Typesafe

Effective 

Henrik Engström

Software Engineer at Typesafe

Arsenal supporter 

@h3nk3

What we do @ Typesafe

- Scala, Akka, Play!, Scala IDE, SBT, Slick, etc.
- Open Source under Apache License v2
- Akka - Scala *and* Java API
- Play! - Scala *and* Java API
- Subscriptions are how we roll

Effective Scala is

“Optimizing your use of the Scala programming language to solve real-world problems without explosions, broken thumbs or bullet wounds”

- Josh Suereth

Agenda

- Basic Stuff
- Object Orientation
- Implicits
- Type Traits
- Collections
- Pattern Matching
- Functional Programming

DA BASICZ

Typesafe

Use the REPL

Become friends with the REPL. It will become a rewarding relationship.

```
> ./scala_home/bin/scala
Welcome to Scala version 2.x ...
> println(“Hello, world!”)
Hello, world!
>
```

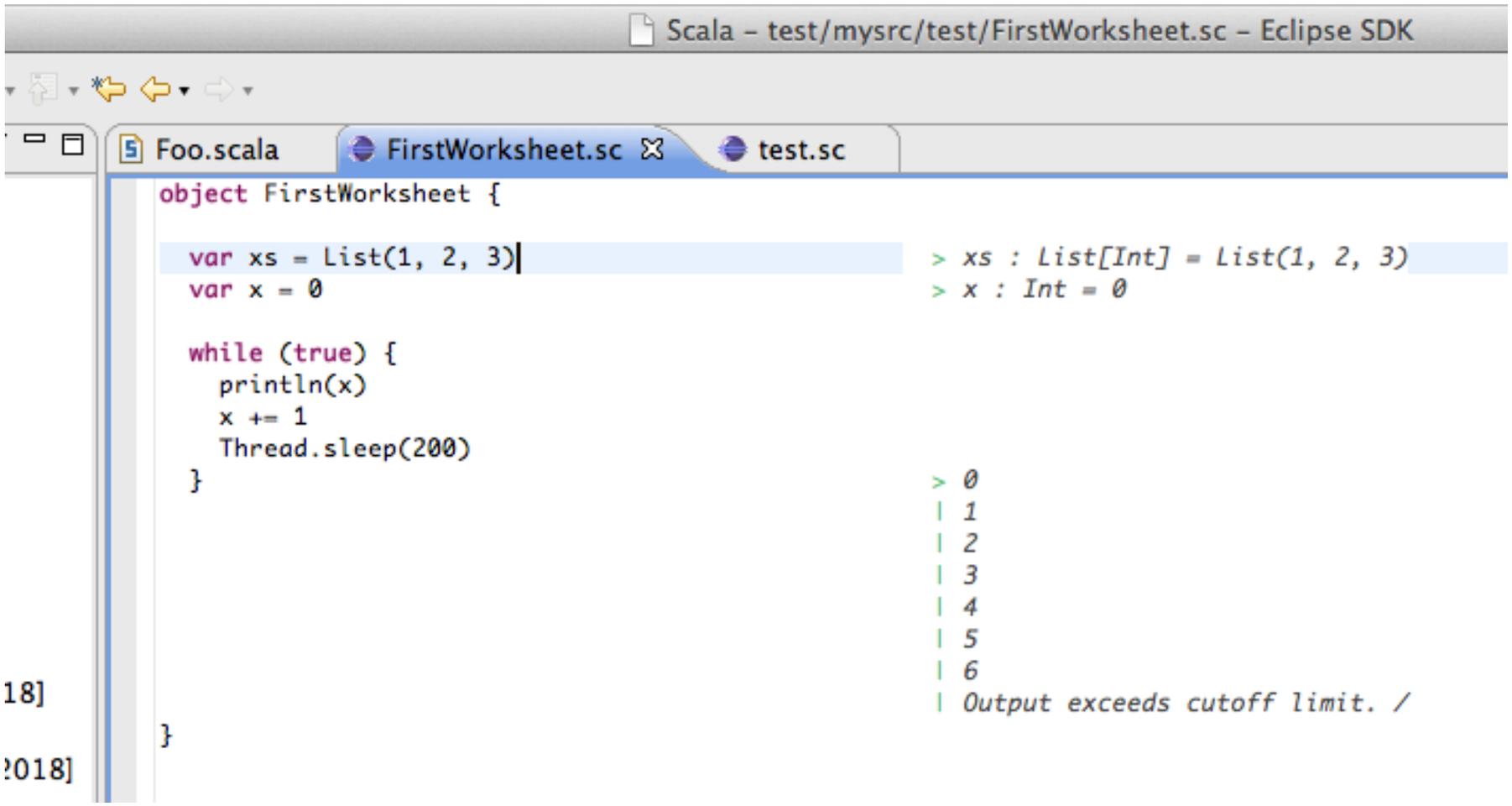
REPL and JARs

Put JARs in scala_home/lib to get access - it is an awesome way to explore a new API

```
> ./scala_home/bin/scala
Welcome to Scala version 2.x ...
> import x.y.z.TheClass
> val instance = new TheClass
> instance.theMethod
```

REPL 2013

- IDE Worksheet
  - Eclipse : awesome
  - IntelliJ : okay



Expressions & Statements

Java

```
String result = null;
if (z < 9) result = “<9” else result = “>=9”;
System.out.println(“Result:“ + result);
```

Scala

```
println(“Result:“ + if (z < 9) “<9” else “>=9”))
```

In Scala everything can be expressed as an expression

Expressions & Statements

Remove temp variables

```
val result =
  try {
    100000 / divisor
  } catch {
    case e:Exception => 0
  }

println(“Your salary is: “ + result)
```

Statement Pop Quiz

What type is variable quiz?

```
var x = 1
val quiz = while(x < 10) {
  println(“X is: “ + x)
  x += 1
}
```

Fishing Instructions

Don’t tell it how to fish!

```
def findPeopleInCity(
    city: String,
    people: Seq[People]): Set[People] = {
  val found =
    new scala.collection.mutable.HashSet[People]()
  for (p <- people) {
    for (a <- p.address) {
      if (a.city == city) {
        found.put(p)
      }
    }
  }
  found
}
```

Typesafe

Just order fish

```
def findPeopleInCity(
    city: String,
    people: Seq[People]): Set[People]) = {
  for {
    p <- people.toSet[People]
    a <- p.addresses
    if a.city == city
  } yield p
}
```

SQL like syntax:

FROM people p, address a

WHERE p.addresses = a

AND a.city = ‘London’

SELECT p

Stay Immutable

Mutable safe code => cloning

Performance degradation!

```
class FootballPlayer {
  private var cars = Array[Car]()

  def setCars(c: Array[Car]): Unit =
    cars = c.clone

  def getCars: Array[Car] =
    cars.clone
}
```

Stay Immutable

Safer code - use an immutable collection

```
class FootballPlayer {
  private var cars = Vector[Car]()

  def setCars(c: Array[Car]): Unit = { cars = c }

  def getCars: Array[Car] = cars
}
```

Case Classes

Make the whole class immutable

```
> case class Car(brand: String)
> case class FootballPlayer(name: String, team: String,
    cars: Vector[Car] = Vector.empty)

> var player = FootballPlayer(“Thierry Henry”, “Arsenal”)
> player.toString
FootballPlayer(Thierry Henry, Arsenal, Vector())

> player = player.copy(cars = Vector(Car(“Porsche”)))
> player.toString
FootballPlayer(Thierry Henry, Arsenal,
  Vector(Car(Porsche)))
```

Immutable Class Benefits

- Simple equality
- Simple hashcode
- No need to lock
- No defensive copying

Scala Case Classes

- *Automatic* equality
- *Automatic* hashcode (MurmurHash)

Local Mutability

This is almost okay... more information about Seq later

```
import scala.collection.mutable.ArrayBuffer

def theData: Seq[Int] {
  val buffer = new ArrayBuffer[Int]
  populateData(buffer)
  buffer.toSeq
}
```

Use Option

Let’s play the null game

```
def authenticate(user: String, pwd: String):
    Privileges = {
  if (user == null || pwd == null || user.isEmpty ||
      pwd.isEmpty || (!canAuthenticate(user, pwd)) {
    withPrivileges(Anonymous)
  } else {
    privilegesFor(user)
  }
}
```

Hello there “Option wall”

```
def authenticate(
    user: Option[String],
    pwd: Option[String]): Privileges = {
  val privileges: Option[Privileges] =
    for {
      u <- user
      p <- pwd
      if (!u.isEmpty && !p.isEmpty)
      if canAuthenticate(u, p)
    } yield priviligesFor(u)

  privileges getOrElse withPrivileges(Anonymous)
}
```

OBJECT

ORIENTATION

Typesafe

val for abstract members

Don’t!

```
trait SquaredShape {
  val width: Int
  val height: Int
  val area: Int = width * height
}

class Rectangle(w: Int, h: Int) extends SquaredShape {
  override val width = w
  override val height = h
}

> val r1 = new Rectangle(1, 314)
> r1.height
res0: Int = 314
> r1.area
res1: Int = 0
```

def is much better

Programmer power => val, var, def

```
trait SquaredShape {
  def width: Int
  def height: Int
  def area: Int = width * height
}

class Rectangle(w: Int, h: Int) extends SquaredShape {
  override val width = w
  override val height = h
}

// or even better

case class Rect(width: Int, height: Int)
  extends SquaredShape
```

Annotate it

Annotate API or non-trivial return types

(if not *you* have to “compile” the code)

```
def convert(x: Int) = x match {
  case 1 => 1.toChar
  case 2 => true
  case z => z.toByte
}

def convert(x: Int): AnyVal = ...
```

Document it

A common use -

document existential property

```
trait Person {
  def name: String
  def age: Option[Int]
}
```

Composition and Inheritance

- Composition preferred over Inheritance
  - easier to modify (e.g. dependency injection)
- Composition can use Inheritance in Scala
- Leads to the famous cake pattern

Let’s bake a cake

```
trait UserRepositoryComponent {
  def userLocator: UserLocator
  def userUpdater: UserUpdater
  trait UserLocator {
    def findAll: Vector[User]
  }
  trait UserUpdater {
    def save(user: User)
  }
}
```

Baking in progress

```
trait JPAUserRepositoryComponent extends
    UserRepositoryComponent {
  def em: EntityManager
  def userLocator = new JPAUserLocator(em)
  def userUpdater = new JPAUserUpdater(em)

  class JPAUserLocator(em: EntityManger) extends
      UserLocator {
    def findAll: Vector[User] =
      em.createQuery(“from User”, classOf[User]).
        getResultList.toVector
  }
  class JPAUserUpdater(em: EntityManger) extends
      UserUpdater {
    def save(user: User) = em.persist(user)
  }
}
```

Service Layer

```
trait UserServiceComponent {
  def userService: UserService
  trait UserService {
    def findAll: Vector[User]
    def save(user: User)
    def checkStatusOf(user: User): String
  }
}
```

Service Layer Impl

```
trait DefaultUserServiceComponent extends
    UserServiceComponent {
  this: UserRepositoryComponent =>
  def userService = new DefaultUserService

  class DefaultUserService extends UserService {
    def findAll = userLocator.findAll
    def save(user: User) = userUpdater.save(user)
    def checkStatus(user: User) =
      s“User $user seems okay to me”
  }
}
```

Use it

```
object TheApplication extends Application {
  val componentService =
    new DefaultUserServiceComponent with
      JPAUserRepositoryComponent {
        def em =
          Persistence.createEntityManagerFactory(
            “cake”).createEntityManager()
      }
  val service = componentService.userService
  // ...
}
```

Test it

```
class MyTest extends WordSpec with MustMatchers with
    Mockito {
  trait MockedEM {
    def em = mock[EntityManager]
    // ...
  }
  “service” must {
    “return all users” in {
      val componentService =
        new DefaultUserServiceComponent
        with JPAUserRepositoryComponent
        with MockedEM
      // Perform the test
    }
  }
}
```

Thanks to @markglh for the Cake example.

Read more great Scala stuff at www.cakesolutions.net/teamblogs

Typesafe

IMPLICITS

Typesafe

What is it good for?

- Removes boilerplate within a specific context
  - compile time safety
  - must be unambiguous

Implicits Example

```
trait AutoRepository {
  def find(regId: String)(implicit dbId: DBId):
    Option[Car]
  def findAll(country: String)(implicit dbId: DBID):
    Seq[Car]
}

class DefaultAutoRepository extends AutoRepository {
  def find(regId: String)(implicit dbId: DBId):
    Option[Car] = { // ... }
  def findAll(country: String)(implicit dbId: DBID):
    Seq[Car] = { // ... }
}
```

Implicits Example

```
class CarFinder {
  val dbId = DbId("Dealer1")

  def findCar(regId: String): Option[Car] = {
    val car = repo.find(regId)(dbId)
    // ...
  }

  def listCars(country: String): Seq[Car] = {
    val cars = repo.findAll(country)(dbId)
    // ...
  }
}
```

Implicits Example

```
class CarFinder {
  implicit val dbId = DbId(“Dealer1”)

  def findCar(regId: String): Option[Car] = {
    val car = repo.find(regId)
    // ...
  }

  def listCars(country: String): Seq[Car] = {
    val cars = repo.findAll(country)
    // ...
  }
}
```

Compiler workout

- Implicits Scope
  - Lexical
    - current scope, explicit import, wildcard imports
  - Companions of parts of the type
    - companion of types, companion of types of arguments, outer objects of nested types, package objects
- Can be expensive in compile time!
- Use it with care

Implicit Values

```
trait Logger { def log(msg: String) }

object Logger {
  implicit object DefaultLogger extends Logger {
    def log(msg: String) = println(“DL> “ + msg)
  }
  def log(msg: String)(implicit logger: Logger) = {
    logger.log(msg)
  }
}
```

Implicit Values

```
> Logger.log(“a small test”)
DL> a small test

> class MyLogger extends Logger {
  def log(msg: String) = println(“my>>> “ + msg)
}

> implicit def myLogger = new MyLogger
> Logger.log(“yet another test”)
my>>> yet another test
```

Implicits Wisdom?

deech @deech

Debugging #scala implicits is like trying to find the farter in a crowded room

TYPE TRAITS

Typesafe

a.k.a Type Classes

“describes generic interfaces using type parameters such that the implementations can be created for any type”

```
trait Encodable[T] {
  def from(t: T): String
  def to(s: String): T
}
object Encodable {
  implicit object IntEncoder extends Encodable[Int] {
    def from(i: Int): String = “int” + i
    def to(s: String): Int =
      s.substring(s.indexOf(“int”)+3, s.length).toInt
  }
}
```

Example Usage

```
class MyHandler {
  def convert[T](t: T)(implicit enc: Encodable[T]):
    String = enc.from(t)
  def convert[T](s: String)
    (implicit enc: Encodable[T]): T = enc.to(s)
}

> val myHandler = new MyHandler
> myHandler.convert(12345)
res0: String = int12345
> myHandler.convert(res0)
res1: Int = 12345
```

Example Usage

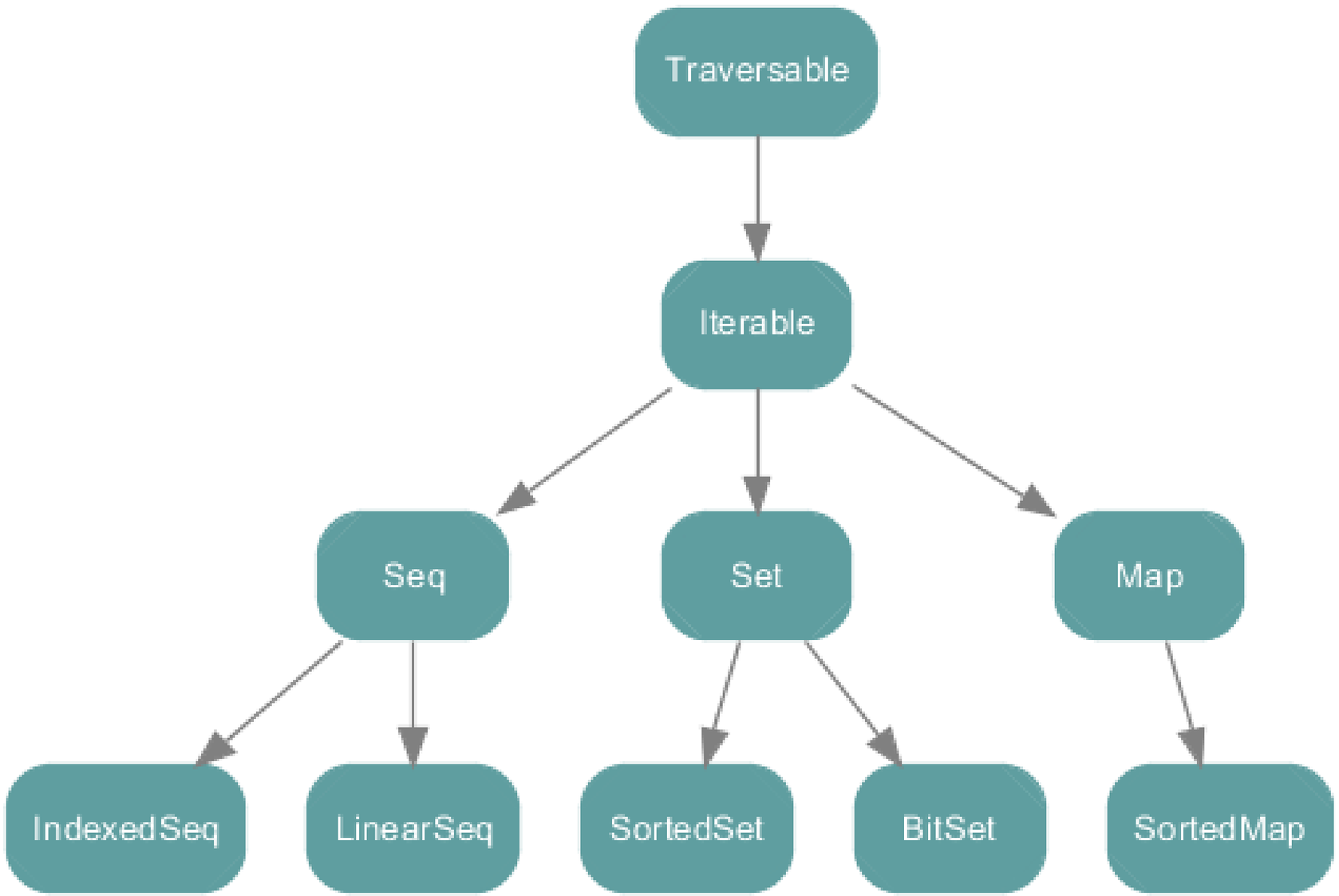
```
> myHandler.convert(12345L)
<console>:15: error: could not find implicit value for
parameter encoder: Encodable[Long]
myHandler.convert(12345L)

> implicit object LongEnc extends Encodable[Long] {
    def from(l: Long): String = “long” + l
    def to(s: String): Long = s.substring(s.indexOf(
      “long”)+4, s.length).toLong
> myHandler.convert(12345L)
res4: String = long12345
```

COLLECTIONS

Typesafe

Collections



Collections

Learn the Collections API - it is *awesome*

```
> val seq = Seq()
> seq.
++               ++:              +:               /:               /:\              :+               :\               addString        aggregate        andThen          apply            applyOrElse
asInstanceOf     canEqual         collect          collectFirst     combinations     companion        compose          contains         containsSlice    copyToArray
copyToBuffer     corresponds      count            diff             distinct         drop             dropRight        dropWhile        endsWith         exists           filter
filterNot        find             flatMap          flatten          fold             foldLeft         foldRight        forall           foreach          genericBuilder   groupBy
grouped          hasDefiniteSize  head             headOption       indexOf          indexOfSlice     indexWhere       indices          init             inits            intersect
isDefinedAt      isEmpty          isInstanceOf     isTraversableAgain iterator       last             lastIndexOf      lastIndexOfSlice lastIndexWhere   lastOption
length           lengthCompare    lift             map              max              maxBy            min              minBy            mkString         nonEmpty         orElse
padTo            par              partition        patch            permutations     prefixLength     product          reduce           reduceLeft       reduceLeftOption
reduceOption     reduceRight      reduceRightOption repr            reverse          reverseIterator  reverseMap       runWith          sameElements     scan
scanLeft         scanRight        segmentLength    seq              size             slice            sliding          sortBy           sortWith         sorted           span
splitAt          startsWith       stringPrefix     sum              tail             tails            take             takeRight        takeWhile        to               toArray          toBuffer
toIndexedSeq     toIterable       toIterator       toList           toMap            toSeq            toSet            toStream         toString         toTraversable    toVector
transpose        union            unzip            unzip3           updated          view             withFilter       zip              zipAll           zipWithIndex
```

Collections

Message to all Java developers:

use Vector not List

Vector is faster than List

Vector is more memory efficient than List

A note on Seq Collections

```
def saveStuff(a: Seq[String]): Unit
def getStuff: Seq[String]
```

Do you know where Seq comes from?

scala.collection.immutable.Seq

scala.collection.mutable.Seq

scala.collection.Seq

PATTERN MATCHING

Typesafe

FP Pattern Matching

```
@scala.annotation.tailrec
def length[A](l: List[A], len: Int): Int = l match {
  case h :: t => length(t, len + 1)
  case Nil => len
}

> length(List(1,2,3,4,5,6), 0)
res0: Int = 6
```

Extracting + instance of

```
def convertedAge(a: Animal): Int = a match {
  case Dog(name, age) => age * 7
  case Human(_, age, _) => age
  case Walrus(“Donny”, age) => age / 10
  case Walrus(name, age) if name == “Walter” => age
  case _ => 0
}
```

Typesafe

PARALLELISM

&

CONCURRENCY

Typesafe

Word of advice

Implementing correct parallel and concurrent code is difficult!

- Actors
  - the Akka ones
- Futures

scala.concurrent._

JAVA INTEGRATION

Typesafe

Write interfaces in **Java**

Prefer **Java** primitives in APIs

Typesafe

FUNCTIONAL PROGRAMMING

Typesafe

Learn patterns from Functional Programming!

Typesafe

FP - Example

API for server 1

```
trait PersonRepository {
  def getPerson(name: String): Future[Person]
  def friendsOf(person: Person): Future[Seq[Person]]
}
```

API for server 2

```
trait InterestRepository {
  def interestsOf(p: Person): Future[Seq[Interest]]
}
```

Front Page

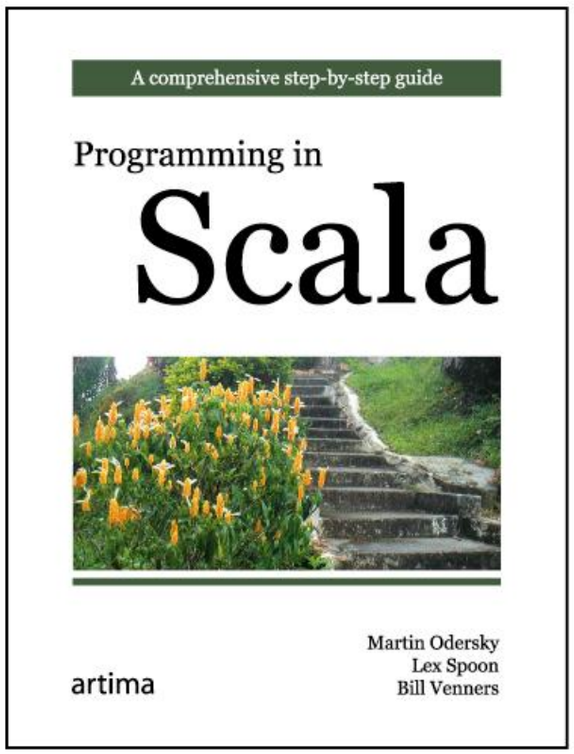
```
case class FrontPageCtx(person: Person, friends:
  Seq[Friends], interests: Seq[Interest])
```

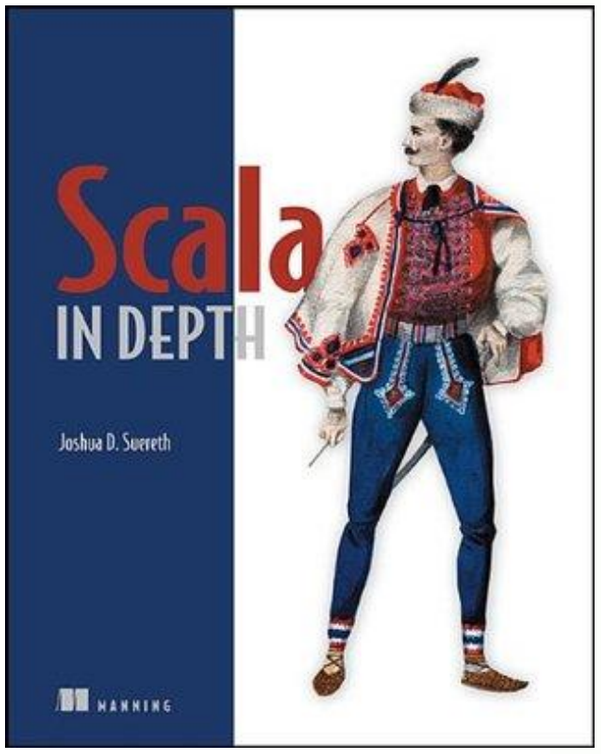
FP - Example

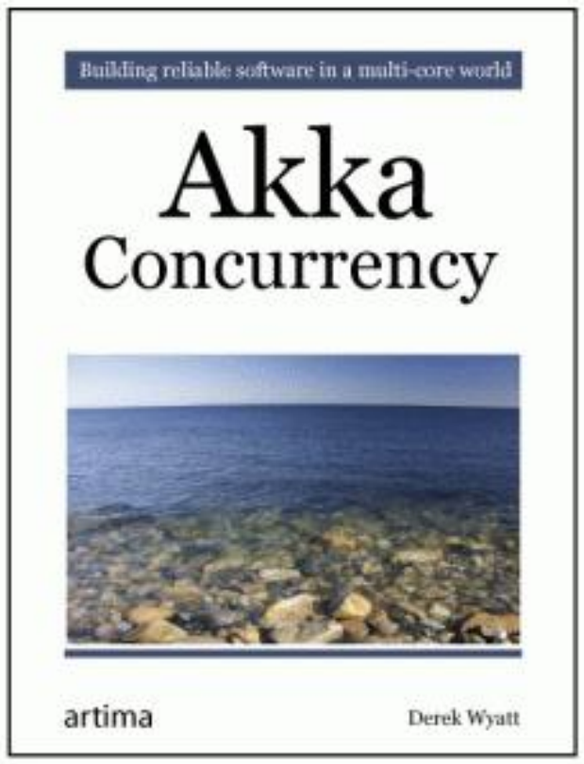
```
def getData(name: String): Future[FrontPageCtx] = {
  for {
    p <- personRepo.getPerson(name)
    (fs, is) <-
         personRepo.friendsOf(p) zip
           interestRepo.interestsOf(p)
  } yield FrontPageCtx(p, fs, is)
}
```

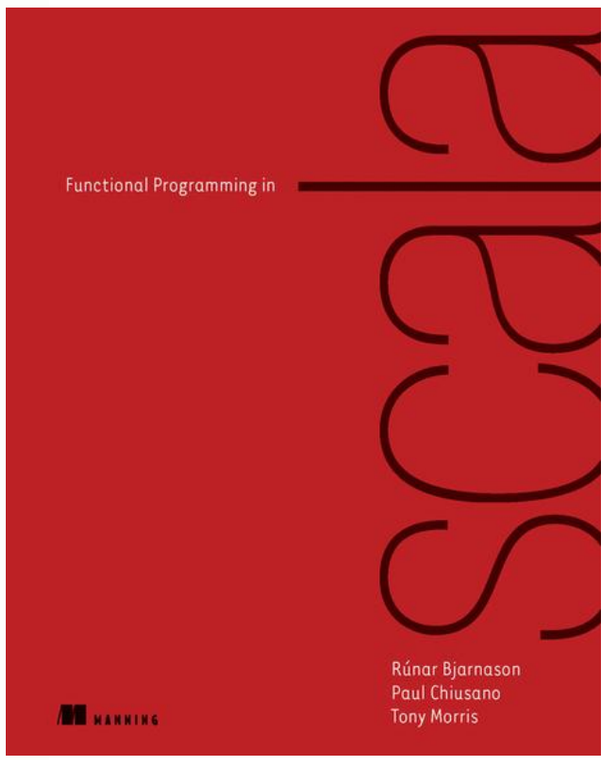
- for expression is a **monad**
- zip is an **applicative functor**

Resources









Typesafe

Scala, Akka, Play questions?

Come by our booth!