Testing with specs

This lesson covers testing with Specs, a Behavior-Driven Design (BDD) Framework for Scala.

extends Specification

Let's just jump in.

```
import org.specs._

object ArithmeticSpec extends Specification {
  "Arithmetic" should {
    "add two numbers" in {
      1 + 1 mustEqual 2
    }
    "add three numbers" in {
      1 + 1 + 1 mustEqual 3
    }
  }
}
```

Arithmetic is the **System Under Specification**

add is a context.

add two numbers and **add three numbers** are examples.

`mustEqual` indicates an **expectation**

`1 mustEqual 1` is a common placeholder **expectation** before you start writing real tests. All examples should have at least one expectation.

Duplication

Notice how two tests both have `add` in their name? We can get rid of that by **nesting** expectations.

```
import org.specs._

object ArithmeticSpec extends Specification {
  "Arithmetic" should {
    "add" in {
      "two numbers" in {
        1 + 1 mustEqual 2
      }
      "three numbers" in {
        1 + 1 + 1 mustEqual 3
      }
    }
  }
}
```

Execution Model

```
object ExecSpec extends Specification {
  "Mutations are isolated" should {
    var x = 0
    "x equals 1 if we set it." in {
      x = 1
      x mustEqual 1
    }
    "x is the default value if we don't change it" in {
      x mustEqual 0
    }
  }
}
```

Setup, Teardown

doBefore & doAfter

```
"my system" should {
  doBefore { resetTheSystem() /** user-defined reset function */ }
  "mess up the system" in {...}
  "and again" in {...}
  doAfter { cleanThingsUp() }
}
```

NOTE `doBefore` / `doAfter` are only run on leaf examples.

doFirst & doLast

`doFirst` / `doLast` is for single-time setup. (need example, I don't use this)

```
"Foo" should {
  doFirst { openTheCurtains() }
  "test stateless methods" in {...}
  "test other stateless methods" in {...}
  doLast { closeTheCurtains() }
}
```

Matchers

You have data, you want to make sure it's right. Let's tour the most commonly used matchers. (See Also Matchers Guide)

mustEqual

We've seen several examples of mustEqual already.

```
1 mustEqual 1

"a" mustEqual "a"
```

Reference equality, value equality.

elements in a Sequence

```
val numbers = List(1, 2, 3)

numbers must contain(1)
numbers must not contain(4)

numbers must containAll(List(1, 2, 3))
numbers must containInOrder(List(1, 2, 3))

List(1, List(2, 3, List(4)), 5) must haveTheSameElementsAs(List(5, List(List(4), 2, 3), 1))
```

Items in a Map

```
map must haveKey(k)
map must notHaveKey(k)

map must haveValue(v)
map must notHaveValue(v)
```

Numbers

```
a must beGreaterThan(b)
a must beGreaterThanOrEqualTo(b)

a must beLessThan(b)
a must beLessThanOrEqualTo(b)

a must beCloseTo(b, delta)
```

Options

```
a must beNone

a must beSome[Type]

a must beSomething

a must beSome(value)
```

throwA

```
a must throwA[WhateverException]
```

This is shorter than a try catch with a fail in the body.

You can also expect a specific message

```
a must throwA(WhateverException("message"))
```

You can also match on the exception:

```
a must throwA(new Exception) like {
  case Exception(m) => m.startsWith("bad")
}
```

Write your own Matchers

```
import org.specs.matcher.Matcher
```

As a val

```
"A matcher" should {
  "be created as a val" in {
    val beEven = new Matcher[Int] {
      def apply(n: => Int) = {
        (n % 2 == 0, "%d is even".format(n), "%d is odd".format(n))
      }
    }
    2 must beEven
  }
}
```

The contract is to return a tuple containing whether the expectation is true, and a message for when it is and isn't true.

As a case class

```
case class beEven(b: Int) extends Matcher[Int]() {
  def apply(n: => Int) =  (n % 2 == 0, "%d is even".format(n), "%d is odd".format(n))
}
```

Using a case class makes it more shareable.

Mocks

```
import org.specs.Specification
import org.specs.mock.Mockito

class Foo[T] {
  def get(i: Int): T
}

object MockExampleSpec extends Specification with Mockito {
  val m = mock[Foo[String]]

  m.get(0) returns "one"

  m.get(0)

  there was one(m).get(0)

  there was no(m).get(1)
}
```

See Also Using Mockito

Spies

Spies can also be used in order to do some "partial mocking" of real objects:

```
val list = new LinkedList[String]
val spiedList = spy(list)

// methods can be stubbed on a spy
spiedList.size returns 100

// other methods can also be used
spiedList.add("one")
spiedList.add("two")

// and verification can happen on a spy
there was one(spiedList).add("one")
```

However, working with spies can be tricky:

```
// if the list is empty, this will throws an IndexOutOfBoundsException
spiedList.get(0) returns "one"
```

`doReturn` must be used in that case:

```
doReturn("one").when(spiedList).get(0)
```

Run individual specs in sbt

```
> test-only com.twitter.yourservice.UserSpec
```

Will run just that spec.

```
> ~ test-only com.twitter.yourservice.UserSpec
```

Will run that test in a loop, with each file modification triggering a test run.