Chapter 2

ScalaCheck versus 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.

2.1 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.

2.2 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.

2.3 Using ScalaCheck

In ScalaCheck, you define *properties* instead of tests. To define a set of properties for our library under test, we extend `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 7.

```
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 these 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.

ScalaCheck's output shows that for the `StringUtils.truncate` property, we encountered a `StringIndexOutOfBoundsException` 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 `truncate` 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 `Prop.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 4.

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.

2.4 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 7.

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 of abstraction above the tests of JUnit. 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 reasoned 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 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—it parameterizes the test cases—but there's no support for automatically producing randomized values the way ScalaCheck does. There's also nothing like ScalaCheck's rich API for defining custom test case generators and properties.

Lastly, 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 cover one particular case, as well as thorough properties that specify 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.

Chapter 3

ScalaCheck Fundamentals

The two most fundamental concepts in ScalaCheck are *properties* and *generators*. This chapter will introduce the classes that represent properties in ScalaCheck, and bring up some technical details about the API. A following chapter will then present the multitude of different methods that exists in ScalaCheck's API for constructing properties.

Generators are the other important part of ScalaCheck's core. A generator is responsible for producing the data passed as input to a property during ScalaCheck's verification phase. Up until now we have sort of glanced over how ScalaCheck actually comes up with the values for the abstract arguments that your properties state truths about. The second part of this chapter will show what a generator is and demonstrate situations where you can make more explicit use of them.

The final section will talk about ScalaCheck's test case simplification feature, that was briefly mentioned in Chapter 1.

3.1 The `Prop` and `Properties` classes

A single property in ScalaCheck is the smallest testable unit. It is always represented by an instance of the `org.scalacheck.Prop` class.

The common way of creating property instances is by using the various methods from the `org.scalacheck.Prop` module. Here are some ways of defining property instances:

```
import org.scalacheck.Prop

val propStringLength = Prop.forAll { s: String =>
```

```
  val len = s.length
  (s+s).length == len+len
}

val propDivByZero =
  Prop.throws(classOf[ArithmeticException]) { 1/0 }

val propListIdxOutOfBounds = Prop.forAll { xs: List[Int] =>
  Prop.throws(classOf[IndexOutOfBoundsException]) {
    xs(xs.length+1)
  }
}
```

The first property is created by using the `Prop.forAll` method that you have seen several times before in this book. The second property uses `Prop.throws` that creates a property that tries to run a given statement each time the property is evaulated. Only if the statement throws the specified exception the property passes. The property `propListIdxOutOfBounds` in the example above shows that `Prop.forAll` not only accepts boolean conditions, but you can also return another property that then must hold for all argument instances.

The property values above are instances of the `Prop` class, and you can give the values to ScalaCheck's testing methods to figure out whether or not they pass.

When defining several related properties, ScalaCheck also has a class named `org.scalacheck.Properties` that can be used to group a bunch of properties together. It provides a way to label the individual property instances, and makes it easier for ScalaCheck to present the test results in a nice way. Using the `Properties` class is the preferred way of defining properties for your code. The code below shows how to use `Properties` to define a set of properties.

```
import org.scalacheck.Properties
import org.scalacheck.Prop.{forAll, throws}

object MySpec extends Properties("MySpec") {

  property("list tail") =
    forAll { (x: Int, xs: List[Int]) =>
      (x::xs).tail == xs
    }
```

```
  property("list head") = forAll { xs: List[Int] =>
    if (xs.isEmpty)
      throws(classOf[NoSuchElementException]) { xs.head }
    else
      xs.head == xs(0)
  }
}
```

The `Properties.property` method is used to add named properties to the set. If we check the property collection in the Scala console we can see the names printed:

```
scala> MySpec.check
+ MySpec.list tail: OK, passed 100 tests.
+ MySpec.list head: OK, passed 100 tests.
```

You mostly don't need to handle individual property instances, but sometimes it can be useful to reuse parts of properties, or combine several properties into one. For example, there is a && operator that creates a new property out of two other property instances. All the operators and methods that can be used to create properties are defined in the `org.scalacheck.Prop` module, and most of them are described in Chapter 5.

3.2 Generators

Up until now, we have never been concerned with how data is generated for our properties. Through the `Prop.forAll` method, we have simply told ScalaCheck to give us arbitrary strings, integers, lists, and so on, and ScalaCheck has happily served us the data when the properties have been evaluated.

However, sometimes we want a bit more control over the test case generation. Or we want to generate values of types that ScalaCheck know nothing about. This section will introduce the generators and show how you can make more explicit use of them.

The Gen class

A generator can be described simply as a function that takes some generation parameters and produces a value. In ScalaCheck, generators are represented by the Gen class, and the essence of this class looks like this:

```
class Gen[+T] {
  def apply(prms: Gen.Params): Option[T]
}
```

As you can see, a generator is parameterized over the type of values it produces. In ScalaCheck, there are default Gen instances for each supported type (Gen[String], Gen[Int], Gen[List], *etc.*). You can also see that the Gen.apply method returns the generated value wrapped in an Option instance. The reason for this is that sometimes a generator might fail to generate a value. In such cases, None will be returned. I will get back to why generators might fail in Chapter 6.

Normally, you don't deal with the Gen class explicitly, even when creating custom generator instances. Instead, you use one or more of the many methods in the module org.scalacheck.Gen. This module is quite independent from the other parts of ScalaCheck, so if you want you can use the generators in a project of your own just for data generation purposes, not only in the ScalaCheck properties you specify.

Let's fire up the Scala interpreter, define a generator, and see how to generate a value with it:

```
scala> import org.scalacheck.Gen
import org.scalacheck.Gen

scala> val myGen = Gen.choose(1,10)
myGen: org.scalacheck.Gen[Int] = Gen()

scala> myGen(Gen.Params())
res0: Option[Int] = Some(7)

scala> myGen.sample
res1: Option[Int] = Some(5)
```

First, we imported the Gen module. Then we created a generator, myGen, using the Gen.choose method. This method creates generators that will

generate random numbers in the given inclusive range. We can see from the type `Gen[Int]` of `myGen` that it will generate integers.

Finally, we used `myGen` to generate values in two different ways. In the first example, we can see how closely a generator resembles a function. We just apply the default generation parameters that are defined in `Gen`, and we get the generated value in return. In the second example, we use the `sample` method that exists on every generator; it is a convenient way of doing exactly the same thing.

In the example, you can also see that the generator returns its value as an `Option` type, which was mentioned previously. The generators you can create by using the `Gen.choose` method will never fail, but will always deliver a `Some`-instance containing a value.

The parameters a generator uses to generate data contain information about which random number generator should be used and how large the generated data should be. Chapter 6 will describe the parameters more closely; for now, you can just use `Gen.Parameters()` or the `sample` method as shown previously.

Defining custom generators

As I've mentioned, there are many methods you can use to create your own generators in the `Gen` module. These methods are called *combinators*, since you can use them as basic building blocks for generating more complex structures and classes. To combine them together, you use Scala's versatile *for* statement, which is mostly used in loop constructs but in fact is much more general. Here is an example of its use with generators:

```
import org.scalacheck.Gen.choose

val myGen = for {
  n <- choose(1, 50)
  m <- choose(n, 2*n)
} yield (n, m)
```

In this example, `myGen` generates randomized tuples of integers, where the second integer always is larger than or equal to the first, but not more than twice as large. With the `sample` method, we can check that it is working as expected:

```
scala> myGen.sample
res0: Option[(Int, Int)] = Some((45,60))

scala> myGen.sample
res1: Option[(Int, Int)] = Some((29,37))
```

You can define generators to build any structure. Consider the following simple types that model shapes and color:

```
trait Color
case object Red extends Color
case object Green extends Color

trait Shape { def color: Color }
case class Line(val color: Color) extends Shape
case class Circle(val color: Color) extends Shape
case class Box(val color: Color,
  val boxed: Shape) extends Shape
```

We can now define generators for the `Color` and `Shape` types:

```
import org.scalacheck.Gen

val genColor = Gen.oneOf(Red, Green)

val genLine = for { color <- genColor } yield Line(color)
val genCircle = for { color <- genColor } yield Circle(color)
val genBox = for {
  color <- genColor
  shape <- genShape
} yield Box(color, shape)

val genShape: Gen[Shape] =
  Gen.oneOf(genLine, genCircle, genBox)
```

In this example, we used `Gen.oneOf`, which takes an arbitrary number of generators (or plain values) and creates a new generator that will use one of the provided generators at random when it is evaluated. As you can see, `genBox` and `genShape` are *recursive* generators. There are some things you should be aware of when defining recursive generators in order to not cause infinite recursions and huge data structures. This will be covered in Chapter 6. However, the above generator definition should be just fine, because it converges quickly as we can see when we try it out:

```
scala> genShape.sample
res0: Option[Shape] = Some(Line(Green))

scala> genShape.sample
res1: Option[Shape] =
  Some(Box(Blue,Box(Red,Circle(Green))))
```

As I've said, data generators are not exclusively related to properties; you can use the `Gen` module as an API for defining data generators for any setting really. Chapter 6 will provide reference information about most of the methods in `Gen`, and also show how to use the generator parameters, both when evaluating generators and when defining them.

Making explicit use of generators in properties

In most of the properties shown earlier, ScalaCheck has automatically picked suitable generator instances and used them behind the scenes when evaluating the properties. However, you can instruct ScalaCheck explicitly to use a certain generator in a property definition. You can use `Prop.forAll` with one extra parameter to inform ScalaCheck which generator to use:

```
import org.scalacheck.{Gen, Prop}

val evenInt = for {
  n <- Gen.choose(-1000, 1000)
} yield 2*n

val propDivide = Prop.forAll(evenInt) { n: Int =>
  val half = n/2
  n == 2*half
}
```

You can also specify several explicit generators for one property:

```
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.{posNum, negNum}

val p = forAll(posNum[Int], negNum[Int]) { (n,m) =>
  n*m <= 0
}
```

Another common usage of explicit generators is to nest `forAll` invocations, and let the inner one use an explicit generator that is defined in terms of the generated value in the outer `forAll`:

```
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.choose

val propPrefix = forAll { s: String =>
  forAll(choose(0, s.length)) { n =>
    val prefix = s.substring(0, n)
    s.startsWith(s)
  }
}
```

Instead of nesting `forAll` calls, we could have defined a custom generator in the following way:

```
import org.scalacheck.Arbitrary.arbitrary
import org.scalacheck.Gen.choose

val genStringWithN = for {
  s <- arbitrary[String]
  n <- choose(0, s.length)
} yield (s,n)
```

We can now specify the property with only one `forAll` call:

```
import org.scalacheck.Prop.forAll

val propPrefix = forAll(genStringWithN) { case (s,n) =>
  val prefix = s.substring(0, n)
  s.startsWith(s)
}
```

Notice that we have to use a `case`-expression since our property takes one tuple as its argument, not two separate arguments.

Whether you use nested `forAll` calls or custom generators is largely a matter of taste. If you have a lot of input arguments to your properties, putting them in a separate generator can make the property easier to read.

Adding implicit support for custom generators

I gave you a quick introduction to defining generators and then using them with the `Prop.forAll` method. However, you can also add implicit support for your own generators so you can write properties for your own classes in exactly the same way you would for the standard types, without explicitly specifying which generator to use in every property.

The key to this lies in Scala's built-in support for implicit methods and values. ScalaCheck can pick up default generators for any type if an implicit instance of the `Arbitrary` class for the given type exists. The `Arbitrary` class is simply a factory that provides a generator for a given type. In the example below, we first define a generator for a simple type and then make an implicit `Arbitrary` instance for it by using the `Arbitrary` module.

```
import org.scalacheck.Gen.{choose, oneOf}

case class Person (
  firstName: String,
  lastName: String,
  age: Int
) {
  def isTeenager = age >= 13 && age <= 19
}

val genPerson = for {
  firstName <- oneOf("Alan", "Ada", "Alonzo")
  lastName <- oneOf("Lovelace", "Turing", "Church")
  age <- choose(1,100)
} yield Person(firstName, lastName, age)
```

Given this `Person` generator, making an implicit `Arbitrary[Person]` instance is simple:

```
scala> import org.scalacheck.Arbitrary
import org.scalacheck.Arbitrary

scala> implicit val arbPerson = Arbitrary(genPerson)
arbPerson: org.scalacheck.Arbitrary[Person] =
  org.scalacheck.Arbitrary$$anon$1@1391f61c
```

As long as `arbPerson` is in scope, we can now write properties like this:

```
scala> import org.scalacheck.Prop.forAll
import org.scalacheck.Prop.forAll

scala> val propPerson = forAll { p: Person =>
         p.isTeenager == (p.age >= 13 && p.age <= 19)
       }
```

3.3 Test case simplification

As soon as ScalaCheck manages to falsify a property, it will try to simplify, or *shrink*, the arguments that made the property false. Then it will re-evaluate the property with the simplified arguments. If the property still fails, simplification will continue. In the end, the smallest possible test case that makes the property false will be presented along with the the original arguments that caused the initial failure. We can demonstrate this by defining a property that is intentionally wrong, to trigger the simplification mechanism in ScalaCheck:

```
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.choose

val propNoPairs = forAll { xs: List[Byte] =>
  forAll(choose(0, xs.length-2)) { i =>
    xs(i) != xs(i+1)
  }
}
```

The property states that there never exists a pair of equal items in a random list, which simply is false. Let's see what happens if we check the property:

```
scala> propNoPairs.check
! Falsified after 11 passed tests.
> ARG_0: List("127", "127")
> ARG_0_ORIGINAL: List("-104", "127", "127", "-1", "89")
> ARG_1: 0
```

ScalaCheck correctly finds a test case (`ARG_0_ORIGINAL`) that makes the property false. Then this value is repeatedly simplified until `ARG_0` remains, that still makes the property false.

ScalaCheck has the ability to simplify most data types for which it has implicit generators. There is no guarantee that the simplification will be perfect in all cases, but they are helpful in many situations. Where ScalaCheck has no built-in simplification support, you can add it yourself, just as you can add implicit generators for custom types. Therefore, you can give your own types and classes exactly the same level of support as the standard ones in ScalaCheck. In Chapter 6, you will be shown how to define such custom simplifiers for your own types.

3.4 Conclusion

This chapter has presented the fundamental parts of ScalaCheck, getting you ready to use it in your own projects. The next chapter will focus less on the technical details of ScalaCheck, and instead provide general techniques and ways to think when coming up with properties for your code. Later chapters will then revisit the topics of this chapter, digging deeper into the details of the API.