Scala Properties and State

Knowing when and how to safely use statefulness

Agenda

1. Principle of Uniform Access
2. Scala modifiers
3. Custom Properties
4. Role of Scopes and Scope Modifiers
5. Dangers of Statefulness
6. Caching
7. Best Practices When Using State

Principle of Uniform Access

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation -- Bertrand Meyer

Paraphrased: There should be no syntactical difference between working with an attribute, pre-computed property, or method/query of an object.

```
val conversionKilosToPounds = 2.20462262185

case class Potato(weightInKilos: Double) {
  val weightInPounds = weightInKilos * conversionKilosToPounds
}

case class Person(name: String, weightInPounds: Double) {
  def weightInKilos = weightInPounds / conversionKilosToPounds
}
```

val vs def

```
val p1 = Potato(0.75)
// > p1: Potato = Potato(0.75)

p1.weightInKilos
// > res0: Double = 0.75

p1.weightInPounds
// > res1: Double = 1.6534669663875001
```

```
val p2 = Person("Fred", 120)
// > p2: Person = Person(Fred,120.0)

p2.weightInPounds
// > res2: Double = 120.0

p2.weightInKilos
// > res3: Double = 54.43108439996978
```

- Both property calls identical
- `def` evaluated each time, `val` evaluated once

Mutable Fields and Modifiers

```
class Person(val name: String, var weightInPounds: Double) {

  def weightInKilos: Double = weightInPounds / conversionKilosToPounds

  def weightInKilos_=(newWeight: Double): Unit = {
    weightInPounds = newWeight * conversionKilosToPounds
  }

}
```

- `var weightInPounds: Double` declares a public mutable field in the class definition (and constructor)
- `def weightInKilos: Double` is a standard property accessor (note, to make this a `val` would be a bug!)
- `def weightInKilos_=(x: Double): Unit` is the way to create a *modifier* method in Scala

Mutable Fields and Modifiers

```
val fred = new Person("Fred", 120)

fred.weightInPounds
// > res0: Double = 120.0

fred.weightInKilos
// > res1: Double = 54.43108439996978

fred.weightInKilos = 60

fred.weightInKilos
// > res2: Double = 60.0

fred.weightInPounds
// > res3: Double = 132.277357311

fred.weightInPounds = 125

fred.weightInPounds
// > res4: Double = 125.0

fred.weightInKilos
// > res5: Double = 56.69904624996852
```

Behind the Scenes

- Scala uses re-writing rules for mutable fields and modifiers:

```
class Person(nm: String, wt: Double) {

  def name: String = nm

  private[this] var wtIbs = wt

  def weightInPounds: Double = wtIbs

  def weightInPounds_=(newWeight: Double): Unit = {
    wtIbs = newWeight
  }

  def weightInKilos: Double = weightInPounds / conversionKilosToPounds

  def weightInKilos_=(newWeight: Double): Unit = {
    weightInPounds = newWeight * conversionKilosToPounds
  }

}
```

- Apart from using less readable variable names, this is what Scala re-writes our original code to

Property modification re-writing

- Likewise, the property modifications get re-written:

```
val fred = new Person("Fred", 120)

fred.weightInPounds
// > res0: Double = 120.0

fred.weightInKilos
// > res1: Double = 54.43108439996978

fred.weightInKilos_=(60)

fred.weightInKilos
// > res3: Double = 60.0

fred.weightInPounds
// > res4: Double = 132.277357311

fred.weightInPounds_=(125)

fred.weightInPounds
// > res6: Double = 125.0

fred.weightInKilos
// > res7: Double = 56.69904624996852
```

The re-writing rules

- A field or parametric field like `val foo: Int = 0` gets re-written to:

```
private[this] val _foo: Int = 0  // must use different name
def foo: Int = _foo              // and this is why...
```

- A field or parametric field like `var bar: Double = 0.0` gets re-written to:

```
private[this] var _bar: Double = 0.0
def bar: Double = _bar
def bar_=(d: Double): Unit = { _bar = d }
```

- A modifier of the form:

```
baz.bar = 1.0
```

  is re-written by the compiler to

```
baz.bar_=(1.0)
```

  calling the `bar_=` modifier method with the value after the `=`

private[this]

- `private[this]` means that only this **instance** can access the field
- Even inner classes or companion objects cannot access it
- Every public, protected or private field in a Scala class gets a `private[this]` field
- Accessors and modifiers are then generated to access those fields with more permissive scopes
- `val`s, `var`s and `def`s share the same namespace, so must have different names so as not to conflict (this includes `private[this]`)
- Otherwise:

```
class Foo {
  private[this] val yo: String = "Yo"
  def yo: String = "Hello"  // will not compile...

  def greet(name: String): String =
    s"$yo $name"  // which yo would it call?
}
```

Abstract Properties

If you make the properties mutable and abstract, like this:

```
trait HeightAndWeight {
  var height: Double
  var weight: Double
}
```

Scala will generate:

```
trait HeightAndWeightAsGenerated {

  def height: Double
  def height_=(d: Double): Unit

  def weight: Double
  def weight_=(d: Double): Unit
}
```

No actual fields are generated for abstracts, only accessors and modifiers

Custom Properties

Knowing this, overriding the properties just means filling in those methods:

```
class Person(val name: String) extends HeightAndWeight {
  private[this] var ht: Double = _
  private[this] var wt: Double = _

  def height: Double = ht
  def height_=(h: Double): Unit = {
    require(h > 0.0, "Height may not be zero or negative")
    ht = h
  }

  def weight: Double = wt
  def weight_=(w: Double): Unit = {
    require(w > 0.0, "Weight may not be zero or negative")
    wt = w
  }
}
```

Custom Properties

With this, setting a property may now result in an exception:

```
fred.weight
// > res2: Double = 0.0

fred.weight = 65.0
// > fred.weight: Double = 65.0

fred.weight
// > res3: Double = 65.0

fred.weight = -5.0
// > java.lang.IllegalArgumentException: requirement failed:
//   Weight may not be zero or negative
```

Without Any Backing Fields

Your implementation may provide any implementation at all that satisfies the type signature:

```
class TruckLoad extends HeightAndWeight {
  import scala.collection.mutable
  private[this] val propsMap = mutable.Map.empty[String, Double]

  def height: Double = propsMap.getOrElse("height", 0.0)
  def height_=(h: Double): Unit = propsMap("height") = h

  def weight: Double = propsMap.getOrElse("weight", 0.0)
  def weight_=(w: Double): Unit = propsMap("weight") = w

}
```

Instead of a simple backing map here, the state could be stored in a database, web service, or may actually not save/restore state at all (though that could be confusing)

State can bleed

```
class Person(val name: String, var weight: Double) {
  override def toString: String = s"Person($name, $weight)"
}
```

```
val alice = new Person("Alice", 123)
val bob = new Person("Bob", 124)

val all = Seq(alice, bob)

def heaviestPerson(people: Seq[Person]): Person =
  people.maxBy(_.weight)
```

```
heaviestPerson(all)
// > res0: Person = Person(Bob, 124.0)

bob.weight = 122
// > bob.weight: Double = 122.0

heaviestPerson(all)
// > res1: Person = Person(Alice, 123.0)
```

- Mutable state can have far-reaching unintended consequences

Caching

- Imagine a slow temperature lookup service:

```
def fakeWeatherLookup(wxCode: String) = {
  Thread.sleep(1000)
  wxCode.toList.map(_.toInt).sum / 10.0
}
```

- Could use a Scala `Map` with your own thread-safety
- Could use a Java `ConcurrentHashMap`
- My advice: use Google's Guava with Futures

Guava Cachebuilder with Futures

```
import com.google.common.cache.{CacheLoader, CacheBuilder}
import scala.concurrent._
import duration._
import ExecutionContext.Implicits.global

object FakeWeatherLookup {
  private val cache = CacheBuilder.newBuilder().
    build {
      new CacheLoader[String, Future[Double]] {
        def load(key: String) = Future(fakeWeatherLookup(key))
      }
    }

  def apply(wxCode: String) = cache.get(wxCode)
}
```

- Futures avoid "key" lock
- Guava has soft/weak references support, eviction, timeouts, maximum sizes, etc.
- Once in Future async space, stay there as long as you can

Best Practices

- Don't use mutable state
- If you do use mutable state, document it well, scaladoc and overview
  - Users will typically expect immutability, you will cause bugs
- Minimize scope of anything mutable
- Migrate to immutable as soon as possible
  - e.g. mutable builder -> immutable result
- For in-memory caching, Guava is a solution for 95%
  - Couple with futures to avoid key-update lock on slow ops