Testing in Scala

Pre and Post Conditions, Unit and Integration Tests, Mocking, Stubbing, Property Driven Testing

Agenda

1. Pre and Post Conditions
2. Scalatest
3. Tests vs Specs
4. Matchers
5. Exception Handling
6. Mocking and Stubbing
7. Property Driven Testing
8. Testing With Futures

Pre and Post Conditions

- Idea from Bertrand Meyer, pre-conditions, post-conditions and invariants

```
assert(2 > 3)  // java.lang.AssertionError: assertion failed
assume(2 > 3)  // java.lang.AssertionError: assumption failed
(1 + 1) ensuring (_ > 3) // java.lang.AssertionError: assertion failed
```

- These can (and should) include String explanations

```
assert(x > 3, "x must be larger than 3")
// java.lang.AssertionError: assertion failed: x must be larger than 3

assume(x > 3, "x must be larger than 3")
//java.lang.AssertionError: assumption failed: x must be larger than 3

(x - 1) ensuring (_ > 3, "x is not large enough")
// java.lang.AssertionError: assertion failed: x is not large enough

def square(x: Int): Int = {
  x * x
} ensuring (_ >= 0, "squares cannot be negative")
```

`assert` and `assume` Can Be Elided

- But `ensuring` still throws…

```
14:28 $ scala -Xdisable-assertions
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.

scala> val x = 2
x: Int = 2

scala> assert(x > 3, "x must be larger than 3")

scala> assume(x > 3, "x must be larger than 3")

scala> (x - 1) ensuring (_ > 3, "x is not large enough")
java.lang.AssertionError: assertion failed: x is not large enough
  at scala.Predef$Ensuring$.ensuring$extension3(Predef.scala:219)
  ... 29 elided
```

Requirements

- Since `assert` and `assume` can be turned off, we have `require` (and `requireState`) which always throw on failure

```
val x = 2

require(x < 0, "x must be negative")
// java.lang.IllegalArgumentException: requirement failed: x must be negative

import org.scalactic.Requirements._
requireState(x < 0, "x must be negative")
// java.lang.IllegalStateException: 2 was not less than 0 x must be negative
```

- `require`, and `requireState` from Scalactic are idiomatic, `assert` and `assume` less so

Testing

- Testing took over from pre-conditions, post-conditions and invariants in most development
- Scala has various testing frameworks:
    - JUnit
    - TestNG
    - Specs2
    - Scalatest
- In the course so far, you have been using Scalatest
- Scalatest supports many styles of testing and offers many features

Scalatest

- Scalatest is a free, open source testing framework that is well documented and widely used
- It supports many different styles of tests: http://www.scalatest.org/user_guide/selecting_a_style
- We will concentrate on a couple of styles:
    - `FunSuite` - very similar to the `koans` style you have been using throughout the couse
    - `FunSpec` - a more BDD influenced suite for writing specifications
- We will also look at the `Matchers` DSL and support for mocking and property testing
- Keep the http://www.scalatest.org/user_guide handy to write your own (there is lots here)

FunSuite

```scala
import org.scalatest._

class TestingSuiteDemo extends FunSuite with Matchers {
  val nums: List[Int] = (1 to 20).toList

  test("Filtering a list") {
    val filtered = nums.filter(_ > 15)
    assert(filtered === Seq(16, 17, 18, 19, 20))
  }

  test("Summing a list") {
    nums.sum should be (210)
  }

  test("Try something else")(pending)
}
```

- `tests` all have a string name, names must be unique in that suite
- First test uses `assert` with `===` (this is not the elidable pre-condition assert)
- Second test uses `Matchers` for more readability
- The `pending` test at the end will put out a warning until defined

FunSpec

- More BDD (behavioral driven design): requirements and specs

```
class TestingSpecDemo extends FunSpec with Matchers {

  describe ("Retrieving the weather from the weather service") {
    it ("should call getWeather only if operational is true")(pending)
    it ("should not call getWeather if service not operational")(pending)
    it ("should not call getWeather if operational throws an exception")(pending)
    it ("should call getWeather with different codes when necessary")(pending)
  }

  describe ("Various matchers") {
    describe("on a list of numbers") {
      they("should allow a wide and varied language for matching")(pending)
    }
    describe("on a case class example") {
      they("should allow easy field checking")(pending)
    }
  }

  describe ("Handling exceptions") {
    it ("should expect and intercept exceptions")(pending)
  }
}
```

Matchers

- `Matchers` is a large DSL that makes tests easier to read (though possibly harder to write)

```
describe ("Various matchers") {
  describe("on a list of numbers") {

    val nums = (1 to 20).toList
    val threeMults = nums.filter(_ % 3 == 0)

    they("should allow a wide and varied language for matching") {
      val x = 10 * 2

      x should be (20)

      threeMults should have size (6)
      threeMults should contain allOf (3, 6, 12, 15)
      threeMults should not contain (10)
      threeMults should be (Vector(3, 6, 9, 12, 15, 18))
      threeMults should be (sorted)
      all(threeMults) should be > (0)
      atLeast(3, threeMults) should be > (10)
    }
  }
```

Matchers

- Also support for fields in case classes, e.g.

```
    describe("on a case class example") {
      they("should allow easy field checking") {
        case class Person(first: String, last: String, age: Int)

        val p1 = Person("Harry", "Potter", 34)

        p1 should have(
          'first ("Harry"),
          'last ("Potter"),
          'age (34)
        )
      }
    }
```

Matchers

- And for checking exceptions:

```
describe ("Handling exceptions") {
  it ("should expect and intercept exceptions") {
    an [IllegalArgumentException] should be thrownBy {
      require(1 == 2, "One equals two?")
    }

    val ae = intercept [ArithmeticException] (1 / 0)
    ae.getMessage should be ("/ by zero")
  }
}
```

- Also checking floating point values within tolerance

```
describe ("Floating point values") {
  they ("should be matched within a tolerance") {
    val sqrt = math.sqrt(2.0)
    sqrt should be (1.41421356 +- 1e-6)
    sqrt should (be > 1.41421355 and be < 1.41421357)
  }
}
```

Unit vs Integration Testing

- Unit tests are small, self contained and fast, requiring no external resources
- Integration tests may use external resources, and are useful for checking a complete system
- Neither type of testing is enough without the other
- To Unit test, one must often isolate from external resources
- This is where fakes, mocks and stubs come in useful

Scalamock

- Scalatest has support for several mocking frameworks, I tend to use *Scalamock*
- Imagine a service to look up external weather details from a web API

```scala
case class Weather(temp: Double, precip: Double, pressure: Double)

trait WeatherService {
  def getWeather(icaoCode: String): Weather
  def operational(): Boolean
}

def lookupWeather(service: WeatherService, icaoCode: String): Option[Weather] = {
  if (!service.operational()) None
    else Some(service.getWeather(icaoCode))
}
```

Unit Testing with Mocks

- For Unit testing, we don't want to use the real service (slow/unreliable), we also might want to simulate failures to test how they are handled:

```
class TestingSpecDemo extends FunSpec with Matchers with MockFactory
```

```
  describe ("Retrieving the weather from the weather service") {
    it ("should call getWeather if operational is true") {
      val mockWS = mock[WeatherService]

      (mockWS.operational _: () => Boolean).expects().returning(true)
      (mockWS.getWeather _).expects("PDX").returning(Weather(55.0, 0.0, 1012.0))

      val weather = lookupWeather(mockWS, "PDX")

      val tolerance = 1e-6

      weather should be (defined)
      weather.get.precip should be (0.0 +- tolerance)
      weather.get.pressure should be (1012.0 +- tolerance)
      weather.get.temp should be (55.0 +- tolerance)
    }
```

Advantages of Mocks

- Mocks allow full scripting of responses (including failures)

```
it ("should not call the getWeather if operational throws an exception") {
  val mockWS = mock[WeatherService]

  (mockWS.operational _: () => Boolean).expects().
    throwing(new IllegalStateException("network failure"))

  val ex = intercept[IllegalStateException] {
    lookupWeather(mockWS, "PDX")
  }

   ex.getMessage should be ("network failure")
}
```

- They also verify calls are made, in the correct quantity (e.g. fail if too many or few)

Mocks vs Stubs

- With Mocks, you script up what you expect
- With Stubs, the emphasis is on verifying what you got:

```
it ("should call the lookup weather with different codes when necessary") {
  val stubWS = stub[WeatherService]

  (stubWS.operational _: () => Boolean).when().returning(true)
  (stubWS.getWeather _).when("PDX").returning(Weather(55.0, 0.0, 1012.0))
  (stubWS.getWeather _).when("SFO").returning(Weather(65.0, 0.3, 1008.0))

  val results1 = lookupWeather(stubWS, "SFO")  // check results1 here
  val results2 = lookupWeather(stubWS, "PDX")  // check results2 here

  (stubWS.operational _: () => Boolean).verify().twice()
  (stubWS.getWeather _).verify("PDX")
  (stubWS.getWeather _).verify("SFO")
}
```

Fakes

```scala
trait DBAccess {
  def save[T](item: T): String
  def load[T](id: String): Option[T]
}

class FakeDBAccess extends DBAccess {
  private[this] var itemMap = Map.empty[String, Any]

  def save[T](item: T): String = {
    val uuid = UUID.randomUUID().toString
    itemMap = itemMap + (uuid -> item)
    uuid
  }

  def load[T](id: String): Option[T] =
    Try(itemMap(id).asInstanceOf[T]).toOption
}

case class Person(name: String, age: Int)
val fake = new FakeDBAccess
val uuid = fake.save(Person("Sally", 23)) // 1a889c78-ea0a-45ae-b8ce-9e6305d6ec24
fake.load(uuid)   // Some(Person(Sally,23))
```

Property Driven Testing

- Scalatest has support for Scalacheck, which can generate random testing data:

```scala
import org.scalatest._
import org.scalatest.prop.PropertyChecks

class TestingSpecDemo extends FunSpec with Matchers with PropertyChecks {
  describe ("Property checks") {
    they ("should ensure abs on all Ints returns a positive number") {
      forAll { (i: Int) =>
        whenever(i != Int.MinValue) {
          i.abs should be >= 0
        }
      }
    }
  }
}
```

- Without that `whenever`:

```
// TestFailedException was thrown during property evaluation.
//   Message: -2147483648 was not greater than or equal to 0
```

Custom Property Generators

```scala
import org.scalacheck.Gen

val validChars = ('a' to 'z') ++ ('A' to 'Z') ++ ('0' to '9')
val char = Gen.oneOf(validChars)
val strGen = for {
  n <- Gen.choose(0, 30)  // 0 to 30 length strings
  seqChars <- Gen.listOfN(n, char)
} yield seqChars.mkString


describe ("Property checks and generators") {
  they ("should test .reverse.reverse on any string gives you back original") {
    forAll(strGen) { str =>
      println(str)
      str.reverse.reverse should be (str)
    }
  }
}
```

- The `strGen` will generate random length, random character strings and run several of them through the test each time it is run.

Testing with Futures

- This should fail:

```
// using...
def calc(n: Int, iterations: Int): Future[Double]

test("Calculating PI Asynchronously") {
  import ExecutionContext.Implicits.global
  val calcPi = new CalcPi

  val resultF = calcPi.calc(500, 1000000)

  for (piBy4 <- resultF) yield {
    println(piBy4*4)
    piBy4*4 should be (10.0 +- 0.001)
  }
}
```

- But this passes, and never prints anything, the test finishes before the future is ready...

Waiting for the Future

- One solution is simply to Await:

```scala
test("Calculating PI Asynchronously") {
  import ExecutionContext.Implicits.global
  val calcPi = new CalcPi

  val resultF = calcPi.calc(500, 1000000)

  val piBy4 = Await.result(resultF, 1.minute)

  println(piBy4*4)
  piBy4*4 should be (10.0 +- 0.001)
}
```

- Now it fails: `3.1414943040000063 was not 10.0 plus or minus 0.001`
- Blocking is OK in main methods and tests, but there are still better options for tests...

whenready

```
import org.scalatest._
import org.scalatest.concurrent._
import PatienceConfiguration._

class TestingSuiteDemo extends FunSuite with Matchers with ScalaFutures {
  test("PI when ready") {
    val calcPi = new CalcPi

    val resultF = calcPi.calc(500, 1000000)

    whenReady(resultF, Timeout(1.minute)) { result =>
      result * 4 should be (3.141 +- 0.001)
    }
  }
}
```

Or… If Everything Being Tested is a Future

```scala
import org.scalatest._

class AsyncTestingSuiteDemo extends AsyncFunSuite with Matchers {
  test ("Calculating PI") {
    val calcPi = new CalcPi

    val resultF = calcPi.calc(500, 1000000)

    for (result <- resultF) yield {
      println(result * 4)
      result * 4 should be (3.141 +- 0.001)
    }
  }
}
```

- Async tests must result in a `Future` (so use `for...yield` usually)
- While writing the tests you can end with `succeed` to stop everything turning red

Exercises for Module 11

- Find the `Module11` class and follow the instructions to make the tests pass
- `Module11` is under `module11/src/test/scala/koans`
- Unlike previous exercises where you had to get the tests to pass, this time you have to write the tests