escalate SOFTWARE

Best Practices and Idioms

Good Ideas, Style and Fashion in Scala

Agenda

1. On Mutability and Immutability
2. Nulls vs Options
3. May the Fors Be With You
4. Untying Unit
5. Try/Either/Or
6. Getting Loopy About Recursion
7. Mastery of Collections
8. The Case for `case` classes
9. If You Use a `match`
10. Fancy a Curry
11. Classy Implicits
12. Operation operator
13. Beware the by-names

On Mutability and Immutability

> If a tree falls in a forest and no one is around to hear it, does it make a sound? -- George Berkeley

> If you use mutability carefully and it does not escape its scope, does anyone know? -- Dick Wall

```
def fibs(ct: Int): Seq[Int] = {
  var n1 = 0
  var n2 = 1
  for (i <- 1 to ct) yield {
    val x = n1 + n2
    n1 = n2
    n2 = x
    x
  }
}

fibs(10)
// res0: Seq[Int] = Vector(1, 2, 3, 5, 8, 13, 21, 34, 55, 89)
```

Mutability++

But if you can make it immutable anyway, then why not?:

```
def facts(ct: Int): Seq[Long] = {
  var f = 1L
  for (i <- 1 to ct) yield {
    f = f * i
    f
  }
}
facts(10)
// res1: Seq[Long] = Vector(1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)
```

vs

```
@tailrec
def facts2(limit: Int, ct: Int = 2, acc: Vector[Long] = Vector(1L)): Seq[Long] =
  if (ct > limit) acc else facts2(limit, ct + 1, acc :+ (acc.last * ct))

facts2(10)
// res2: Seq[Long] = Vector(1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)
```

- Can you do this for the Fibonacci example?

Nix your Nulls

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965...
>
> This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. -- Tony Hoare

```
val s1: String = "Hello"
val s2: String = null

val l1: Int = s1.length  // fine
val l2: Int = s2.length  // oops - runtime error
```

None is the new null

```
val os1: Option[String] = Some("Hello")
val os2: Option[String] = None

val ol1: Option[Int] = os1.map(_.length) // Some(5)
val ol2: Option[Int] = os2.map(_.length) // None
```

Smooth Options

Using Options with `for` expressions is really a sweet spot

```
case class Address(street: String, city: String, state: String, zipCode: String)
case class Person(first: String, last: String, address: Option[Address])

val p1 = Some(Person("Fred", "Bloggs", None))
val p2 = Some(Person("Simon", "Jones",
        Some(Address("123 Main", "Fakesville", "AZ", "12345"))))
val p3: Option[Person] = None

scala> for (p <- p1; a <- p.address) yield a.zipCode
res3: Option[String] = None

scala> for (p <- p2; a <- p.address) yield a.zipCode
res4: Option[String] = Some(12345)

scala> for (p <- p3; a <- p.address) yield a.zipCode
res5: Option[String] = None

scala> p2.flatMap(_.address.map(_.zipCode))
res6: Option[String] = Some(12345)
```

Options and Collections

If you find yourself mixing collections and options in a for expression it's a good idea to use `.toSeq` on all of the options to turn them into collections.

```
val optPeople = Some(Seq(p1, p2, p3))

for {
  people <- optPeople.toSeq
  personOpt <- people
  person <- personOpt.toSeq
  address <- person.address.toSeq
} yield address.zipCode
```

If you don't do this you will sometimes see an error like:

```
Error:(43, 14) type mismatch;
 found   : Seq[String]
 required: Option[?]
  personOpt <- people
```

Try writing the equivalent expression out using flatMap and map only and the incompatible types will become clear.

May the `fors` be with you

> Nothing makes you more popular at parties than a thorough knowledge of the Scala for expression -- Dick Wall

```
val forLineLengths =
  for {
    file <- filesHere
    if file.getName.endsWith(".sc")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(".*for.*")
  } yield trimmed.length
```

The four Gs of for expressions:

- **G**enerator: `file <- filesHere`, denoted by the `<-`, all generators in the same for block must be of the same type (e.g. a collection, or a Try).
- **G**uard: `if file.getName.endsWith(".sc")`, guards short-circuit the for expression, causing filtering behavior.
- inline assi**G**nment: `trimmed = line.trim`, a simple val expression that may be used in both the remainder of the for block, and in the yield block.
- **G**ive (or the yield), after the `yield` keyword comes the payoff of the for block setup.

For is for more than just looping

E.g. Futures:

```
import scala.concurrent._
import duration._
import ExecutionContext.Implicits.global

val f1 = Future(1.0)
val f2 = Future(2.0)
val f3 = Future(3.0)

val f4 = for {
  v1 <- f1
  v2 <- f2
  v3 <- f3
} yield v1 + v2 + v3

Await.result(f4, 10.seconds)
```

- Curly braces for semicolon inference
- Easy asynchronous programming

Untying Unit

- Any expression resulting in `Unit` must have a side effect to do anything useful
- Expressions/common methods that produce `Unit`:
  - `while` loops
  - `.foreach`
  - `for` without `yield`
  - `def someMethod(i: Int) { i * i }` (i.e. forgotten =)
- From the above, the last two are common mistakes
- While absence of `Unit` does not imply functional, `Unit` makes it certainly not.

```
def someMethod(i: Int) { i * i }
```

is re-written by Scala to:

```
def someMethod(i: Int) { i * i; () }
```

Try/Either/Or, Try...

- Exceptions interrupt control flow, this is not functional behavior
- Try gives a more functional style alternative:

```
val d1 = Try(2/1)  // Success(2)
val d2 = Try(2/0)  // Failure(java.lang.ArithmeticException: / by zero)

val m1 = for (x <- d1) yield x * 2  // Success(4)
val m2 = for (x <- d2) yield x * 2  // Failure(java.lang.ArithmeticException...)

m1.isFailure  // false
m2.isFailure  // true

m1.getOrElse(0)  // 4
m2.getOrElse(0)  // 0

m1.get           // 4
m2.get           // throws ArithmeticException
```

Or Either...

- Try is a specific case of type T or a `Throwable`
- If you want an exception more specific than Throwable (or another type entirely), consider `Either`:

```
val e1: Either[ArithmeticException, Int] = Right(2)
val e2: Either[ArithmeticException, Int] = Left(new ArithmeticException)

val em1 = for (x <- e1) yield x * 2  // Right(4)
val em2 = for (x <- e2) yield x * 2  // Left(ArithmeticException)

em1.right.getOrElse(0)  // 4
em2.right.getOrElse(0)  // 0
```

- `Either` is right biased since Scala 2.12
- Convention says the "right" answer goes on the Right, failure on the Left

Either those or try Or

```
import org.scalactic._

def makeInt(str: String): Int Or ErrorMessage = {
  try {
    Good(str.toInt)
  } catch {
    case NonFatal(_) => Bad("Invalid Number: " + str)
  }
}
```

```
val or1 = makeInt("1")  // Good(1)
val or2 = makeInt("uno") // Bad(Invalid Number: uno)

for (x <- or1) yield x * 2  // Good(2)
for (x <- or2) yield x * 2  // Bad(Invalid Number: uno)
```

- Or can accumulate errors, for full validation behavior
- http://www.scalactic.org/user_guide/OrAndEvery

Getting Loopy About Recursion

Loopy loops!

```
import scala.collection.mutable

def factSeq(n: Int): List[Long] = {
  def fact(v: Int): Long = {
    var prod = 1L
    for (i <- 1 to v) prod *= i
    prod
  }
  val buf = mutable.ArrayBuffer.empty[Long]
  for (i <- 1 to n) buf.append(fact(i))
  buf.toList
}

factSeq(8)  // List(1, 2, 6, 24, 120, 720, 5040, 40320)
```

- No mutability escapes, but do we need any at all?

Radical Recursion

```scala
@scala.annotation.tailrec
def factSeqFun(lim: Int, cur: Int = 2,
               xs: List[Long] = List(1L)): List[Long] =
    if (cur > lim) xs.reverse else
        factSeqFun(lim, cur + 1, cur * xs.head :: xs)

factSeqFun(8)  // List(1, 2, 6, 24, 120, 720, 5040, 40320)
```

- `tailrec` ensures tail call optimization
- Default parameters used to allow same API with recursion
- New items added to head of list (constant time)
- Termination reverses list to keep order like original

Mastery of Collections

- The Scala collections API will likely be your most used API, therefore to improve at Scala, improve with the collections API.
- Some of the more interesting methods: `count`, `diff`, `intersect`, `union`, `distinct`, `head`, `headOption`, `tail`, `take`, `drop`, `dropWhile`, `exists`, `filter`, `filterNot`, `find`, `foldLeft`, `reduceLeft`, `foldRight`, `reduceRight`, `forall`, `groupBy`, `grouped`, `hasDefiniteSize`, `isTraversableAgain`, (`indexOf` functions), `max`, `min`, `product`, `sum`, `mkString`, `isEmpty`, `nonEmpty`, `partition`, `patch`, `sameElements`, `slice`, `updated`, `sliding`, `sortBy`, `sortWith`, `sorted`, `span`, `splitAt`, `startsWith`, `endsWith`, `zip`, `unzip`, `zipWithIndex`, `view`, `withFilter`, (`conversions`) etc.

Inference vs Explicit Typing

What is the return type of this function?

```
def safeDiv(n: Int, d: Int) =
    if (d == 0) None else n / d
```

Inference vs Explicit Typing

What is the return type of this function?

```
def safeDiv(n: Int, d: Int) =
    if (d == 0) None else n / d
```

in use...

```
List(0,1,2).map(safeDiv(5, _)).flatten
error: could not find implicit value for parameter asTraversable:
  (Any) => Traversable[B]
       res7.flatten
```

If we explicitly typed the method

```
def safeDiv(n: Int, d: Int): Option[Int] =
    if (d == 0) None else n / d
error: type mismatch;
 found   : Int
 required: Option[Int]
           if (d == 0) None else n / d
```

The Case for `case` classes

Case classes offer much for free, and are often preferable to just using tuples (for no extra cost)

```
def calc(al1: Allele, al2: Allele): (Double, Double, (Boolean, Boolean))

val (r2, dprime, flags) = calc(a1, a2) // what are the flags?

// vs.

case class LDResult(r2: Double, dPrime: Double, valid: Boolean, linked: Boolean)

def calc(al1: Allele, al2: Allele): LDResult

val ldResult = calc(a1, a2)
ldResult.valid     // etc.

val LDResult(r2, dPrime, valid, linked) = calc(a1, a2)
```

In addition to richer field names, you get a new specific type as well

If You Use a `match`

Instead of if expressions...

```
def describeSign(n: Int): String =
    if (n == 0) "Zero" else
        if (n > 0) "Positive" else
          "Negative"
```

try matches with guards...

```
def describeSign(n: Int): String = n match {
    case 0 => "Zero"
    case v if v > 0 => "Positive"
    case _ => "Negative"
}
```

lines up nicely with `case class` uses:

```
def ldLinked(ldResult: LDResult): String = ldResult match {
  case LDResult(_, _, _, true) => "linked"
  case _ => "not linked"
}
```

Fancy a Curry?

Scala allows multiple parameter lists, often referred to as curried parameter lists. These can be effective when used with functions:

```
def times(n: Int)(fn: => Unit): Unit =
  for (_ <- 1 to n) fn

times(5) {
  println("hello")
}
```

Or:

```
def withLinesFromFile[A](file: File)(fn: Seq[String] => A): A = {
  val src = Source.fromFile(file)
  try {
    fn(src.getLines().toSeq)
  } finally src.close()
}

withLinesFromFile(new File("somefile.txt")) { lines =>
  lines.length
}
```

Classy Implicits

Favor implicit classes over implicit conversions when possible, for example:

```
implicit class TimesDo(n: Int) {
  def times(fn: => Unit): Unit =
    for (_ <- 1 to n) fn
}

5 times {
  println("hello")
}
```

or use AnyVal classes if you can...

```
implicit class TimesDo(val n: Int) extends AnyVal {
  def times(fn: => Unit): Unit =
    for (_ <- 1 to n) fn
}
```

Remember that Scala makes a hidden implicit conversion when you do this, so implicit classes can only go where `defs` can be used.

Smooth Operator

Scala does not have operator overloading...

It does have symbolic method names...

```
case class Rational(n: Int, d: Int) {
  require(d != 0)
  def this(n: Int) = this(n, 1) // auxiliary constructor
  override def toString = s"$n/$d"
  def +(that: Rational): Rational =
    Rational(
      n * that.d + that.n * d,
      d * that.d
    )
}

Rational(1, 2) + Rational(2, 3)  // 7/6
```

It also has defined precedence based on the first symbol:

http://scala-lang.org/files/archive/spec/2.12/06-expressions.html#infix-operations

Abusing Operators

- Would you call a method `sdlgkjlgkjerg`?
- Then why would you call an operator `|-->` or `<++=` or `|@|`?
- Use operators only when their meaning is obvious to your library user, otherwise spell it out with a name.

```
val optRes1 = optv1 |@| optv2 |@| optv3

// vs:
import Options
val optRes2 = Options.zip(optv1, optv2, optv3)
```

- I know which I find more readable (and easier to type!)
- In the 1.0 RC of Cats, `|@|` has been deprecated in favor of a simple syntax: `(o1, o2).mapN(_ + _)`

Beware the by-names

By-name functions can be accidentally invoked:

```
case class FruitLoopStep(p: Boolean, fn: Unit) // oops, do you see it?

@tailrec
def fruitLoopInner(fl: FruitLoopStep): Unit =
  if (fl.p) {
    fl.fn
    fruitLoopInner(fl)
  }

def fruit(p: => Boolean)(fn: => Unit) =
  fruitLoopInner(FruitLoopStep(p, fn))

var i = 0
// if we change the test to < 1 now?
fruit(i < 0) {
  println(i * i)
  i += 1
}
```

This is based on a real bug observed in ScalaZ code.