

The Seductions of Scala

Dean Wampler
dean@deanwampler.com
[@deanwampler](https://twitter.com/deanwampler)
polyglotprogramming.com/talks

April 17, 2012



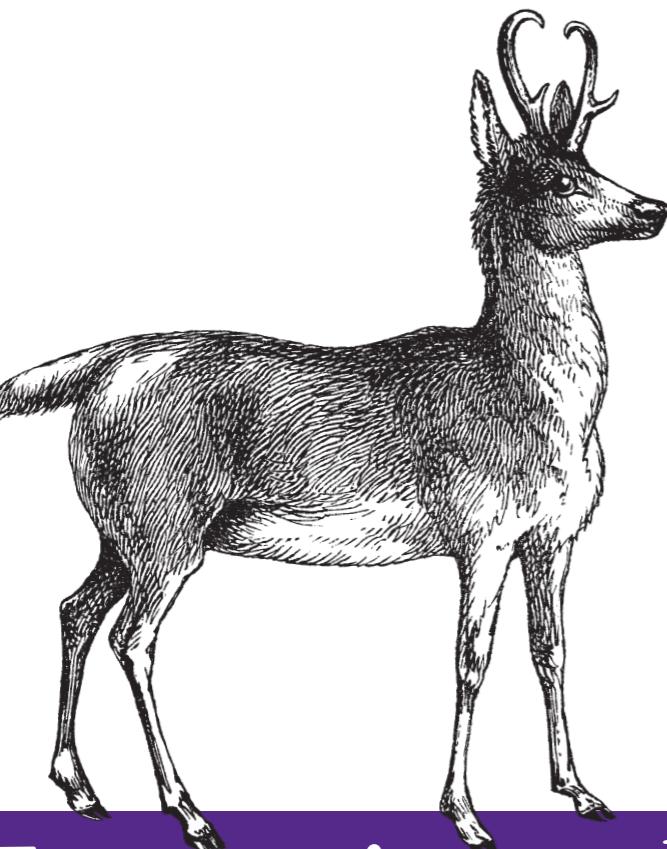
1

Monday, June 17, 13

The online version contains more material. You can also find this talk and the code used for many of the examples at github.com/deanwampler/Presentations/tree/master/SeductionsOfScala.

Copyright © 2010–2013, Dean Wampler. Some Rights Reserved – All use of the photographs and image backgrounds are by written permission only. The content is free to reuse, but attribution is requested.

<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>



Functional Programming

for Java Developers

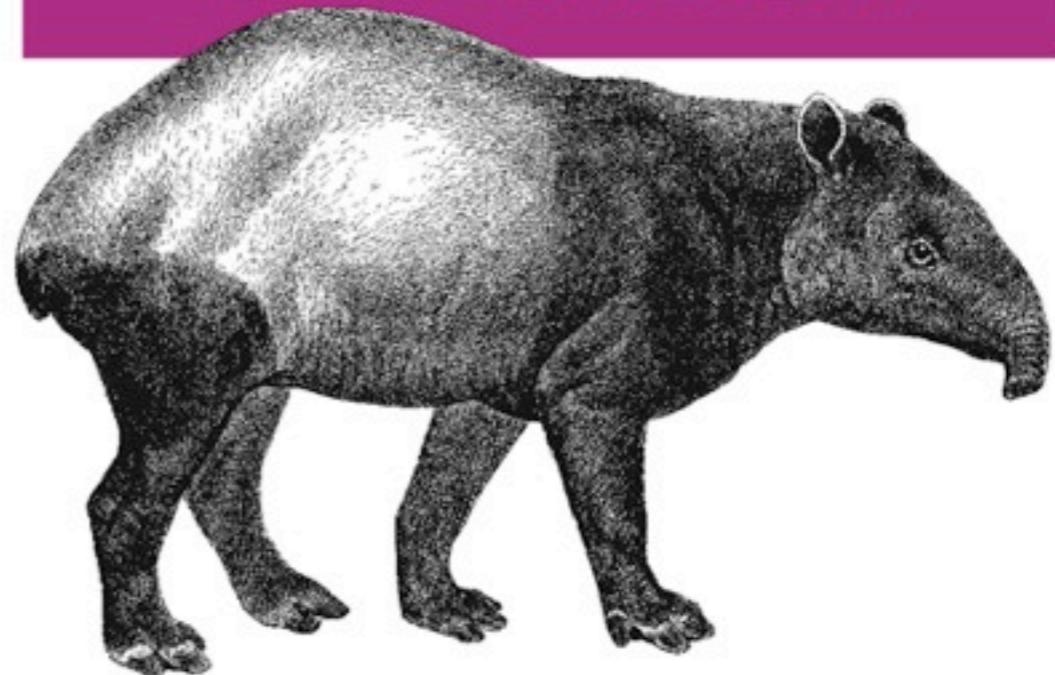
O'REILLY®

Dean Wampler

Scalability = Functional Programming + Objects

Programming

Scala



O'REILLY®

Dean Wampler & Alex Payne

[polyglotprogramming.com/
fpjava](http://polyglotprogramming.com/fpjava)

programmingscala.com

Why do we need a new language?

3

Monday, June 17, 13

I picked Scala to learn in 2007 because I wanted to learn a functional language. Scala appealed because it runs on the JVM and interoperates with Java. In the end, I was seduced by its power and flexibility.

I

We need *Functional* Programming

...

4

Monday, June 17, 13

First reason, we need the benefits of FP.

... for concurrency.
... for concise code.
... for correctness.

#2

We need a better
Object Model

• • •

... for composability.
... for scalable designs.

Scala's Thesis: Functional Prog. Complements Object-Oriented Prog.

Despite surface contradictions...

But we want to
keep our *investment*
in Java.

Scala is...

- A JVM language.
- *Functional* and *object oriented*.
- *Statically typed*.
- An *improved Java*.

10

Monday, June 17, 13

There has also been work on a .NET version of Scala, but it seems to be moving slowly.

Martin Odersky

- Helped design java *generics*.
- Co-wrote *GJ* that became *javac* (v1.3+).
- Understands Computer Science theory and industry's needs.

11

Monday, June 17, 13

Odersky is the creator of Scala. He's a prof. at EPFL in Switzerland. Many others have contributed to it, mostly his grad. students. GJ had generics, but they were disabled in javac until v1.5.

A wide-angle photograph of a serene lake nestled in a mountainous region. The foreground is dominated by the dark, calm water of the lake. In the middle ground, a small, densely forested island is visible in the center. The background features a majestic range of mountains with their peaks partially covered in snow. The sky is a clear, pale blue, suggesting either sunrise or sunset. The overall atmosphere is peaceful and natural.

Everything can
be a *Function*

12

Monday, June 17, 13

Not all objects are functions, but they can be...

Objects as Functions

```
class Logger(val level:Level) {  
  
    def apply(message: String) = {  
        // pass to Log4J...  
        Log4J.log(level, message)  
    }  
}
```

makes “level” a field

```
class Logger(val level:Level) {
```

```
    def apply(message: String) = {  
        // pass to Log4J...  
        Log4J.log(level, message)  
    }  
}
```

method

*class body is the
“primary” constructor*

```
class Logger(val level:Level) {  
  
    def apply(message: String) = {  
        // pass to Log4J...  
        Log4J.log(level, message)  
    }  
}  
  
val error = new Logger(ERROR)  
  
...  
error("Network error.")
```

```
class Logger(val level:Level) {  
  
    def apply(message: String) = {  
        // pass to Log4J...  
        Log4J.log(level, message)  
    }  
}
```

apply is called

“function object”

...
error("Network error.")

When you put
an argument *list*
after any *object*,
apply is called.

A wide-angle photograph of a serene lake nestled among majestic mountains. The sky is a soft, warm orange and yellow, suggesting either sunrise or sunset. The mountains in the background are rugged and partially covered in snow. In the foreground, the calm water of the lake reflects the surrounding beauty. The overall atmosphere is peaceful and inspiring.

Everything is an Object

19

Monday, June 17, 13

While an object can be a function, every “bare” function is actually an object, both because this is part of the “theme” of scala’s unification of OOP and FP, but practically, because the JVM requires everything to be an object!

Int, Double, etc.
are true *objects*.

But they are compiled to primitives.

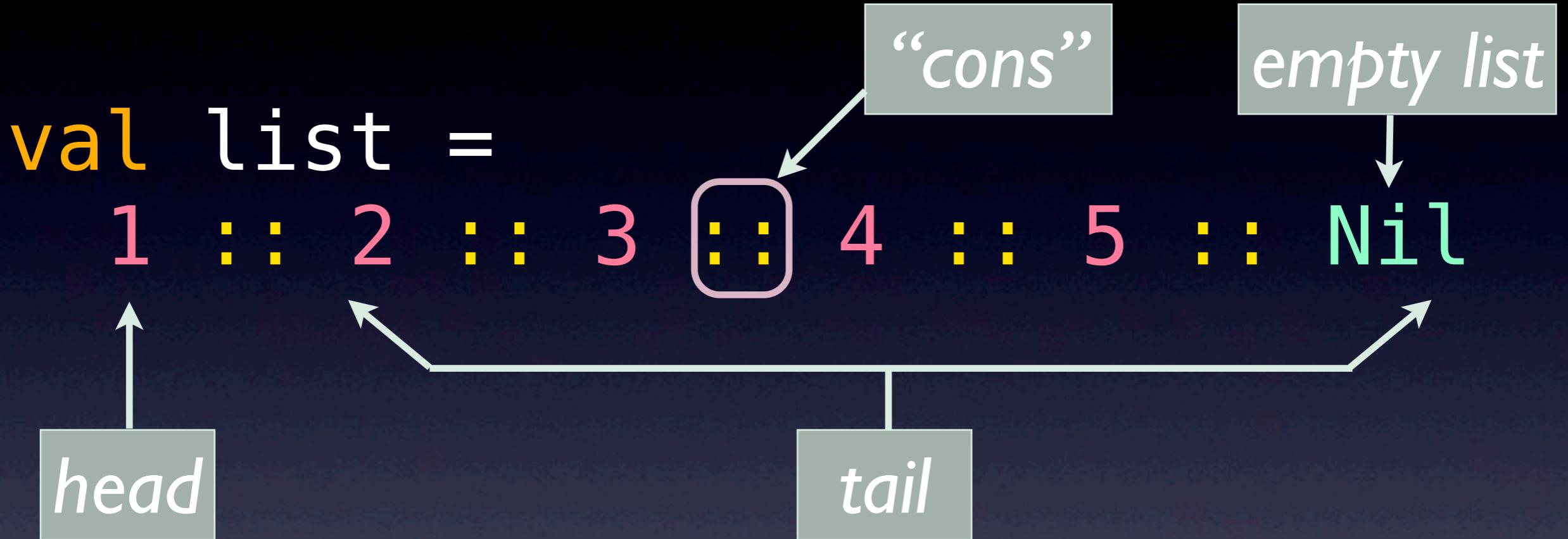
Lists

List.apply()

```
val list = List(1, 2, 3, 4, 5)
```

The same as this “list literal” syntax:

```
val list =  
  1 :: 2 :: 3 :: 4 :: 5 :: Nil
```



Baked into the Grammar?

```
val list =  
  1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

No, just method calls!

```
val list = Nil :: (5) :: (4) :: (3) :: (2) :: (1)
```

```
val list =  
  1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
val list = Nil.::(5).:::(4).:::(  
  3).:::(2).:::(1)
```

Method names can contain almost any character.

```
val list =  
  1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
val list = Nil :: (5) :: (4) :: (3) :: (2) :: (1)
```

Any method ending in “::” binds to the right!

```
val list =  
  1 :: 2 :: 3 :: 4 :: 5 :: Nil
```

```
val list = Nil :: (5) :: (4) :: (3) :: (2) :: (1)
```

If a method takes one argument, you can drop the “.” and the parentheses, “(“ and “)”.

Infix Operator Notation

"hello" + "world"

is actually just

"hello".+("world")

Oh, and Maps

```
val map = Map(  
  "name" -> "Dean",  
  "age"  -> 39)
```

Monday, June 17, 13

Maps also have a literal syntax, which should look familiar to you Ruby programmers ;) Is this a special case in the language grammar?

Oh, and Maps

```
val map = Map(  
    "name" -> "Dean",  
    "age" -> 39)
```

“baked” into the
language grammar?

No, just method calls...

Oh, and Maps

```
val map = Map(  
  "name" -> "Dean",  
  "age"  -> 39)
```

*What we like
to write:*

```
val map = Map(  
  Tuple2("name", "Dean") ,  
  Tuple2("age", 39))
```

*What Map.apply()
actually wants:*

Oh, and Maps

```
val map = Map(  
  "name" -> "Dean",  
  "age"  -> 39)
```

*What we like
to write:*

```
val map = Map(  
  ("name", "Dean") ,  
  ("age",   39) )
```

*What Map.apply()
actually wants:*

*More succinct
syntax for Tuples*

We need to get from this,

"name" -> "Dean"

to this,

`Tuple2("name", "Dean")`

There is no String-> method!

Implicit Conversions

```
class ArrowAssoc[T1](t:T1) {  
  def -> [T2](t2:T2) =  
    new Tuple2(t1, t2)  
}
```

```
implicit def  
toArrowAssoc[T](t:T) =  
  new ArrowAssoc(t)
```

Back to Maps

```
val map = Map(  
  "name" -> "Dean",  
  "age"  -> 39)
```

*toArrowAssoc called for each pair,
then ArrowAssoc.-> called*

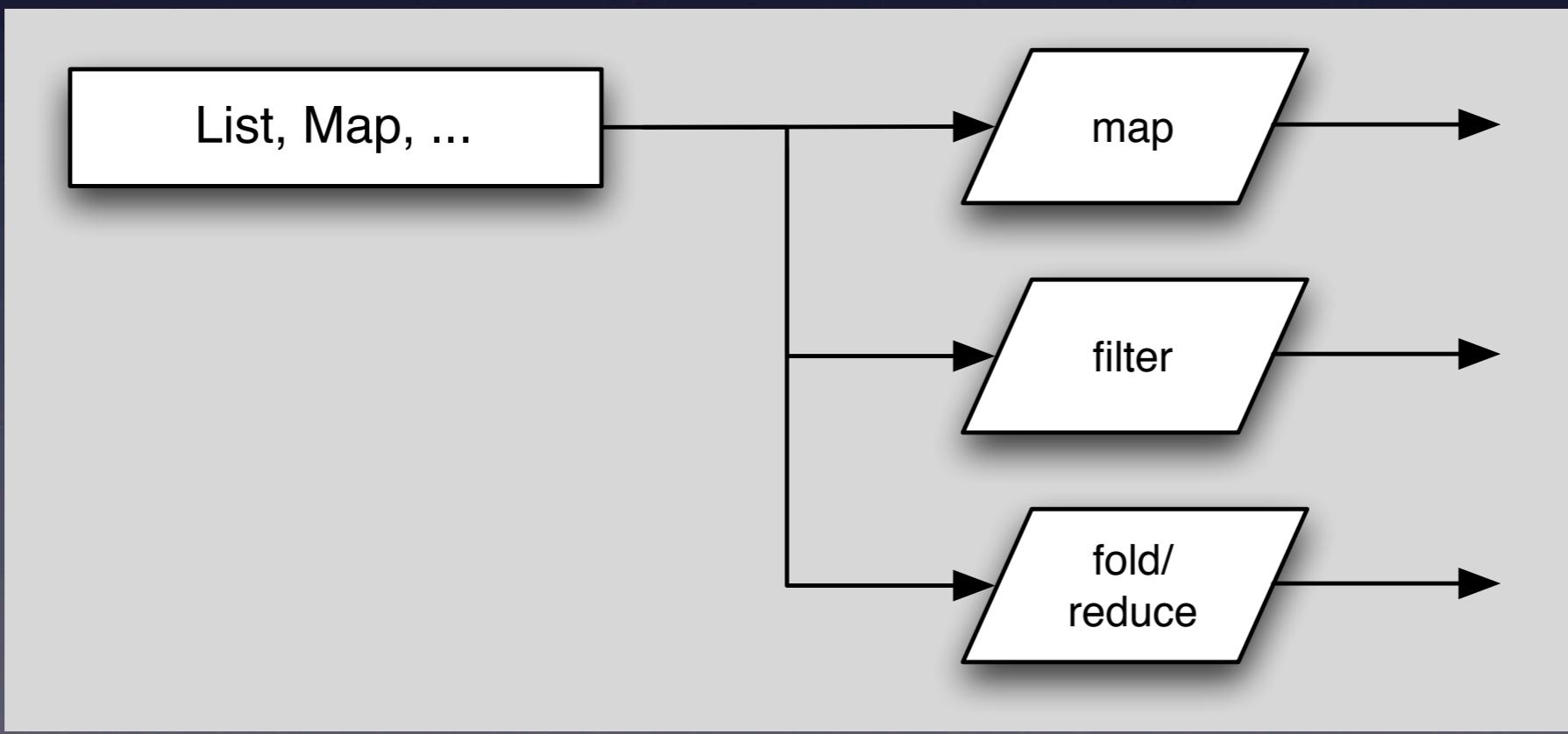
```
val map = Map(  
  Tuple2("name", "Dean"),  
  Tuple2("age", 39))
```

Similar mini-DSLs
have been defined
for other types.

Also in many third-party libraries.

Functions as Objects

Classic Operations on “Container” Types



37

Monday, June 17, 13

Collections like List and Map are containers. So are specialized containers like Option (Scala) or Maybe (Haskell) and other “monads”.

```
val list = "a" :: "b" :: Nil  
  
list map {  
    s => s.toUpperCase  
}  
  
// => "A" :: "B" :: Nil
```

map called on *list*
(dropping the ".")

argument to map
(using "{}" vs. "()")

list **map** {}
S => S.toUpperCase

}

*function
argument list*

function body

“function literal”

```
list map {  
    s => s.toUpperCase  
}  
  
list map {  
    (s:String) => s.toUpperCase  
}
```

inferred type

Explicit type

So far,
we have used
type inference
a lot...

How the Sausage Is Made

```
class List[A] {  
    ...  
    def map[B](f: A => B): List[B]  
    ...  
}
```

Parameterized type

Declaration of `map`

The function argument

map's return type

```
graph TD; A[Parameterized type] --> Class; B[Declaration of map] --> Method; C[The function argument] --> F; D[map's return type] --> Return
```

How the Sausage Is Made

like an “abstract” class

“contravariant”,
“covariant” typing

```
trait Function1[-A,+R] {
```

```
def apply(a:A): R
```

```
} ...
```

No method body:
=> abstract

What the Compiler Does

(s:String) => s.toUpperCase

What you write.

```
new Function1[String, String] {  
    def apply(s:String) = {  
        s.toUpperCase  
    }  
}
```

No “return”
needed

*What the compiler
generates*

An anonymous class

Functions as Objects

```
val list = "a" :: "b" :: Nil  
list map { ← {...} ok, instead of (...)  
    s => s.toUpperCase  
}  
  
// => "A" :: "B" :: Nil
```

Function “object”

A wide-angle photograph of a serene lake nestled in a mountainous region. The water is calm, reflecting the warm, golden light of the setting sun. In the background, a range of majestic mountains with snow-capped peaks rises against a clear sky. A small, dark island with a cluster of trees is visible in the middle distance. The overall atmosphere is peaceful and natural.

Big Data DSLs

46

Monday, June 17, 13

FP is going mainstream because it is the best way to write robust concurrent software. Here's an example...

Scalding: Scala DSL for Cascading

- *FP idioms* are a better fit for data than *objects*.
- <https://github.com/twitter/scalding>
- [http://blog.echen.me/2012/02/09/
movie-recommendations-and-more-
via-mapreduce-and-scalding/](http://blog.echen.me/2012/02/09/movie-recommendations-and-more-via-mapreduce-and-scalding/)

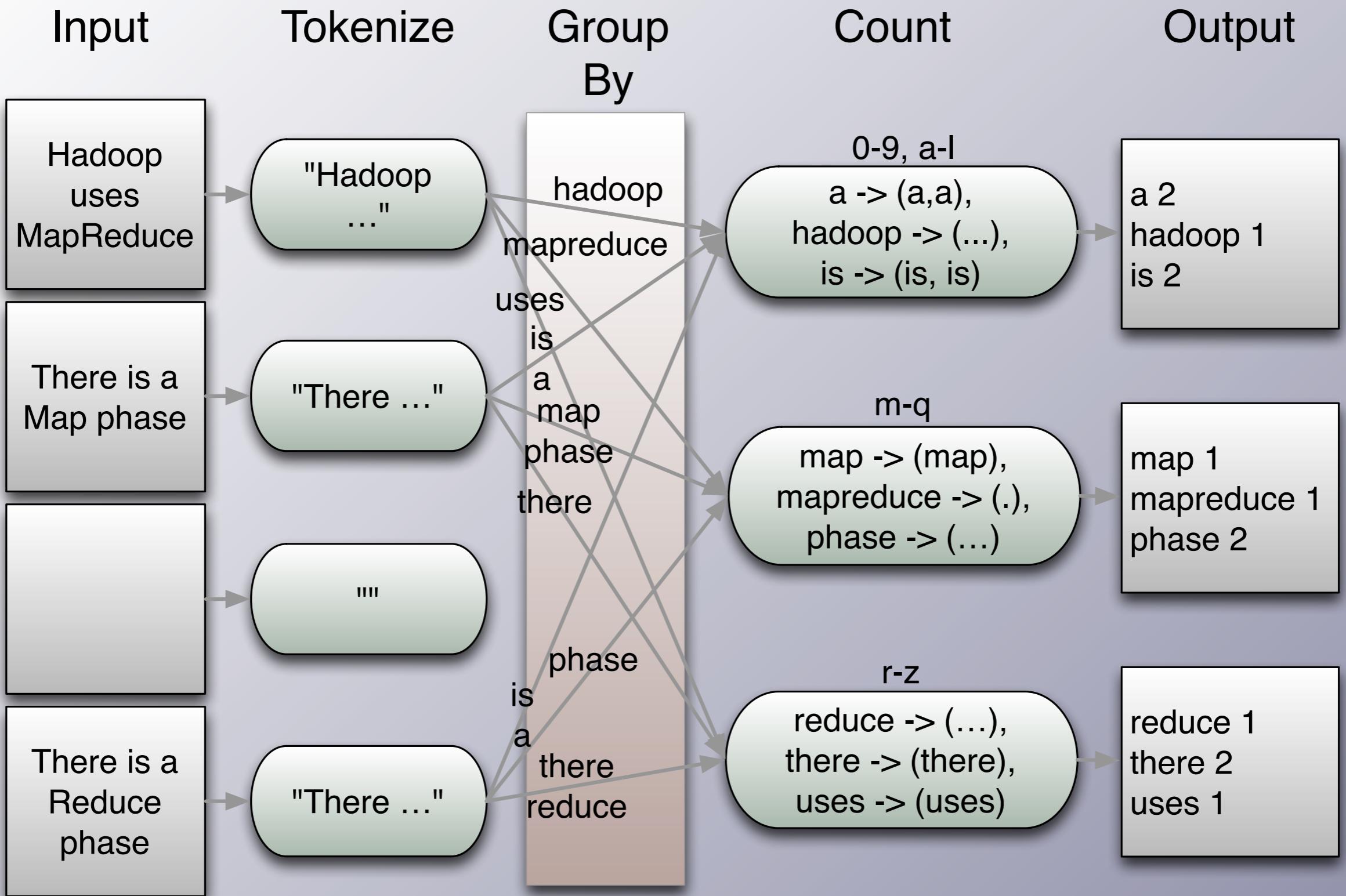
47

Monday, June 17, 13

Cascading is a Java toolkit for Hadoop that provides higher-level abstractions like pipes and filters composed into workflows. Using Scala makes it much easier to write concise, focused code.

Scalding is one of many Scala options. See also Scrunch, a Scala DSL for the Java Crunch library, and Spark, a different framework that can work with the Hadoop Distributed File System (HDFS).

Let's look at
the classic
Word Count
algorithm.



Schematic of the Word Count algorithm. Starting with 4 input docs (one of which is empty), tokenize it into words and emit every word found, then group by the words, finally counting them to output the word count to one or more files.

```
class WordCount(args : Args)
  extends Job(args) {
  TextLine(args("input"))
    .read
    .flatMap('line -> 'word) {
      line: String =>
      line.toLowerCase.split("\\s")
    }.groupBy('word) {
      group => group.size
    }.write(Tsv(args("output")))
}
```

```
class WordCount(args : Args)
  extends Job(args) {
  TextLine(args("input"))
    .read
    .flatMap('line -> 'word) {
      line: String =>
      line.toLowerCase.split("\\s")
    }.groupBy('word) {
      group => group.size
    }.write(Tsv(args("output")))
}
```

A workflow “job”.

```
class WordCount(args : Args)
  extends Job(args) {
    TextLine(args("input"))
      .read
      .flatMap('line ->
        line: String =>
        line.toLowerCase())
      .groupBy('word) {
        group => group.size
      }.write(Tsv(args("output")))
}
```

Read the text file
given by the “`--input`
...” argument.

```
class WordCount(args : Args)
  extends Job(args) {
  TextLine(args("input"))
    .read
    .flatMap('line -> 'word) {
      line: String =>
      line.toLowerCase.split("\\s")
    }.groupBy('word)
      group => group.size
    }.write(Tsv(args("output")))
}
```

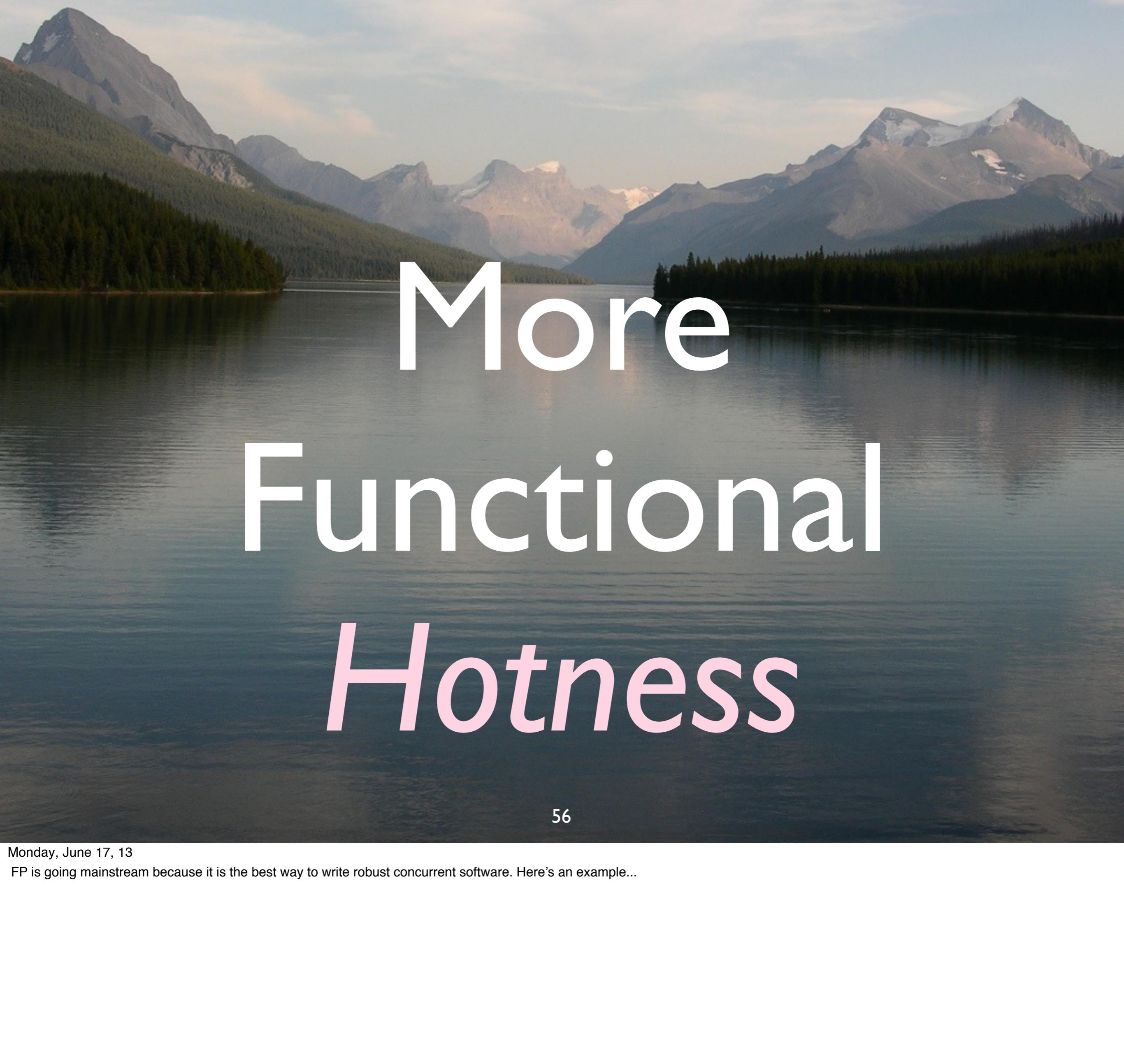
*Tokenize lines into
lower-case words.*

```
class WordCount(args : Args)
  extends Job(args) {
  TextLine(args("input"))
    .read
    .flatMap('line -> 'word) {
      line: String =>
      line.toLowerCase.split("\\s")
    }.groupBy('word) {
      group => group.size
    }.write(Tsv(args("output")))
}
```

*Group by word and
count each group size.*

```
class WordCount(args : Args)
  extends Job(args) {
  TextLine(args("input"))
    .read
    .flatMap('line -> 'word) {
      line: String =>
      line.toLowerCase.split("\\s")
    }.groupBy('word) {
      group => group.size
    }.write(Tsv(args("output")))
}
```

Write to tab-delim. output.

A scenic landscape featuring a calm lake in the foreground, framed by a dense forest of evergreen trees. In the background, a range of majestic mountains is visible, their peaks partially obscured by a soft, warm glow from the setting sun. The sky is a mix of pale blue and orange, creating a peaceful and inspiring atmosphere.

More Functional Hotness

56

Monday, June 17, 13

FP is going mainstream because it is the best way to write robust concurrent software. Here's an example...

Avoiding Nulls

```
sealed abstract class Option[+T]  
{...}
```

```
case class Some[+T](value: T)  
extends Option[T] {...}
```

```
case object None  
extends Option[Nothing] {...}
```

An Algebraic Data Type

57

Monday, June 17, 13

I am omitting MANY details. You can't instantiate Option, which is an abstraction for a container/collection with 0 or 1 item. If you have one, it is in a Some, which must be a class, since it has an instance field, the item. However, None, used when there are 0 items, can be a singleton object, because it has no state! Note that type parameter for the parent Option. In the type system, Nothing is a subclass of all other types, so it substitutes for instances of all other types. This combined with a property called covariant subtyping means that you could write "val x: Option[String] = None" and it would type correctly, as None (and Option[Nothing]) is a subtype of Option[String]. Note that Options[+T] is only covariant in T because of the "+" in front of the T.

Also, Option is an algebraic data type, and now you know the scala idiom for defining one.

```
// Java style (schematic)
class Map[K, V] {
    def get(key: K): V = {
        return value || null;
    }
}
```

```
// Scala style
class Map[K, V] {
    def get(key: K): Option[V] = {
        return Some(value) || None;
    }
}
```

Which is the better API?

In Use:

```
val m = Literal syntax for map creation  
  Map("one" -> 1, "two" -> 2)
```

...

```
val n = m.get("four") match {  
  case Some(i) => i  
  case None      => 0 // default  
}
```

Use pattern matching to extract the value (or not)

Option Details: sealed

```
sealed abstract class Option[+T]  
{...}
```

*All children must be defined
in the same file*

60

Monday, June 17, 13

I am omitting MANY details. You can't instantiate Option, which is an abstraction for a container/collection with 0 or 1 item. If you have one, it is in a Some, which must be a class, since it has an instance field, the item. However, None, used when there are 0 items, can be a singleton object, because it has no state! Note that type parameter for the parent Option. In the type system, Nothing is a subclass of all other types, so it substitutes for instances of all other types. This combined with a proper called covariant subtyping means that you could write "val x: Option[String] = None" it would type correctly, as None (and Option[Nothing]) is a subtype of Option[String].

Case Classes

```
case class Some[+T](value: T)
```

- Provides factory method, pattern matching, equals, toString, etc.
- Makes “value” a field without **val** keyword.

Object

```
case object None  
extends Option[Nothing] {...}
```

A “singleton”. Only one *instance* will exist.

62

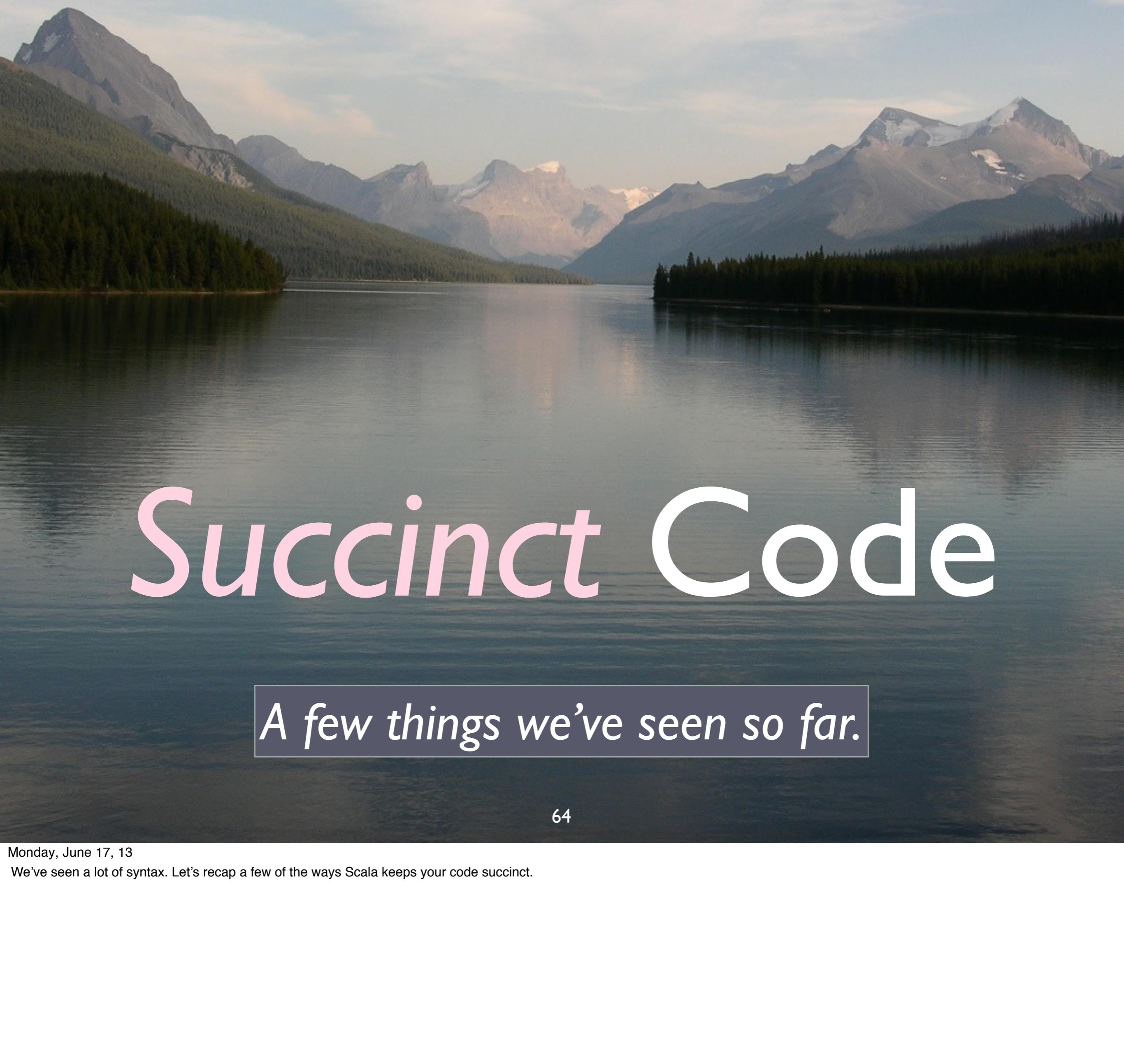
Monday, June 17, 13

The scala runtime controls (lazy) instantiation of the single instance. Since the user can't instantiate the instance, objects can't have constructor argument lists, but they are allowed to define fields inside the class body (i.e., primary - and only - constructor body).

Nothing

```
case object None  
  extends Option[Nothing] {...}
```

Special child type of all other types. Used for this special case where no actual instances required.

The background of the slide features a wide-angle photograph of a mountainous landscape. In the foreground is a calm lake with a small, dark island in the center. The middle ground shows a range of mountains with dense forests on their lower slopes. The background consists of more mountain peaks, some with snow or ice, under a clear sky.

Succinct Code

A few *things* we've seen so far.

64

Monday, June 17, 13

We've seen a lot of syntax. Let's recap a few of the ways Scala keeps your code succinct.

Infix Operator Notation

"hello" + "world"

same as

"hello".+("world")

Type Inference

```
// Java  
HashMap<String, Person> persons =  
new HashMap<String, Person>();
```

VS.

```
// Scala  
val persons  
= new HashMap[String, Person]
```

Type Inference

```
// Java
```

```
HashMap<String, Person> persons =  
    new HashMap<String, Person>();
```

VS.

```
// Scala
```

```
val persons  
  = new HashMap[String, Person]
```

67

Monday, June 17, 13

Java (and to a lesser extent C#) require explicit type “annotations” on all references, method arguments, etc., leading to redundancy and noise.
Note that Scala uses [] rather than <>, so you can use << and >> as method names!

```
// Scala  
val persons  
= new HashMap[String, Person]
```

↑
*no () needed.
Semicolons inferred.*

User-defined Factory Methods

```
val words =  
  List("Scala", "is", "fun!")
```

*no **new** needed.
Can return a subtype!*

```
class Person {  
    private String firstName;  
    private String lastName;  
    private int    age;  
  
    public Person(String firstName, String lastName, int age){  
        this.firstName = firstName;  
        this.lastName  = lastName;  
        this.age       = age;  
    }  
  
    public void String getFirstName() {return this.firstName;}  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public void String getLastname() {return this.lastName;}  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public void int getAge() {return this.age;}  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

Typical Java

70

Monday, June 17, 13

Typical Java boilerplate for a simple “struct-like” class.
Deliberately too small to read...

```
class Person(  
    var firstName: String,  
    var lastName: String,  
    var age: Int)
```

Typical Scala!

*Class body is the
“primary” constructor*

Parameter list for c’tor

```
class Person(  
    var firstName: String,  
    var lastName: String,  
    var age: Int)
```

*Makes the arg a field
with accessors*

*No class body {...}.
nothing else needed!*

Even Better...

```
case class Person(  
  firstName: String,  
  lastName: String,  
  age: Int)
```

*Constructor args are automatically
vals, plus other goodies.*

73

Monday, June 17, 13

Case classes also get equals, hashCode, toString, and the “factory” method we’ve mentioned (e.g., for List and Map) that, by default, creates an instance of the type without the “new”. We’ll see more about case classes later.

The background of the slide features a wide-angle photograph of a mountainous landscape. In the foreground, there is a calm lake reflecting the light. A dense forest of evergreen trees lines the shore of the lake. In the background, several majestic mountains rise, their peaks partially obscured by a hazy sky. The lighting suggests either sunrise or sunset, casting a warm glow over the scene.

Scala's Object Model: *Traits*

Composable Units of Behavior

74

Monday, June 17, 13

Fixes limitations of Java's object model.

We would like to
compose *objects*
from *mixins*.

Java: What to Do?

```
class Server  
  extends Logger { ... }
```

“Server is a Logger”?

```
class Server  
  implements Logger { ... }
```

Logger isn’t an interface!

Java's object model

- *Good*
 - Promotes abstractions.
- *Bad*
 - No *composition* through reusable *mixins*.

Traits

Like interfaces with
implementations,

Traits

... or like
abstract classes +
multiple inheritance
(if you prefer).

Logger as a Mixin:

```
trait Logger {  
    val level: Level // abstract  
  
    def log(message: String) = {  
        Log4J.log(level, message)  
    }  
}
```

Traits don't have constructors, but you can still define fields.

Logger as a Mixin:

```
trait Logger {  
    val level: Level // abstract  
    ...  
}  
  
val server =  
    new Server(...) with Logger {  
        val level = ERROR  
    }  
server.log("Internet down! !")
```

mixed in Logging

↓
*abstract
member defined*

The background of the slide features a wide-angle photograph of a serene lake nestled in a mountainous region. The mountains in the distance are partially covered in snow, and their reflections are visible on the calm water. A dense forest of evergreen trees lines the shore of the lake. The overall atmosphere is peaceful and natural.

Actor Concurrency

82

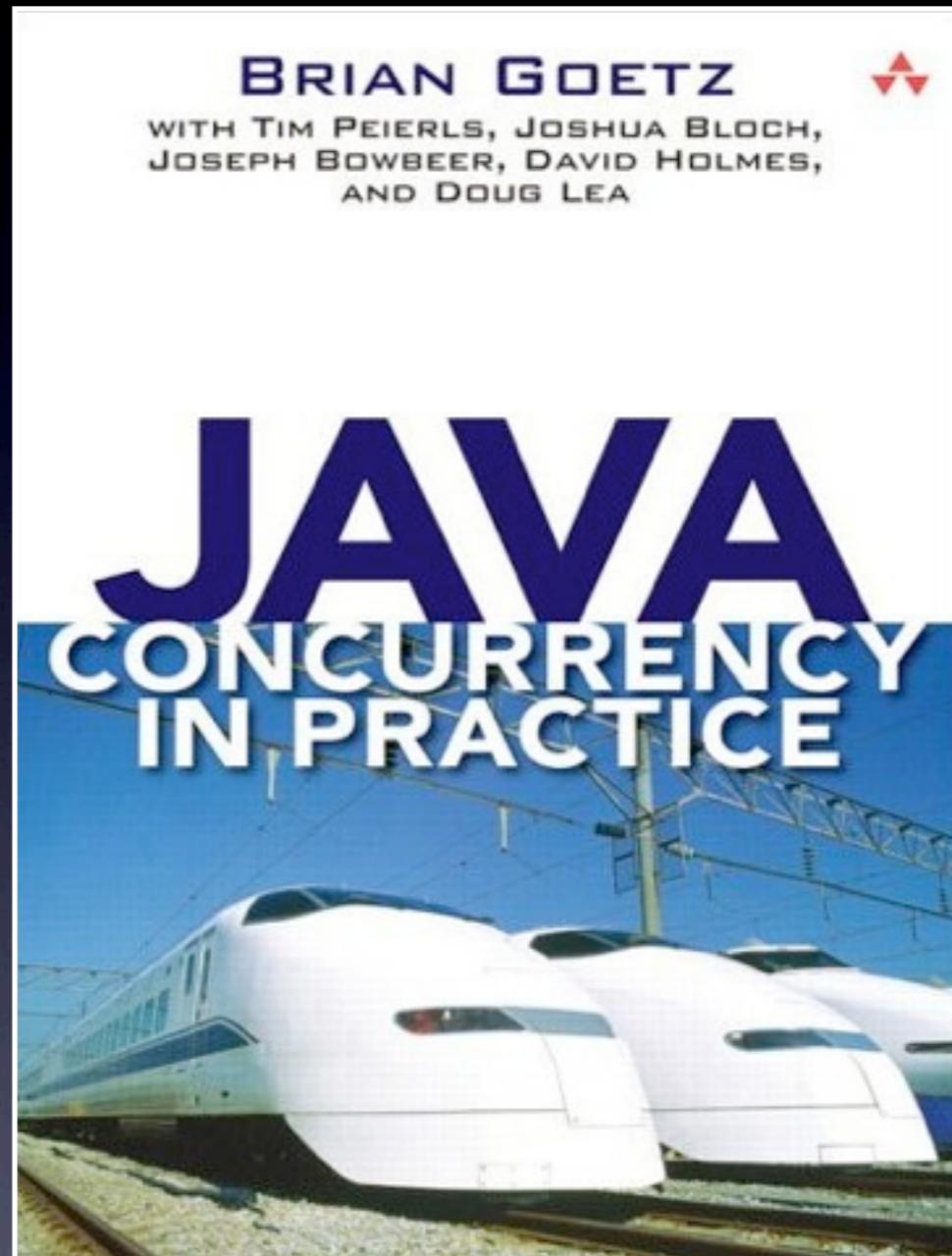
Monday, June 17, 13

FP is going mainstream because it is the best way to write robust concurrent software. Here's an example...

NOTE: The full source for this example is at <https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala/code-examples/actor>.

When you share mutable state...

Hic sunt dracones
(Here be dragons)



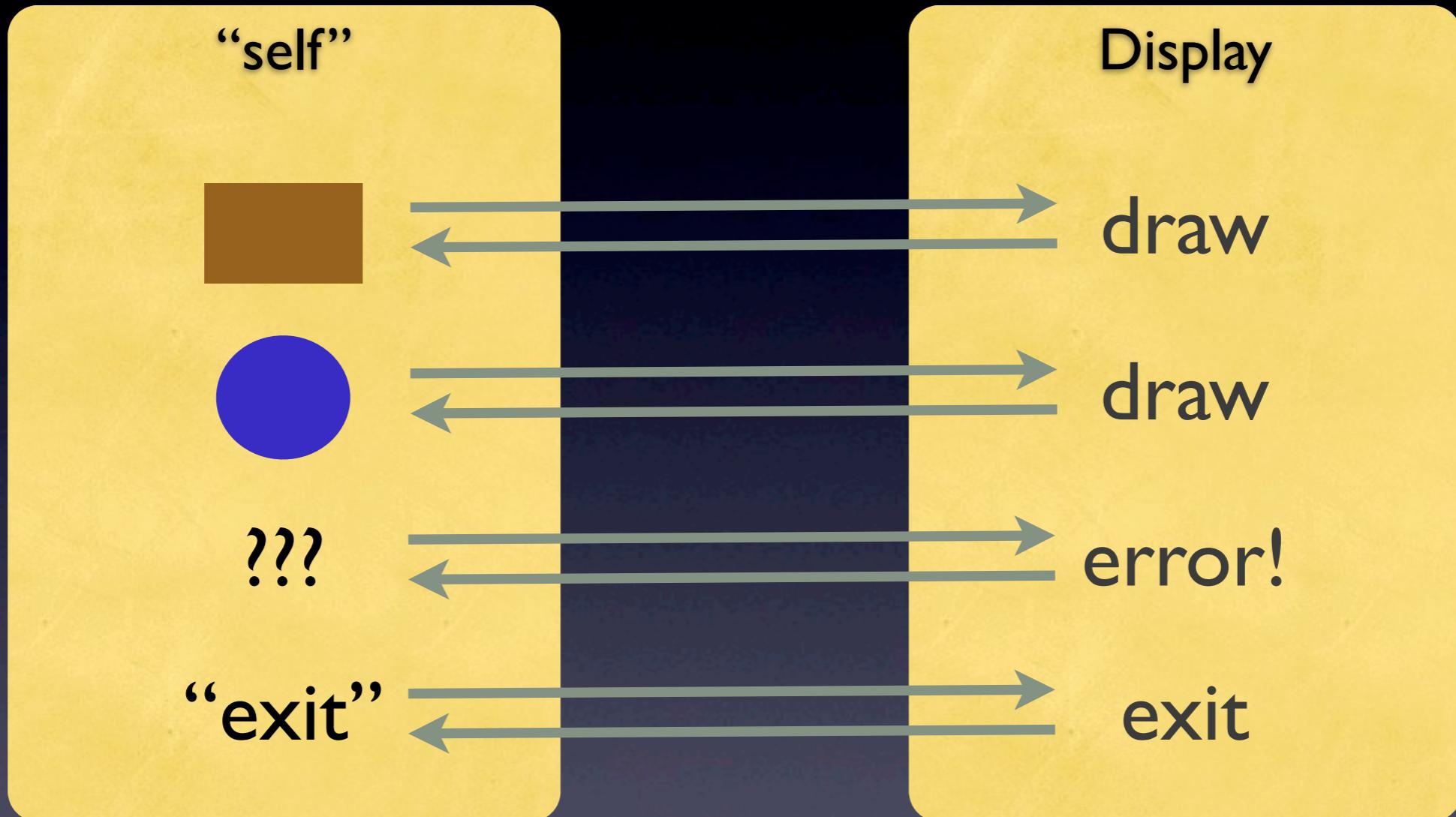
Actor Model

- Message passing between autonomous *actors*.
- No shared (mutable) state.

Actor Model

- First developed in the 70's by Hewitt, Agha, Hoare, etc.
- Made “famous” by *Erlang*.
 - Scala’s Actors patterned after Erlang’s.

2 Actors:



```
package shapes
```

```
case class Point(  
  x: Double, y: Double)
```

```
abstract class Shape {  
  def draw()  
}
```

abstract “draw” method

Hierarchy of geometric shapes

87

Monday, June 17, 13

“Case” classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape. The “case” keyword makes the arguments “vals” by default, adds factory, equals, etc. methods. Great for “structural” objects.

(Case classes automatically get generated equals, hashCode, toString, so-called “apply” factory methods - so you don’t need “new” - and so-called “unapply” methods used for pattern matching.)

NOTE: The full source for this example is at <https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala/code-examples/actor>.

```
case class Circle(  
  center: Point, radius: Double)  
  extends Shape {  
    def draw() = ...  
  }
```

*concrete “draw”
methods*

```
case class Rectangle(  
  ll: Point, h: Double, w: Double)  
  extends Shape {  
    def draw() = ...  
  }
```

```
package shapes  
import akka.actor._
```

*Use the “Akka”
Actor library*

```
class ShapeDrawingActor extends Actor {  
    def receive = {  
        ...  
    }  
}
```

*receive and handle
each message*

Actor for drawing shapes

89

Monday, June 17, 13

An actor that waits for messages containing shapes to draw. Imagine this is the window manager on your computer. It loops indefinitely, blocking until a new message is received...

Note: This example uses the Akka Frameworks Actor library (see <http://akka.io>), which I prefer over Scala’s native actors.

Receive
method

```
receive = {  
    case s:Shape =>  
        print("-> "); s.draw()  
        self.reply("Shape drawn.")  
    case "exit" =>  
        println("-> exiting...")  
        self.reply("good bye!")  
    case x =>          // default  
        println("-> Error: " + x)  
        self.reply("Unknown: " + x)  
}
```

Actor for drawing shapes

90

Monday, June 17, 13

“Receive” blocks until a message is received. Then it does a pattern match on the message. In this case, looking for a Shape object, the “exit” message, or an unexpected object, handled with the last case, the default.

```

receive = {
    case s:Shape =>
        print("-> "); s.draw()
        self.reply("Shape drawn")
    case "exit" =>
        println("-> exiting...")
        self.reply("good bye!")
    case x => // default
        println("-> Error: " + x)
        self.reply("Unknown: " + x)
}

```

*pattern
matching*

Actor for drawing shapes

```

receive = {
  case s:Shape =>
    print("-> "); s.draw()
    self.reply("Shape drawn.")
  case "exit" =>
    println("-> exiting...")
    self.reply("good bye!")
  case x => // default
    println("-> Error: " + x)
    self.reply("Unknown: " + x)
}

```

*draw shape
& send reply*

done

unrecognized message

```
package shapes
import akka.actor._
class ShapeDrawingActor extends Actor {
  receive = {
    case s:Shape =>
      print("-> "); s.draw()
      self.reply("Shape drawn.")
    case "exit" =>
      println("-> exiting...")
      self.reply("good bye!")
    case x =>           // default
      println("-> Error: " + x)
      self.reply("Unknown: " + x)
  }
}
```

Altogether

```
import shapes._  
import akka.actor._  
import akka.actor.Actor
```

a “singleton” type to hold main

```
object Driver {  
  def main(args: Array[String]) = {  
    val driver = actorOf[Driver]  
    driver.start  
    driver ! "go!"  
  }  
}  
class Driver ...
```

*! is the message
send “operator”*

driver to try it out

Its “companion” object Driver
was on the previous slide.

```
...  
class Driver extends Actor {  
    val drawer =  
        actorOf[ShapeDrawingActor]  
    drawer.start  
    def receive = {  
        ...  
    }  
}
```

driver to try it out

```

def receive = {
    case "go!" => ← sent by main
        drawer ! Circle(Point(...),...)
        drawer ! Rectangle(...)
        drawer ! 3.14159
        drawer ! "exit"
    case "good bye!" => ← sent by drawer
        println("<- cleaning up...")
        drawer.stop; self.stop
    case other =>
        println("<- " + other)
}

```

driver to try it out

```
case "go!" =>
  drawer ! Circle(Point(...),...)
  drawer ! Rectangle(...)
  drawer ! 3.14159
  drawer ! "exit"
```

```
// run Driver.main (see github README.md)
-> drawing: Circle(Point(0.0,0.0),1.0)
-> drawing: Rectangle(Point(0.0,0.0),
2.0,5.0)
-> Error: 3.14159
-> exiting...
<- Shape drawn.
<- Shape drawn.
<- Unknown: 3.14159
<- cleaning up...
```

“<” and “->” messages
may be interleaved!!

97

Monday, June 17, 13

NOTE: The full source for this example is at <https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala/code-examples/actor>. See the README.md for instructions.

Note that the -> messages will always be in the same order and the <- will always be in the same order, but the two groups may be interleaved!!

```
...  
// ShapeDrawingActor  
receive = {  
    case s:Shape =>  
        s.draw() ←  
        self.reply("...")  
    case ...  
    case ...  
}  
...
```

*Functional-style
pattern matching*

*Object-
oriented-style
polymorphism*

**“Switch” statements
are not evil!**



Recap

99

Scala is...

100

a *better* Java,

*object-oriented
and
functional,*

*succinct,
elegant,
and
powerful.*

Questions?

Dean Wampler
dean@deanwampler.com
[@deanwampler](https://twitter.com/deanwampler)
polyglotprogramming.com/talks

April 16, 2012



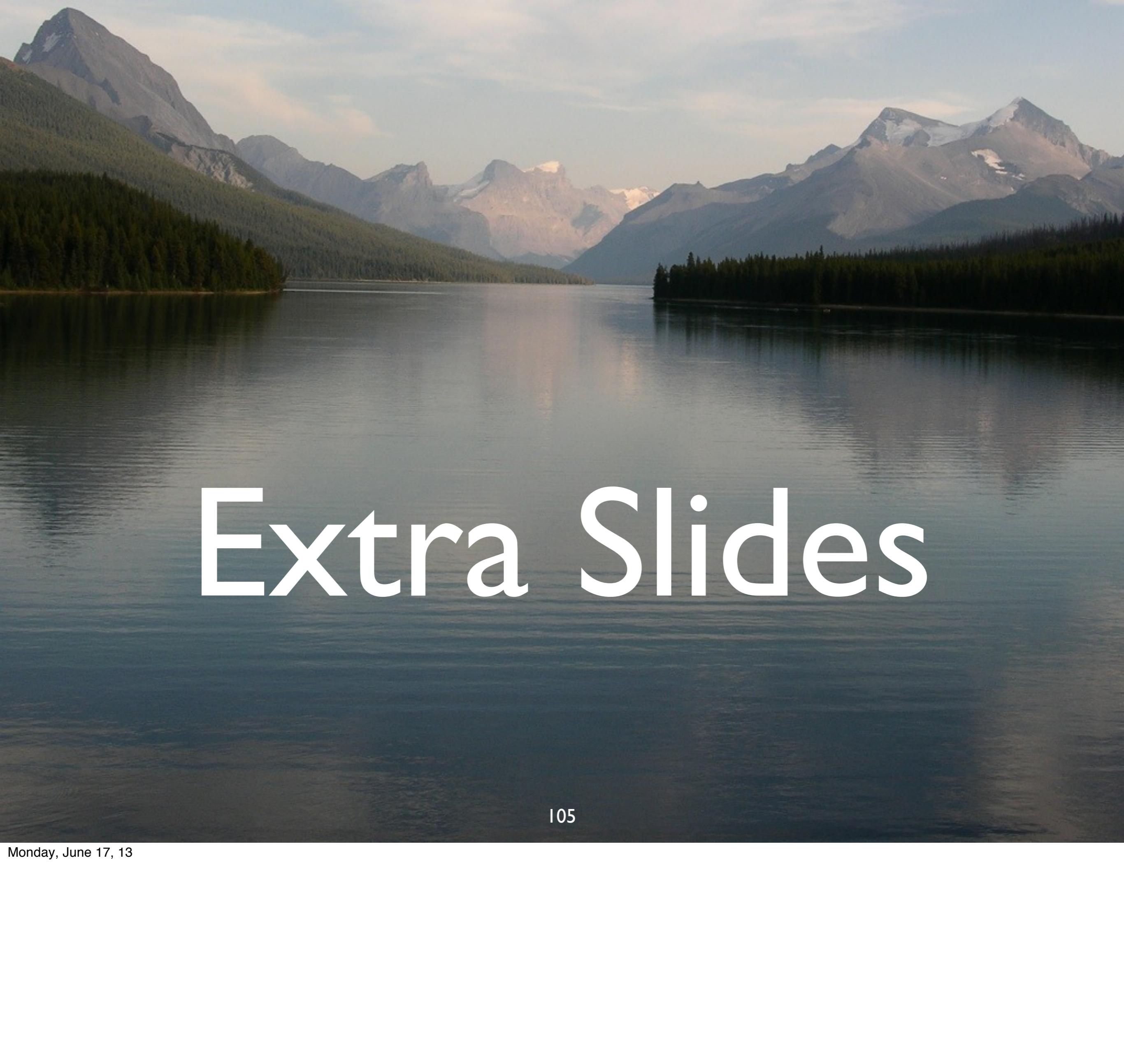
104

Monday, June 17, 13

The online version contains more material. You can also find this talk and the code used for many of the examples at github.com/deanwampler/Presentations/tree/master/SeductionsOfScala.

Copyright © 2010–2013, Dean Wampler. Some Rights Reserved – All use of the photographs and image backgrounds are by written permission only. The content is free to reuse, but attribution is requested.

<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

A wide-angle photograph of a serene lake nestled in a mountainous region. The foreground is dominated by the dark, calm water of the lake. In the background, a range of majestic mountains rises, their peaks partially obscured by a light, hazy sky. The mountains are covered with dense forests of evergreen trees, particularly visible along the shoreline. The overall atmosphere is peaceful and natural.

Extra Slides

105

Modifying Existing Behavior with Traits

106

Example

```
trait Queue[T] {  
    def get(): T  
    def put(t: T)  
}
```

A *pure abstraction* (in this case...)

107

Log put

```
trait QueueLogging[T]
  extends Queue[T] {
  abstract override def put(
    t: T) = {
    println("put(" + t + ")")
    super.put(t)
  }
}
```

Log put

```
trait QueueLogging[T]
  extends Queue[T] {
    abstract override def put(
      t: T) = {
      println("put(" + t + ")")
      super.put(t)
    }
}
```

What is “super” bound to??

```
class StandardQueue[T]
  extends Queue[T] {
  import ...ArrayBuffer
  private val ab =
    new ArrayBuffer[T]
  def put(t: T) = ab += t
  def get() = ab.remove(0)
  ...
}
```

Concrete (boring) implementation

110

Monday, June 17, 13

Our concrete class. We import `scala.collection.mutable.ArrayBuffer` wherever we want, in this case, right where it's used. This is boring; it's just a vehicle for the cool traits stuff...

```
val sq = new StandardQueue[Int]  
  with QueueLogging[Int]
```

```
sq.put(10)           // #1  
println(sq.get())   // #2  
// => put(10)      (on #1)  
// => 10            (on #2)
```

Example use

III

Monday, June 17, 13

We instantiate StandardQueue AND mixin the trait. We could also declare a class that mixes in the trait.
The “put(10)” output comes from QueueLogging.put. So “super” is StandardQueue.

Mixin composition;
no class required

```
val sq = new StandardQueue[Int]  
with QueueLogging[Int]
```

```
sq.put(10)           // #1  
println(sq.get())   // #2  
// => put(10)      (on #1)  
// => 10            (on #2)
```

Example use

112

Monday, June 17, 13

We instantiate StandardQueue AND mixin the trait. We could also declare a class that mixes in the trait.
The "put(10)" output comes from QueueLogging.put. So "super" is StandardQueue.

Like Aspect-Oriented Programming?

Traits give us *advice*,
but not a *join point*
“query” *language*.

*Traits are a powerful
composition
mechanism!*

114

Monday, June 17, 13

Not shown, nesting of traits...

The background of the slide features a wide-angle photograph of a serene lake nestled among majestic, rugged mountains. The sky is a soft, warm orange and yellow, suggesting either sunrise or sunset. The water of the lake is calm, with gentle ripples, and its surface reflects the surrounding mountains and the sky. In the foreground, the dark, silhouetted shapes of coniferous trees are visible along the shore. The overall atmosphere is peaceful and inspiring.

Functional Programming

115

What is Functional Programming?

Don't we already write “functions”?

$y = \sin(x)$

Based on *Mathematics*

117

Monday, June 17, 13

“Functional Programming” is based on the behavior of mathematical functions and variables.

$$y = \sin(x)$$

Setting x fixes y

\therefore variables are *immutable*

`20 += | ??`

We never modify
the 20 “object”

Concurrency

No *mutable state*

\therefore *nothing to synchronize*

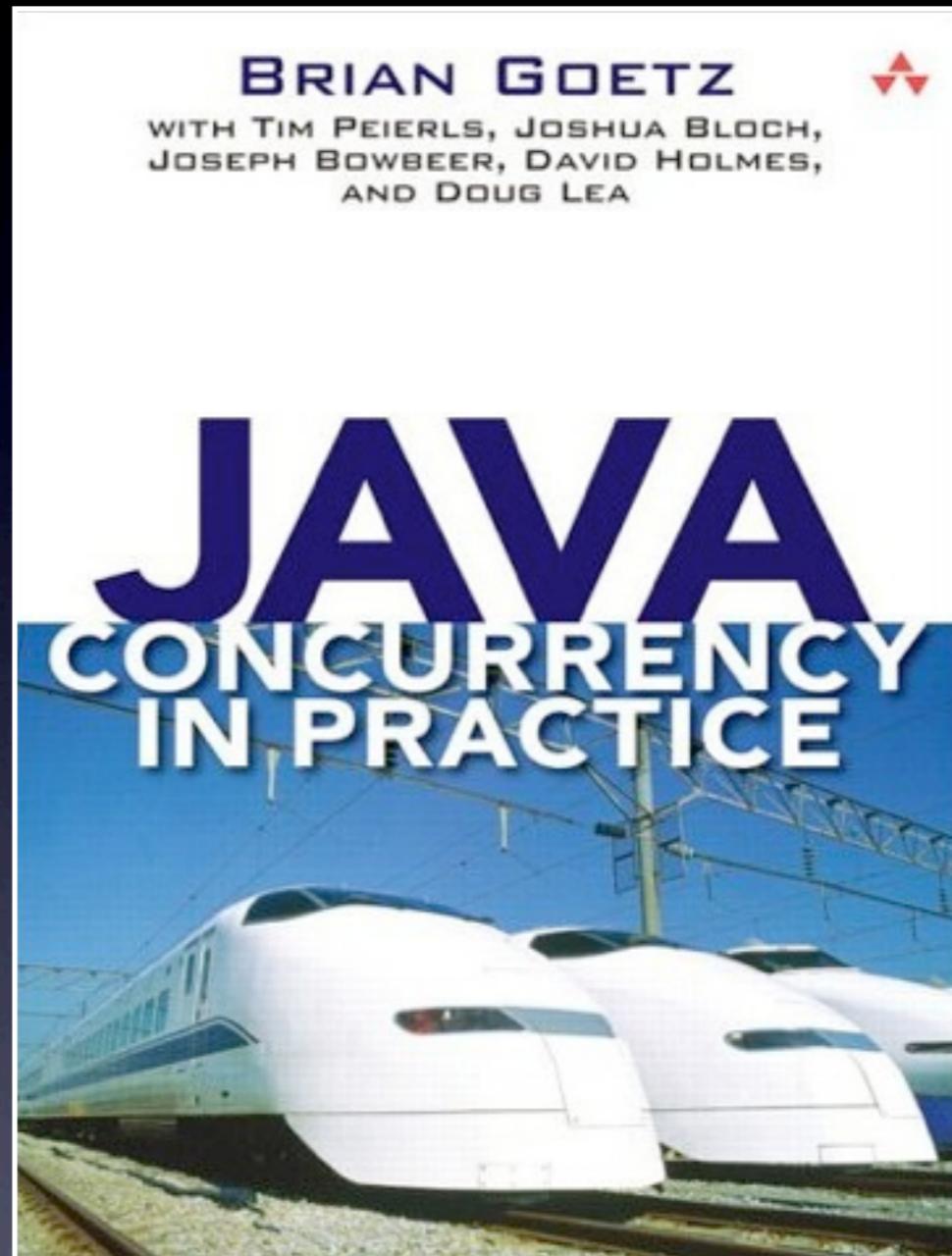
120

Monday, June 17, 13

FP is breaking out of the academic world, because it offers a better way to approach concurrency, which is becoming ubiquitous.

When you share mutable state...

Hic sunt dracones
(Here be dragons)



$$y = \sin(x)$$

Functions don't
change state
∴ *side-effect free*

122

Monday, June 17, 13

A math function doesn't change any "object" or global state. All the work it does is returned by the function. This property is called "referential transparency".

Side-effect free functions

- Easy to *reason about behavior*.
- Easy to invoke *concurrently*.
- Easy to invoke *anywhere*.
- Encourage *immutable objects*.

123

Monday, June 17, 13

side-effect free functions are far less likely to introduce subtle integration bugs, especially in concurrent systems. By encouraging immutable objects (e.g., when methods return new objects, rather than modify existing ones), that improves concurrency robustness.

$$\tan(\Theta) = \sin(\Theta)/\cos(\Theta)$$

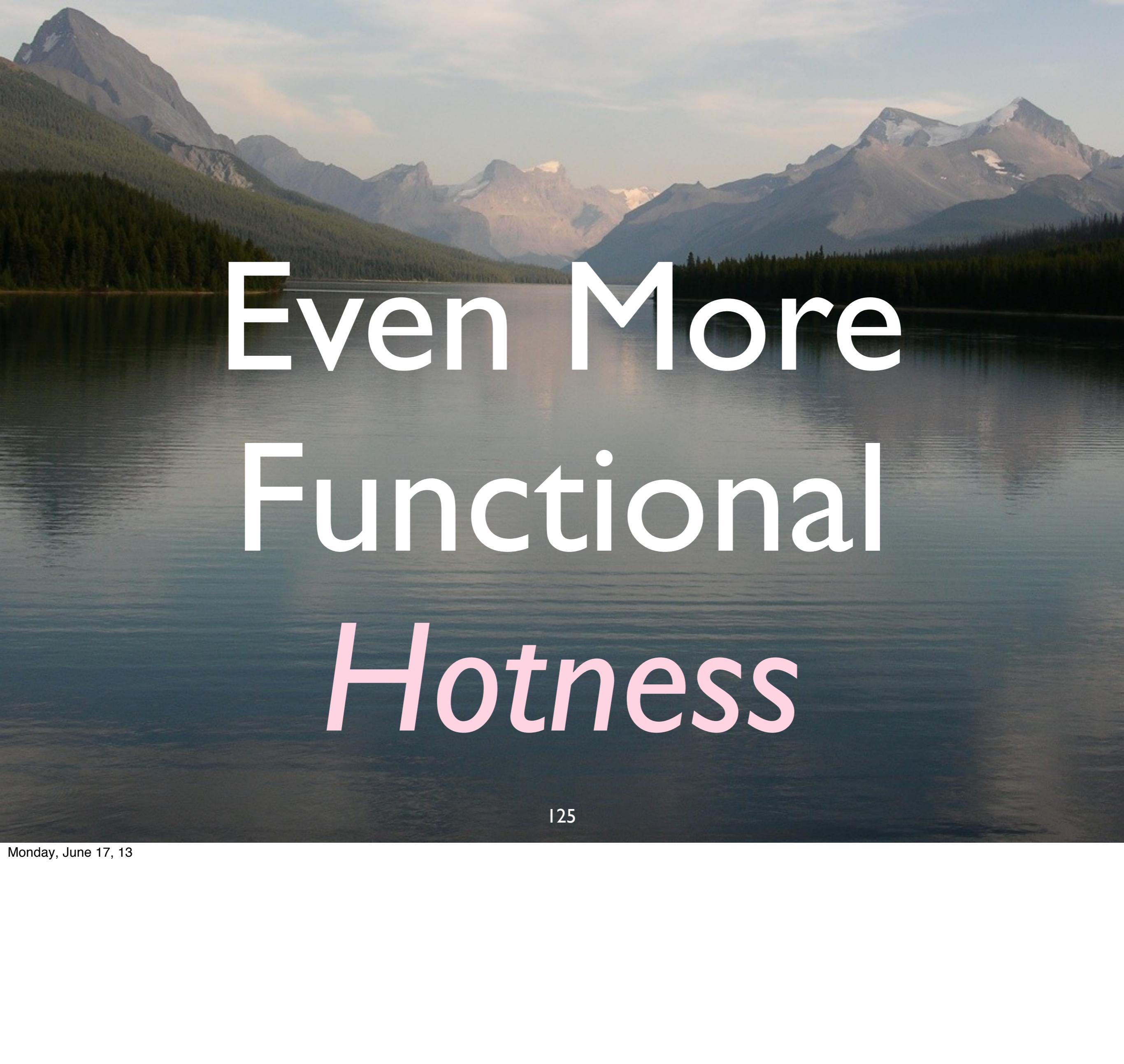
*Compose functions of
other functions*

∴ first-class citizens

124

Monday, June 17, 13

Function are “first-class citizens”; you can assign them to variables and pass them to other (higher-order) functions, giving you composable behavior.

A scenic landscape featuring a calm lake in the foreground, a dense forest along the shore, and a range of majestic mountains in the background under a clear sky.

Even More Functional Hotness

125

For “Comprehensions”

```
val l = List(  
  Some("a"), None, Some("b"),  
  None, Some("c"))
```

```
for (Some(s) <- l) yield s  
// List(a, b, c)
```

No “if” statement

*Pattern match; only
take elements of “l”
that are Somes.*

126

Monday, June 17, 13

We’re using the type system and pattern matching built into case classes to discriminate elements in the list. No conditional statements required. This is just the tip of the iceberg of what “for comprehensions” can do and not only with Options, but other containers, too.