# Spock Testing Framework


**Prakash Badhe**

**prakash.badhe@vishwasoft.in**

# Groovy

- Groovy is high level language on top of java designed to make java programming more faster and productive.
- Groovy is a powerful dynamic scripting language.
- Supports static-typing and static compilation capabilities.
- Integrates smoothly with any Java program and supports declarative programming.

# Groovy Program

- A groovy program is written as script or class.
- The groovy interpreter dynamically generates the groovy to code to java byte code and executes line by line.
- Optionally the groovyc compiler can be used to compile groovy code java .class files to be used in java programs.
- The groovy works with other java classes by importing and referring just as java code.

# Groovy : Scaffold for Java

- For writing an java program you need to write everything from importing class files, class declaration to method bodies and main(optional).
- With groovy this code is generated as default for  every groovy program.
- The groovy bootstraps the java code!

# Spock



Spock is a testing and specification framework for Java and Groovy applications.
Spock is based on java platform.

# Spock Web Console



- http://meetspock.appspot.com/?id=9001
- Here you instantly view, edit, run, and even publish Spock specifications.
- It is the perfect place to toy around with Spock without making any commitments.

# Test Class with Spock



- The Test class is a groovy class that extends from spock.lang.Specification.
- Class Specification contains a number of useful methods for writing specifications.
- It instructs JUnit to run specification with Sputnik, Spock's JUnit runner.

# Testing with Spock

- Spock supports **Behavior Driven Development** for test applications.
- Spock supports writing test class as *specifications* which describe expected *features* (properties, aspects) to be tested for system of interest.
- The system of interest could be anything between a single class and a whole application and is termed the *system under specification or **SUS***.
- The system to test the features is called as feature's *fixture*.

`

# Spock vs. JUnit



| **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) |

# Spock Fixture: Test Life cycle



- **def setupSpec**() {} *// runs once - before the first feature method*
- **def setup**() {} *// runs before every feature method*
- **def cleanup**() {} *// runs after every feature method*
- **def cleanupSpec**() {} *// runs once - after the last feature method*

Fixture methods are responsible for setting up and cleaning up the environment in which feature methods are run.

# Spock Features : Test Cases



- In a class extending Specification you write test cases as feature methods
- Feature methods are the heart of a specification(test cases).
- Theses describe the features (properties, aspects) that you expect to find in the system under specification.
- By convention, feature methods are named with String literals.
- **def** “Testing for Saving Account Existence"() {
- *// code for testing* }

# Feature Method Phases

The feature method consists of four phases:

- Set up the feature's fixture
- Provide a *stimulus* to the system under specification (inputs)
- Describe the *response* expected from the system(assert the output)
- Clean up the feature's fixture

# Blocks : Built-in Feature Parts



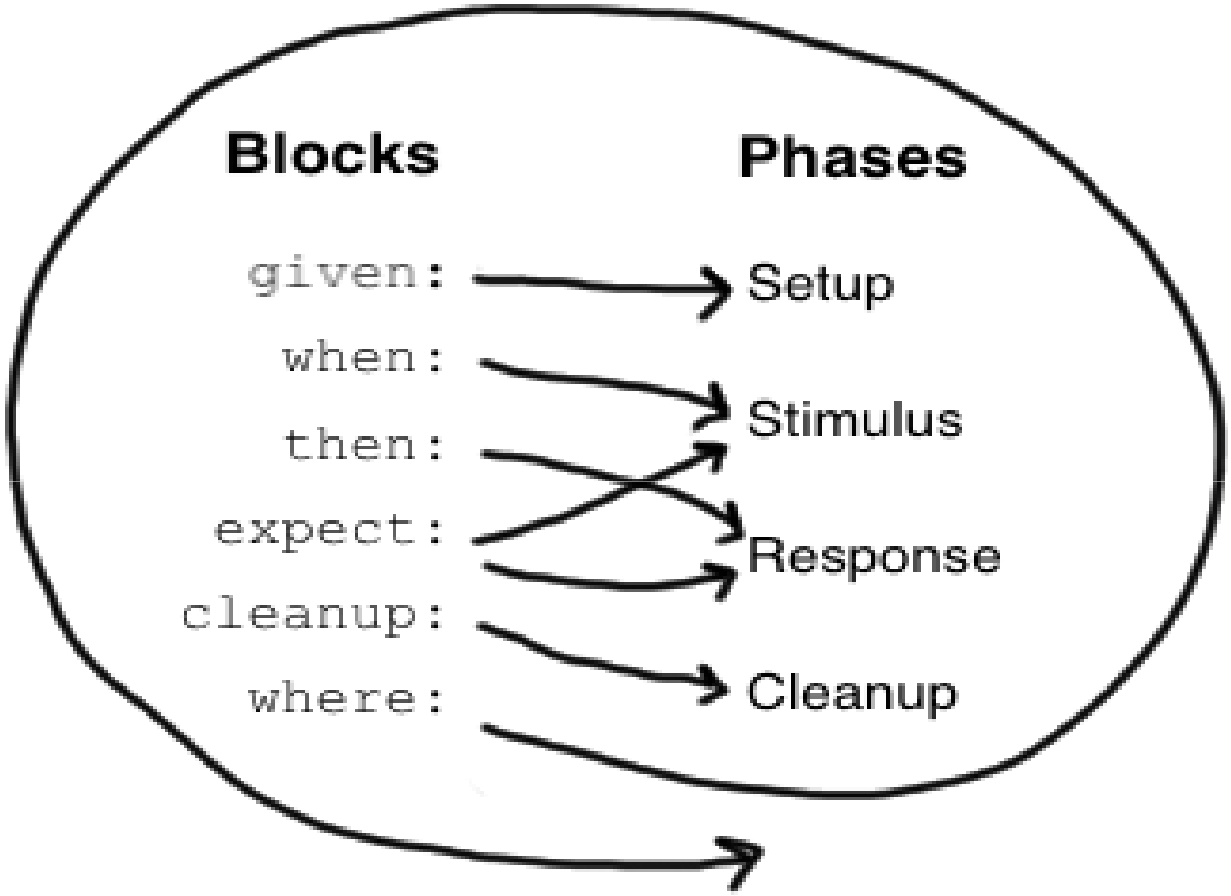
- The feature methods are structured into so-called *blocks*.
- Blocks start with a label, and extend to the beginning of the next block, or the end of the method.
- There are six kinds of blocks:
- given, when, then, expect, cleanup, and where blocks.
- Any statements between the beginning of the method and the first explicit block belong to an implicit given block.

# Feature with 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 blocks in feature method






given:
**def** stack = **new** Stack()
**def** elem = "push me"

# Test for Expectations



- Conditions describe an expected state, much like JUnit's assertions.
- The  conditions are written as plain boolean expressions, eliminating the need for an assertion API.
- The conditions are written inside the 'then' block.

# Condition Example



- when:
  - stack.push(elem)
- then:
  - !stack.empty
  - stack.size() == 1
  - stack.peek() == elem

# Implicit and explicit conditions



- Conditions are an essential ingredient of then blocks and expect blocks.
- All top-level expressions in these blocks are implicitly treated as conditions.
- To use conditions in other places, you need to use them with Groovy's assert keyword:
  - **def setup**() {
  - stack = **new** Stack()
  - **assert** stack.empty
  - }

  When an explicit condition is violated, it produce the same diagnostic message.

# Expected Exception



- Exception conditions are used to describe that a when block should throw an exception.
- They are defined using the thrown() method, passing along the expected exception type.
- when: stack.pop()
- then: thrown(EmptyStackException) stack.empty

# Interactions

- Whereas conditions describe an object's state, interactions describe how objects communicate with each other.
- To describe the flow of events from a publisher to its subscribers.

# The expect block

- 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.

# Expect Example

- when:
- **def** x = Math.max(1, 2)
- then: x == 2
- expect:
- Math.max(1, 2) == 2

# The cleanup block



- The cleanup block may only be followed by a where block, and may not be repeated.
- 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.

# The where block



- The where block always comes last in a method, and may not be repeated.
- It is used to write data-driven feature methods.
- The where block effectively creates multiple "versions" of the feature method based on the parameters as like in parameterized test methods.

# Reporting the failures

- Spock displays  the log of success and failures.
- To clearly identify the details of failures, pock uses @Unroll.
- Spock makes it loud and clear which iteration failed, rather than just reporting the failure.
- A method annotated with @Unroll will have its iterations reported independently.
- The unrolling has no effect on how the method gets executed; it is only an alternation in reporting.
- @Unroll **def** "maximum of two numbers"() {

# Spock Extensions

- Spock offers lots of functionality for writing specifications.
- When something else is needed, pock provides an interception-based extension mechanism.
- Extensions are activated by annotations called *directives*.
  - **@Timeout:** Sets a timeout for execution of a feature or fixture method.
  - **@Ignore:** Ignores any feature method carrying this annotation.

# More extensions

- **@IgnoreRest:** Any feature method carrying this annotation will be executed and all others will be ignored. Useful for quickly running just a single method.
- **@FailsWith:** Expects a feature method to complete abruptly.
- The 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.

# Mocking Support

- Built-in mock library is provided.
- The Mock objects are created with the MockingApi.Mock() method.
  - **def** subscriber = Mock(Subscriber)
  - Subscriber subscriber = Mock()
- Inject the mocks
  - **def setup**() {
  - publisher.subscribers << subscriber
  - *// << is a Groovy shorthand for List.add()*

# Verify Mock Interactions



- then:
- 1 * subscriber.receive("hello")
- 1 * subscriber2.receive("hello") }
- When the publisher sends a 'hello' message, then both subscribers should receive that message exactly once.

# Cardinality of Interactions



- The cardinality of an interaction describes how often a method call is expected.
- It can either be a fixed number or a range.
- 1 * subscriber.receive("hello") *// exactly one call* 0 * subscriber.receive("hello") *// zero calls* (1..3) * subscriber.receive("hello") *// between one and three calls (inclusive)* (1.._) * subscriber.receive("hello") *// at least one call* (_..3) * subscriber.receive("hello") *// at most three calls* _ * subscriber.receive("hello") *// any number of calls, including zero // (rarely needed; see 'Strict Mocking')*

# Cardinality

- 1 * subscriber.receive("hello")
- *--exactly one call*
- 0 * subscriber.receive("hello")
- */--zero calls*
- (1..3) * subscriber.receive("hello")
- *--between one and three calls (inclusive)*
- (1.._) * subscriber.receive("hello")
- *---at least one call* (_..3)
- * subscriber.receive("hello")
- *--at most three calls*
- _ * subscriber.receive("hello")
- *--any number of calls, including zero /*

# The constraints on Mock



- Target
- The target constraint of an interaction describes which mock object is expected to receive the method call:
- 1 * subscriber.receive("hello") *// a call to 'subscriber'*
- 1 * _.receive("hello") *// a call to any mock object*

# Method constraints

- The method constraint of an interaction describes which method is expected to be called:
- 1 * subscriber.receive("hello")
- *--a method named 'receive‘*
- 1 * subscriber./r.*e/("hello")
- *--a method whose name matches the given regular expression* : *method name starts with 'r' and ends in 'e')*
- 1 * subscriber.status
- *-- same as: 1 * subscriber.getStatus()*
- 1 * subscriber.setStatus("ok")

# Arguments Constraints

- The argument constraints of an interaction describe which method arguments are expected.
- 1 * subscriber.receive("hello")
- *--an argument that is equal to the String "hello"*
- 1 * subscriber.receive(!"hello")
- *---an argument that is unequal to the String "hello"*

# More Arg Constraints

- 1 * subscriber.receive() // the empty argument list (would never match in our example)
- 1 * subscriber.receive(_) // any single argument (including null)
- 1 * subscriber.receive(*_) // any argument list (including the empty argument list)
- 1 * subscriber.receive(!null) // any non-null argument
- 1 * subscriber.receive(_ as String) // any non-null argument that is-a String
- 1 * subscriber.receive(endsWith("lo")) // any non-null argument that is-a String
- 1 * subscriber.receive({ it.size() > 3 && it.contains('a') })

# Type constraints with Mock



- The type constraint checks for the type/class of the argument, like the negating constraint it is also a compound constraint.
- The ‘_ ‘ as Type, which is a combination of the wildcard constraint and the type constraint.
- 1 * subscriber.receive({ it.contains(Data')} as String)
- --assert that it is a String before executing the code constraint to check if it contains Data.

# Matching to anything

- 1 * subscriber._(*_)  // any method on subscriber, with any argument list
- 1 * subscriber._  // shortcut for he above

- 1 * _._  // any method call on any mock object
- 1 * _  // shortcut for the above