## Specs \_ what it is

Scala library for Behaviour-Driven Development.

```
import org.specs._
object HelloWorldSpec extends Specification {
   "'hello world' has 11 characters" in { "hello world".size must_== 11 }
   "'hello world' matches 'h.* w.*'" in { "hello world" must beMatching("h.* w.*") }
}
```

```
[info] Running specs tests...
[info]
[info] Testing scalamelb.specs.HelloWorldSpec ...
[info] specifies
[info] + 'hello world' has 11 characters
[info] + 'hello world' matches 'h.* w.*'
[info]
[info] All tests PASSED.
```

## Specs \_ structure

- BDD builds on TDD to allow tests to be written in the business domain language.
- The Specs structure:
  Specify system (with context) and examples
  - "An empty stack" should "be of size zero"

```
"A Nigerian prince with access to an email account" should {
    "contact you in relation to his most generous offer" in {
        // exercise the logic and make assertions ...
    }
}
```

## Specs \_ structure

Examples can be nested

```
"A snowflake" should { [info] A snowflake should "be cold" in { [info] + be cold "and icy" in { ... } [info] + and icy "and unique" in { ... } [info] + and unique
```

Systems can't

```
"A snowflake" should {
    "in the desert" should {
        "melt" in { ... }
```

Specifications can, sort of

```
object SnowflakeSpec extends Specification {
   "A snowflake" isSpecifiedBy SnowflakeInTheDesertSpec, SnowflakeInTheFrideSpec,
        SnowflakeOnTheTongueSpec
}
```

## Specs \_ matchers

- Matchers are the assertion mechanism.
  - Allow natural language assertions
  - 100+ are documented at time of writing (v1.4.4)
  - Can write custom matchers. Can be combinatorial.

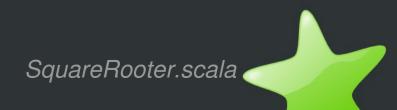
```
"A phone bill" should {
   "export calls to file" in {
    bill.exportFile must (beReadable and not(beHidden))
```

### Specs \_ scalacheck

- Matcher for <u>ScalaCheck</u>
  - (A test case automation tool)
  - Default of 100 arbitrary test input values per statement

```
object SquareRooterSpec extends Specification with ScalaCheck {
   "A Square Rooter" should {
      "find the originally squared value (as long as it wasn't negative)" in {
        // forAll { (n: Int) => scala.Math.sqrt(n*n) == n } must pass // fails on -1
        forAll { n: Int => n >= 0 ==> (scala.Math.sqrt(n*n) == n) } must pass
```

• For all n (where  $n \ge 0$ )  $\sqrt{n^2}$  must equal n



## Specs \_ tear up / down

- Traditional setup & teardown mechanisms available for before and after:
  - Specification
  - each System
  - each Example

```
object SetupAndTeardown extends Specification {
   doBeforeSpec{ println("Specification setup") }
    doFirst{ println("System setup") }
   doBefore{ println("Example setup") }
...
   doAfter{ println("example teardown") }
   doLast{ println("System teardown") }
   doAfterSpec{ println("Specification teardown") }
}
```

### **Specs** \_ shared contexts

• Before and after context blocks can be encapsulated in Context values and "threaded" into Systems.

```
var listOfStrings:List[String] = Nil
val withThreeValues = beforeContext {
   listOfStrings = "bob" :: "harry" :: "dorothy" :: Nil }

"A list of three Strings" ->-(withThreeValues) should {
   "be of three strings long" in {
     listOfStrings.length must_== 3
   }
}
```

• They come with a warning – expect the unexpected if shared and mutated in different Systems within the Specification. And don't feed after midnight.

### **Specs** \_ system contexts

- Alternatively, System Contexts can be constructed to always return a system in a given state.
- They come in two flavours:
  - Internal (a SystemContext instance)
  - External (a case class extending SystemContext)

```
object MonkeyKingSpec extends Specification with SystemContexts {
   /* Demonstrates explicit, internal system context. See example for more. */
   "The monkey king under attack" should {
    val monkeyKingUnderAttack = systemContext { new MonkeyKing(true) }
    "summon warriors from his ear hairs".withA(monkeyKingUnderAttack) {
        monkeyKing => monkeyKing.summonsWarriorsFromEarHair must beTrue
```

## Specs \_ mocking

- Both jMock and Mockito can be mixed-in
  - Specs adds syntactic sugar to ease mocking

```
object USBLightSpec extends Specification with Mockito {
    "A green USB light" should {
        "be observed but not adjusted" in {
            val light = mock[Light]
            light.getColour returns Green

            val colour = LightObserver(light).observeLightsColour colour must_== Green

            light.getColour was called
            light.setIntensity _ wasnt called
```

# Specs \_ alternatives

#### • ScalaTest

```
class StackSpec extends Spec with ShouldMatchers {
  describe("A newly created Stack") {
    val stack = new Stack[Any]
    it("should be empty") {
       stack should be ('empty)
    }
  }
}
```

#### • Instinct

```
class ANewlyCreatedStack {
  val stack = new Stack[Any]
  @Specification def shouldBeEmpty = {
    expect that stack.depth isEqualTo 0
  }
}
```

