artima

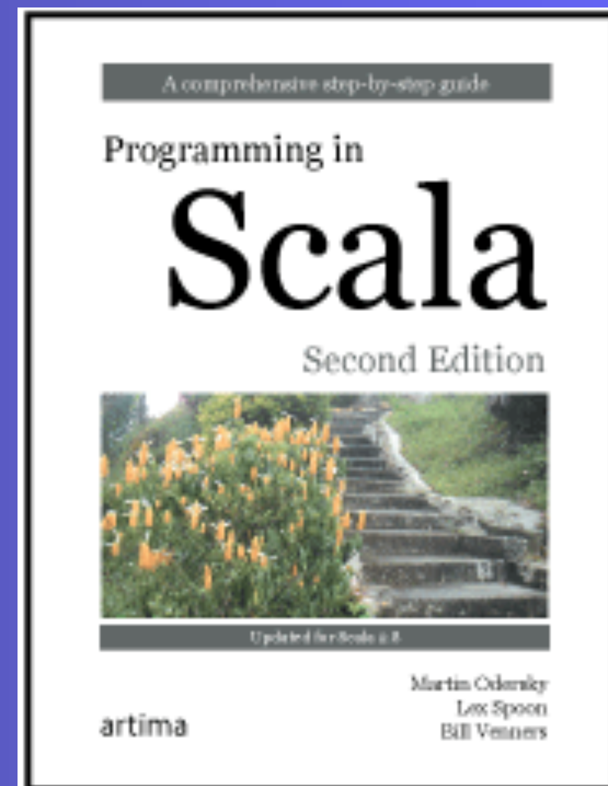
ScalaDays 2011

Effective Scala

Bill Venners
Dick Wall

www.artima.com

Copyright (c) 2010-2011 Artima Inc. All Rights Reserved.



Effective Scala goal

Examine some of the idioms and "best practices" that are becoming more commonplace in Scala development.

Favor Immutability

- val vs var
- scala.collection.immutable vs scala.collection.mutable

```
val numsToStars = {
    var nToS = Map.empty[Int, String]
    for (i <- 0 to 10) nToS += i -> "*" * i
    nToS
}

// or
val numsToStars2 = Map.empty ++ (0 to 10).map { i =>
    (i, "*" * i)
}
```

Favor Immutability Ctd.

```
def buildList(maxNum: Int): List[String] = {
  val newList = mutable.ListBuffer.empty[String]
  for (i <- 1 to maxNum) {
    newList += "*" * i
  }
  newList.toList   // turn the mutable into immutable
}

scala> buildList(5)
res5: List[String] = List(*, **, ***, ****, *****)
```

Avoid Nulls

```
scala> var s: String = _    // Danger, Will Robinson
s: String = null

scala> val s: Option[String] = None
s: Option[String] = None

scala> val s2 = Some("Hello, World!")
s2: Option[String] = Some(Hello, World!)
```

Idiomatic Options

```
case class Address(street: String, city: String, state: String, zip: 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.zip
res3: Option[String] = None
scala> for (p <- p2; a <- p.address) yield a.zip
res4: Option[String] = Some(12345)
scala> for (p <- p3; a <- p.address) yield a.zip
res5: Option[String] = None
scala> p2.flatMap(_.address.map(_.zip))
res6: Option[String] = Some(12345)
```

Know Your For Expressions

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

- Use {} instead of () for semicolon inference
- Inline assignments, guards, nested <-
- Can be combined with idiomatic Option handling too

Consider map or flatMap above foreach

```
(1 to 10).foreach { i => println("*" * i) }

val stars = (1 to 10).map { i => "*" * i }
println(stars.mkString("\n"))

val sumOfSquares = (1 to 10).map { i => i * i }.sum

val lst = List(None, Some("hey"), None, Some("nonny"))
scala> lst.flatten
res4: List[java.lang.String] = List(hey, nonny)
scala> lst.flatten.map { s => s.toList }
res5: List[List[Char]] = List(List(h, e, y), List(n, o, n, n, y))
scala> lst.flatten.flatMap { s => s.toList }
res6: List[Char] = List(h, e, y, n, o, n, n, y)
```

Consider map or flatMap (ctd.)

```
val wl = List("horse", "after", "found", "snarf", "glean")

scala> for (i <- 0 to 4) yield wl(i).toList(i)
res8: IndexedSeq[Char] = Vector(h, f, u, r, n)

scala> wl.zipWithIndex.map { case (w, i) => w.toList(i) }
res10: List[Char] = List(h, f, u, r, n)
```

Consider Either as Alternative to Exceptions

```
scala> def makeInt(str: String): Either[String, Int] = {
          try {
             Right(str.toInt)
          } catch {
             case _ => Left("Invalid Number: " + str)
          }
       }

makeInt: (str: String)Either[String,Int]
```

Using Either as Alternative to Exceptions

```
scala> val a = makeInt("5")
a: Either[String,Int] = Right(5)
scala> val b = makeInt("fred")
b: Either[String,Int] = Left(Invalid Number: fred)

scala> a.fold(error => error, number => number * 2)
res1: Any = 10
scala> b.fold(error => error, number => number * 2)
res2: Any = Invalid Number: fred

scala> makeInt("5").right.map(_ * 10)
res6: Product with Either[String,Int] with Serializable = Right(50)

scala> makeInt("fred").right.map(_ * 10)
res7: Product with Either[String,Int] with Serializable = Left(Invalid
Number: fred)
```

Consider Alternatives to Loops and Mutables

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

scala> factSeq(8)
res13: List[Long] = List(1, 2, 6, 24, 120, 720, 5040, 40320)
```

Favor Tail Recursion, and Immutables

```
// Recursion and List cons
@scala.annotation.tailrec
def factSeq(lim: Int, cur: Int = 2,
                  l: List[Long] = List(1L)): List[Long] =
    if (cur > lim) l.reverse else
        factSeq(lim, cur + 1, cur * l.head :: l)

scala> factSeq(8)
res14: List[Long] = List(1, 2, 6, 24, 120, 720, 5040, 40320)
```

Know Your Collections

- Really knowing collections is time well spent
- e.g. LinearSeq: 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.

Consider Always Providing Return Types on Functions and Methods

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

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

Consider case classes for more than match

- Remember what you get:
  - toString, apply, unapply, properties, equals, hashCode, copy, etc.
- You can override these as desired
- You can define other methods, state, like regular class
- Remember when you should avoid them:
  - Inheritance heirarchies of case classes
  - Want superclass equals/hashcode behavior instead
  - Want constructor params but not properties

Favor Case Classes over Return Tuples

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

val (r2, dprime, flags) = calc(a1, a2)   // eww - and 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.
```

Favor Traits Over Classes When Possible

```
class Car { val wheels = 4 }
class Motorcycle { val wheels = 2 }
abstract class ElectricVehicle { def noise() = "Whizz" }
abstract class GasVehicle { def noise() = "Vroom" }
class SportsBike extends Motorcycle with GasVehicle //oops

trait Car { val wheels = 4 }
trait Motorcycle { val wheels = 2 }
trait ElectricVehicle { def noise() = "Whizz" }
trait GasVehicle { def noise() = "Vroom" }
class SportsBike extends Motorcycle with GasVehicle //fine
```

Trait Rules and Notes

- Traits can't take parameters
- Can Get Trait Instance: new Object with Motorcycle...
  - But typically use classes if you want an instance
- Trait linearization: Scaladoc
- Traits can specify what they may be mixed in to:

```
trait Database { def newSession() {...} }
trait Transaction { this: Database =>  // requires Database
  def begin() = newSession().beginTransaction()
}
```

Consider match Over if..else if..else

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


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

artima

Function Literals and Closures

General Tips:

- Try to avoid closures around mutable state
  - Consider copying state to an immutable first
- Higher order functions are a boilerplate buster
- By-name functions can be pretty in use
- Closures are not serializable
- Can provide an implicit conversion from interfaces
- Commonly used functions may be a candidate for object method definitions

Consider Currying for Function Parameters

```
def doTimes(fn: => Unit, n: Int) {
  for (i <- 1 to n) { fn }   // do it n times
}

doTimes(println("Hello, World!"), 5)  // multi-line functions?

def doTimes(n: Int)(fn: => Unit) {
  for (i <- 1 to n) { fn }   // do it n times
}

doTimes(5) {
  println("Multi-line luxury")
  println("Hello, World!")
}
```

Object Methods for Common Functions

```
val isEven: Int => Boolean = (x: Int) => x % 2 == 0

// vs

object Filters {
  def isEven2(x: Int): Boolean = x % 2 == 0
}

scala> List(1,2,3,4,5,6).filter(isEven)
res3: List[Int] = List(2, 4, 6)
scala> import Filters._
scala> List(1,2,3,4,5,6).filter(isEven2)
res2: List[Int] = List(2, 4, 6)
```

Know and Use Scala Annotations

- @scala.volatile
- @scala.throws(classOf[IOException])
- @scala.annotation.tailrec
- @scala.specialized
- @scala.reflect.BeanInfo, @scala.reflect.BeanProperty
- @scala.native
- @scala.serializable // DEPRECATED NOW
- @scala.transient
- @scala.remote

Other Tips and Tricks

- Syntax Coloring in IDE (e.g. turn vars red)
- Pair program/review with another Scala developer
- Sweep through existing code as a "relaxation" exercise
- Look for patterns that work well in your own code
- More importantly, look for anti-patterns, spread the word