For the `StringUtils.truncate` property, we encountered a `StringIndexOutOfBoundsExc` during testing. The arguments that caused the exception were an empty string and the integer value `-1`. These arguments correspond to the parameters `s` and `n` in the `truncate` property definition in `StringUtilsProps.scala`. If we look at the library code, the failure is not hard to understand. The given arguments will lead to an invocation of `"".substring(0, -1)`, and the API documentation for the `String` class clearly states that such indices will cause an exception to be thrown.

There are several ways to make the `truncate` property pass, and we must now decide exactly how we want the `truncate` method to behave. Here is a list of alternatives:

1. Let the `truncate` method throw an exception for invalid input, and clearly specify the kind of exception it will throw. Either we can leave the method as it is, throwing the same exception as `String.substring` does, or we can throw another type of exception. In any case, we'll have to do something about the property, since we want it to verify that the correct exception is thrown.

2. Let the `truncate` method be completely unspecified for invalid inputs. We simply state a precondition for the method and if the caller breaks that condition, there's no guarantee for how `truncate` will behave. This can be a reasonable approach in some situations, but we still need to make our property respect the precondition.

3. Handle invalid inputs in another reasonable way. For example, if a negative integer is used in a call to `truncate`, then it could make sense to return an empty string. This approach requires us to change both the implementation (the `truncate` method) and the specification (the property).

Notice how ScalaCheck forced us to think about the general behavior of `truncate` and not just about a few concrete test cases. If you are experienced in writing unit tests, you might spot the exception case above and write tests covering it. However, ScalaCheck seemed to spot it for free.

Now, for each possible alternative in the list above, let's see how we would change code.

1. **Throw the exception** We let the implementation remain the same, and update the property to respect the fact that an exception should be thrown for invalid input:

```
property("truncate") =
  Prop.forAll { (s: String, n: Int) =>
    lazy val t = StringUtils.truncate(s, n)
    if (n < 0)
      Prop.throws(
        classOf[StringIndexOutOfBoundsException]
      ) { t }
    else
      (s.length <= n && t == s) ||
      (s.length > n && t == s.take(n)+"...")
  }
```

   The new version of the property uses a handy feature of the Scala language called *lazy evaluation*. By marking the variable `t` with the keyword `lazy`, the expression to the right of the assignment operator is not evaluated until the value of `t` is used. Therefore, the exception is not thrown during assignment. We then use ScalaCheck's `Props.throws` operator, which makes sure that the property passes only if the correct type of exception is thrown. The `classOf` operator is built into Scala and used for retrieving the `java.lang.Class` instance for a particular type.

2. **Remain unspecified** The precondition for the `truncate` method is simply that the integer parameter must be greater than or equal to zero. We state this in the property by using ScalaCheck's *implication* operator, `==>`. To get access to this operator, we need to import `Prop.BooleanOperators` that makes some boolean property operators implicitly available in the importing scope. By specifying a precondition in this way, we keep ScalaCheck from testing the property with input values that don't fulfill the condition.

```
import Prop.BooleanOperators

property("truncate") =
```

```
Prop.forAll { (s: String, n: Int) =>
  (n >= 0) ==> {
    val t = StringUtils.truncate(s, n)
    (s.length <= n && t == s) ||
    (s.length > n && t == s.take(n)+"...")
  }
}
```

Preconditions in ScalaCheck properties are discussed in Chapter **??**.

3. **Handle it** In the third alternative, we wanted our method to return an empty string when confronted with invalid inputs. This is a simple change in the implementation:

```
public static String truncate(String s, int n) {
  if(n < 0) return "";
  else if(s.length() <= n) return s;
  else return s.substring(0, n) + "...";
}
```

The property is updated to cover the empty string case:

```
property("truncate") =
  Prop.forAll { (s: String, n: Int) =>
    val t = StringUtils.truncate(s, n)
    if(n < 0) t == ""
    else
      (s.length <= n && t == s) ||
      (s.length > n && t == s.take(n)+"...")
  }
```

Each solution above makes the `truncate` property pass; it's up to the implementer to decide exactly how the method should behave. If we run the tests again, after having picked one of the alternatives, we get the following output:

```
$ scala -cp .:scalacheck.jar StringUtilProps
 + StringUtils.truncate: OK, passed 100 tests.
 ! StringUtils.tokenize: Falsified after 3 passed tests.
 > ARG_0: List("")
 > ARG_0_ORIGINAL: List("", "")
 > ARG_1: 9
 + StringUtils.contains: OK, passed 100 tests.
```

Now only the `tokenize` property fails. We can see that the property was given a single string `""` (an empty string) and the delimiter token 2. However, to debug the property and implementation, it would be nice to see more information about the property evaluation. For example, it would be beneficial if we could somehow see the value produced by `tokenize` when given the generated input. In fact, there are several ways to collect data from the property evaluation, which I'll describe in Chapter 5. In this specific case, the simplest solution is to use a special equality operator of ScalaCheck instead of the ordinary one. We import `Prop.AnyOperators` that makes a number of property operators implicitly available, and then simply change == to ?= in the property definition:

```
property("tokenize") = {
  import Prop.AnyOperators
  Prop.forAll(listOf(alphaStr), numChar) { (ts, d) =>
    val str = ts.mkString(d.toString)
    StringUtils.tokenize(str, d).toList ?= ts
  }
}
```

Let's see what ScalaCheck tells us now:

```
$ scala -cp .:scalacheck.jar StringUtilProps
 + StringUtils.truncate: OK, passed 100 tests.
 ! StringUtils.tokenize: Falsified after 3 passed tests.
 > Labels of failing property:
 Expected List("") but got List()
 > ARG_0: List("")
 > ARG_0_ORIGINAL: List("", "E", "zd")
 > ARG_1: 4
 + StringUtils.contains: OK, passed 100 tests.
```

Because ScalaCheck generates random input, the exact results of each run are not the same. Don't worry if the output you see is different.

ScalaCheck now reports a *label* for the failing property. Here, we can see exactly what went wrong in the comparison at the end of our property definition. Apparently, `tokenize` doesn't regard that empty string in the middle as a token. Actually, this is a feature of the standard Java `StringTokenizer` class. If there are no characters between two delimiters, `StringTokenizer` doesn't regard that as an empty string token, but instead as no token. Whether this is a bug or not is completely up to the person who is defining the specification. In this case, I would probably change the implementation to match the property, but you could just as well adjust the specification.

Conclusion

I won't take this example further here. After this quick overview, the upcoming chapters will describe ScalaCheck's features in greater detail. However, let me summarize what I wanted to show with this exercise.

First, while there are many differences between ScalaCheck and JUnit, they are quite similar on the surface. Instead of writing JUnit tests, you write ScalaCheck properties. Often you can replace several tests with one property. You manage and test your property collections in much the same way as your JUnit test suites. In this chapter, I only showed the console test runner of ScalaCheck, but other ways of running tests are shown in Chapter 4.

The differences between JUnit and ScalaCheck lie in the way you *think* about your code and its specification. In JUnit, you throw together several small usage examples for your code units, and verify that those particular samples work. You describe your code's functionality by giving some usage scenarios.

In property-based testing, you don't reason about usage examples. Instead, you try to capture the desired code behavior in a general sense, by abstracting over input parameters and states. The properties in ScalaCheck are one level above the tests of JUnit, so to speak. By feeding abstract properties into ScalaCheck, many concrete tests will be generated behind the scenes. Each automatically generated test is comparable to the tests that you write manually in JUnit.

What does this buy us, then? In Chapter 1, I tried to reason about the

advantages of property-based testing theoretically, and hopefully this chapter has demonstrated some of it practically. What happened when we ran our JUnit tests in the beginning of this chapter? They all passed. And what happened when we tested the ScalaCheck properties? They didn't pass. Instead, we detected several inconsistencies in our code. We were forced to think about our implementation and its specification, and difficult corner cases surfaced immediately. This is the goal of property-based testing; its abstract nature makes it harder to leave out parts and create holes in the specification.

It should be said that all the inconsistencies we found with ScalaCheck could have been found with JUnit as well, if we had picked more tests, with greater care. You could probably come a long way with JUnit tests just by applying a more specification-centered mindset. There's even a feature in JUnit 4 called *theories* that resembles property-based testing by parameterizing the test cases, but there's no support for automatically producing randomized values in the way ScalaCheck does. There's also nothing like ScalaCheck's rich API for defining custom test case generators and properties.

Lastly, as I've mentioned before, there is no need for an all-or-nothing approach when it comes to property-based testing. Cherry-picking is always preferred. Sometimes it feels right using a property-based method; in other situations, it feels awkward. Don't be afraid to mix techniques, even in the same project. With ScalaCheck, you can write simple tests that just cover one particular case, as well as complete properties that specifies the behavior of a method completely.

I hope that you are now intrigued by ScalaCheck's possibilities. The next chapter describes the fundamental parts of ScalaCheck and their interactions.