Introduction

Peter Niederwieser, The Spock Framework TeamVersion 1.1

Spock is a testing and specification framework for Java and Groovy applications. What makes it stand out from the crowd is its beautiful and highly expressive specification language. Thanks to its JUnit runner, Spock is compatible with most IDEs, build tools, and continuous integration servers. Spock is inspired from [JUnit](http://junit.org/), [jMock](http://www.jmock.org/), [RSpec](http://rspec.info/), [Groovy](http://groovy-lang.org/), [Scala](http://scala-lang.org/), [Vulcans](https://en.wikipedia.org/wiki/Vulcan_(Star_Trek)), and other fascinating life forms.

# Getting Started

Peter Niederwieser, The Spock Framework TeamVersion 1.1

It’s really easy to get started with Spock. This section shows you how.

## **Spock Web Console**

[Spock Web Console](http://webconsole.spockframework.org/) is a website that allows you to instantly view, edit, run, and even publish Spock specifications. It is the perfect place to toy around with Spock without making any commitments. So why not run [Hello, Spock!](http://webconsole.spockframework.org/edit/9001) right away?

## **Spock Example Project**

To try Spock in your local environment, clone or download/unzip the [Spock Example Project](https://github.com/spockframework/spock-example). It comes with fully working Ant, Gradle, and Maven builds that require no further setup. The Gradle build even bootstraps Gradle itself and gets you up and running in Eclipse or IDEA with a single command. See the README for detailed instructions.

# Spock Primer

Peter Niederwieser, The Spock Framework TeamVersion 1.1

This chapter assumes that you have a basic knowledge of Groovy and unit testing. If you are a Java developer but haven’t heard about Groovy, don’t worry - Groovy will feel very familiar to you! In fact, one of Groovy’s main design goals is to be the scripting language alongside Java. So just follow along and consult the [Groovy documentation](http://groovy-lang.org/documentation.html) whenever you feel like it.

The goals of this chapter are to teach you enough Spock to write real-world Spock specifications, and to whet your appetite for more.

To learn more about Groovy, go to <http://groovy-lang.org/>.

To learn more about unit testing, go to <http://en.wikipedia.org/wiki/Unit_testing>.

## **Terminology**

Let’s start with a few definitions: Spock lets you write specifications that describe expected features (properties, aspects) exhibited by a system of interest. The system of interest could be anything between a single class and a whole application, and is also called system under specification (SUS). The description of a feature starts from a specific snapshot of the SUS and its collaborators; this snapshot is called the feature’s fixture.

The following sections walk you through all building blocks of which a Spock specification may be composed. A typical specification uses only a subset of them.

## **Imports**

**import** spock.lang.\*

Package spock.lang contains the most important types for writing specifications.

## **Specification**

**class** **MyFirstSpecification** **extends** Specification {

*// fields*

*// fixture methods*

*// feature methods*

*// helper methods*

}

A specification is represented as a Groovy class that extends from spock.lang.Specification. The name of a specification usually relates to the system or system operation described by the specification. For example, CustomerSpec, H264VideoPlayback, and ASpaceshipAttackedFromTwoSides are all reasonable names for a specification.

Class Specification contains a number of useful methods for writing specifications. Furthermore it instructs JUnit to run specification with Sputnik, Spock’s JUnit runner. Thanks to Sputnik, Spock specifications can be run by most modern Java IDEs and build tools.

## **Fields**

**def** obj = **new** ClassUnderSpecification()

**def** coll = **new** Collaborator()

Instance fields are a good place to store objects belonging to the specification’s fixture. It is good practice to initialize them right at the point of declaration. (Semantically, this is equivalent to initializing them at the very beginning of the setup()method.) Objects stored into instance fields are not shared between feature methods. Instead, every feature method gets its own object. This helps to isolate feature methods from each other, which is often a desirable goal.

@Shared res = **new** VeryExpensiveResource()

Sometimes you need to share an object between feature methods. For example, the object might be very expensive to create, or you might want your feature methods to interact with each other. To achieve this, declare a @Shared field. Again it’s best to initialize the field right at the point of declaration. (Semantically, this is equivalent to initializing the field at the very beginning of the setupSpec() method.)

**static** **final** PI = 3.141592654

Static fields should only be used for constants. Otherwise shared fields are preferable, because their semantics with respect to sharing are more well-defined.

## **Fixture Methods**

**def** **setup**() {} *// run before every feature method*

**def** **cleanup**() {} *// run after every feature method*

**def** **setupSpec**() {} *// run before the first feature method*

**def** **cleanupSpec**() {} *// run after the last feature method*

Fixture methods are responsible for setting up and cleaning up the environment in which feature methods are run. Usually it’s a good idea to use a fresh fixture for every feature method, which is what the setup() and cleanup()methods are for.

All fixture methods are optional.

Occasionally it makes sense for feature methods to share a fixture, which is achieved by using shared fields together with the setupSpec() and cleanupSpec() methods. Note that setupSpec() and cleanupSpec() may not reference instance fields unless they are annotated with @Shared.

If fixture methods are overridden in a specification subclass then setup() of the superclass will run before setup() of the subclass. cleanup() works in reverse order, that is cleanup() of the subclass will execute before cleanup() of the superclass. setupSpec() and cleanupSpec() behave in the same way. There is no need to explicitly call super.setup() or super.cleanup() as Spock will automatically find and execute fixture methods at all levels in an inheritance heirarchy.

## **Feature Methods**

**def** "pushing an element on the stack"() {

*// blocks go here*

}

Feature methods are the heart of a specification. They describe the features (properties, aspects) that you expect to find in the system under specification. By convention, feature methods are named with String literals. Try to choose good names for your feature methods, and feel free to use any characters you like!

Conceptually, a feature method consists of four phases:

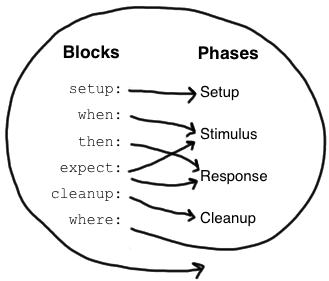
1. Set up the feature’s fixture
2. Provide a stimulus to the system under specification
3. Describe the response expected from the system
4. Clean up the feature’s fixture

Whereas the first and last phases are optional, the stimulus and response phases are always present (except in interacting feature methods), and may occur more than once.

### **Blocks**

Spock has built-in support for implementing each of the conceptual phases of a feature method. To this end, feature methods are structured into so-called blocks. Blocks start with a label, and extend to the beginning of the next block, or the end of the method. There are six kinds of blocks: setup, when, then, expect, cleanup, and where blocks. Any statements between the beginning of the method and the first explicit block belong to an implicit setup block.

A feature method must have at least one explicit (i.e. labelled) block - in fact, the presence of an explicit block is what makes a method a feature method. Blocks divide a method into distinct sections, and cannot be nested.



The picture on the right shows how blocks map to the conceptual phases of a feature method. The where block has a special role, which will be revealed shortly. But first, let’s have a closer look at the other blocks.

#### **Setup Blocks**

setup:

**def** stack = **new** Stack()

**def** elem = "push me"

The setup block is where you do any setup work for the feature that you are describing. It may not be preceded by other blocks, and may not be repeated. A setup block doesn’t have any special semantics. The setup: label is optional and may be omitted, resulting in an implicit setup block. The given: label is an alias for setup:, and sometimes leads to a more readable feature method description (see [Specifications as Documentation](http://spockframework.org/spock/docs/1.1/spock_primer.html#specs-as-doc)).

#### **When and Then Blocks**

when: *// stimulus*

then: *// response*

The when and then blocks always occur together. They describe a stimulus and the expected response. Whereas whenblocks may contain arbitrary code, then blocks are restricted to conditions, exception conditions, interactions, and variable definitions. A feature method may contain multiple pairs of when-then blocks.

##### **Conditions**

Conditions describe an expected state, much like JUnit’s assertions. However, conditions are written as plain boolean expressions, eliminating the need for an assertion API. (More precisely, a condition may also produce a non-boolean value, which will then be evaluated according to Groovy truth.) Let’s see some conditions in action:

when:

stack.push(elem)

then:

!stack.empty

stack.size() == 1

stack.peek() == elem

|  |  |
| --- | --- |
| **TIP** | Try to keep the number of conditions per feature method small. One to five conditions is a good guideline. If you have more than that, ask yourself if you are specifying multiple unrelated features at once. If the answer is yes, break up the feature method in several smaller ones. If your conditions only differ in their values, consider using a [data table](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#data-tables). |

What kind of feedback does Spock provide if a condition is violated? Let’s try and change the second condition tostack.size() == 2. Here is what we get:

Condition not satisfied:

stack.size() == 2

| | |

| 1 false

[push me]

As you can see, Spock captures all values produced during the evaluation of a condition, and presents them in an easily digestible form. Nice, isn’t it?

##### **Implicit and explicit conditions**

Conditions are an essential ingredient of then blocks and expect blocks. Except for calls to void methods and expressions classified as interactions, all top-level expressions in these blocks are implicitly treated as conditions. To use conditions in other places, you need to designate them with Groovy’s assert keyword:

**def** **setup**() {

stack = **new** Stack()

**assert** stack.empty

}

If an explicit condition is violated, it will produce the same nice diagnostic message as an implicit condition.

##### **Exception Conditions**

Exception conditions are used to describe that a when block should throw an exception. They are defined using thethrown() method, passing along the expected exception type. For example, to describe that popping from an empty stack should throw an EmptyStackException, you could write the following:

when:

stack.pop()

then:

thrown(EmptyStackException)

stack.empty

As you can see, exception conditions may be followed by other conditions (and even other blocks). This is particularly useful for specifying the expected content of an exception. To access the exception, first bind it to a variable:

when:

stack.pop()

then:

**def** e = thrown(EmptyStackException)

e.cause == null

Alternatively, you may use a slight variation of the above syntax:

when:

stack.pop()

then:

EmptyStackException e = thrown()

e.cause == null

This syntax has two small advantages: First, the exception variable is strongly typed, making it easier for IDEs to offer code completion. Second, the condition reads a bit more like a sentence ("then an EmptyStackException is thrown"). Note that if no exception type is passed to the thrown() method, it is inferred from the variable type on the left-hand side.

Sometimes we need to convey that an exception should **not** be thrown. For example, let’s try to express that a HashMapshould accept a null key:

**def** "HashMap accepts null key"() {

setup:

**def** map = **new** HashMap()

map.put(null, "elem")

}

This works but doesn’t reveal the intention of the code. Did someone just leave the building before he had finished implementing this method? After all, where are the conditions? Fortunately, we can do better:

**def** "HashMap accepts null key"() {

setup:

**def** map = **new** HashMap()

when:

map.put(null, "elem")

then:

notThrown(NullPointerException)

}

By using notThrown(), we make it clear that in particular a NullPointerException should not be thrown. (As per the contract of Map.put(), this would be the right thing to do for a map that doesn’t support null keys.) However, the method will also fail if any other exception is thrown.

##### **Interactions**

Whereas conditions describe an object’s state, interactions describe how objects communicate with each other. Interactions are devoted a whole [chapter](http://spockframework.org/spock/docs/1.1/interaction_based_testing.html), and so we only give a quick example here. Suppose we want to describe the flow of events from a publisher to its subscribers. Here is the code:

**def** "events are published to all subscribers"() {

**def** subscriber1 = Mock(Subscriber)

**def** subscriber2 = Mock(Subscriber)

**def** publisher = **new** Publisher()

publisher.add(subscriber1)

publisher.add(subscriber2)

when:

publisher.fire("event")

then:

1 \* subscriber1.receive("event")

1 \* subscriber2.receive("event")

}

#### **Expect Blocks**

An expect block is more limited than a then block in that it may only contain conditions and variable definitions. It is useful in situations where it is more natural to describe stimulus and expected response in a single expression. For example, compare the following two attempts to describe the Math.max() method:

when:

**def** x = Math.max(1, 2)

then:

x == 2

expect:

Math.max(1, 2) == 2

Although both snippets are semantically equivalent, the second one is clearly preferable. As a guideline, use when-thento describe methods with side effects, and expect to describe purely functional methods.

|  |  |
| --- | --- |
| **TIP** | Leverage [Groovy JDK](http://docs.groovy-lang.org/docs/latest/html/groovy-jdk/) methods like any() and every() to create more expressive and succinct conditions. |

#### **Cleanup Blocks**

setup:

**def** file = **new** File("/some/path")

file.createNewFile()

*// ...*

cleanup:

file.delete()

A cleanup block may only be followed by a where block, and may not be repeated. Like a cleanup method, it is used to free any resources used by a feature method, and is run even if (a previous part of) the feature method has produced an exception. As a consequence, a cleanup block must be coded defensively; in the worst case, it must gracefully handle the situation where the first statement in a feature method has thrown an exception, and all local variables still have their default values.

|  |  |
| --- | --- |
| **TIP** | Groovy’s safe dereference operator (foo?.bar()) simplifies writing defensive code. |

Object-level specifications usually don’t need a cleanup method, as the only resource they consume is memory, which is automatically reclaimed by the garbage collector. More coarse-grained specifications, however, might use a cleanupblock to clean up the file system, close a database connection, or shut down a network service.

|  |  |
| --- | --- |
| **TIP** | If a specification is designed in such a way that all its feature methods require the same resources, use acleanup() method; otherwise, prefer cleanup blocks. The same trade-off applies to setup() methods and setup blocks. |

#### **Where Blocks**

A where block always comes last in a method, and may not be repeated. It is used to write data-driven feature methods. To give you an idea how this is done, have a look at the following example:

**def** "computing the maximum of two numbers"() {

expect:

Math.max(a, b) == c

where:

a << [5, 3]

b << [1, 9]

c << [5, 9]

}

This where block effectively creates two "versions" of the feature method: One where a is 5, b is 1, and c is 5, and another one where a is 3, b is 9, and c is 9.

Although it is declared last the where block is evaluated before feature method runs.

The where block will be further explained in the [Data Driven Testing](http://spockframework.org/spock/docs/1.1/data_driven_testing.html) chapter.

## **Helper Methods**

Sometimes feature methods grow large and/or contain lots of duplicated code. In such cases it can make sense to introduce one or more helper methods. Two good candidates for helper methods are setup/cleanup logic and complex conditions. Factoring out the former is straightforward, so let’s have a look at conditions:

**def** "offered PC matches preferred configuration"() {

when:

**def** pc = shop.buyPc()

then:

pc.vendor == "Sunny"

pc.clockRate >= 2333

pc.ram >= 4096

pc.os == "Linux"

}

If you happen to be a computer geek, your preferred PC configuration might be very detailed, or you might want to compare offers from many different shops. Therefore, let’s factor out the conditions:

**def** "offered PC matches preferred configuration"() {

when:

**def** pc = shop.buyPc()

then:

matchesPreferredConfiguration(pc)

}

**def** **matchesPreferredConfiguration**(pc) {

pc.vendor == "Sunny"

&& pc.clockRate >= 2333

&& pc.ram >= 4096

&& pc.os == "Linux"

}

The new helper method matchesPreferredConfiguration() consists of a single boolean expression whose result is returned. (The return keyword is optional in Groovy.) This is fine except for the way that an inadequate offer is now presented:

Condition not satisfied:

matchesPreferredConfiguration(pc)

| |

false ...

Not very helpful. Fortunately, we can do better:

**void** matchesPreferredConfiguration(pc) {

**assert** pc.vendor == "Sunny"

**assert** pc.clockRate >= 2333

**assert** pc.ram >= 4096

**assert** pc.os == "Linux"

}

When factoring out conditions into a helper method, two points need to be considered: First, implicit conditions must be turned into explicit conditions with the assert keyword. Second, the helper method must have return type void. Otherwise, Spock might interpret the return value as a failing condition, which is not what we want.

As expected, the improved helper method tells us exactly what’s wrong:

Condition not satisfied:

**assert** pc.clockRate >= 2333

| | |

| 1666 false

...

A final advice: Although code reuse is generally a good thing, don’t take it too far. Be aware that the use of fixture and helper methods can increase the coupling between feature methods. If you reuse too much or the wrong code, you will end up with specifications that are fragile and hard to evolve.

## **Using with for expectations**

As an alternative to the above helper methods, you can use a with(target, closure) method to interact on the object being verified. This is especially useful in then and expect blocks.

**def** "offered PC matches preferred configuration"() {

when:

**def** pc = shop.buyPc()

then:

with(pc) {

vendor == "Sunny"

clockRate >= 2333

ram >= 406

os == "Linux"

}

}

Unlike when you use helper methods, there is no need for explicit assert statements for proper error reporting.

When verifying mocks, a with statement can also cut out verbose verification statements.

**def** service = Mock(Service) *// has start(), stop(), and doWork() methods*

**def** app = **new** Application(service) *// controls the lifecycle of the service*

when:

app.run()

then:

with(service) {

1 \* start()

1 \* doWork()

1 \* stop()

}

## **Specifications as Documentation**

Well-written specifications are a valuable source of information. Especially for higher-level specifications targeting a wider audience than just developers (architects, domain experts, customers, etc.), it makes sense to provide more information in natural language than just the names of specifications and features. Therefore, Spock provides a way to attach textual descriptions to blocks:

setup: "open a database connection"

*// code goes here*

Individual parts of a block can be described with and::

setup: "open a database connection"

*// code goes here*

and: "seed the customer table"

*// code goes here*

and: "seed the product table"

*// code goes here*

An and: label followed by a description can be inserted at any (top-level) position of a feature method, without altering the method’s semantics.

In Behavior Driven Development, customer-facing features (called stories) are described in a given-when-then format. Spock directly supports this style of specification with the given: label:

given: "an empty bank account"

*// ...*

when: "the account is credited $10"

*// ...*

then: "the account's balance is $10"

*// ...*

As noted before, given: is just an alias for setup:.

Block descriptions are not only present in source code, but are also available to the Spock runtime. Planned usages of block descriptions are enhanced diagnostic messages, and textual reports that are equally understood by all stakeholders.

## **Extensions**

As we have seen, Spock offers lots of functionality for writing specifications. However, there always comes a time when something else is needed. Therefore, Spock provides an interception-based extension mechanism. Extensions are activated by annotations called directives. Currently, Spock ships with the following directives:

|  |  |
| --- | --- |
| @Timeout | Sets a timeout for execution of a feature or fixture method. |
| @Ignore | Ignores a feature method. |
| @IgnoreRest | Ignores all feature methods not carrying this annotation. Useful for quickly running just a single method. |
| @FailsWith | Expects a feature method to complete abruptly. @FailsWith has two use cases: First, to document known bugs that cannot be resolved immediately. Second, to replace exception conditions in certain corner cases where the latter cannot be used (like specifying the behavior of exception conditions). In all other cases, exception conditions are preferable. |

To learn how to implement your own directives and extensions, go to the [Extensions](http://spockframework.org/spock/docs/1.1/extensions.html) chapter.

## **Comparison to JUnit**

Although Spock uses a different terminology, many of its concepts and features are inspired from JUnit. Here is a rough comparison:

| **Spock** | **JUnit** |
| --- | --- |
| Specification | Test class |
| setup() | @Before |
| cleanup() | @After |
| setupSpec() | @BeforeClass |
| cleanupSpec() | @AfterClass |
| Feature | Test |
| Feature method | Test method |
| Data-driven feature | Theory |
| Condition | Assertion |
| Exception condition | @Test(expected=…​) |
| Interaction | Mock expectation (e.g. in Mockito) |

# Data Driven Testing

Peter Niederwieser, The Spock Framework TeamVersion 1.1

Oftentimes, it is useful to exercise the same test code multiple times, with varying inputs and expected results. Spock’s data driven testing support makes this a first class feature.

## **Introduction**

Suppose we want to specify the behavior of the Math.max method:

**class** **MathSpec** **extends** Specification {

**def** "maximum of two numbers"() {

expect:

*// exercise math method for a few different inputs*

Math.max(1, 3) == 3

Math.max(7, 4) == 7

Math.max(0, 0) == 0

}

}

Although this approach is fine in simple cases like this one, it has some potential drawbacks:

* Code and data are mixed and cannot easily be changed independently
* Data cannot easily be auto-generated or fetched from external sources
* In order to exercise the same code multiple times, it either has to be duplicated or extracted into a separate method
* In case of a failure, it may not be immediately clear which inputs caused the failure
* Exercising the same code multiple times does not benefit from the same isolation as executing separate methods does

Spock’s data-driven testing support tries to address these concerns. To get started, let’s refactor above code into a data-driven feature method. First, we introduce three method parameters (called data variables) that replace the hard-coded integer values:

**class** **MathSpec** **extends** Specification {

**def** "maximum of two numbers"(**int** a, **int** b, **int** c) {

expect:

Math.max(a, b) == c

...

}

}

We have finished the test logic, but still need to supply the data values to be used. This is done in a where: block, which always comes at the end of the method. In the simplest (and most common) case, the where: block holds a data table.

## **Data Tables**

Data tables are a convenient way to exercise a feature method with a fixed set of data values:

**class** **MathSpec** **extends** Specification {

**def** "maximum of two numbers"(**int** a, **int** b, **int** c) {

expect:

Math.max(a, b) == c

where:

a | b | c

1 | 3 | 3

7 | 4 | 7

0 | 0 | 0

}

}

The first line of the table, called the table header, declares the data variables. The subsequent lines, called table rows, hold the corresponding values. For each row, the feature method will get executed once; we call this an iteration of the method. If an iteration fails, the remaining iterations will nevertheless be executed. All failures will be reported.

Data tables must have at least two columns. A single-column table can be written as:

where:

a | \_

1 | \_

7 | \_

0 | \_

## **Isolated Execution of Iterations**

Iterations are isolated from each other in the same way as separate feature methods. Each iteration gets its own instance of the specification class, and the setup and cleanup methods will be called before and after each iteration, respectively.

## **Sharing of Objects between Iterations**

In order to share an object between iterations, it has to be kept in a @Shared or static field.

|  |  |
| --- | --- |
| **NOTE** | Only @Shared and static variables can be accessed from within a where: block. |

Note that such objects will also be shared with other methods. There is currently no good way to share an object just between iterations of the same method. If you consider this a problem, consider putting each method into a separate spec, all of which can be kept in the same file. This achieves better isolation at the cost of some boilerplate code.

## **Syntactic Variations**

The previous code can be tweaked in a few ways. First, since the where: block already declares all data variables, the method parameters can be omitted.[[1](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_footnote_1)] Second, inputs and expected outputs can be separated with a double pipe symbol (||) to visually set them apart. With this, the code becomes:

**class** **MathSpec** **extends** Specification {

**def** "maximum of two numbers"() {

expect:

Math.max(a, b) == c

where:

a | b || c

1 | 3 || 3

7 | 4 || 7

0 | 0 || 0

}

}

## **Reporting of Failures**

Let’s assume that our implementation of the max method has a flaw, and one of the iterations fails:

maximum of two numbers FAILED

Condition not satisfied:

Math.max(a, b) == c

| | | | |

| 7 4 | 7

42 false

The obvious question is: Which iteration failed, and what are its data values? In our example, it isn’t hard to figure out that it’s the second iteration that failed. At other times this can be more difficult or even impossible. [[2](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_footnote_2)] In any case, it would be nice if Spock made it loud and clear which iteration failed, rather than just reporting the failure. This is the purpose of the @Unroll annotation.

## **Method Unrolling**

A method annotated with @Unroll will have its iterations reported independently:

@Unroll

**def** "maximum of two numbers"() {

...

Why isn’t @Unroll the default?

One reason why @Unroll isn’t the default is that some execution environments (in particular IDEs) expect to be told the number of test methods in advance, and have certain problems if the actual number varies. Another reason is that @Unroll can drastically change the number of reported tests, which may not always be desirable.

Note that unrolling has no effect on how the method gets executed; it is only an alternation in reporting. Depending on the execution environment, the output will look something like:

maximum of two numbers[0] PASSED

maximum of two numbers[1] FAILED

Math.max(a, b) == c

| | | | |

| 7 4 | 7

42 false

maximum of two numbers[2] PASSED

This tells us that the second iteration (with index 1) failed. With a bit of effort, we can do even better:

@Unroll

**def** "maximum of #a and #b is #c"() {

...

This method name uses placeholders, denoted by a leading hash sign (#), to refer to data variables a, b, and c. In the output, the placeholders will be replaced with concrete values:

maximum of 3 and 5 is 5 PASSED

maximum of 7 and 4 is 7 FAILED

Math.max(a, b) == c

| | | | |

| 7 4 | 7

42 false

maximum of 0 and 0 is 0 PASSED

Now we can tell at a glance that the max method failed for inputs 7 and 4. See [More on Unrolled Method Names](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_more_on_unrolled_method_names) for further details on this topic.

The @Unroll annotation can also be placed on a spec. This has the same effect as placing it on each data-driven feature method of the spec.

## **Data Pipes**

Data tables aren’t the only way to supply values to data variables. In fact, a data table is just syntactic sugar for one or more data pipes:

...

where:

a << [1, 7, 0]

b << [3, 4, 0]

c << [3, 7, 0]

A data pipe, indicated by the left-shift (<<) operator, connects a data variable to a data provider. The data provider holds all values for the variable, one per iteration. Any object that Groovy knows how to iterate over can be used as a data provider. This includes objects of type Collection, String, Iterable, and objects implementing the Iterablecontract. Data providers don’t necessarily have to be the data (as in the case of a Collection); they can fetch data from external sources like text files, databases and spreadsheets, or generate data randomly. Data providers are queried for their next value only when needed (before the next iteration).

## **Multi-Variable Data Pipes**

If a data provider returns multiple values per iteration (as an object that Groovy knows how to iterate over), it can be connected to multiple data variables simultaneously. The syntax is somewhat similar to Groovy multi-assignment but uses brackets instead of parentheses on the left-hand side:

@Shared sql = Sql.newInstance("jdbc:h2:mem:", "org.h2.Driver")

**def** "maximum of two numbers"() {

expect:

Math.max(a, b) == c

where:

[a, b, c] << sql.rows("select a, b, c from maxdata")

}

Data values that aren’t of interest can be ignored with an underscore (\_):

...

where:

[a, b, \_, c] << sql.rows("select \* from maxdata")

## **Data Variable Assignment**

A data variable can be directly assigned a value:

...

where:

a = 3

b = Math.random() \* 100

c = a > b ? a : b

Assignments are re-evaluated for every iteration. As already shown above, the right-hand side of an assignment may refer to other data variables:

...

where:

where:

row << sql.rows("select \* from maxdata")

*// pick apart columns*

a = row.a

b = row.b

c = row.c

## **Combining Data Tables, Data Pipes, and Variable Assignments**

Data tables, data pipes, and variable assignments can be combined as needed:

...

where:

a | \_

3 | \_

7 | \_

0 | \_

b << [5, 0, 0]

c = a > b ? a : b

## **Number of Iterations**

The number of iterations depends on how much data is available. Successive executions of the same method can yield different numbers of iterations. If a data provider runs out of values sooner than its peers, an exception will occur. Variable assignments don’t affect the number of iterations. A where: block that only contains assignments yields exactly one iteration.

## **Closing of Data Providers**

After all iterations have completed, the zero-argument close method is called on all data providers that have such a method.

## **More on Unrolled Method Names**

An unrolled method name is similar to a Groovy GString, except for the following differences:

* Expressions are denoted with # instead of $ [[3](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_footnote_3)], and there is no equivalent for the ${…​} syntax.
* Expressions only support property access and zero-arg method calls.

Given a class Person with properties name and age, and a data variable person of type Person, the following are valid method names:

**def** "#person is #person.age years old"() { *// property access*

**def** "#person.name.toUpperCase()"() { *// zero-arg method call*

Non-string values (like #person above) are converted to Strings according to Groovy semantics.

The following are invalid method names:

**def** "#person.name.split(' ')[1]" { *// cannot have method arguments*

**def** "#person.age / 2" { *// cannot use operators*

If necessary, additional data variables can be introduced to hold more complex expression:

**def** "#lastName"() { *// zero-arg method call*

...

where:

person << [**new** Person(age: 14, name: 'Phil Cole')]

lastName = person.name.split(' ')[1]

}

[**1**](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_footnoteref_1). The idea behind allowing method parameters is to enable better IDE support. However, recent versions of IntelliJ IDEA recognize data variables automatically, and even infer their types from the values contained in the data table.

[**2**](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_footnoteref_2). For example, a feature method could use data variables in its setup: block, but not in any conditions.

[**3**](http://spockframework.org/spock/docs/1.1/data_driven_testing.html#_footnoteref_3). Groovy syntax does not allow dollar signs in method names.

# References

http://spockframework.org/spock/docs/1.1/index.html