NFJS 2013 TOUR SERIES

No Fluff Just Stuff.com

Local Venue, World-Class Event

- St. Louis - Gateway Software Symposium
  - April 12-13, 2013
  - Topics
    - Languages on the JVM: Scala, Groovy, Clojure
    - Automation Tools: Gradle, Git, Hudson, Sonar
    - Continuous Deployment: Puppet, Chef
    - Enterprise Java
    - Core Java, Java 7
    - Agility
    - Testing: Geb, Spock, Easyb
    - REST
    - NoSQL: MongoDB, Cassandra
    - Hadoop
    - Spring 3
    - HTML5, Ajax, jQuery, Usability
    - Mobile Applications - iPhone and Android
    - More ...

5 concurrent tracks

50 talks/workshops

NFJS No Fluff Just Stuff.com 2013 TOUR SERIES Local Venue, World-Class Event

- Why attend?
  - Excellent speakers
    - authors, project leaders, consultants, and professional trainers
  - Network with your local developer community
  - No marketing or vendor promos
  - Great venue - St. Louis Marriott West, Maryville Center
  - Limited attendance
- What is the cost?
  - $875 through 3/18; $975 after that
  - group discounts
  - $50 JUG discount - promo code nfjsusergroup50
  - $100 alumni discount - promo code nfjsalumni100

only highest discount applies

NFJS 2013 TOUR SERIES
No Fluff Just Stuff.com
Local Venue, World-Class Event

- What do you get?
  - all access pass
  - all meals / snacks
  - session materials
  - NFJS, the Magazine 2013 subscription
  - NFJS backpack
  - NFJS 2012 t-shirt
  - NFJS 4 gig USB drive
  - Wi-Fi access
  - great raffle giveaways
- http://www.nofluffjuststuff.com/conferences/st_louis/2013/04/home

Use Scala Now

Ryan Richt

*The contents of this presentation do **not** represent Monsanto Company in any way*

Ryan Richt

Monsanto Genome Integration Lead

Washington University in St. Louis

Washington University in St. Louis
Olin Business School

The Genome Institute at Washington University

COFACTOR genomics

MONSANTO

Scala @ $MON

100,000 Lines of Code
1,200 Source Files
3,100 Classes
11,000 .class files

Just the beginning!

TOC

1. Whirlwind Tour of Scala Features
2. Complex Constraints with Type-Foolery
3. Abstraction with the Typeclass Pattern

I DONT CODE SCALA OFTEN...
...BUT WHEN I DO ITS NOT COMPLICATED
memegenerator.net

Expressions

Even more OO than Java

Expressions

Even more OO than Java

1. Almost everything has a return value
2. Almost everything is a method call

```
var myCat    = "Sam"
val bestLang = "Scala"
val three    =  1  + 2
```

Semicolon inference

Type inference

No operators, only methods

```
var myCat     = "Sam"
val bestLang  = "Scala"
val three     = (1).+(2)
```

Semicolon inference

Type inference

No operators, only methods

```
var myCat     = "Sam"
val bestLang  = "Scala"
val three     = (1).+(2)
```

Semicolon inference

Type inference

No operators, only methods

```
val snugglier =
    if (myCat < bestLang) myCat else bestLang
```

If is an expression

```scala
val betterLang =
  try {
    val source = io.Source.fromFile("crystal.ball”)
    source.getLines().toSeq.head
  } catch {
    case _: FileNotFoundException => “Scala”
  }
```

The handy underscore

A partial function

Classes

The Java You Love, Simplified

Classes

The Java You Love, Simplified

1. The body is the constructor
2. Variables are bean fields
3. Statics go in “companion”

```scala
package com.monsanto.Triffids
import java.util.UUID
import scala.util.Random
```

val/var declarations create fields

```scala
class Person(val weight: Int)
class Triffid private (var hasOil: Boolean,
                       serial: String){

  private[this] var belly: Option[Int] = None

  def eat(food: Person) {
     belly  = Some(food.weight)
     hasOil = true
  }
  def hungry = belly.isEmpty
}
object Triffid{
 def apply = new Triffid(false,UUID.randomUUID.toString)
}
```

```scala
package com.monsanto.Triffids
import java.util.UUID
import scala.util.Random

class Person(val weight: Int)
class Triffid private (var hasOil: Boolean,
                       serial: String){

  private[this] var belly: Option[Int] = None

  def eat(food: Person) {
      belly  = Some(food.weight)
      hasOil = true
  }
  def hungry = belly.isEmpty
}
object Triffid{
 def apply = new Triffid(false,UUID.randomUUID.toString)
}
```

Unit / void method

Expression method

```
package com.monsanto.Triffids
import java.util.UUID
import scala.util.Random

class Person(val weight: Int)
class Triffid private (var hasOil: Boolean,
                       serial: String){

  private[this] var belly: Option[Int] = None

  def eat(food: Person) {
     belly  = Some(food.weight)
     hasOil = true
  }
  def hungry = belly.isEmpty
}
object Triffid{
 def apply = new Triffid(false,UUID.randomUUID.toString)
}
```

Statics in "Companion"

```
val justAte   = new Triffid(true, “fluffy”)
val carnivore = Triffid()
val noSugar   = Triffid.apply()
```

constructors

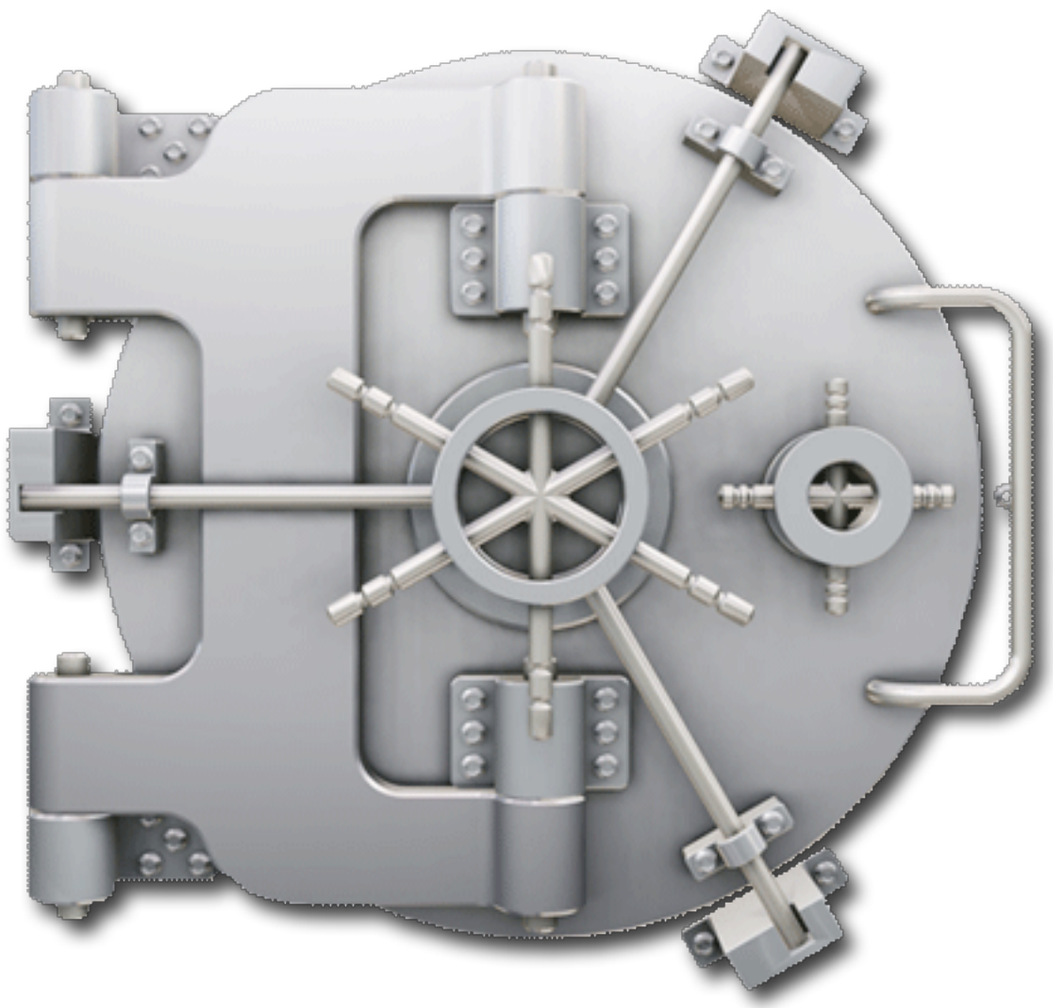
```
carnivore.hasOil = false
carnivore.hasOil_=(false)
```

fields accessed through getter/setters

```
carnivore.eat( new Person(125) )

val isHungry = carnivore.hungry
```

paren-less method

Scala



Java

Stronger Access Modifiers (& other toys)

```
package com.monsanto.carnivores
```

named & default parameters

```
class VenusFlyTrap(name: String = "Sweet Sweets"){

    public                      val isHungry          = ???
    private                     val trapCount         = ???
    protected                   val spikesPerTrap     = ???
    private[monsanto]           val isRoundupReady    = ???
    private[this]               def triggerTrap       = ???
    final                       val hasChloroplasts   = ???

    lazy                  val matureHeight = measure(grow())
    ...
}
```

!!!

expensive vals computed "on-demand"

Pattern Matching

Complex Conditionals Made Simple

```scala
case class Color(name: String = “Green”)
case class PitcherPlant(name: String, color: Color)
...
val boring = PitcherPlant(“Tiger”,    Color()       )
val cool   = PitcherPlant(“Precious”, Color(“Pink”))

val PitcherPlant(name, color) = boring

type PP = PitcherPlant
def coolPlantsOnly(plant: PP): Option[PP] =
 plant match {
  case PitcherPlant(n, Color(c)) if c != “Green”
         => Some(plant)
  case _ => None
}
```

free apply/unapply

```
case class Color(name: String = “Green”)
case class PitcherPlant(name: String, color: Color)
...
val boring = PitcherPlant(“Tiger”,    Color()      )
val cool   = PitcherPlant(“Precious”, Color(“Pink”))

val PitcherPlant(name, color) = boring
```

destructuring

```
type PP = PitcherPlant
def coolPlantsOnly(plant: PP): Option[PP] =
 plant match {
  case PitcherPlant(n, Color(c)) if c != “Green”
         => Some(plant)
  case _ => None
}
```

```
case class Color(name: String = “Green”)
case class PitcherPlant(name: String, color: Color)
...
val boring = PitcherPlant(“Tiger”,    Color()      )
val cool   = PitcherPlant(“Precious”, Color(“Pink”))

val PitcherPlant(name, color) = boring
```

```
type PP = PitcherPlant
def coolPlantsOnly(plant: PP): Option[PP] =
 plant match {
  case PitcherPlant(n, Color(c)) if c != “Green”
          => Some(plant)
  case _ => None
}
```

arg is a partial function

guards

```
case class PitcherPlant(name: String, color: Color)
```

```
object PitcherPlant{
    def apply(name: String, color: Color): PitcherPlant
    def unapply(plant: PitcherPlant): Option[(String, Color)]
}
```

Traits

Multiple Inheritance without Diamonds

Traits

Multiple Inheritance without Diamonds

1. Traits are “mix-ins”
2. No constructor parameters
3. Interface + Implementation
4. Linearization

```
trait Plant { def desc = “Plant” }
```

implementation

```
trait Commercial extends Plant {
  def output: String
  abstract override def desc =
      output + “-producing ” + super.desc
}
```

interface only

interface + implementation

```
trait GMO extends Plant {
   abstract override def desc =
       “Roundup-Ready ” + super.desc
}
```

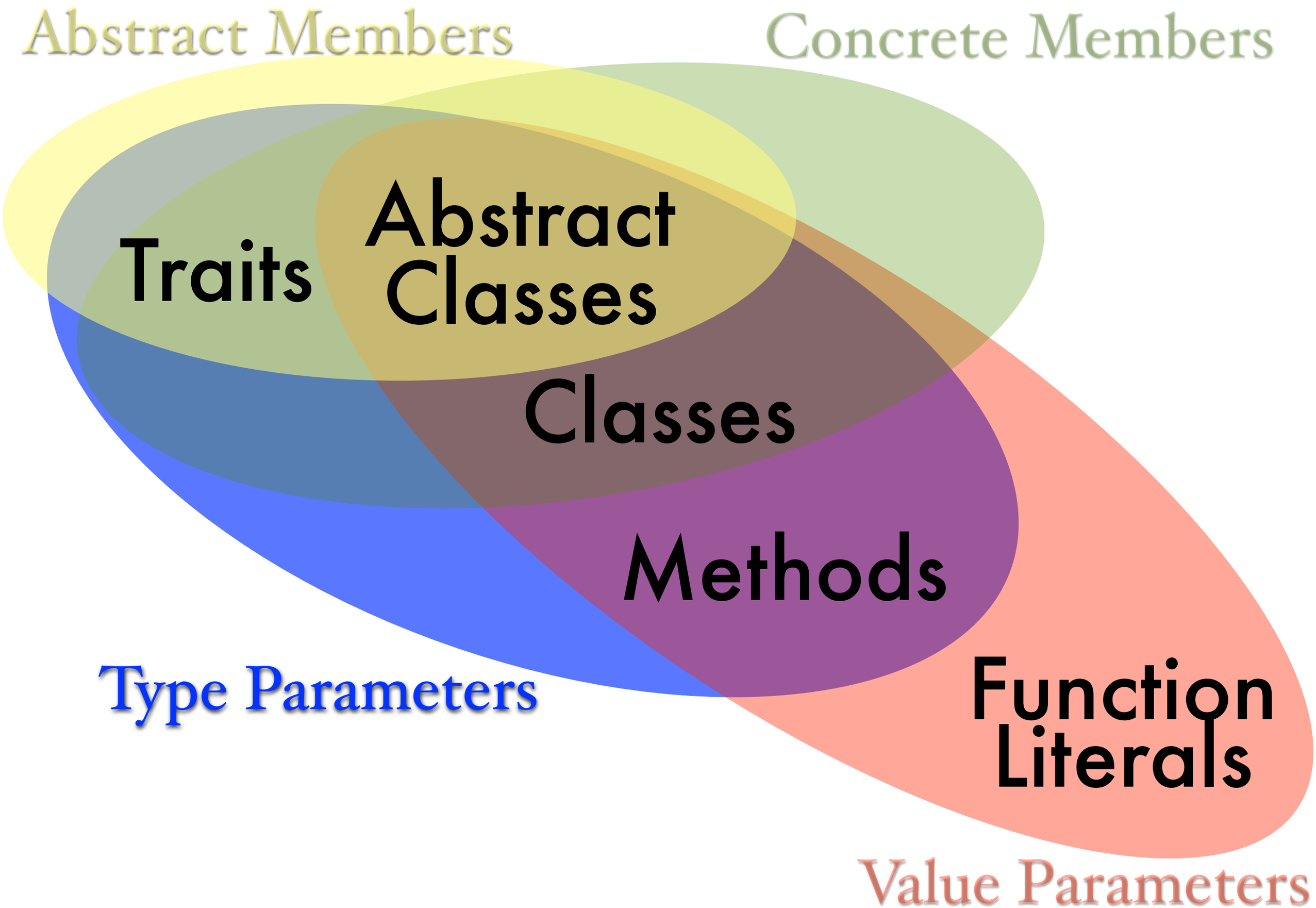
no value parameters

order matters!

```
class Triffid extends Plant with Commercial with GMO{
   val output = “Oil”
   override def desc = “A Triffid is a ” + super.desc
}

new Triffid().desc
```

Triffid GMO Commercial Plant

“A Triffid is a Roundup-Ready Oil-producing Plant”


Abstract Members
Concrete Members
Traits
Abstract Classes
Classes
Methods
Type Parameters
Function Literals
Value Parameters

</basics>

AKA the boring part

Higher-Order Functions

A Functional Programming Abstraction without Recursion

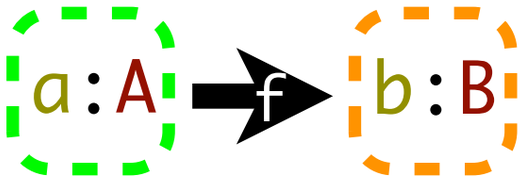
Higher-Order Functions

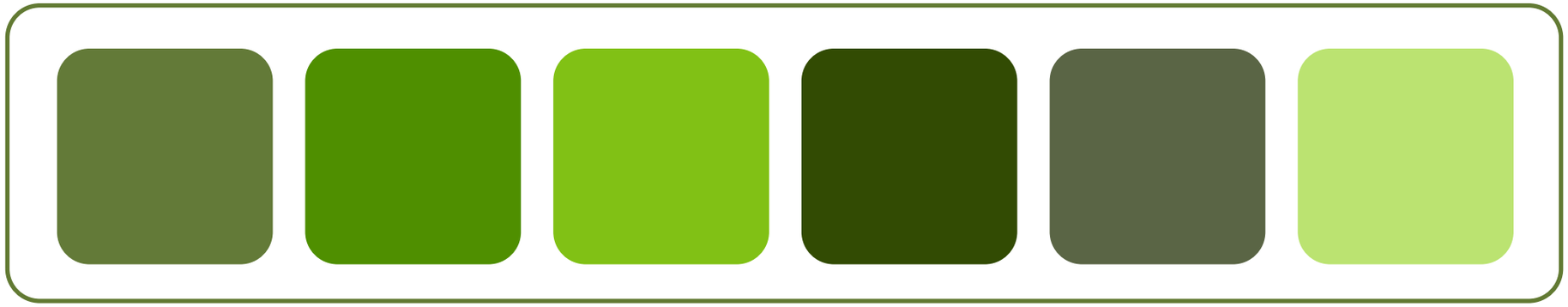
A Functional Programming Abstraction without Recursion

1. Functions that take functions as arguments
2. Functions that produce functions as results
3. Simpler but isomorphic to recursion in many cases

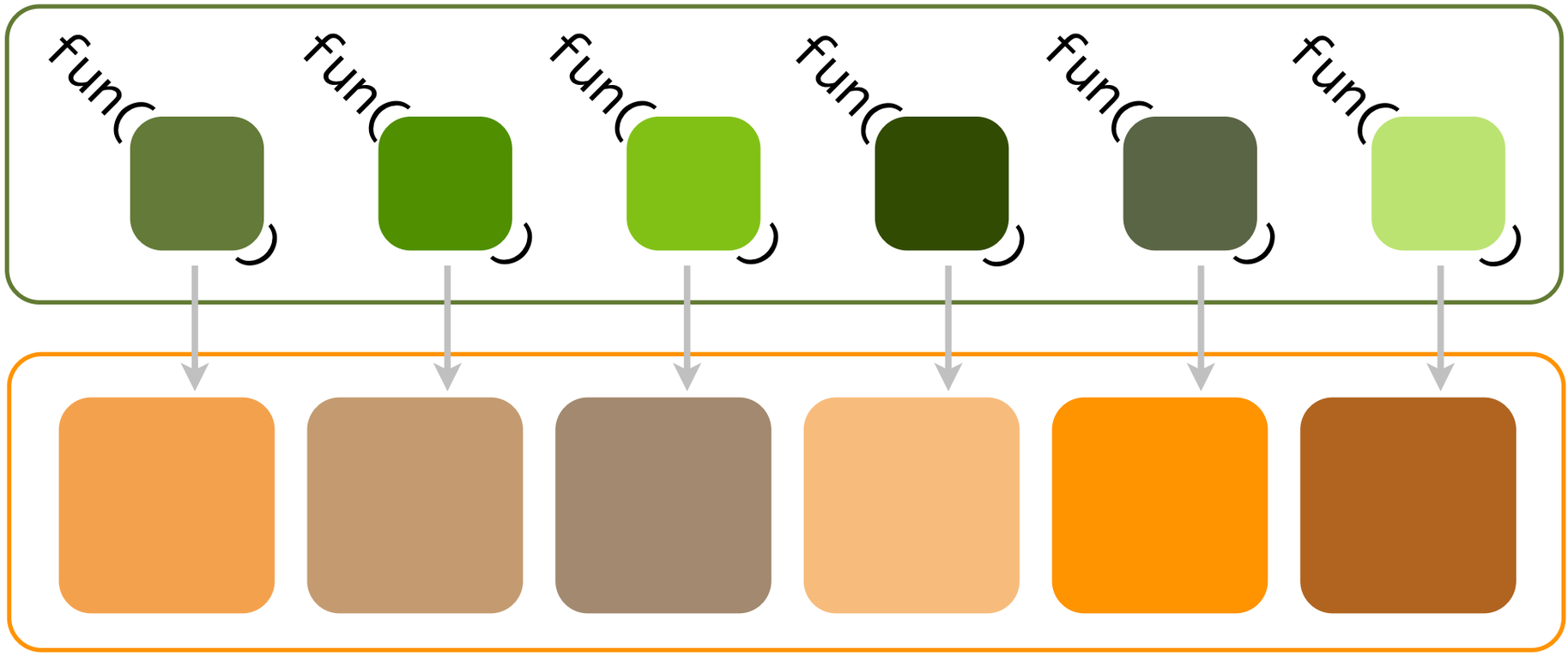
Map

```
val fun  =    (a: A) => myManipulate(a).myTransform
```



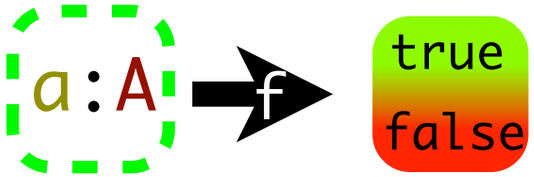



```
val res  =    list.map(fun)
```

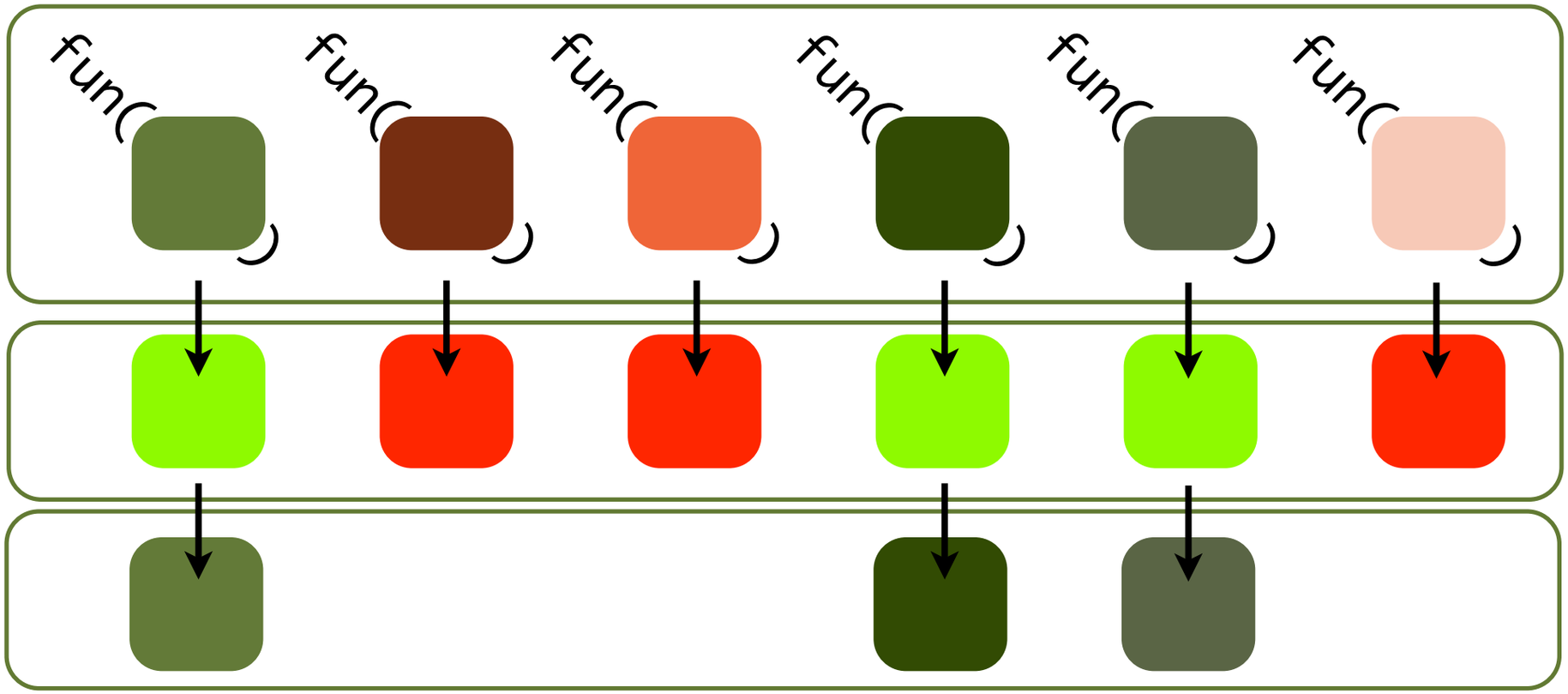


Filter

```
val fun  =    (a: A) => isFooLike(a)
```

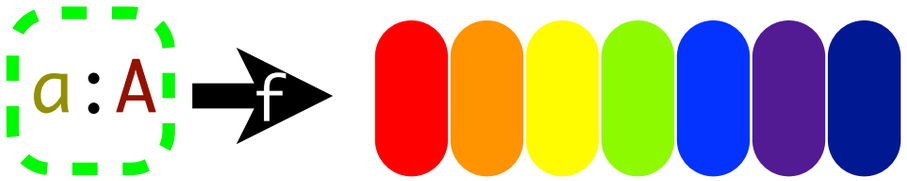


```
val res  =    list.filter(fun)
```

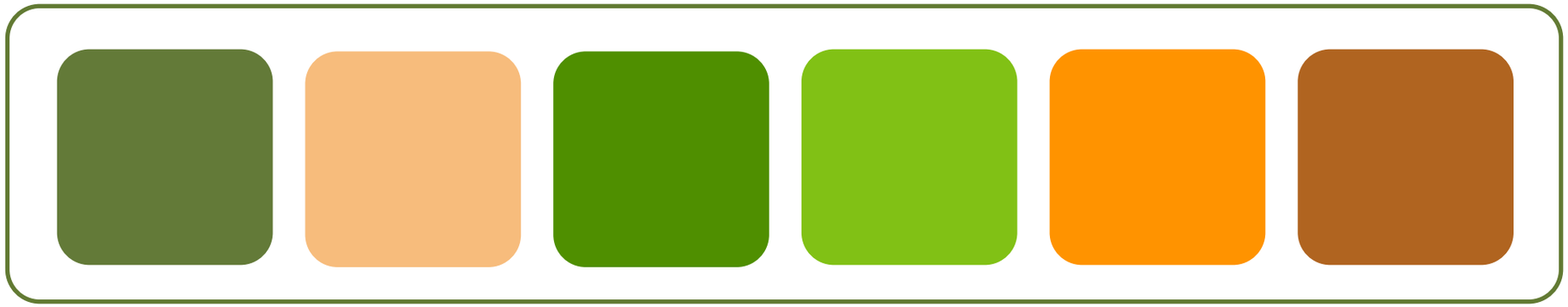


GroupBy

```
val fun =    (a: A) => categorize(a)
```

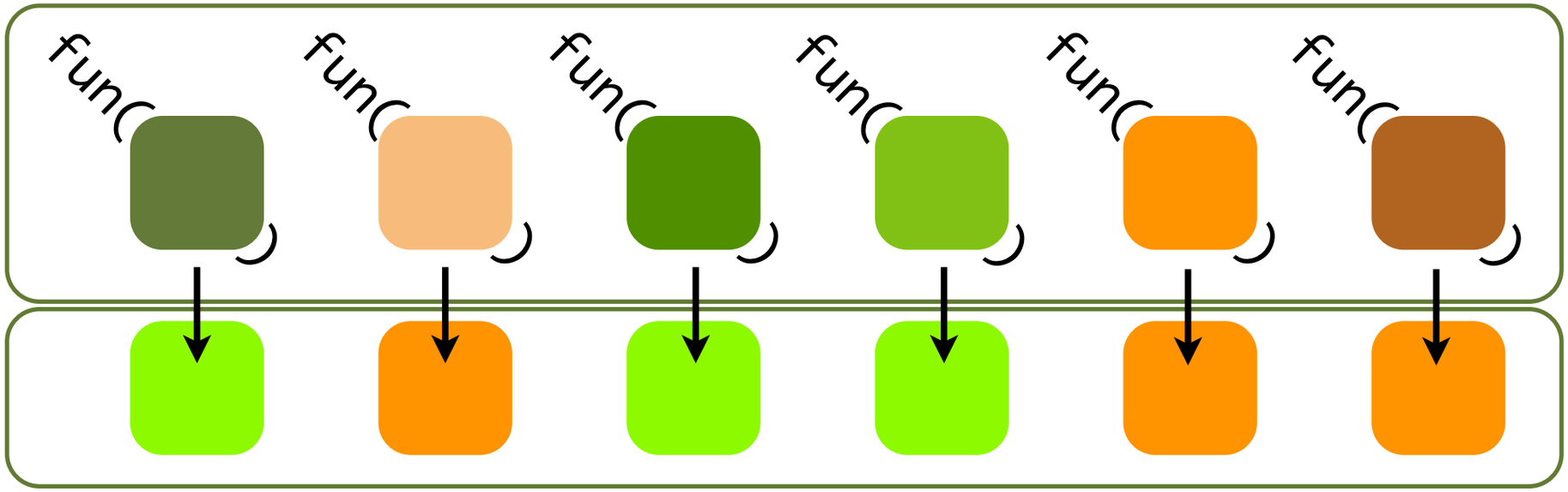


val list =

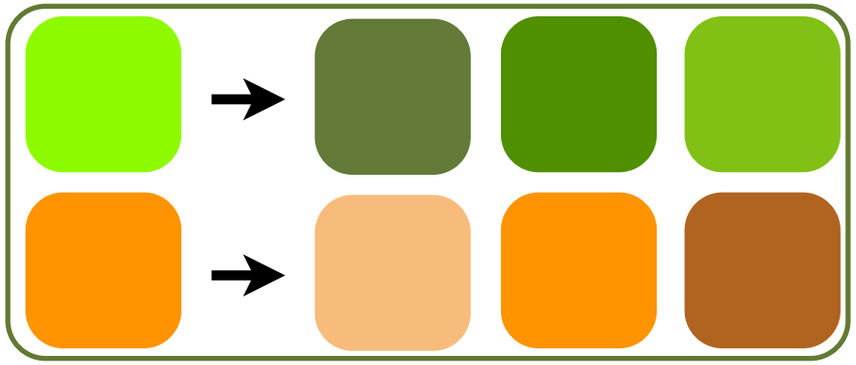


```
val res =    list.groupBy(fun)
```

=

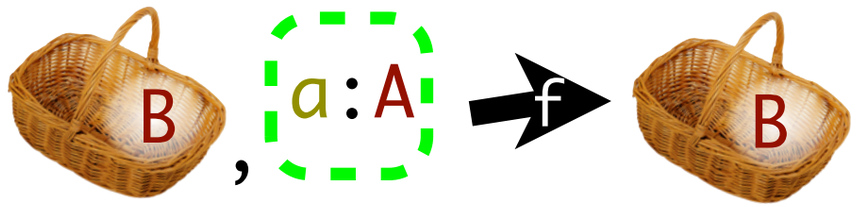


=

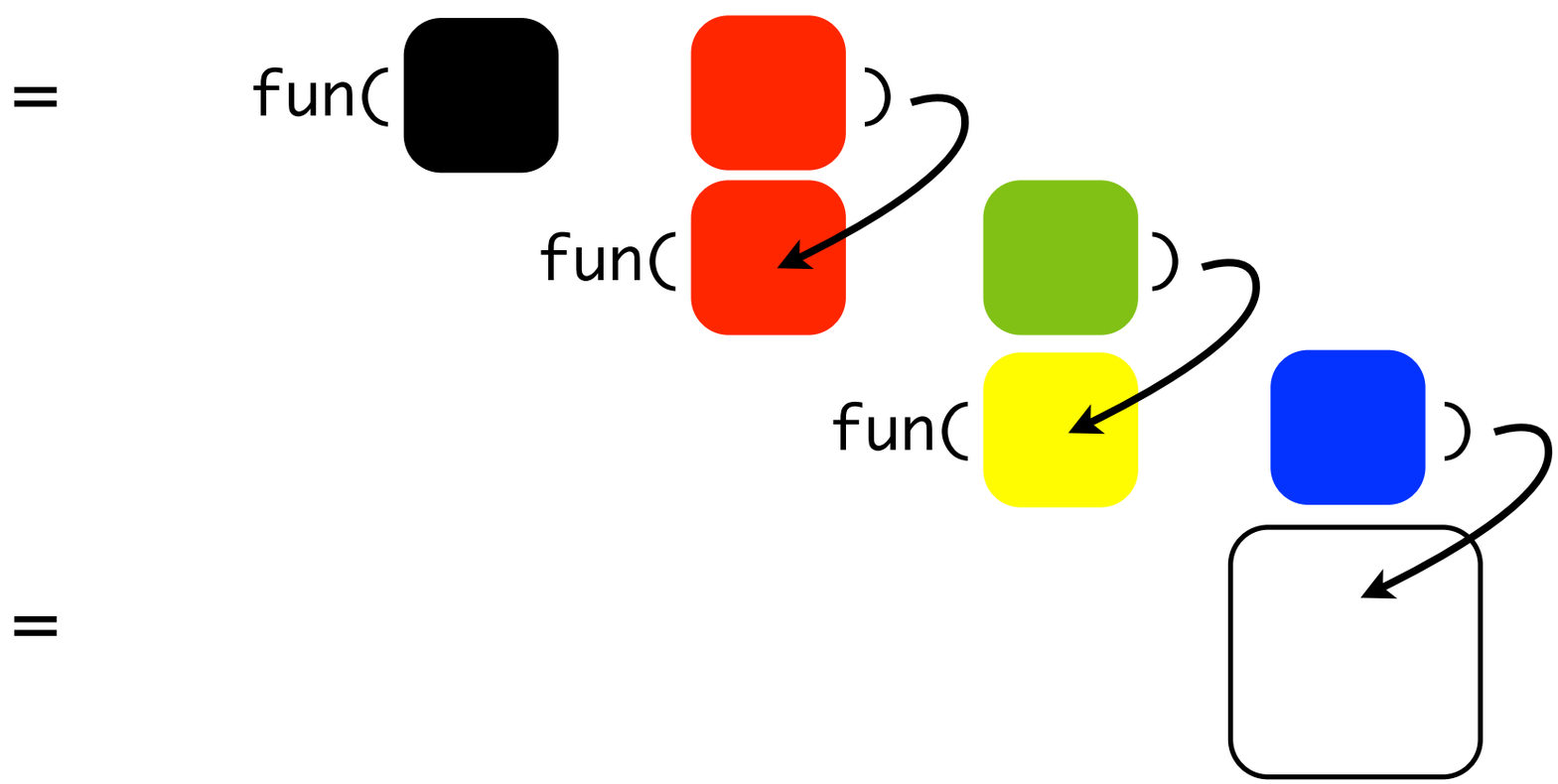


Reduce (foldLeft)

```
val fun  =    (acc: B, a: A) => new B(acc + a)
```



```
val res  =    list.foldLeft(zero)(fun)
```



```scala
case class Sundew(bugsCaught = 0)

val plants = Seq(Sundew(3), Sundew(), Sundew(7))

val withFood = plants.filter( s => s.bugsCaught > 0 )
val withFood = plants.filter( _.bugsCaught > 0 )

val counts = plants.map(_.bugsCaught) // Seq(3,0,7)

val totalBugs = counts.foldLeft(_ + _) // 10
```

```scala
val nums = Seq(1,2,3,4,5)

val dubs = nums.map(_ * 2)
val dubs = nums.foldLeft(Seq.empty[Int]){_ :+ _ * 2}

val evens = nums.filter(_ % 2 == 0)
val evens = nums.foldLeft(Seq.empty[Int]){ (acc,e) =>
    if(e % 2 == 0) acc :+ e else acc
}

def isPrime(i: Int): String // “prime”||“comp”
val primeSort = nums.groupBy(isPrime)
val zero = Map[String,Seq[Int]]().withDefault(_ => Seq[Int]())
val primeSort = nums.foldLeft(zero){ (acc,e) =>
    val category = isPrime(e)
    acc.updated(category, acc(category) :+ e)
}
```

The Loan Pattern

```
package com.YourBankHere.loans
class Cust(var mood: String = “happy”){
   def screw { mood = “flabbergasted” }
}
class Bank{
   def claimLoan(c:Cust, amt:Int, spend:Int=>String)={
         val awarded = amt / 2
         try {
            spend(awarded)
         } finally {
            c.screw
         }
   }
}
val burnMoney = (d: Int) => “fire ” * d
new Bank().claimLoan(new Cust(), 10, burnMoney)
```

prework

potentially dangerous

don’t forget to screw the customer!

Collections

Simple, Expressive, Parallel

```
val dna = Set("A","C","G","T")

val plants = Map("boring" -> Seq("ficus"),
                 "cool" -> Seq("triffid", "sundew"))

val nums = Map( "evens"-> Seq(2,4,6),
                "odds" -> Seq(1,3,5) )

val nums: Map[String,Int] =
        Map[String,Int]( "evens"->Seq(2,4,6),
                         "odds"->Seq(1,3,5)   )

val successor = mutable.Map(1->2)
successor(3) = 4 // successor = Map(1->2, 3->4)

Seq(1,3,1,3).map(successor) // Seq(2,4,2,4)
```

Parallel & Lazy Collections

fork/join pool!

```
Range(1,1000000).par.fold(0)(_ + _) // 1783293664
Range(1,1000000).par.map(_ * 2)
```

ordered yields ordered

```
Range(1,1000).view.
                 map(_ + 1).
                  map(_ * 2).
                   filer(_ % 4 == 0) … .sum
```

don't materialize intermediate collections

...until "forcing" operation

TOC

1. Whirlwind Tour of Scala Features
2. Complex Constraints with Type-Foolery
3. Abstraction with the Typeclass Pattern

Implicits

Compile-time, Type-driven Boilerplate Buster

There can be Only One

Implicits - Compiler Discovered & Passed

```
case class Hour(i: Int){
    def am = Hour(i)
    def pm = Hour(i + 12)
}
implicit def intToHour(i: Int) = Hour(i)

case class CatFur(count: Int = 3)

def sitOnCouch(time: Hour)(implicit h: CatFur) = ???
def foldLaundry(time: Hour)(implicit h: CatFur) = ???
def openLaptop(time: Hour)(implicit h: CatFur) = ???

implicit val samFur = CatFur()
sitOnCouch(5 pm)
implicit val zoeFur = CatFur()
openLaptop(8 am) // does not compile!!
```

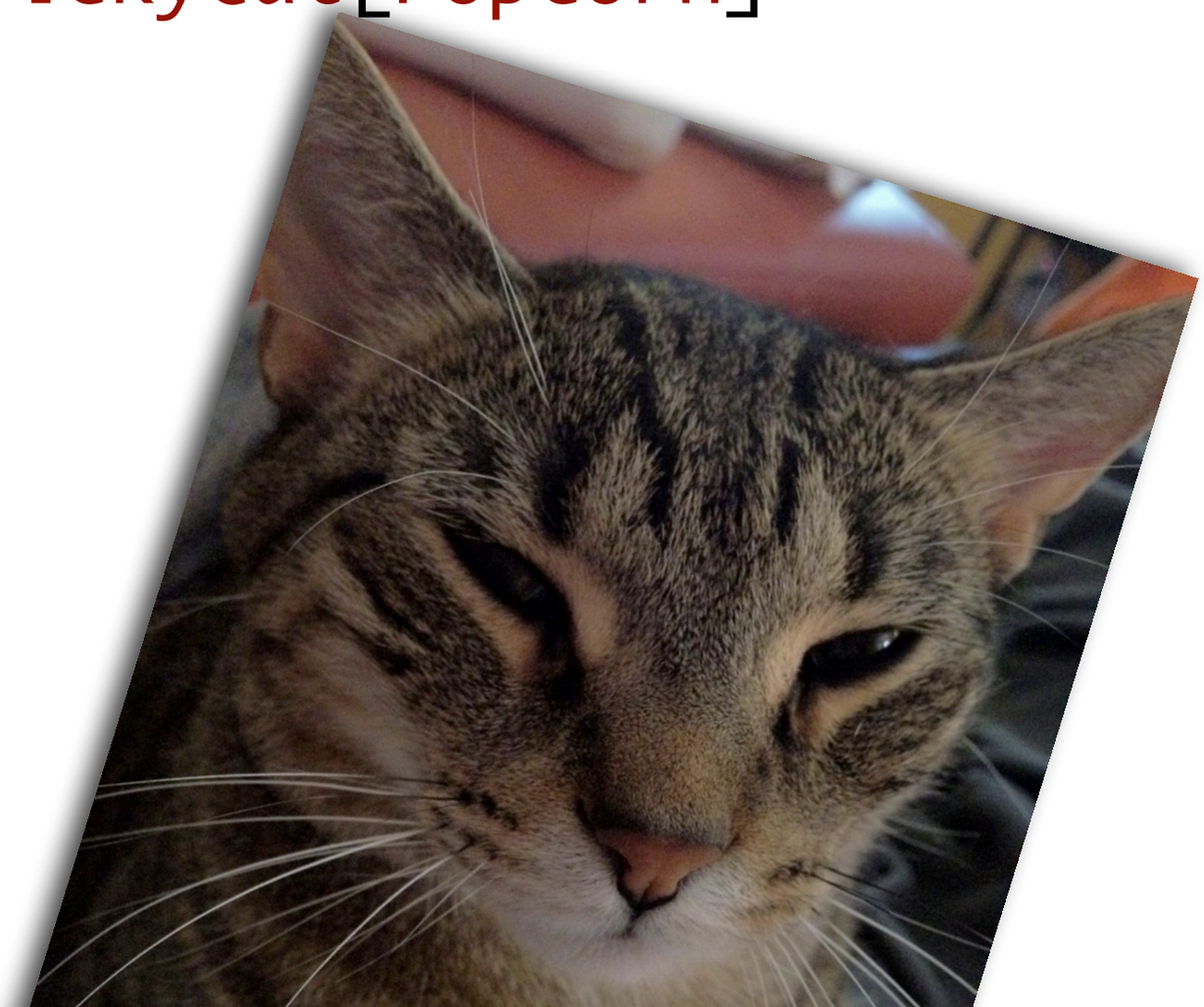
Implicits - Compiler Discovered & Passed

```
case class Hour(i: Int){
   def am = Hour(i)
   def pm = Hour(i + 12)
}
implicit def intToHour(i: Int) = Hour(i)

case class CatFur(count: Int = 3)

def sitOnCouch(time: Hour)(implicit h: CatFur) = ???
def foldLaundry(time: Hour)(implicit h: CatFur) = ???
def openLaptop(time: Hour)(implicit h: CatFur) = ???

implicit val samFur = CatFur()
sitOnCouch(5 pm)
implicit val zoeFur = CatFur()
openLaptop(8 am) // does not compile!!
```

Type Parameters

Apply Liberally after Type Inference

```scala
trait Food
trait PickyCat[Favorite <: Food]{
   def eat(bite: Favorite) = “mrow”
}


case class Popcorn(kernels: Int = 1) extends Food
case class SamCat extends PickyCat[Popcorn]
val sam = new SamCat
sam.eat( Popcorn() )
```



```scala
trait Food
trait PickyCat[Favorite <: Food]{
   def eat(bite: Favorite) = “mrow”
}


case class Popcorn(kernels: Int = 1) extends Food
case class SamCat extends PickyCat[Popcorn]
val sam = new SamCat
sam.eat( Popcorn() )


case class Carrier[Cat <: PickyCat[_]](c: Cat)
val samInCrate = Carrier(sam)
```

preserves your cat’s class upon exiting

```scala
case class LuxCarrier
            [F <: Food, Cat <: PickyCat[F]](c: Cat){
   def stuffFoodInWindow(f: F) = c.eat(f)
}

val luxTrip = LuxCarrier[Popcorn,SamCat](sam)
```

:-(*no inference*

```scala
trait Picky2{
  type Fav <: Food
  def eat(bite: Favorite) = “mrow”
}

case class Sam2 extends Picky2{ type Fav = Popcorn}

case class Lux2[F, Cat <: Picky2]
                (c: Cat{type Fav = F}){

   def stuffFoodInWindow(f: F) = c.eat(f)
}

val trip2 = Lux2(Sam2()).stuffFoodInWindow(Popcorn())
```

type member

```
trait Picky2{
  type Fav <: Food
  def eat(bite: Favorite) = “mrow”
}

case class Sam2 extends Picky2{ type Fav = Popcorn}

case class Lux2[F, Cat <: Picky2]
                (c: Cat{type Fav = F}){

   def stuffFoodInWindow(f: F) = c.eat(f)
}

val trip2 = Lux2(Sam2()).stuffFoodInWindow(Popcorn())
```

type member

“refinement”

!!!

My Favorite (useful) Type Trick

```scala
trait Cloneable[C <: Cloneable[C]]{ self: C =>
   def cloneIt: C
}
```

as in DNA clone!

```scala
case class PartnerInCrime(i: Int = 1)
case class Sneaky extends Cloneable[PartnerInCrime]{
   def cloneIt = PartnerInCrime()
}
```

does not compile

```scala
case class Tribble extends Cloneable[Tribble]{
   def cloneIt = Tribble()
}
```

TOC

1. Whirlwind Tour of Scala Features

2. Complex Constraints with Type-Foolery

3. Abstraction with the Typeclass Pattern

A Magic Box

The Mechanics of Type Classes

Defeating Type Erasure

```
class BlackMagicBox[T : Manifest](c: Option[T]){
   val spell = implicitly[Manifest[T]].runtimeClass

   def summoned = copy(c = Some(spell.newInstance))

   def vanished = copy(c = None)
}
```

```scala
trait Summoner[T]{ def summon: T }

case class MagicBox[T: Summoner](c: Option[T]=None){
   val spell = implicitly[Summoner[T]]
   def summoned = copy(c = Some(spell.summon))
   def vanished = copy(c = None)
}
```

the typeclass instance

```scala
case class Dove
object Dove{
 implicit object conjureDove extends Summoner[Dove]{
   def summon = new Dove
 }
}

MagicBox[Dove]().summoned.c // Some(Dove) !
```

JSON Serialization

Toy example of Typeclasses in Action

```scala
case class JSON(s:String){override def toString = s}
object JSON{
   def serialize[T : JsonFormat](elem: T) =
      implicitly[JsonFormat[T]].toJson(elem)
}
trait JsonFormat[T]{ def toJson(e: T): JSON }
object JsonFormat{
   implicit object intF extends JsonFormat[Int] {
      def toJson(i: Int) = JSON(i.toString)
   }
   implicit def seqF[T : JsonFormat] =
      new JsonFormat[Seq[T]] {
         def toJson(s: Seq[T]) = {
            val f = implicitly[JsonFormat[T]]
            JSON( s.map(f.toJson(_)).
                  mkString("[","," , "]") )
         }
      }
}
```

```scala
case class JSON(s:String){override def toString = s}
object JSON{
   def serialize[T : JsonFormat](elem: T) =
      implicitly[JsonFormat[T]].toJson(elem)
}
```

Use the Typeclass instance

```scala
trait JsonFormat[T]{ def toJson(e: T): JSON }
```

An interface for our Typeclass

Typeclass instance for Int

```
object JsonFormat{
    implicit object intF extends JsonFormat[Int] {
        def toJson(i: Int) = JSON(i.toString)
    }

}
```

“Recursive” Typeclass instance for Seq

it’s generic!

```
object JsonFormat{

    implicit def seqF[T : JsonFormat] =
        new JsonFormat[Seq[T]] {
            def toJson(s: Seq[T]) = {
                val f = implicitly[JsonFormat[T]]
                JSON( s.map(f.toJson(_)).
                        mkString(“[”,“,”, “]”) )
            }
        }
}
```

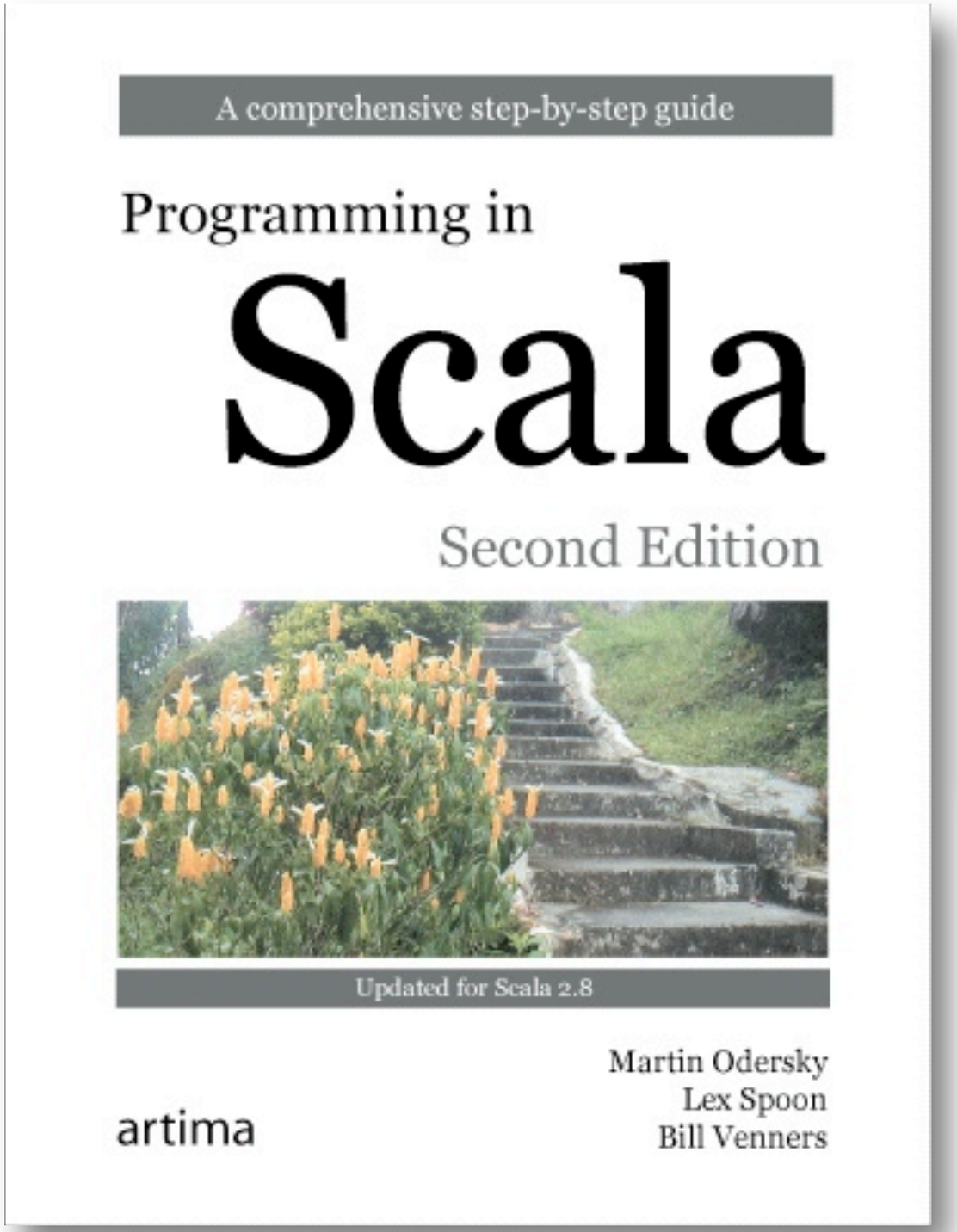
```scala
val json = JSON.serialize(Seq(1,2,3))
```

```scala
scala> val j1 = JSON.serialize(Seq(1,2,3))
j1: JSON = [1,2,3]

scala> val j2 = JSON.serialize(Seq(Seq(1,2,3), Seq(4,5,6)))
j2: JSON = [[1,2,3],[4,5,6]]
```

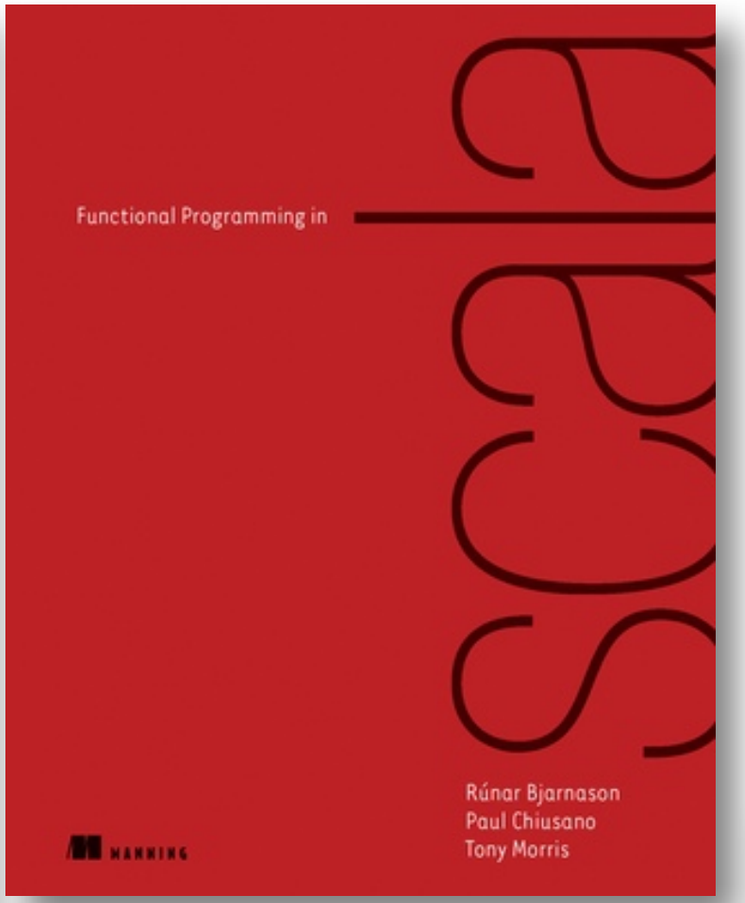
Other Great Stuff

- Simple conversion to Java classes
- @specialized / @tailrec / -optimise
- Spire, ScalaCL
- Akka & the actor model
- SBT
- IntelliJ support
- Implicit conversions (!?)
- Parser Combinators
- Play, Slick, Spray, Scalaz
- Macros
- OMG the collections library

Resources

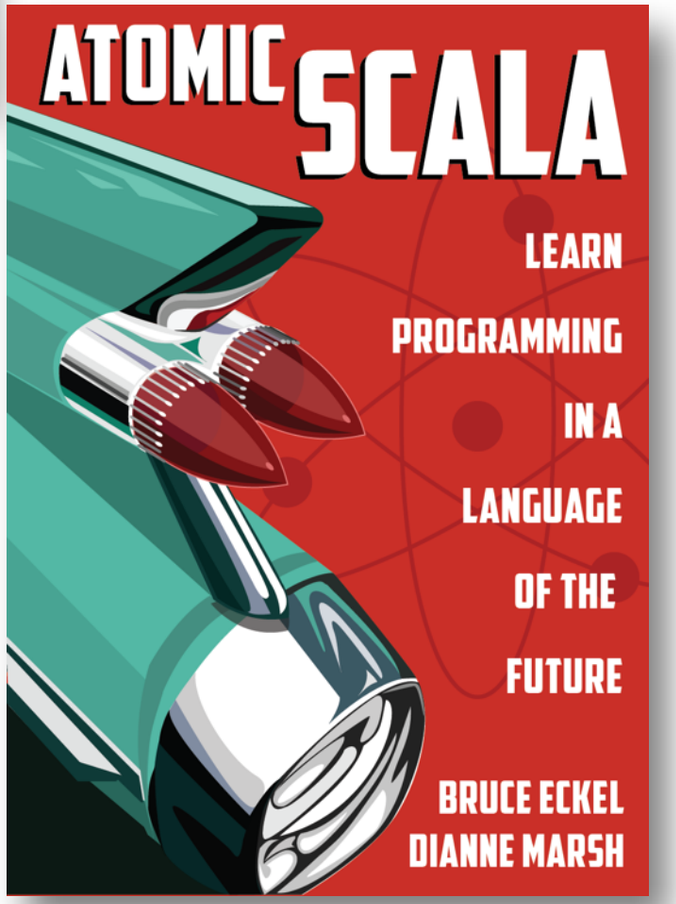


Odersky, Spoon, Venners



Bjarnason, Chiusano

Eckel, Marsh



Use Scala Now

Ryan Richt

*The contents of this presentation do **not** represent Monsanto Company in any way*