Accenture Architecture Innovation

ScalaCheck Cookbook

Contents

1 INTRODUCTION

1.1 Overview

The purpose of this document is to introduce Scala and Java developers to ScalaCheck, how to build test cases and how to integrate ScalaCheck cases with existing unit test case suites.

Some of the examples can be run from Scala's console by simply copying and pasting the code, but all of the code used throughout this document is available as a separate package (Maven 2.x or 3.x, and Simple Build Tool 0.7.x are required to build the examples).

1.2 Knowledge Requirements

In order to maximize the value of the content in this deliverable and to be able to create property tests and custom generators with ScalaCheck, it is essential to understand the following Scala concepts:

- Case classes
- Implicit conversions and parameters
- Generic typing

2 WHAT IS SCALACHECK?

ScalaCheck is a unit testing library that provides unit testing of code using property specifications with random automatic test data generation.

ScalaCheck is currently hosted in http://code.google.com/p/scalacheck/, including documentation and readymade JAR files, and the source code is currently available from github.com: https://github.com/rickynils/scalacheck.

Other features provided by ScalaCheck include:

- Test case minimization (i.e. what is the shortest test case the fails?)
- Support for custom data generators
- Collection of statistical information about the input data

2.1 Benefits of ScalaCheck compared to traditional unit testing

ScalaCheck is not meant to replace traditional unit testing frameworks (JUnit, Specs, ScalaTest), but to extend them and provide some advantages for new and existing unit testing suites:

- Automatic generation of data allows developers to focus on defining the purpose of the actual test case, rather than spending time and energy looking for corner cases by themselves
- Property-based tests provide a lot more testing for a lot less code than assertion-based tests
- Can help in scenarios where whole classes of test cases have to be tested and it is not feasible to write tests for all distinct test scenarios

2.2 Use Cases

The following are use cases and scenarios where ScalaCheck could prove itself useful for existing projects:

- Testing of code that expects any kind of input values within the boundaries of a specific data type or domain type, i.e. code that works on strings or code that works on domain classes, where ScalaCheck can create any kind of string or any kind of random domain object for validating purposes
- Application logic that maintains mutable state, where ScalaCheck generates random commands/input that affect the internal state that gets validated at the end of the property check
- State machines, where ScalaCheck can be used to generate random input and to verify the correctness of the end state
- Parsers
- Data processors
- Hadoop's mappers and reducers
- Validators (i.e. validators of web form fields in Spring or equivalent)

3 GETTING STARTED

3.1 Properties and Generators

ScalaCheck is based around the concept of Properties and Generators. According to the official ScalaCheck documentation, a property is "*the testable unit in ScalaCheck*", while a Generator is defined as "*responsible for generating test data in ScalaCheck*".

Properties are used to test the output of a specific function while generators are used by properties to generate random data for the logic checked by the property, and the property will confirm whether the property holds true based on the random test data given by the Generator, or if it is falsified.

3.2 Creating tests with ScalaCheck

Creating property checks with ScalaCheck requires that we:

- define the logic for the property check
- define the data requirements; are the checks going to be performed using basic data types, or will they be performed using custom objects/domain objects from our application?

It is important to know that ScalaCheck properties are evaluated in terms of Boolean logic: either the property holds or the property is falsified (i.e. the code must return to true or false as its result). ScalaCheck does not provide support for assertions so all property checks are generally validated using the equality == operator. The only exception is exceptions (no pun intended) as ScalaCheck provides support to validate a property based on whether it throws an exception or not.

Once the logic for the check has been defined, we need to define the kind of data to be used.

ScalaCheck's main feature is that it relies on automatic test data generation to ensure that we don't miss any case when writing our unit tests. In order to do that, it provides mechanisms out-of-the-box to generate random data for basic data types (strings, integers, double-precision numbers, amongst other basic types), as well as containers for those types (lists, sequences, options, amongst others).

If our property checks do not use basic data types, we will have to write our own custom data generators before we can write the property checks. Fortunately, writing custom generators is not a very complex task (and in most cases it is an essential one).

The next two sections will walk us through the process of creating our first ScalaCheck properties.

3.3 Running the sample snippets

In addition to the accompanying package with code examples, throughout the content below there are a lot of code snippets that show concepts as they are introduced to the reader. The code example package (see link in Appendix 1) should be deployed to a directory on the local machine to facilitate execution of the snippets below.

Most of the snippets are executable when typed directly in a Scala console. The beginning of each main section or subsection will point to a folder with the relevant source code and other related files.

Please refer to Appendix 1 for more details on the tools required.

4 WORKING WITH PROPERTIES

The examples in this section can be run from the scalacheck-basic folder. In a command prompt window, switch to the scalacheck-basic folder and run the command "*mvn scala:console*".

4.1 Defining a simple property

The simplest possible property-based test uses the *forAll* object, with a closure/function as the logic for the check, and uses an existing generator to generate random data (two lists of integers in the example below):

```
import org.scalacheck.Prop.forAll

val propConcatLists = forAll { (l1: List[Int], l2: List[Int]) => l1.size + l2.size == (l1
::: l2).size }
```

In order to execute a property test, the *check* method can be used:

```
propConcatLists.check
```

When run like this, ScalaCheck will use the default generators in scope (more on this later), and run all the possible inputs, up to 100 by default, through our property check and report whether the property held true for all of them or if any of them was falsified.

All this code can be run from within the Scala console (REPL). When running this code, the following should be visible on the screen:

```
+ OK, passed 100 tests.
```

org.scalacheck.Prop.forAll takes a function as a parameter, and creates a property that can be tested with the *check* method. The function passed to *forAll* should implement the test/check logic, and should return a Boolean value (defining whether the test was successful or not).

The function provided to *forAll* can receive a parameter of any type as long as there's a suitable *Arbitrary* instance in the function scope. The Arbitrary type will be described in detail later on in this document but in a nutshell, it is a special wrapper type in ScalaCheck that is used to generate random data. ScalaCheck provides built-in generators for standard data types, but it is also possible to build custom data generators for our own data types (e.g., domain objects from our application). See the chapter below on custom generators and arbitrary types.

It is possible to have more than one property to test the same function, and to combine multiple properties into a single one to check a single function.

4.2 Grouping properties

While it is possible to define multiple properties as immutable values (Scala's *val*s) or nullary methods (methods with no parameters), it is usually more convenient to logically group related properties in a single class/object extending the *Properties* class. Definition of property checks when using the *Properties* class is done using the *property* method:

```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
object BiggerSpecification extends Properties("A bigger test specification") {
  property("testList") = forAll { (l1: List[Int], l2: List[Int]) =>
    l1.size + l2.size == (l1 ::: l2).size
  }
  property("Check concatenated string") = forAll { (a:String, b:String) =>
    a.concat(b) == a + b}
}
```

When using the *Properties* class, all the enclosed properties can be run at once with the *Properties.check* or *Properties.check(params)* methods (depending on whether we want to provide testing parameters or run with default parameters)

```
BiggerSpecification.check
```

Additionally, the Properties class provides the include() method that allows the creation of groups of specifications based on other groups of specifications:

```
object AllSpecifications extends Properties("All my specifications") {
  include(BiggerSpecification)
  include(AnotherSpefication)
  include(...)
}
```

4.3 Advanced Property Usage

4.3.1 Conditional properties

Often, it's useful to run our properties for a specific range or subset of data. This can be implemented with a generator that only generates the values that we want or, depending on the kind of subset, properties can be defined to select valid data for their check with a filter condition.

For the former, examples of custom generators will be shown later. For the latter usage with filters, we have to provide a Boolean predicate that upon evaluation to true will run the provided property check. The predicate is linked with the property check using the ==> method:

```
import org.scalacheck.Prop._
val propMakeList = forAll { (n: Int) =>
  (n >= 0 && n <20000) ==> (List.fill(n)("").length == n)
}
```

As usual, the property above can be run with *propMakeList.check*.

It is important to keep in mind not to be too strict when defining the conditions, as ScalaCheck will eventually give up if only a very low amount of test data fulfills the condition and will report that very little data was available for the property. Compare the following property check with the one above:

```
import org.scalacheck.Prop._
val propMakeListPicky = forAll { n: Int =>
  (n >= 0 && n <1) ==> (List.fill(n)("").length == n)
}
```

When running *propMakeListPicky.check*,tit would produce the following output:

```
! Gave up after only 42 passed tests. 500 tests were discarded.
```

This was a rather extreme example but there may be real world situations where this may become an issue. Please note that the number of passed tests will vary from execution to execution, due to the random nature of the input data.

4.3.2 Combining properties

ScalaCheck also provides Boolean operators and methods to combine several different properties into one:

prop1 && prop2	Returns a new property that holds if and only if both prop1 and prop2 hold. If one of the properties doesn't generate a result, the new property will generate false
prop1 || prop2	Returns a new property that holds if either prop1 or prop2 (or both) hold
prop1 == prop2	Returns a new property that holds if and only if prop1 holds exactly when prop2 holds. In other words, every input which makes prop1 true also makes prop2 true and every input which makes prop1 false also makes prop2 false.
all(prop1, prop2, …, propN)	Combines properties into one which is true if and only if all the properties are true
atLeastOne(prop1, …, propN)	Combines properties into one which is true if at least one of the properties is true

5 GENERATORS

The samples in this section can be run using the contents of the scalacheck-basic, folder, but the content needs to be compiled. In a command window, switch to the scalacheck-basic folder and then first type “mvn compile” and then “mvn scala:console”.

5.1 Built-in generators

ScalaCheck provides built-in generators for common data types: Boolean, Int, Long, Double, String, Char, Byte, Float, containers with those data types (Lists, Arrays), and other commonly used objects such as Date, Throwable, Scala's Option and Either, Tuples, and Function objects. All these built-in generators are in class *org.scalacheck.Arbitrary*.

These generators can be used as-is in our own property checks, or can be combined to create more complex generators and arbitrary objects for our own property checks.

5.2 Custom generators

Often, when testing our own domain objects and functionality, it is be necessary to create our own custom generators to act as random data sources for our property checks. This allows us to define property checks with functions that take domain objects as their parameters.

Custom generators in ScalaCheck are implemented using class *org.scalacheck.Gen*.

In order to present custom generators, the Rectangle class will be used as our domain object. This is a simple case class that encapsulates the (simple) logic for handling a rectangle with height and width, with the following constructor signature:

```
case class Rectangle(val width: Double, val height: Double) {
  // note that there's a bug in the function below!
  lazy val area =  if(width % 11 ==0) (width * 1.0001 * height) else (width * height)
  // correct version of the area method
  lazy val areaCorrect = (width * height)
  lazy val perimeter = (2*width) + (2*height)
  def biggerThan(r:Rectangle) = (area > r.area)
}
```

Please note that in the snippet above the *Rectangle.area* method contains a small bug in it in order to force it to fail for certain values. The correct area method is called *areaCorrect*.

This class also provides methods *area*, *perimeter* and *biggerThan*. The full source code for this class is available as part of this deliverable (see References in the Appendix).

At their simplest level, generators are functions that optionally take some input data (where this input data is used to customize the kind of random data that is generated) and generate output for ScalaCheck to use as input in a property check.

In the case of the Rectangle class, we'll create a generator function that returns a Rectangle object as well as the random height and the width values that were used to create the object. The purpose for this generator is to create a property check to verify that the internal calculation for the area and the width is correct.

Generators return Option objects, which means that they do not necessarily have to return an actual value if the input data is not suitable to generate output values.

In our example, the generator will be defined as a function that returns a tuple of 3 elements of type *(Rectangle, Double, Double)*, where the Double values represent the height and the width that were used to create the Rectangle object:

```
import org.scalacheck.Gen
val rectangleGen: Gen[(Rectangle, Double, Double)] = for {
  height <- Gen.choose(0,9999)
  width <- Gen.choose(0,9999)
} yield((Rectangle(width, height), width, height))
```

It is common to implement generators using a for comprehension in order to keep the code short and concise, but as generators are normal functions, they can be implemented in any way as long as they return a value of the required type (or no value, see below)

Now that we've created the generator, it can also be run as a standalone object using its *apply* method in Scala's console; the *apply* method returns an Option[T] object (so that generators can also return no value using the None object, if needed) where T is the type of the generator, in this case *(Rectangle, Double, Double)*. The *apply* method also needs a *Params* object in order to run, which is used to encapsulate all the parameters needed for data generation:

```
rectangleGen(new org.scalacheck.Gen.Params)
```

Alternatively, the parameterless *sample* method can be used instead to generate sample values from an existing generator.

In the console, the result of running either one of the methods above will produce something like this:

```
Option[(Rectangle, Double, Double)] = Some((Rectangle(9714.0,7002.0),9714.0,7002.0))
```

Now our property check can be written using our new custom generator by providing it as a parameter to the *forAll* method. Once the generator is in place, the same data type that is produced by the generator must be used as the type for the input parameter(s) of the property check function. In this case, the tuple *(Rectangle, Double, Double)* is used as the return type by the generator and the type for the *input* parameter. If the types do not match, the Scala compiler will complain.

The code for the property check using the generator is:

```
import org.scalacheck.Prop._
import org.scalacheck.{Arbitrary, Properties, Gen}

object RectangleSpecification extends Properties("Rectangle specification") {
  property("Test area") = forAll(rectangleGen) { (input: (Rectangle,Double,Double)) =>
    input match {
      case(r, width, height) => r.area == width * height
  }}
}
```

In the example code above, pattern matching is used so that we can extract named parameters from the tuple; this is not necessary and was only implemented for clarity reasons, as we could have also referred to the values within the tuple using their indexes (*input._1*, *input._2* and *input._3*)

To run this property check, use the *RectangleSpecification.check* method. As we've used incorrect logic to calculate the area, the output should be failure:

```
! Rectangle specification.Test area: Falsified after 15 passed tests.
```

```
> ARG_0: (Rectangle(4169.0,1988.0),4169.0,1988.0)
```

5.3 The Gen companion object

The Gen companion object provides some commodity methods and implicit methods to generate all sorts of data. Two of the most useful ones are the *Gen.choose* method, which is able to generate random data of any type (as long as there's an implicit Choose[T] object in scope for the type, but this is always the case for basic data types) within the given limits and *Gen.oneOf*, which returns an element from the given set.

5.4 The Arbitrary generator

The Arbitrary generator is a special generator in ScalaCheck. Arbitrary generators are built on existing generators.

The Arbitrary generator allows us to simplify our properties because the data generation logic is implemented as an implicit function that is brought into the scope of the property check. This means that when using Arbitrary objects, ScalaCheck can automatically generate test data for any custom type without the need to provide an explicit reference to the generator(s) required in the call to *forAll*.

In this example, we'll use the same Rectangle case class but we'll create an implicit value which wraps the Rectangle generator and returns an Arbitrary[Rectangle] object every time it is called.

First of all, we need a generator of Rectangle objects. Note that this time we're only returning Rectangle objects:

```
val arbRectangleGen: Gen[Rectangle] = for {
  height<- Gen.choose(0,9999)
  width<- Gen.choose(0,9999)
} yield(Rectangle(width, height))
```

Once the generator is in place, defining the arbitrary generator is a rather straightforward affair using the Arbitrary object and providing our generator as a parameter to its *apply* method:

```
import org.scalacheck.Arbitrary
implicit val arbRectangle: Arbitrary[Rectangle] = Arbitrary(arbRectangleGen)
```

In some cases, the Scala compiler will not be able to infer the type of our arbitrary object, therefore the explicit type may need to be provided as part of the val definition (as we did above).

Finally, we create our new property that checks the correctness of the *Rectangle.biggerThan* method:

```
object ArbitraryRectangleSpecification extends Properties("Arbitrary Rectangle spec") {
  property("Test biggerThan") = forAll{ (r1: Rectangle, r2: Rectangle) =>
    (r1 biggerThan r2) == (r1.area > r2.area)
  }
}
```

Note how the *forAll* method is not provided an explicit reference to an existing generator, because it's using the implicit Arbitrary generator *arbRectangle* that we just defined, which is conveniently in the scope of the property check. This means that our new property check is not bound to any specific generator but only to the one that is currently in scope. It also means that the same property check can be transparently reused with different generators only by importing different implicit Arbitrary generators.

Compare the property above with the same version of the property check but without an arbitrary generator:

```
object ... extends Properties(“...”) {
property("Test biggerThan") = forAll(rectangleGen, rectangleGen){ (r1: Rectangle, r2:
Rectangle) =>
    (r1 biggerThan r2) == (r1.area > r2.area)
  }
}
```

The main difference is that now two custom generators have been provided, one for each one of the parameters of the function provided to *forAll*. In order to keep our code focused on the actual testing logic and remove all the typing boilerplate, it is recommended to create the implicit Arbitrary generator.

5.5 More generators

In addition to the topics described above, ScalaCheck also provides more specialized generators such as sized generators, conditional generators and generators of containers (such as lists and arrays). Please refer to the ScalaCheck user guide as well as to the ScalaCheck online API documentation for more information.

6 ANALYZING TEST DATA

In addition to generating test data, ScalaCheck can also collect the data, group it using custom rules, and report the statistics upon test completion. This feature is useful to visualize, as part of the output of the property check execution, whether ScalaCheck is using sufficiently equally distributed data.

The two key methods for data gathering and grouping are *classify* and *collect*, both part of the *Prop* companion object. When using either one of these two methods, ScalaCheck will print the results of the grouping to the console upon completion of the property check.

The *collect* method is the easiest way to collect all the data generated by ScalaCheck, as it can use any kind of criteria to group our data. Data can be grouped directly based on its actual values, such as the example below where the input of the property is used as the grouping criterion:

```
object DataCollectionSpecification extends Properties("Data collection examples") {
  property("data collection spec") = forAll { (r: Rectangle) =>
    collect((r.width, r.height)) {
      r.areaCorrect == (r.width * r.height)
    }
  }
}
```

In the previous example, the console output will show a long list of pairs of height and width, each one of them with a 1% frequency, similar to this:

```
+ Data collection examples.data collection spec: OK, passed 100 tests.
```

```
> Collected test data:
1% (1059.0,3685.0)
1% (8004.0,2400.0)
1% (5543.0,751.0)
1% (2941.0,6644.0)
1% (277.0,9994.0)
1% (973.0,289.0)
1% (7662.0,3924.0)
(...up to 100 lines...)
```

It is clear that a smarter mechanism for grouping the data is needed here. In the case of the *collect* method, ScalaCheck will use the values provided to the method call to group the data, so it's possible to provide a custom function of the input data that provides for coarser grouping of the data. In this case, we are going to group rectangles in three categories based on the size their perimeter ("small", "medium" and "large") using the following function:

```
val collector: PartialFunction[Rectangle, String] = {
  case r if r.perimeter < 10000 => "small"
  case r if r.perimeter > 10000 && r.perimeter < 25000 => "medium"
  case r if r.perimeter > 25000 => "large"
}
```

A partial function is used here as opposed to a traditional function because partial functions allow us to directly plug in pattern matching with guards that are used to define how our input data gets classified. (This partial function is actually a full function, but if we had declared it as a full function we would have need to include an artificial default case, e.g., *case _=> ""*. to avoid a " match is not exhaustive!" warning. Alternatively, the function could be defined using a chain of if…else blocks. The partial function syntax is more concise and clear in this situation.) The return value of the function is a string as this function must always return something which is used by ScalaCheck as the grouping criterion for our data. Now we only have to slightly rewrite our property check to ensure that our *collector* function is called with each one of the input data:

```
object DataCollectionSpecificationWithCollect extends Properties("Grouping with collect") {
  property("Prop.collectwith a grouping function") = forAll { (r: Rectangle) =>
    collect(collector(r)) {
      r.areaCorrect == (r.width * r.height)
    }
  }
}
```

Now the console output is much more useful:

```
> Collected test data:
63% medium
24% large
13% small
```

The *Prop.classify* method offers similar features to *Prop.collect* but one key advantage of *Prop.classify* is that it allows multi-level grouping of data. In our example scenario, data could also be grouped based on whether the rectangle was wider or taller in addition to the size of its perimeter:

```
object DataCollectionWithGroupAndCollect extends Properties("Grouping with both") {
  property("Prop.classify with Prop.collect") = forAll { (r: Rectangle) =>
    classify(r.height > r.width, "taller", "wider") {
      collect(collector(r)) {
        r.areaCorrect == (r.width * r.height)
      }
    }
  }
}
```

The console output after running DataCollectionWithGroupAndCollect.check will be something like this:

```
> Collected test data:
32% medium, taller
26% medium, wider
21% large, wider
10% large, taller
9% small, wider
2% small, taller
```

Here we've used the "binary" version of *Prop.classify* to classify our random rectangle objects into "taller" or "wider", and then with *Prop.collect* we've collected the data using our previous function. The output now is a two-level grouping of our data.

The previous example also shows how *Prop.classify* and *Prop.collect* can be combined within the same property check, and even multiple calls can be nested to obtain a more granular classification (this is because according to their method signature, *Prop.classify* and *Prop.collect* return a Prop object, and Prop objects can be chained).

All in all, both *classify* and *collect* provide powerful mechanisms to examine the distribution of the random data being used by ScalaCheck for testing our property checks.

7 SCALACHECK INTEGRATION

7.1 Using ScalaCheck to test Java code

Since Java and Scala run side-by-side in the same JVM using the same bytecode, there are no special requirements to get ScalaCheck to validate existing Java code.

The set of sample code provided with this document (see References in the Appendix) contains one such example where a class encapsulating the functionality of a bank account implemented in Java is tested using property checks written in ScalaCheck. The Account class is a common example of a mutable domain object. The code used throughout the examples below can be found in the **java-scalacheck** folder.

7.1.1 The scenario

The Java code being tested is a simple class that represents a bank account, on which deposit and withdrawal operations can be performed. The operations modify the internal state of the object. Additionally, there are methods for retrieving the balance, account age, account rate, and depositing some credit interest back into the account:

```
public class Account {

  public final static int GOLD_AGE = 50;
  public final static double GOLD_BALANCE = 10000;
  public final static double STD_INTEREST = .02;
  public final static double GOLD_INTEREST = .03;

  private int id;
  private int age;
  private double balance;

  public Account(int id, int age, double balance) {
    // set internal attributes id, age and balance
  }

  public double getBalance() {...}

  public int getAge() {...}

  public void deposit(double amt) {
    assert (amt > 0);
    balance += amt;
  }

  public void withdraw(double amt) throws InsufficientFundsException {
    assert(amt > 0);
    if (amt <= this.balance)
      balance -= amt;
    else
     throw new InsufficientFundsException();
  }

   public double getRate() {
    if (balance < Account.GOLD_BALANCE && age < Account.GOLD_AGE)
      return(Account.STD_INTEREST);
    else
      return(Account.GOLD_INTEREST);
  }

    public void creditInterest() {
    deposit(getRate() * balance);
  }
}
```

7.1.2 The tests

The code in this section requires the Java classes to be compiled and available in the classpath, and we can let Maven take care of that. In order to run the examples below please switch to the java-scalacheck folder in the examples package, and run *mvn compile scala:console*. Please note that this example is presented as one single large specification in the example code but as smaller independent ones here, so that they can be separately tested.

Since all property checks require Account domain objects, the GenAccount singleton provides a generator for Account domain objects as well as an Arbitrary generator:

```
import org.scalacheck.Prop._
import org.scalacheck.{Arbitrary, Properties, Gen}
import com.company.account.Account

object GenAccount {
  import com.company.account.Account._

  val MAX_ID: Int = 999999
  val MAX_AGE: Int = 200
  val MAX_BALANCE: Double = 10 * GOLD_BALANCE

  def genAccount(maxId: Int, maxAge: Int, maxBalance: Double): Gen[Account] = for {
    id <- Gen.choose(0, maxId)
    age <- Gen.choose(0, maxAge)
    balance <- Gen.choose(0, maxBalance)
  } yield new Account(id, age, balance)

  implicit val arbAccount: Arbitrary[Account] =
    Arbitrary(genAccount(MAX_ID, MAX_AGE, MAX_BALANCE))
}

import GenAccount._
```

In the *genAccount* generator method, the choose method in the *Gen* companion object is used extensively to create random numbers. Those random numbers are used to initialize a new entity of the *Account* domain class using a *for* comprehension. Once we have got the generator in place, building an Arbitrary object is very straightforward, but please notice how in this example the values generated by the generator can be customized based on its input parameters, and how those parameters are provided to the generator when building the Arbitrary object (the generators seen previously did not use any parameters).

Property checks for the Account class are grouped into a separate *AccountSpecification* object. In this object two groups of property checks have been defined: one of the groups only uses an Account object, while the second group uses an Account object as well as a Double entity that represents an amount to deposit or withdraw.

For the first group of property checks, ScalaCheck will use the implicit arbitrary generator that we just defined. For example:

```
object AccountCreditCheckSpecification extends Properties("Credit interest check") {
  property("CreditInterest") = forAll { acct: Account =>
    val oldBalance = acct.getBalance()
    acct.creditInterest()
    acct.getBalance() == oldBalance + (oldBalance * acct.getRate())
  }
}
```

For the second group of property checks, there's a need to have a second generator that will generate random tuples of two elements of type *(Account, Double)*. The generator will use the arbitrary generator of Account objects previously defined so there is no need to duplicate any code, while the Double value will be generated using *Gen.choose*:

```
val genAcctAmt: Gen[(Account, Double)] = for {
  acct <- Arbitrary.arbitrary[Account]
  amt <- Gen.choose(0.01, MAX_BALANCE)
} yield (acct, amt)
```

In this case there is no arbitrary generator for tuples of type (Account, Double) – even though there could be, hence we'll need to provide an explicit reference to this generator in those property checks that require this tuple:

```
object AccountWithdrawalSpecification extends Properties("Withdrawal specification") {
  property("Withdraw-normal") = forAll(genAcctAmt) {
    case (acct: Account, amt: Double) =>
      amt <= acct.getBalance() ==> {
        val oldBalance = acct.getBalance()
        acct.withdraw(amt)
        acct.getBalance() == oldBalance - amt
       }
  }
}
```

For comparison's sake, the package with the code used throughout this document contains a version of the same property check but using an implicit object generator. The arbitrary generator as well as the modified version of the property check using the generator would look as follows:

```
implicit val arbAccountAmount: Arbitrary[(Account, Double)] = Arbitrary(genAcctAmt)

object AccountDepositWithArbitrary extends Properties("Account deposit") {
property("Deposit-with-Arbitrary") = forAll { (input: (Account, Double)) =>
  input match {
    case (acct: Account, amt: Double) =>
      val oldBalance = acct.getBalance()
      acct.deposit(amt)
      acct.getBalance() == oldBalance + amt
    }
  }
}
```

There's very little difference between the two examples above; creating the arbitrary object once the generator function already exists is rather easy, and later the generator function is not passed as a parameter to *Prop.forAll* anymore.

The rest of the sample code contains property checks built not unlike the examples described so far, where the arbitrary generator of Account objects is used as part of property checks that validate certain functionality as in the “Rate-highbalance”, “Rate-highAge” and “CreditInterest” property checks.

The “Rate-lowBalance, lowAge” property check uses a custom generator that is scoped to remain within the boundaries of that one specific property check:

```
object RateLowBalanceLowAgeSpecification extends Properties("Low balance, low age spec") {
  import com.company.account.Account._
  property("Rate-lowBalance, lowAge") = {
    val gen = genAccount(MAX_ID, GOLD_AGE - 1, GOLD_BALANCE - .01)
    forAll(gen) {
      acct: Account => acct.getRate() == STD_INTEREST
    }
  }
}
```

In this example, we are using the custom generator of Account objects with some very specific values, and it is kept within the local scope because it is not used in any other property check.

Since the Java code is using exceptions for handling some error scenarios, lastly the “Widthdraw-overdraft” property check uses *Prop.throws* to add a condition that validates that the code throws exceptions under certain circumstances:

```
import org.scalacheck.Prop
object WithdrawOverdraftSpecification extends Properties("Withdraw overdraft spec") {
  import com.company.account.InsufficientFundsException
  property("Withdraw-overdraft") = forAll(genAcctAmt) {
    case (acct: Account, amt: Double) =>
      amt > acct.getBalance() ==> {
      val oldBalance = acct.getBalance()
      Prop.throws(acct.withdraw(amt), classOf[InsufficientFundsException]) &&
acct.getBalance() == oldBalance
    }
  }
}
```

The previous property check will only run when the given random amount is greater than the remaining balance in the account; if this condition is fulfilled, the code will use *Prop.throws* to ensure that the when calling the *Account.withdraw* method, an exception of type *InsufficientFundsException* is thrown every time when the amount withdrawn is greater than the current amount in the account.

7.2 Using ScalaCheck with Scala Specs

The Specs behavior-driven design library also provides integration of ScalaCheck’s properties when mixing in the *ScalaCheck* trait. The example code below can be found in the **scalacheck-integration-specs** folder.

In order to run the snippets below, switch to the scalacheck-integration-specs folder and run the following command: “sbt update compile console” (essentially, we are telling SBT to update/download the project dependencies, compile the code and start a Scala console, all in one single command).

Specs2’s *expectations* when using ScalaCheck properties are written by providing the property function to Specs’s *check* method:

```
import org.specs2.mutable._
import org.specs2.ScalaCheck
object SimplePropertySpec extends Specification with ScalaCheck {
  "Strings" should {
    "String.concat property" ! check { (a:String, b:String) =>
      a.concat(b) == a + b
    }
  }
}
```

Please note how Specs uses! ("bang") as the operator that links Specs's descriptions with ScalaCheck property check logic. This is specific to Specs and is part of its own domain specific language for writing test specifications.

It is possible to run Specs2 specifications from the command line but we have to use Specs2's own runner class (we can no longer use the check method as that's not part of Specs2's *Specification* or *ScalaCheck* classes):

```
specs2.run(SimplePropertySpec)
```

Please note that if running under Windows, the console will display some gibberish where it should be displaying colours, as the Windows console is not compatible with Specs2's ANSI colors:

```
SimplePropertySpec
Strings should
String.concat property

Total for specification SimplePropertySpec
Finished in 34 ms
example, 100 expectations, 0 failure, 0 error

res11: Either[org.specs2.reporter.Reporter,Unit] = Right(())
```

As per Scala's unwritten convention, returning a *Right* object means success; in case of error, the output of the execution would be a *Left* object with an error message. Please refer to Scala's Either type for more details on Either, Left and Right.

ScalaCheck-style properties can be mixed with Specs2's usual style of writing expectations:

```
import org.specs2.mutable._
import org.specs2.ScalaCheck
object SimpleMixedPropertySpec extends Specification with ScalaCheck {
  "String" should {
    "String.concat property" ! check { (a: String, b: String) =>
      a.concat(b) == a + b
    }
    "reverse non-empty strings" ! check {(a:String) =>
      (a.length > 0) ==> (a.charAt(a.length-1) == a.reverse.charAt(0))
    }
  }
}
```

ScalaCheck features such as conditional properties work just fine when integrated with Specs2. Contrast this with the integration with ScalaTest below.

Generators and arbitrary generators can also be used from Specs2 expectations just like they would be used (and created) when using "pure" ScalaCheck test cases. The Specs2 cases only need to import the implicit arbitrary object and use the check method just like before:

```
import org.specs2.mutable._
import org.specs2.ScalaCheck
import org.scalacheck.{Arbitrary, Properties, Gen}

case class Rectangle(val width:Double, val height:Double) {
  lazy val area =  width * height
  lazy val perimeter = (2*width) + (2*height)
  def biggerThan(r:Rectangle) = (area > r.area)
}

object RectangleGenerator {
  // same generator for the Rectangle case class as before
  val arbRectangleGen: Gen[Rectangle] = for {
    height<- Gen.choose(0,9999)
    width<- Gen.choose(0,9999)
  } yield(Rectangle(width, height))
  implicit val arbRectangle: Arbitrary[Rectangle] = Arbitrary(arbRectangleGen)
}

object ArbitraryRectangleSpec extends Specification with ScalaCheck {
  import RectangleGenerator._

  "Rectangle" should {
    "correctly calculate its area" ! check { (r: Rectangle) =>
      r.area == r.width * r.height
    }
  }
}
```

Running this code with *specs2.run(ArbitraryRectangleSpec)* should report success.

7.3 Using ScalaCheck with ScalaTest

ScalaTest, a popular Scala unit testing and behavior-driven design framework, offers integration with ScalaCheck for writing property style unit tests. The examples discussed below are located in folder **scalacheck-integration-scalatest**.

In order to run the snippets below, switch to the scalacheck-integration-scalatest folder and run the following command with multiple targets: "*sbt update compile console*".

ScalaTest provides support for ScalaCheck-style specifications by mixing either the PropertySpec trait or the Checkers trait. The former allows us to write properties in ScalaTest style, while the latter allows us to do it in ScalaCheck style.

The advantage of using ScalaTest together with ScalaCheck with the PropertySpec is that ScalaTest's matchers (MustMatchers and ShouldMatchers) can be used to validate if a property holds, which provides additional readability to the code.

The following is an example of a very simple property check written in ScalaTest using the ScalaTest style:

```
import org.scalatest.PropSpec
import org.scalatest.prop.{PropertyChecks, Checkers}
import org.scalatest.matchers.ShouldMatchers
import org.scalacheck.Prop._

class SimplePropertySpec extends PropSpec with PropertyChecks with ShouldMatchers {
  property("String should append each other with the concat method") {
    forAll { (a: String, b: String) =>
      a.concat(b) should be (a + b)
    }
  }
}
```

In order to run ScalaTest specifications from the Scala console, we can use the execute() method (with nocolor=false in Windows to avoid the pesky ANSI characters for color):

```
(new SimplePropertySpec).execute(color=false)
```

The output should be something like this:

```
$read$$iw$$iw$$iw$$iw$SimplePropertySpec:
- String should append each other with the concat method
```

The first line in the output looks very cryptic but it's because of the dynamic nature of the Scala console; it will not appear like that in normal executions (e.g. when using SBT's test action).

In the code above, *property* is ScalaTest's own way to define property checks, provided by the *PropSpec* trait. Please note that even though it shares the name with ScalaCheck's Properties.property method, it is **not** the same method.

Within the property block we find the *forAll* method (please note that this is also ScalaTest's own *forAll* method, and not ScalaCheck's). Within the *forAll* block we can define the logic of our property check implemented as usual as a function. As part of the property checks, the "should be" matcher has been used, which is provided by the *ShouldMatchers* trait (*MustMatchers* can also be used instead). Being able to use ScalaTest's matchers can make property checks more readable (even though this is purely a matter of taste, as it will make our property checks fully dependent on ScalaTest).

One important difference when using *PropertyChecks* is that some features from ScalaCheck are not available, such as the ==> function for implementing conditional properties; this method is replaced by the *whenever* function, as follows:

```
import org.scalatest.prop.{PropertyChecks, Checkers}
import org.scalatest.matchers.ShouldMatchers
import org.scalacheck.Prop._

class ReversePropertySpec extends PropSpec with PropertyChecks with ShouldMatchers {
  property("Reverse non-empty strings correctly") {
    forAll { (a: String) =>
      whenever(a.length > 0) {
        a.charAt(a.length-1) should be (a.reverse.charAt(0))
      }
    }
  }
}
```

When using the Checkers trait, the *check* method should be used to specific property checks. This method takes the output of ScalaCheck's *forAll* function, containing the definition of a property check. The following example is the same property shown above, but now written using the ScalaCheck style with the *Checkers* trait:

```
import org.scalatest.prop.{PropertyChecks, Checkers}
import org.scalatest.matchers.ShouldMatchers
import org.scalacheck.Prop._

class SimplePropertyCheckersSpec extends PropSpec with Checkers {
  property("Reverse non-empty strings correctly") {
    check(forAll { (a: String) =>
      (a.length > 0) ==> (a.charAt(a.length-1) == (a.reverse.charAt(0)))
    })
  }
}
```

For developers more familiar with the terse ScalaCheck style, this approach might be more suitable.

Lastly, ScalaTest-ScalaCheck integration also offers the possibility of using generators and Arbitrary objects:

```
import org.scalatest.prop.{PropertyChecks, Checkers}
import org.scalatest.matchers.ShouldMatchers
import org.scalacheck.Prop._

class ArbitraryRectangleWithCheckersSpec extends PropSpec with Checkers {
  import com.company.scalacheck.RectangleGenerator._
  import com.company.scalacheck.Rectangle

  property("A rectangle should correctly calculate its area") {
    check(forAll { (r: Rectangle) =>
      r.area == (r.width * r.height)
    })
  }
  property("A rectangle should be able to identify which rectangle is bigger") {
    check(forAll { (r1: Rectangle, r2: Rectangle) =>
      (r1 biggerThan r2) == (r1.area > r2.area)
    })
  }
}
```

The code of the generator has been omitted for brevity reasons since it's the exact same code of the Rectangle class used in all the examples so far (when running this example in the Scala console, the code for the Rectangle class and its generator is being imported at the top of the specification class).

When using the Checkers style, the code shown above is not very different from pure ScalaCheck code except for the *check* method.

7.4 Using ScalaCheck with JUnit

Integration between ScalaCheck and JUnit could be desirable when an existing large Java code base with JUnit tests exists and the project would like to introduce ScalaCheck and some Scala code to improve the quality of existing test code (leveraging the random data generation features of ScalaCheck).

Combined use of JUnit and ScalaCheck can be achieved in several different ways, in increasing degree of integration:

- Call ScalaCheck properties and evaluate them using JUnit's *assertTrue*

- Create our own JUnit runner for "plain" ScalaCheck code; as opposed to ScalaTest and Specs, ScalaCheck does not provide a JUnit runner class even though a runner would provide the highest level of integration with JUnit. Fortunately, creating a custom JUnit runner is a fairly straightforward process.

The snippets below can be run from within the Scala console, using "*maven compile test-compile scala:console*" from the command line after switching to the **scalacheck-integration-junit** folder. Maven works on Linux, OS X and Windows, but please note that these instructions require that the "mvn" command is your PATH, which may require some configuration in your operating system.

7.4.1 Using JUnit's assertTrue

The easiest way to integrate ScalaCheck with JUnit is to write a JUnit test suite as a Scala class in JUnit's own style, write the ScalaCheck properties as usual and then create one or multiple JUnit tests that ensure that the property holds true using JUnit's *assertTrue*.

Implementing the integration using this approach requires some familiarity with the ScalaCheck API and class structure.

Instead of using the *Properties.check* method to execute the property check, we need to execute the property check but collect the Boolean output (passed/not passed) so that we can provide it to JUnit's assertTrue. For that purpose, ScalaCheck's *check* method in the *Test* class will be used to run a given property check. The output of Test.check is of type Test.Result, and the boolean attribute Result.passed can be checked after executing the property. This is the value that will be provided to assertTrue:

```
import org.junit.Assert._
import org.junit.Test
import org.scalacheck.Test.{Params=>TestParams}
import org.scalacheck.{ConsoleReporter, Prop, Test => SchkTest}
import org.scalacheck.Prop._

class ScalaJUnitSimpleTest {
  val validProperty = Prop.forAll { (a: String) =>
    (a.length > 0) ==> (a + a == a.concat(a))
  }

  @Test def testConcat = {
    assertTrue(SchkTest.check(TestParams(testCallback = ConsoleReporter()),
validProperty).passed)
  }
}
```

There is some boilerplate required to call the *Test.check* method (here renamed as *SchkTest.check* so that it doesn't collide with JUnit's *@Test* annotation), but it is the same for all calls to the same logic can be easily abstracted into a separate reusable method.

In order to run this code from the Scala console, the testConcat method can be called directly:

```
(new ScalaJUnitSimpleTest).testConcat
```

assertTrue does not generate any output when execution is successful , but it will throw an exception in case of failure. When these unit test suites are run from Maven or a Java development environment with a runner, the output will be slightly different.

The approach described so far is pretty straightforward for small amounts of property checks. If the number of properties is larger than a handful, there will be as many calls to assertTrue as ScalaCheck properties, which makes this approach fairly verbose.

7.4.2 Using a custom JUnit runner

Creating a custom JUnit runner is the alternative that provides the highest degree of integration between ScalaCheck and JUnit, as each one of the property checks will be transparently handled by JUnit as a separate unit test case.

When using a custom runner, the classes to be run with JUnit need to be annotated with JUnit's @RunWith annotation, with our own JUnit-ScalaCheck runner class as a parameter. This annotation will be the only difference between these classes and plain ScalaCheck Properties classes:

```
import org.scalacheck.Prop._
import org.junit.runner.RunWith
import com.company.scalacheck.support.ScalaCheckJUnitPropertiesRunner
import org.scalacheck.{Gen, Properties}

@RunWith(classOf[ScalaCheckJUnitPropertiesRunner])
class ScalaCheckRunnerTest extends Properties("Rectangle property suite") {
  property("Failed test") = forAll {(a: Int) =>
    a == 1
  }

  property("Test with collection of data") = forAll {(a: Int) =>
    (a > 0 && a <= 10) ==> collect(a) {
      2 * a == a + a
    }
  }
}
```

In the example above, the class is annotated with *com.company.scalacheck.support.ScalaCheckJUnitPropertiesRunner*, which is the custom class that takes care of providing the JUnit integration features. Describing the implementation of our custom JUnit runner class is outside of the scope of this document, but the source code is available in the source code package and is ready to be used. Additionally, there is an official version of the ScalaCheck JUnit 4 runner maintained separately, which is currently stored in the scalacheck-contrib repository in Github: https://github.com/oscarrenalias/scalacheck-contrib.

Please note that it is not easily possible to run the code above in the Scala console with the custom JUnit runner, because we would need the support of the Maven JUnit plugin to set everything up for us; it is much easier to use the traditional *Properties.check* method, which works regardless the property is annotated with @RunWith or not. In order to run the tests use command "*mvn test*" from a command window.

8 REAL WORLD SCENARIO: TESTING A HADOOP MAP-REDUCE JOB WITH SCALACHECK

8.1 Hadoop

For those not familiar with Hadoop, this scenario may require a basic introduction.

Hadoop is built around the concepts of mappers and reducers. Following are the definitions from the original MapReduce paper *(J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS, pages 137–150, 2004.)*:

> *"The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: map and reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function.*
>
> *The reduce function, also written by the user, accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory."*

Hadoop is a MapReduce library. As a Java library, Hadoop represents map functions as Mapper classes and reduce functions as Reducer classes. Naturally, in a distributed environment there will be many mapper instances and many reducer instances running as concurrent tasks. Hadoop ensures that each mapper task receives a manageable chunk of the input data, and will dynamically decide how many reducer tasks are needed and which data to feed into each one of them. When the reducer tasks have run, Hadoop collects their output and makes it available in the file system.

Between the map and the reduce computation phases, Hadoop will group and sort all the keys and values from the mapper tasks, so that the input for a reducer task is a single key and a list of values that mapper tasks generated for that key.

Regarding input and output, Hadoop can read any kind of file format from the file system (either Hadoop's own Hadoop File System – HFS, or the local file system when testing) but generally the input is based on text files; mappers usually read one line at a time from a file and produce keys and values based on specific logic. Reducers process this data and write it back to temporary output, which is consolidated by Hadoop at the end of the job.

8.2 The scenario

This scenario describes how to use ScalaCheck to test a Hadoop MapReduce job which is written in Java.

In this scenario we'll create a couple of property checks to verify the correctness of the map and reduce functions used in Hadoop's WordCount example (http://wiki.apache.org/hadoop/WordCount). WordCount is a job that is used to count the number of times that a word appears in the file(s) provided to the job. WordCount is a simple job, but the concepts for creating suitable ScalaCheck property checks remain applicable in more complex Hadoop scenarios.

ScalaCheck is ideal for these use cases as usually both mappers and reducers in a Hadoop job will work on arbitrary data, and ScalaCheck's random data generation features can be used to their full extent to generate a suitable range of inputs for the jobs.

8.2.1 WordCount mapper

In WordCount, the input for the mapper is, as previously described, a key-value pair; the key is the position of the line within the file (offset in bytes), while the value is a line of text.

This is the code for the mapper:

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, one);
    }
  }
}
```

In WordCount, the mapper splits incoming lines so that the output keys are the words from the line while the output values are always the value "1" (each word instance counts as one). The mapper ignores the input key because in this case, it is not needed for our logic.

8.2.2 WordCount reducer

Later on the reducer will simply sump up all the "1" values for a given word (key), to determine how many times a particular word appeared in the text. Since Hadoop will group all the same keys into one single key with its values being a list of occurrences (a list of number 1s), all the mapper needs to do is add up the list and generate as its output, the same word as the key and the total sum value as the value:

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
      if(key.getLength() == 0)      // do nothing for empty keys
        return;

      int sum = 0;
      for (IntWritable value : values)  // otherwise sump up the values
```

```
        sum += value.get();
      context.write(key, new IntWritable(sum));
  }
}
```

8.2.3 Input and output

The input for the job will be taken from a text file (or files) stored in the local filesystem. The contents of the file(s) will be plain text.

The output of the job will be another text file where each line will contain a word from the source file as well as the number of times that the same word occurred throughout the text.

8.2.4 Other requirements

In this particular example, we'll use Cloudera's MrUnit library to help us take care of the internal Hadoop plumbing for unit testing (initialization of input and output data types, initialization of contexts, job execution, etc.). The correct library version and configuration is provided in the Maven project needed to run these test cases.

8.3 Defining the test cases

In this scenario, the following test cases will be executed:

1. Ensure that the mapper can handle lines with one single word. The input for this property check will be a key (with type LongWritable) and single word (with type Text). Only basic generators will be required for this check.

2. Ensure that the mapper can handle lines with an arbitrary number of words. The input for this check will be a key (with type LongWritable) and a tuple containing a line of text with the pre-calculated number of words in that line. A more complex custom generator will be required to pre-calculate the number of words.

3. Verify that the reducer can correctly process any input. The input for this check will be a key (with type Text) and a pair with a list of values representing the number of appearances of the given word in the input file given to the Hadoop job, as well as the pre-calculated sum of those values (which is the same value that the reducer should generate). As in the previous test case, a more complex custom generator will be required to pre-calculate the expected output values from the mapper.

The splitting of input files into key-value (offset-line) pairs precedes the map phase of processing and is outside the scope of our tests below. Similarly, the generation of the output file following the reduce phase is outside the scope of our tests.

8.4 Building the generators

In order to build property checks for the scenario above, we're going to need some specific data generators for the Hadoop data types as, unfortunately for us, Hadoop internally only works with its own wrapper types that are optimized for size when serialized.

The code snippets presented below can be run from within the Scala console, but must be added in the same sequence as presented in this document. In order to start the Scala console, switch to the **java-hadoop-scalacheck** folder and run "*mvn scala:console*".

8.4.1 Building the basic generators

We'll start by building some basic generators to create random strings, words, blank spaces (random amounts of them), and then we'll put those in lines, lists and "bags" (please see below). These generators are generic and do not depend on Hadoop types.

All these generators are in the GenText object.

First, generators to create random strings either from a list of characters, or from a generator of random characters:

```
import org.scalacheck.Gen
import org.scalacheck.Gen._

def genString(genc: Gen[Char]): Gen[String] =  for {
    lst <- Gen.listOf(genc)
  } yield {
    lst.foldLeft(new StringBuilder)(_+=_).toString()
  }

def genString(chars: Seq[Char]): Gen[String] = genString(Gen.oneOf(chars))
```

The first generator generates a randomly long list of characters, as selected from the random character generator. The random character generator can be initialized with an arbitrary and unlimited number of characters. The list of character is converted to a string using StringBuilder.

The second generator builds on the first one, and the list of characters can be provided ad-hoc rather than through another character generator.

The next set of generators used the one we have just defined to create strings that satisfy the property that are never empty:

```
def genNonemptyString(genc: Gen[Char]): Gen[String] =
   genc.combine(genString(genc))((x,y) => Some(x.get + y.get))

def genNonemptyString(chars: Seq[Char]): Gen[String] =
  genNonemptyString(Gen.oneOf(chars))
```

Just like before, the first generator contains the generation logic; it uses another random character generator to create the characters for the non-empty string while the second generator allows providing a pre-defined static list of characters that will be used as part of the string.

```
implicit def seqToChar(coll: Seq[Int]): Seq[Char] = for {
  c <- coll
} yield(c.toChar)

val spaceChars: Seq[Char] = Vector(9, 32)
val nonWhitespaceChars: Seq[Char] = (33 to 126)

val genWord = genNonemptyString(nonWhitespaceChars)
val genWhitespace = genNonemptyString(spaceChars)
```

These generators build non-empty words, or strings which contain characters considered blank spaces (in the ASCII range 9 to 32). They are used by other generators and property checks below.

```
val genLineWithList = for {
  lst <- Gen.listOf(genWord)
} yield (lst.foldLeft(new StringBuilder)(_++=_ + genWhitespace.sample.get).toString.trim(),
lst)

val genLineWithBag = for {
  lst <- Gen.listOf(genWord)
} yield (lst.foldLeft(new StringBuilder)(_++=_ + genWhitespace.sample.get).toString.trim(),
bagOf(lst))
```

Both generators above generate tuples, where the first item is a line of text and the second is either the list or "bag" of the words in the line.

Finally, we can create some arbitrary generators for later usage:

```
val arbGenWord = Arbitrary(genWord)
var arbGenLineWithList = Arbitrary(genLineWithList)
val arbGenLineWithBag = Arbitrary(genLineWithBag)
```

Now we can move onto creating generators of Hadoop-specific data.

8.4.2 Hadoop generators

The following are all generators specific for Hadoop data types:

```
import org.apache.hadoop.io.{Text, LongWritable, IntWritable}

def genLongWritable(upperRange:Int) = for {
  num <- Gen.choose(0, upperRange)
} yield(new LongWritable(num))

val genIntWritable = for {
  num <- Gen.choose(0, 9999)
} yield(new IntWritable(num))

val genIntWritableList = Gen.listOf(genIntWritable)
def genLongWritableList(upperRange:Int) = Gen.listOf(genLongWritable(upperRange))
```

Based on the knowledge acquired so far, the first four generators are rather straightforward: generators for single values of IntWritable and LongWritable, and generators of lists of those two types using Gen.listOf.

All of Hadoop's own types are simple wrappers on Java's basic types so can use the default generators in the Gen object to generate random values of the basic type, and then wrap those values using Hadoop's own type. In the case of the *LongWritable* generator, we're even parameterizing it by allowing to specify and upper value range for the generated values (may not always be needed, but it was included here for the sake of showing a parameterized generator)

The last generator is used to create a tuple where the first element is a list of IntWritable objects (created with the previous generator) and the second is the sum of all the values in the first list. This will be required in a specific test scenario later on:

```
import com.company.hadoop.tests.HadoopImplicits._
val genIntWritableListWithTotal: Gen[(List[IntWritable], Int)] = for {
   l <- genIntWritableList
} yield((l, l.foldLeft(0)((total, x) => x + total)))
```

In this case we require the support of the HadoopImplicits object, that contains a handful of useful implicit conversions between Hadoop's types and the basic Scala data types (e.g. IntWritable converted to and from Integer, LongWritable to Long, and so on). These implicit conversions are more heavily used in the actual property checks, and there's some more detailed information about them below.

8.4.3 Testing the Generators

We've built quite a few custom generators so that we can build property checks for our selected scenarios. But how do we know that our generators are generating the data that they say will generate? In other words, how can we be sure that they generate valid data?

To ensure the correctness of our generators we're going to build a few simple ScalaCheck property checks for generators; property checks that check custom generators used in other property checks:

```
object GeneratorSpecification extends Properties("Generator tests") {
  import GenText._
  import HadoopGenerators._

  property("genNonemptyString property") = forAll(genNonemptyString(nonWhitespaceChars))
{(s:String) =>
    s != "" &&
    s.forall(_ != ' ')
  }

  property("genWord property") = forAll(genWord) { (w:String) =>
    w != "" &&
    w.forall(_ != ' ')
  }

  property("genLineWithList property") = forAll(genLineWithList) { case (line, list) =>
    list.forall(w => w.forall(_ != ' '))  &&
    list.filterNot(w => line.indexOf(w) > 0).size == 0
  }

  property("genIntWritableListWithTotal property") = forAll(genIntWritableListWithTotal)  {
    case(list, total) =>
      list.foldLeft(0)((x, sum) => x + sum) == total
  }
}
```

The property checks consist of simple logic that validates that the data that they produce is according to their definition, e.g. words generated by *genWord* are always non-empty, that the list of words produced by *genLineWithList* does not contain more words than the line of text, and so on.

We've created more generators than the four tested above, but they are not tested because they are either used as part of the ones tested above (*genString*, *genWhitespace*), or they are so basic that they do not need testing (*genIntWritable*, *genIntWritableList*, etc).

8.5 Building the property checks

Now that we've got all the generators that we're going to need and we've ensured that the key ones generate correct data, we can proceed to build the actual property checks. Since the generators are doing most of the work now by pre-calculating the correct expected values, the logic in the property checks will mostly consists of running the mapper/reducer and then comparing their output values with the expected values.

8.5.1 Some implicits to make our life easier

Implicit conversions are a very useful feature in the Scala language toolset, that provide Scala with a dynamic language flavor even though it's statically typed, by providing transparent conversion between different types through implicit functions/methods.

In our example, implicits are very handy because they allow our code to transparently use Scala native types in place of Hadoop's own wrapper types, e.g. use Int instead of IntWritable in a Hadoop method call. This simplifies our code because we do not have to write all those conversions manually.

For our scenarios we have defined two-way conversions between Int and IntWritable, Long and LongWritable, Text and String:

```
object HadoopImplicits {
  import org.apache.hadoop.mrunit.types.Pair
  implicit def IntWritable2Int(x:IntWritable) = x.get
  implicit def Int2WritableInt(x:Int) = new IntWritable(x)
  implicit def LongWritable2Long(x:LongWritable) = x.get
  implicit def Long2LongWritable(x:Long) = new LongWritable(x)
  implicit def Text2String(x:Text) = x.toString
  implicit def String2Text(x:String) = new Text(x)
  implicit def Pair2Tuple[U,T](p:Pair[U,T]):Tuple2[U,T] = (p.getFirst, p.getSecond)
}
```

Hadoop has a few more wrapper types but only the ones above are required here.

The last implicit conversion between the Pair type in MrReduce and Scala's built-in tuple type is used when dealing with the results of the MapDriver and ReduceDriver classes, used later on for unit testing mappers and reducers.

8.5.2 Handling of single words by the mapper

The first property check consists of ensuring that the mapper can handle lines with one word only; the expected output is a single pair where the key is the given word (which was provided as a parameter as the key) and the value is 1 (represented here by the *one* static object):

```
import org.scalacheck.Properties

object WordCountMapperSingleWordProperty extends Properties("Mapper property") {
  import org.scalacheck.Prop._
  import com.company.hadoop.tests.GenText._
  import com.company.hadoop.tests.HadoopGenerators._
  import com.company.hadoop.WordCount._
  import scala.collection.JavaConversions._
  import org.apache.hadoop.mrunit.mapreduce.MapDriver
  import com.company.hadoop.tests.HadoopImplicits._
  import org.apache.hadoop.io.{IntWritable, LongWritable}

  val mapper = new Map
  val one = new IntWritable(1)

  property("The mapper correctly maps single words") = {
    forAll(genLongWritable(99999), genWord) {(key:LongWritable, value:String) =>

    val driver = new MapDriver(mapper)
    val results = driver.withInput(key, value).run

    results.headOption.map(pair =>
      pair._1.toString == value && pair._2 == one).get
    }
  }
}
```

First of all, this property requires a handful of import statements to make sure that we've got all needed classes in the scope. This includes our own class with implicit conversions (since we'll be transparently converting between Scala types and Hadoop wrapper types) as well as Scala's *scala.collection.JavaConversions*, imported into the scope so that we can easily convert to and from Scala and Java lists and types (makes it easier to deal with Java's Array and Iterable types as they're transparently converted to their Scala equivalent)

MapDriver is part of the MrUnit framework and takes care of setting up Hadoop's internal structures that allow us to run map and reduce jobs for testing purposes. The job is actually run when calling the *run* method in the *driver* object, after providing some test input data key and input value.

In this example, the test key is coming from the *longWritableGen* generator that simply generates a random *LongWritable* object representing an offset from an input file just like Hadoop would do. The value is a random word generated by the *genWord* generator.

The output of the mapper will be a list (a Java Iterable, actually) of *Pair* objects. *Pair* objects are produced by MrUnit and are not part of the Hadoop data types. The validation will consist of getting the first one from the list (there should only be one, as there was only one word as the input), and we do that by using *Iterable.headOption*, which will return the first element wrapped in a Scala *Some* (as an *Option* type).

Once we've got that, we can use *Option.map* to execute the validation without extracting the actual value from the *Some* wrapper in a very functional way. There's an implicit conversion between the *Pair* type and Scala's *Tuple* type (defined in class *com.company.hadoop.tests. HadoopImplicits*), and that allows us to access the contents of the pair variable using the _1 and _2 notation as with tuples. In this case, pair._1 refers to the key produced by the mapper (which should be the same word that was provided as the mapper's input as the value) and the value, accessed as pair._2, should be 1. If the comparison is fulfilled, then the property check is validated.

8.5.3 Mapping longer lines

The next property check extends the first one to ensure that the mapper can process lines of with arbitrary numbers of words, including empty lines (where the key is still present but the value is an empty line).

In this property check the *LongWritable* is again provided by the *genLongWritable* generator while the second parameter is a tuple where the first element is a line of text and the second value is the list of words in the line, split in the same way as the mapper should do it. Please note that the line of text may contain arbitrary numbers of blank spaces between words.

The generator responsible for this is *genLineWithList*:

```
import org.scalacheck.Properties

object WordCountMapperLinesProperty extends Properties("Mapping lines of text") {
  import com.company.hadoop.WordCount._
  import scala.collection.JavaConversions._
  import org.apache.hadoop.mrunit.mapreduce.MapDriver
  import org.scalacheck.Prop._
  import com.company.hadoop.tests.GenText._
  import com.company.hadoop.tests.HadoopGenerators._
  import com.company.hadoop.tests.HadoopImplicits._
  import org.apache.hadoop.io.{IntWritable, LongWritable}

  val mapper = new Map
  val one = new IntWritable(1)

  property("The mapper correctly maps lines with multiple words") =
    forAll(genLongWritable(99999), genLineWithList) {
      case (key, (line, list)) =>
        val driver = new MapDriver(mapper)
        val results = driver.withInput(key, line).run

        (results.forall(one == _._2) && results.map(_._1.toString).sameElements(list))
    }
}
```

After setting up the MrUnit driver, the property check consists of two checks: ensuring that all words (keys) produced by the mapper have a value of 1, and that the mapper produced as many keys as indicated by the generator (which by itself already knew how many words were there in the string).

The first check is rather straightforward and implemented by processing all *pairs* in the *results* iterable using *results.forall* (returns true if all items in the iterable match the given Boolean check),

For the second check, method *sameElements* in Scala's List type comes very handy as it checks if the given list has the exact same elements as the current list, and in the same order; here were are assuming that the mapper will generate its results in the same sequence as the words appeared in the line, which is a valid assumption.

Please note that before comparing the two lists, we have to convert the values in the list with results that was generated by the mapper from Hadoop's *Text* type to *String* (essentially, the list variable is of type List[Text], not List[String]). Comparing a Text object with a String object, even if the content is the same, will always yield Boolean false. We do the conversion using the *map* method in the List object, to generate another list of Strings.

8.5.4 Reducing data

The last property check is focused on the reducer, and is used to ensure that given a word (key) a number representing the number of times it appears in the input string, the reducer is able to correctly produce an output where the keyis the same word but the value is the sum of the input values.

The *textGen* generator is used to generate a random word, while *intWritableListWithSum* is a generator that produces a tuple where one element is the list of occurrences for the word while the second element is the sum of the values that the reducer is expected to produce.

```
import org.scalacheck.Properties
```

```
object WordCountReducerCheck extends Properties("Reducer spec") {
import com.company.hadoop.WordCount._
  import scala.collection.JavaConversions._
  import org.apache.hadoop.mrunit.mapreduce.ReduceDriver
  import org.scalacheck.Prop._
  import com.company.hadoop.tests.GenText._
  import com.company.hadoop.tests.HadoopGenerators._
  import com.company.hadoop.tests.HadoopImplicits._

  val reducer = new Reduce

  property("The reducer correctly aggregates data") =
    forAll(genWord, genIntWritableListWithTotal) {
      case(key, (list, total)) => {
        val driver = new ReduceDriver(reducer)
        val results = driver.withInput(key, list).run

        results.headOption.map(_._2.get == total).get == true
     }
  }
}
```

After setting up MrData and running the job, the results will be provided as an iterable object where there should only be one element; reducers are only provided one key at a time and we know that the reducer in our sample code should only generate one output key and value.

The pattern is the same as before: obtain the only element using *Iterable.headOption* and then use the *Option.map* method to process the wrapped value to make sure that the *IntWritable* value representing the sum of the input values is the same that as pre-calculated by the generator. If it is, then the property check holds true.

8.6 Putting it all together

With the property checks in place, the only missing thing is to implement that calls the properties' *check* method so that we can execute the tests. We could also integrate the code with the ScalaCheck JUnit runner but since we've only got three scenarios, we can do it manually this time.

When using the sample code provided with this deliverable, the code can now be easily run with maven:

```
mvn scala:run -Dlauncher=test
```

And it should generate the following output:

```
+ Mapper and reducer tests.The mapper correctly maps single words: OK, passed 100 tests.
+ Mapper and reducer tests.The mapper correctly maps lines with multiple words: OK, passed
100 tests.
+ Mapper and reducer tests.The reducer correctly aggregates data: OK, passed 100 tests.
```

Please note that the file that implements these property checks in the sample code package does it by grouping all three properties in the same Properties class, instead of having three separate Properties classes. In any case, the final outcome is the same.

9 APPENDIX 1: REFERENCES

9.1 Sample code

All the code used in this document is currently available from Github: https://github.com/oscarrenalias/scalacheck-examples.

The code is structured in folders based on the same structure used throughout this document. Examples where Java and Scala code are mixed (scalacheck-basic, java-scalacheck, scalacheck-integration-junit) require Maven 2.x or 3.x to build and run. Pure Scala examples (scalacheck-integration-specs, scalacheck-integration-scalatest) require Simple Build Tool version 0.11 or greater.

There is no top-level project at the root folder above the sub-folders, so each project is an independent unit by itself.

Folder	Contents	How to run
scalacheck-basic	Basic ScalaCheck examples, showing most of its basic features (basic property checks, data grouping, conditional properties)	Switch to the project's subfolder and run: `mvn scala:run -Dlauncher=test`
java-scalacheck	Integration of ScalaCheck property checks with Java code	Switch to the project's subfolder and run: `mvn scala:run -Dlauncher=test`
scalacheck-integration-scalatest	Examples of integration of ScalaTest with with ScalaCheck. Requires SBT.	Switch to the project's subfolder and run: `sbt reload test`
scalacheck-integration-specs	Examples of integration of Specs with ScalaCheck. Requires SBT.	Switch to the project's subfolder and run: `sbt reload test`
scalacheck-integration-junit	Examples of integration of ScalaCheck with JUnit, as well as JUnit support traits and a JUnit 4 runner for ScalaCheck tests written as Properties classes. It uses the Maven JUnit test runner (Surefire plugin) to run.	Switch to the project's subfolder and run: `mvn test`
java-hadoop-scalacheck	Code and ScalaCheck test cases that test Hadoop's WordCount example	Switch to the project's subfolder and run: `mvn scala:run -Dlauncher=test`

9.2 Links

The following are some useful links and references on ScalaCheck:

- ScalaCheck wiki documentation: http://scalacheck.googlecode.com/svn/artifacts/1.9/doc/api/index.html
- ScalaCheck JAR files for download: http://code.google.com/p/scalacheck/downloads/list
- ScalaCheck source code: https://github.com/rickynils/scalacheck
- ScalaCheck API documentation (Scaladoc): http://scalacheck.googlecode.com/svn/artifacts/1.9/doc/api/index.html#package
- Better unit tests with ScalaCheck (and specs): http://etorreborre.blogspot.com/2008/01/better-unit-tests-with-scalacheck-and.html
- ScalaCheck generators for JSON: http://etorreborre.blogspot.com/2011/02/scalacheck-generator-for-json.html
- Specs2 user guide: http://etorreborre.github.com/specs2/guide/org.specs2.UserGuide.html
- Specs2 ScalaCheck integration example: https://github.com/etorreborre/specs2/blob/1.5/src/test/scala/org/specs2/examples/ScalaCheckExamplesSpec.scala
- ScalaTest: http://www.scalatest.org/
- JUnit: http://www.junit.org/

10 APPENDIX 2: DEPLOYING SCALACHECK

10.1 Using JAR files

The easiest way to get started with ScalaCheck or to get ScalaCheck incorporated into an existing code base is to deploy the required JAR file. If using a build tool such as Ant, pre-compiled JAR files for different versions of ScalaCheck are available from the Downloads section in the project's page at Google Code (link). Only one JAR file is required, as ScalaCheck has no external dependencies.

10.2 Maven

We will assume that Maven 2.x or 3.x is installed and that a suitable pom.xml file has already been created, and that it has been configured to use the Scala compiler plugin to compile Scala code in the project. If not, please refer to the Scala Maven documentation or use one of the existing Maven archetype templates for Scala development to get the project started.

If not already present, the following repository needs to be added to our project's pom.xml file in the repositories section:

```
<repository>
  <id>scala-tools.org</id>
  <name>Scala-Tools Maven2 Repository</name>
  <url>http://scala-tools.org/repo-releases</url>
</repository>
```

Then, the following dependency needs to be declared:

```
<dependency>
	<groupId>org.scala-tools.testing</groupId>
	<artifactId>scalacheck_2.9.0-1</artifactId>
	<version>1.9</version>
</dependency>
```

Bear in mind that ScalaCheck is cross-compiled for multiple releases of Scala, so it is important to know which version of Scala we are currently using, as it is defined as part of the dependency name (2.9.0-1 in the example above).

10.3 Simple Build Tool (0.11.x)

SBT is the preferred tool for Scala development for "pure" Scala projects. The instructions below assume that SBT 0.11.x has been installed, that the sbt startup script is in the path and that a suitable SBT project exists in the current folder. Please note that these instructions are not applicable when using SBT 0.7.x, as there have been extensive changes in the way that SBT handles project definitions starting with version 0.10.

To get ScalaCheck as a dependency in our SBT project the following lines of code need to be added to our project's *build.sbt* file:

```
libraryDependencies += "org.scala-tools.testing" %% "scalacheck" % "1.9" % "test"
```

Get into the SBT console by running "*sbt*" in your project folder, and then run the "*reload*" and "*update*" commands to get SBT to refresh the project file (in case SBT did not do it automatically) and to update its dependencies.

When defining a new dependency in *build.sbt*, please note that there's no need to specify the current version of Scala in use as part of the ScalaCheck dependency, since by using "%%" SBT will figure it out automatically for us.

11 APPENDIX 3: COMMON PROBLEM SCENARIOS

The following are some common issues that may appear during the development and execution of ScalaCheck properties.

11.1 Property check function does not evaluate to true or false

This error will manifest during compile time with the following error message:

```
[ERROR] /.../src/test/scala/TestFile.scala:27: error: No implicit view available from Unit
=>org.scalacheck.Prop.
```

The actual type causing the error may differ (it could be Long, String, etc =>org.scalacheck.Prop.), but what the error message indicates is that our property check is not evaluating in terms of Boolean logic. Property checks must always return true or false and, if they do not comply with that requirement, the compiler will emit this error. In the specific example used above, the function that was being tested produced no result, hence Unit (Scala's equivalent of *void*) was being shown as the type.

11.2 Current Scala version and ScalaCheck's version do not match

If in our preferred build tool we have mismatched the version of ScalaCheck with our version of Scala, the compiler may generate strange and unrelated error messages. Especially when using Maven for building, it is important that we provide the correct name for the ScalaCheck dependency including the version of Scala. This is not an issue when using SBT as it is capable of figuring out the right package version.

11.3 The condition for the property is too strict

During property execution, if we have defined a condition for the execution of our property check with the ==> operator and the condition forces ScalaCheck to discard too much data, it will eventually give up and mark the property check as incomplete.

Such cases will be visible in the console as part of ScalaCheck's output:

```
! Gave up after only 42 passed tests. 500 tests were discarded.
```

In these scenarios, the only possibility is to redesign our property check so that we can use a less strict condition, or create a tailored generator that will only provide the data that we really need.

REVISION HISTORY

Date	Author	Remark
V0.1, 13.09.2011	Oscar Renalias	First version, ready for review
V0.2, 14.09.2011	Oscar Renalias	Incorporated feedback from the first review.
v0.3, 12.10.2011	Oscar Renalias	Added some content about use cases for ScalaCheck and the scenario describing integration between ScalaCheck and a Hadoop MR job
v0.4, 07.11.2011	Oscar Renalias	Updates as part of the review of version 0.3.
v0.5, 12.01.2012	Oscar Renalias	Large amount of updates to the Hadoop section based on the feedback received
V1.0, 16.01.2012	Oscar Renalias	Ready for release, all feedback incorporated and approved

LICENSE