



The Seductions of Scala

Dean Wampler

dean.wampler@typesafe.com

[@deanwampler](https://twitter.com/deanwampler)

polyglotprogramming.com/talks

November 19, 2013



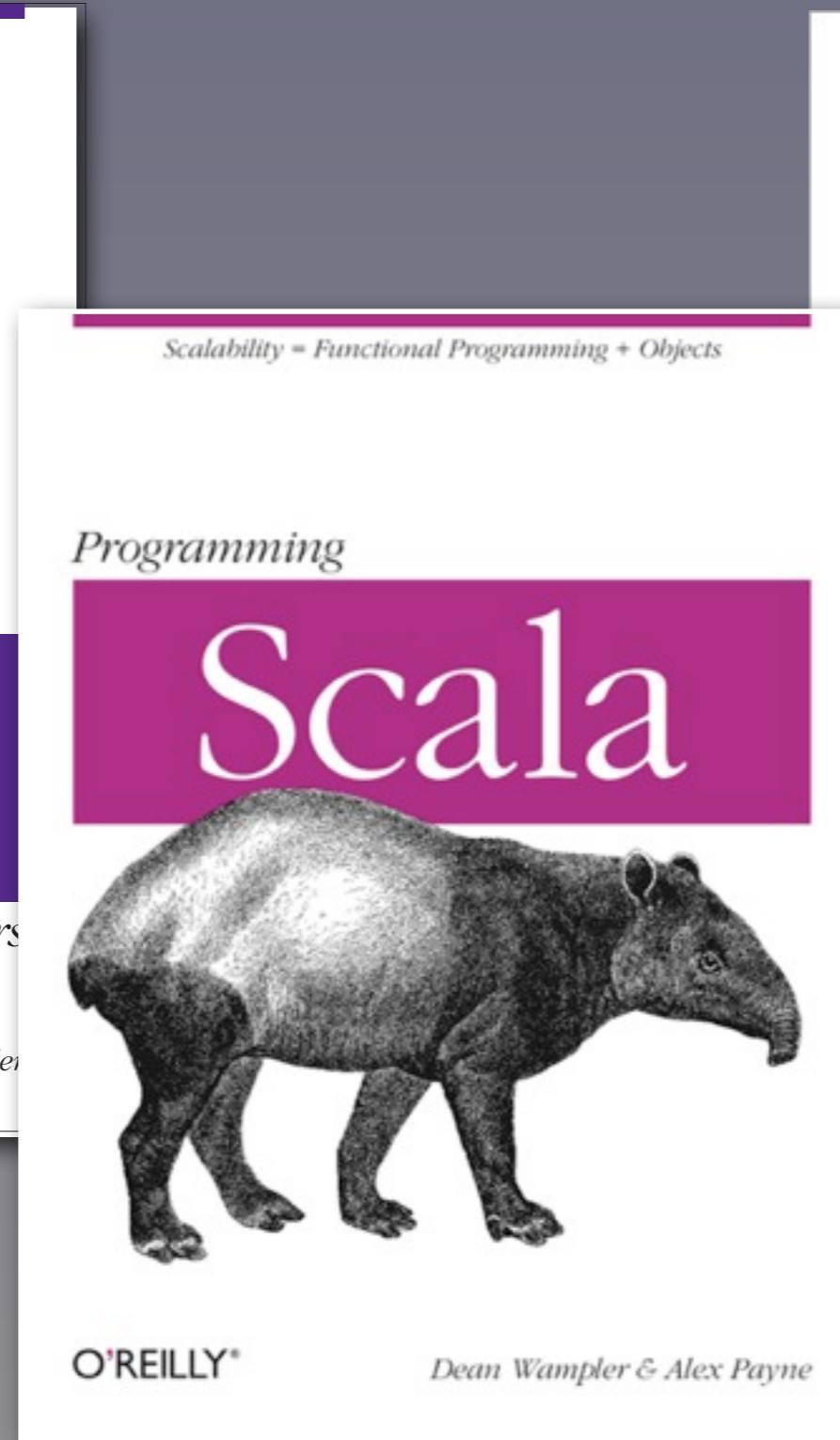
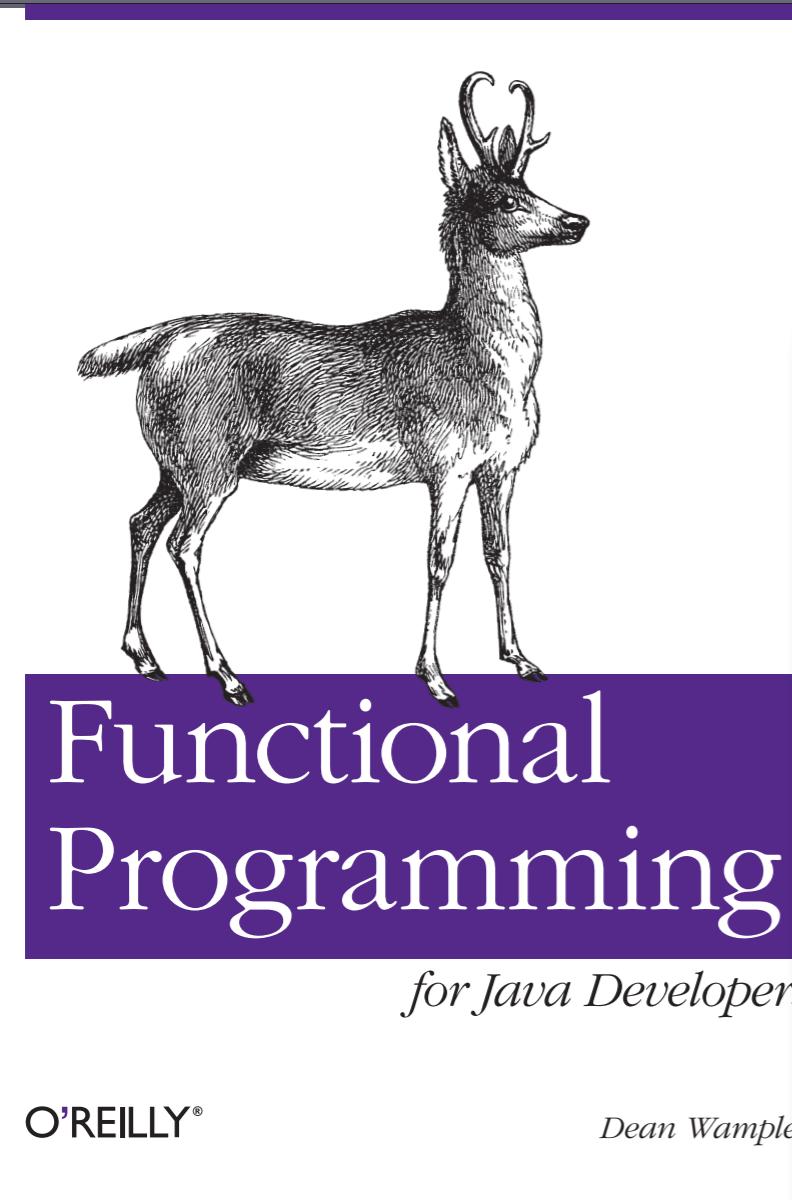
1

Saturday, November 16, 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>



<shameless-plug/>

2

Saturday, November 16, 13

Available now from oreilly.com, Amazon, etc.

Copious Data: The “Killer App” for Functional Programming

This Thursday!

GOTO Chicago Nights

<http://gotochicago.eventbrite.com/>

<shameless-plug/>

Copyright © 2011-2013, Dean Wampler, All Rights Reserved

Saturday, November 16, 13

Want to hear more about why FP is important today (with or without Scala), join me this Thursday night for the GOTO Chicago Nights session at the Thoughtworks Office, Aon building.

Why do we need a new language?

4

Saturday, November 16, 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.

#1

We need *Functional* Programming

• • •

5

- ... 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 need
to keep
our *investment*
in *Java*.

10

Saturday, November 16, 13

We rarely have the luxury of starting from scratch...

Scala is...

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

II

Saturday, November 16, 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 CS theory and industry's needs.

A wide-angle photograph of a serene lake nestled in a mountainous region. The foreground is dominated by the calm water of the lake. In the middle ground, two small, densely forested islands are visible. The background features a range of majestic mountains with their peaks partially covered in snow. The sky above is a clear, pale blue.

*Objects
can be
Functions*

13

Saturday, November 16, 13

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

```
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

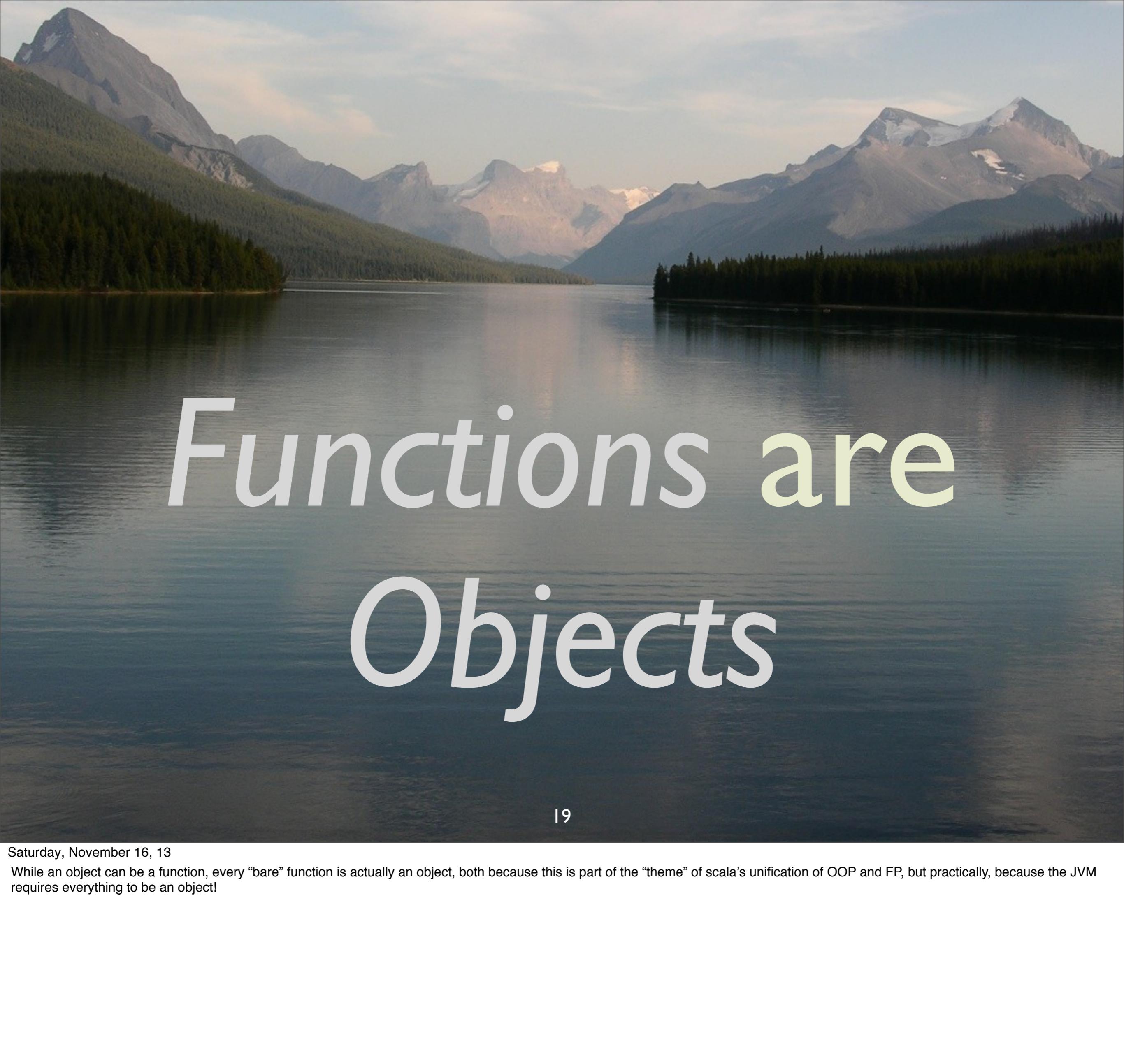
...
error("Network error.")

“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, indicating 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.

Functions are Objects

19

Saturday, November 16, 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!



First, let's
discuss
Lists and Maps

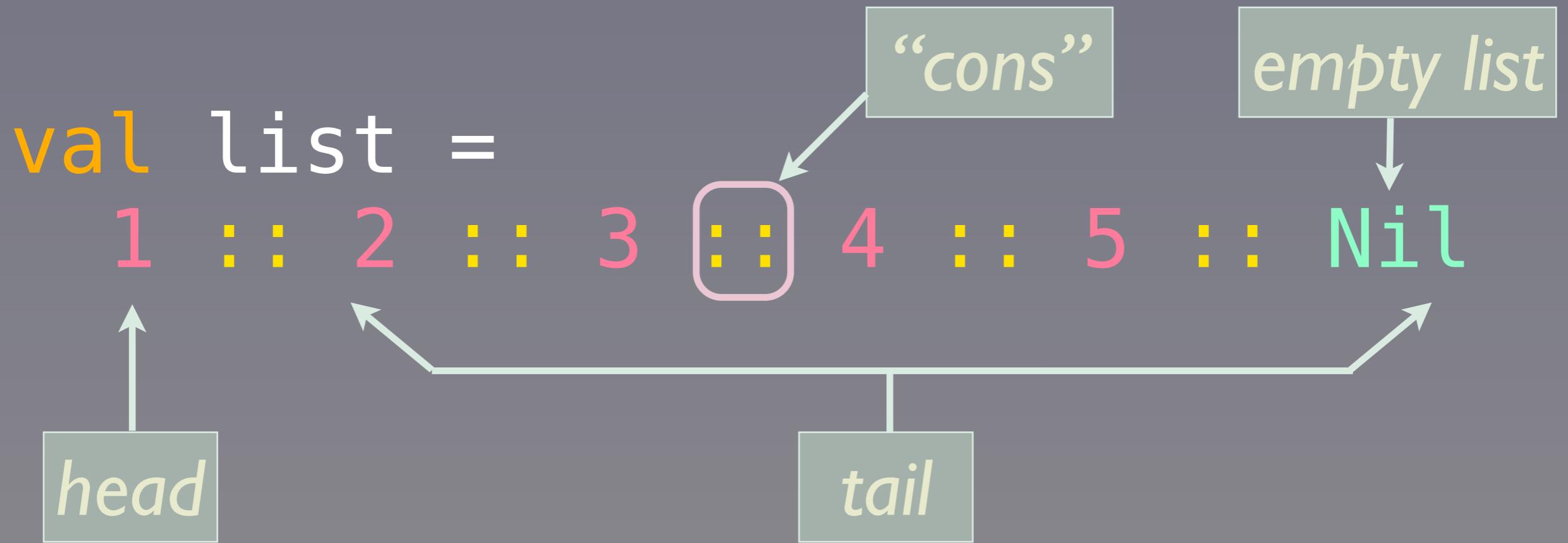
Lists

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")

Note:
Int, Double, etc.
are true *objects*, but
Scala compiles them
to *primitives*.

This means that generics just work.

```
val l = List.empty[Int]
```

An empty list of Ints.

Java: ... List<Int>

Maps

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

Saturday, November 16, 13

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

(Why is there no “new” again? There is a companion object named “Map”, like the one for List, with an apply method that functions as a factory.)

Maps

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

“baked” into the
language grammar?

No! Just method calls...

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:*

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!

1. People want to pretend that String has a -> method.
2. Map really wants tuple arguments...

Implicit Conversions

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

Back to Maps

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

An `ArrowAssoc` is created for each left-hand string, then `->` called.

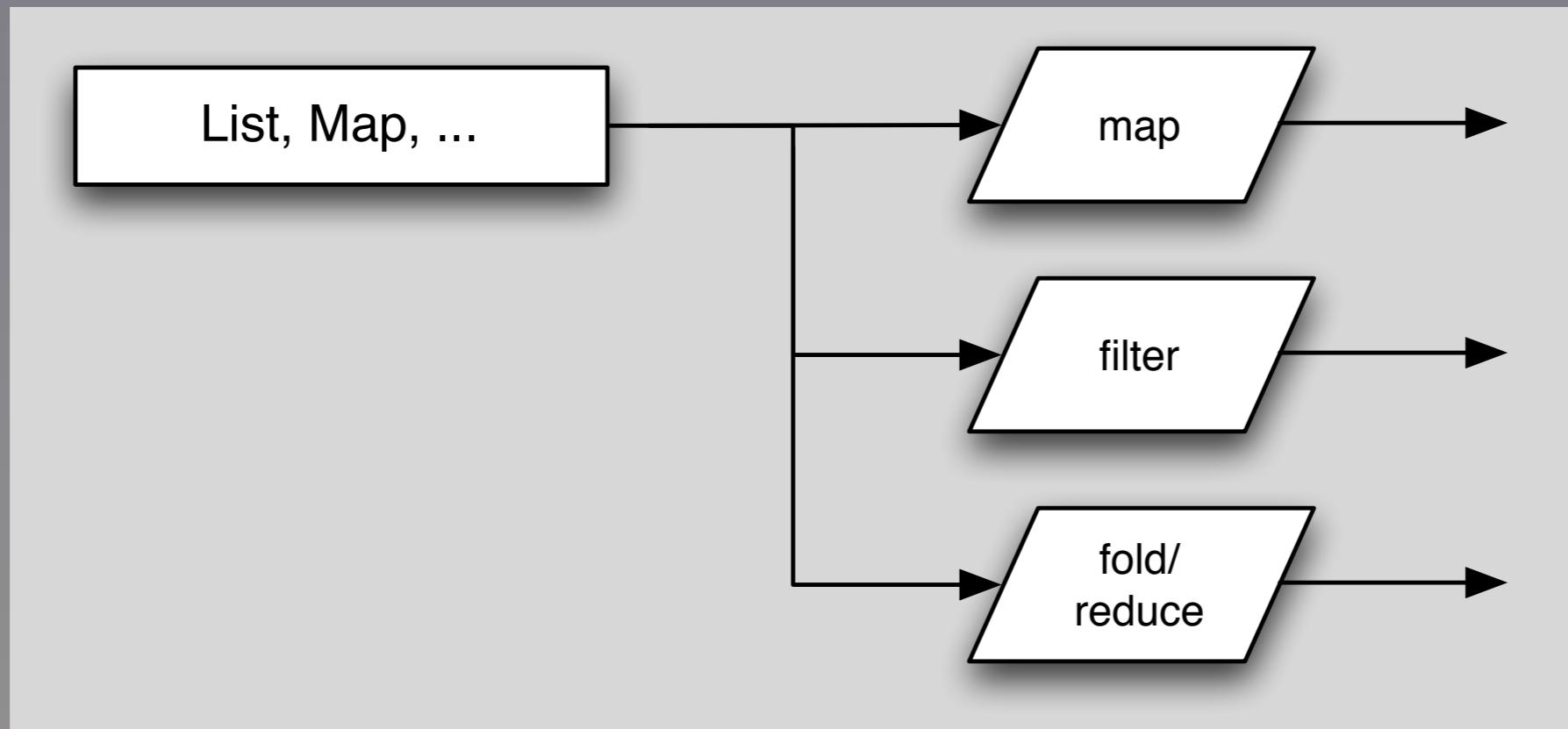
```
val map = Map(  
  Tuple2("name", "Dean"),  
  Tuple2("age", 39))
```

Similar *internal* DSLs
have been defined
for other types,
and in 3rd-party
libraries.

Back to Functions as Objects

38

Classic Operations on Container Types



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

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

argument to **map**: can
use “{...}” or “(...)”

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

{}

*function
argument list*

function body

“function literal”

Typed Arguments

```
list map {  
  s => s.toUpperCase  
}
```

inferred type

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

Explicit type

But wait! There's more!

```
list map {  
  s => s.toUpperCase  
}
```

Placeholder

```
list map (_ .toUpperCase)
```

Watch this...

```
list map (s => println(s))
```

```
list map (println)
```

// or

```
list map println
```

“Point-free” style

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

How the Sausage Is Made

like an abstract class



*“contravariant”,
“covariant” typing*



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

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

```
...  
}
```

*No method body,
therefore it is 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 are Objects

```
val list = "a" :: "b" :: Nil
```

```
list map {  
    s => s.toUpperCase  
}
```

Function “object”

```
// => "A" :: "B" :: Nil
```

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-right. The background features a range of majestic mountains with their peaks partially covered in snow. The sky is a clear, pale blue with a few wispy clouds.

Big Data DSLs

50

Saturday, November 16, 13

FP is going mainstream because it is the best way to write robust data-centric software, such as for “Big Data” systems like Hadoop. 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/)

51

Saturday, November 16, 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.

```
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")))
}
```

Scalding

```
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")))
}
```

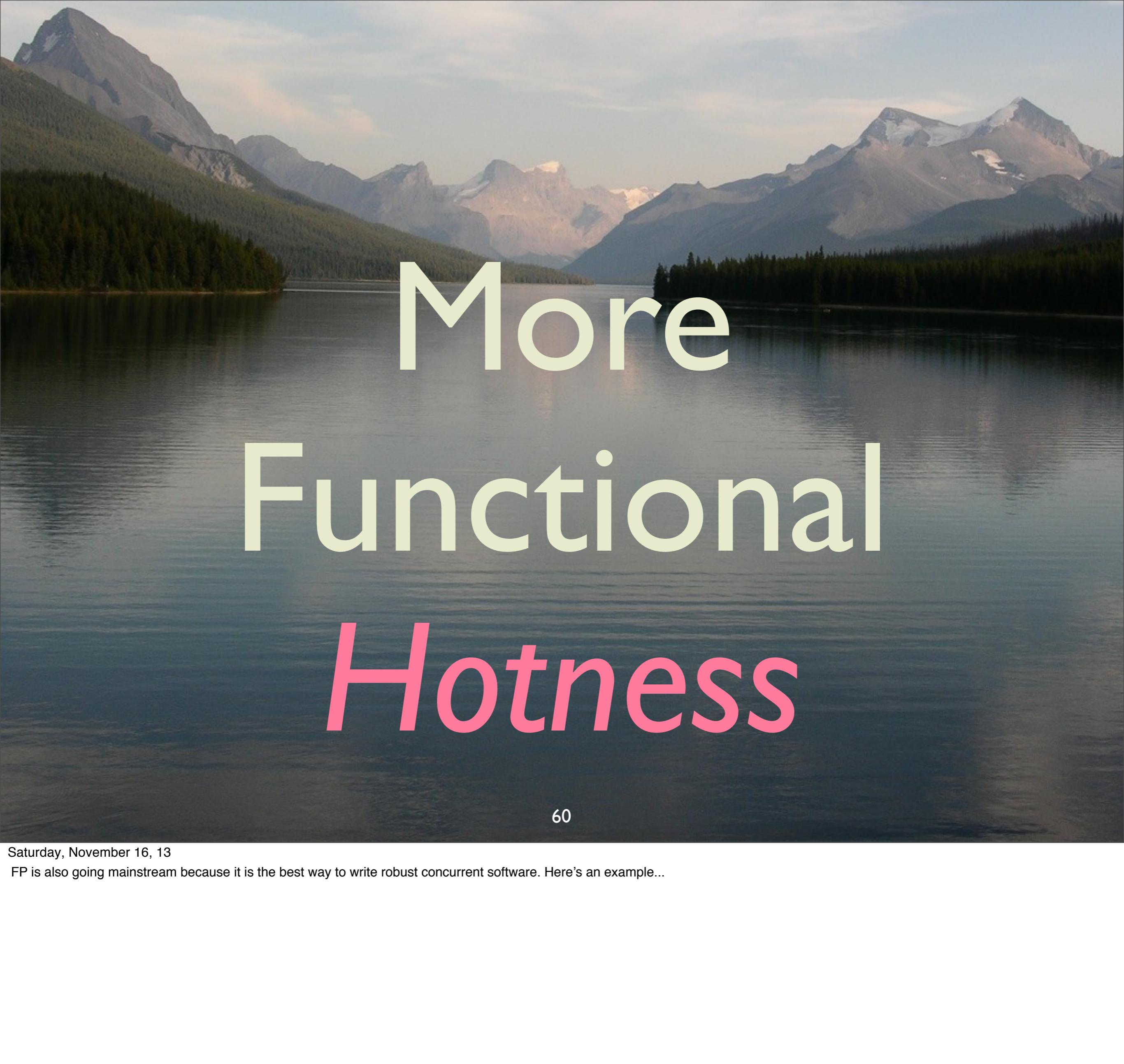
*Group by word and
count each group size.*

```
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.

For more on
Scalding see my talk:

Scalding for Hadoop

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 soft orange, creating a peaceful and inspiring atmosphere.

More Functional Hotness

60

Saturday, November 16, 13

FP is also 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] {...}
```

```
// 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 =  
  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*

64

Saturday, November 16, 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)
```

- **case** keyword creates a *companion object* with a **factory apply method**, and pattern matching support.

Case Classes

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

- **case** keyword **toString**, **equals**, and **hashCode** methods to the class.

Case Classes

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

- **case** keyword makes the value argument a field without the **val** keyword we had before.

```
class Some<T>
    private T value;

    public Some(T value){
        this.value = value;
    }

    public void T get() { return this.value; }

    public boolean equals(Object other) {
        ...
    }

    public int hashCode() {
        ...
    }

    public String toString() {
        ...
    }
}
```

Boilerplate

68

Saturday, November 16, 13

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

Or This:

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

Object

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

A singleton. Only one instance will exist.

70

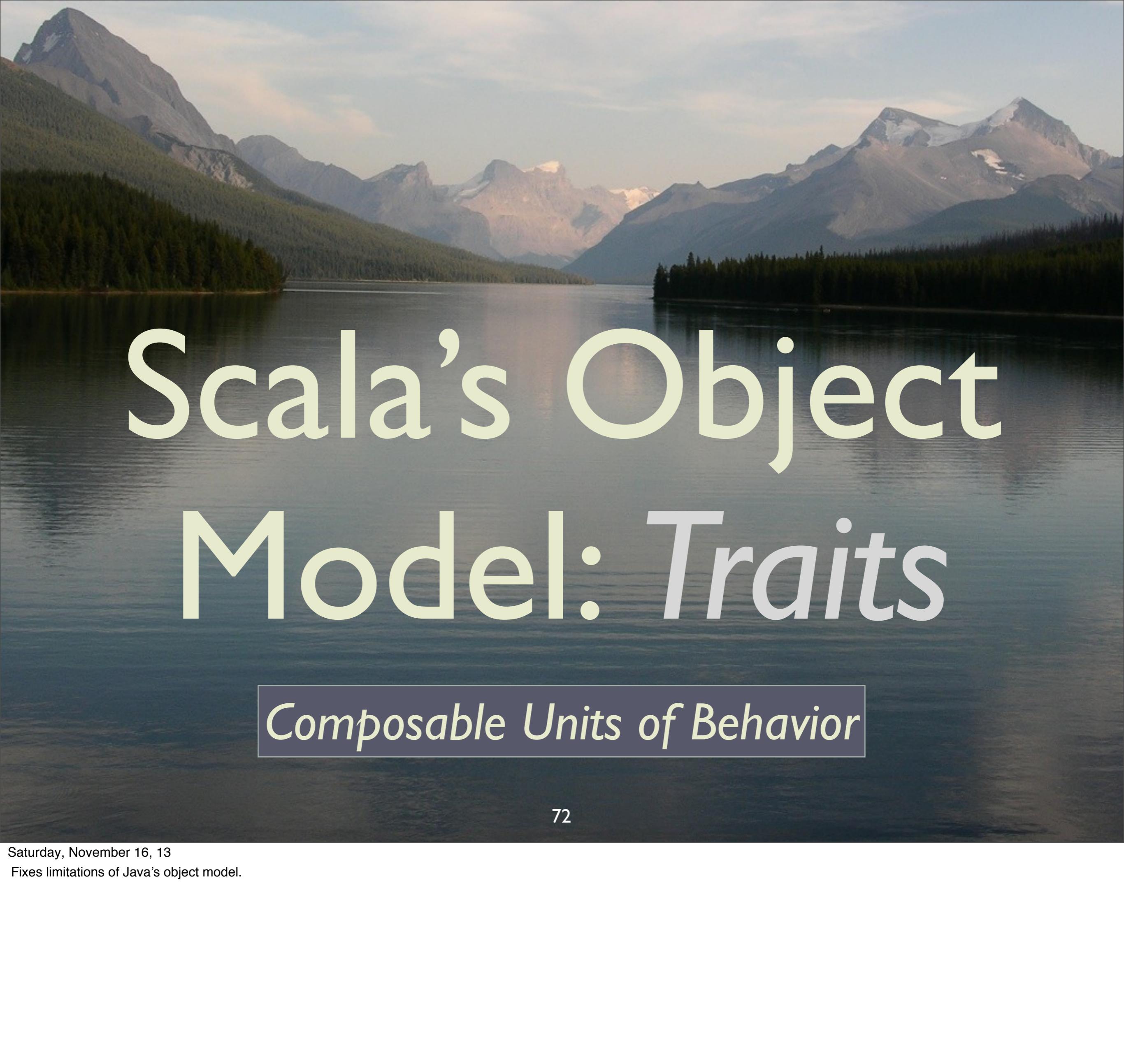
Saturday, November 16, 13

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.



Scala's Object Model: Traits

Composable Units of Behavior

72

Saturday, November 16, 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 or...

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

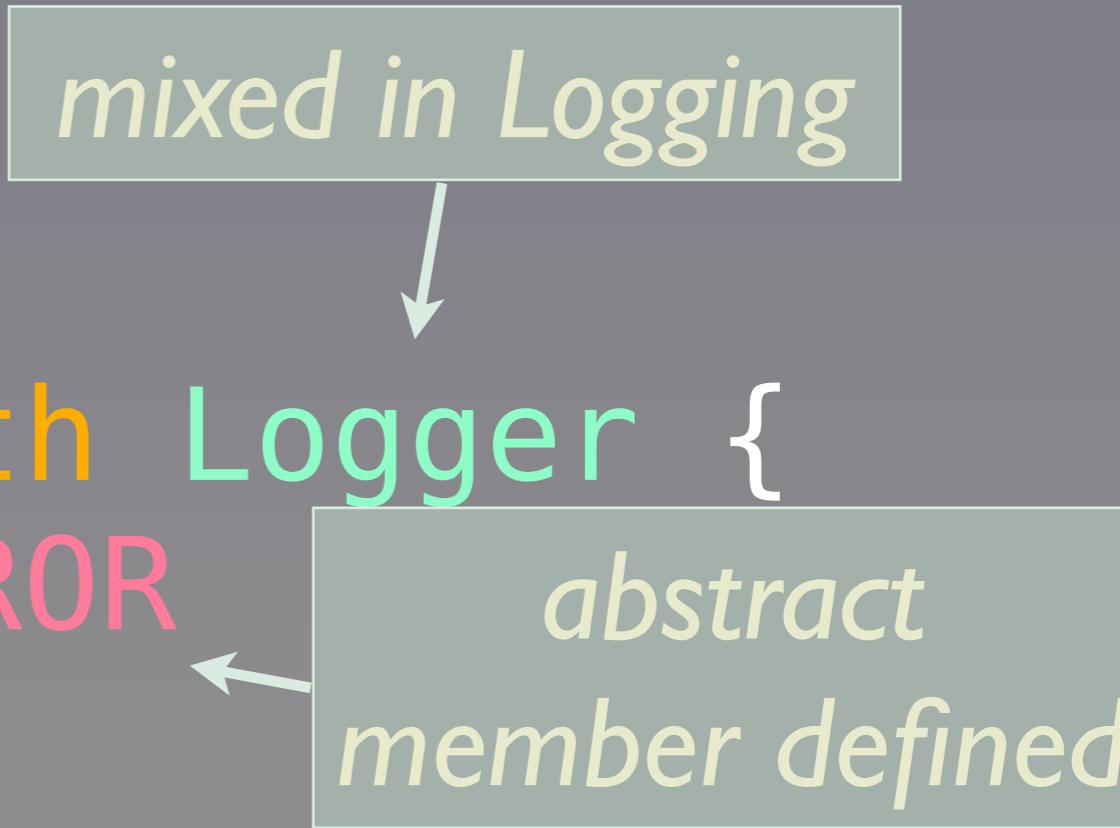
Logger as a Mixin:

```
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! !")
```



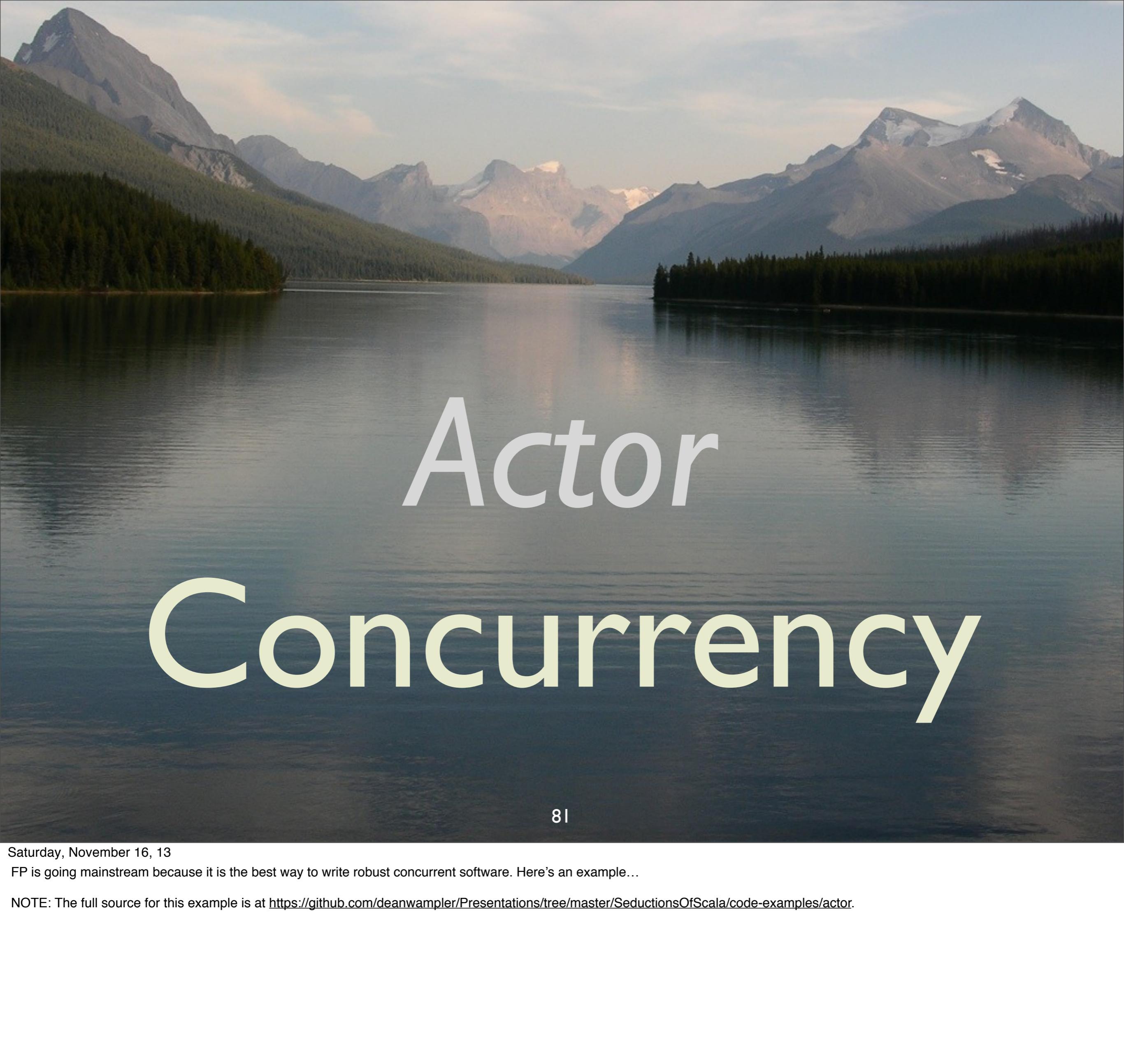
Like Java 8 Interfaces?

- ✓ *Default methods*
 - Can define method bodies.
- ✗ *Fields*
 - J8 fields remain *static final*,
not *instance* fields.

80

Saturday, November 16, 13

Java 8 interfaces aren't quite the same as traits. Fields remain static final, for backwards compatibility, but now you can define method bodies, which will be the defaults used if a class doesn't override the definition.

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 range of majestic mountains with their peaks partially covered in snow. The sky is overcast with soft, grey clouds.

Actor Concurrency

81

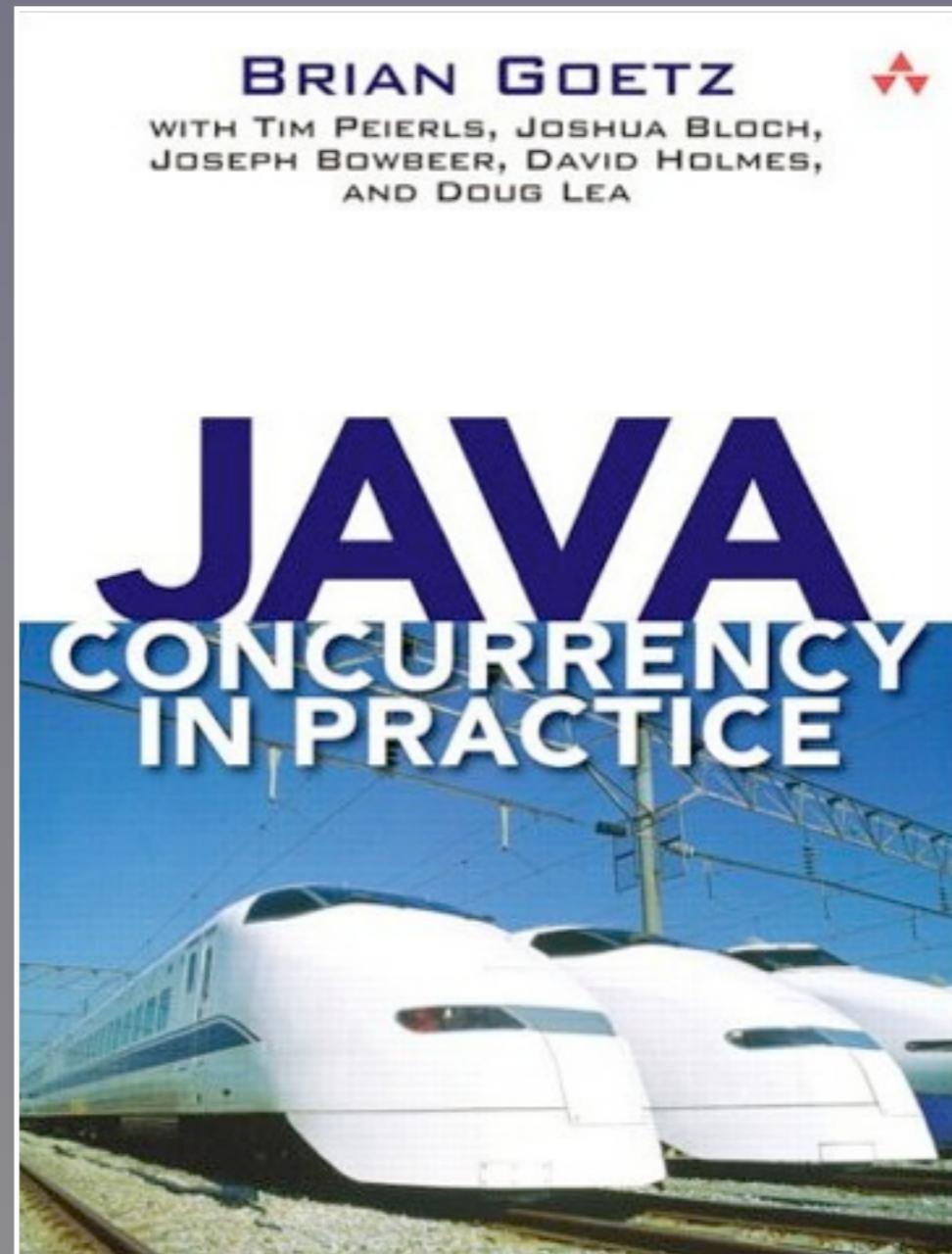
Saturday, November 16, 13

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*.

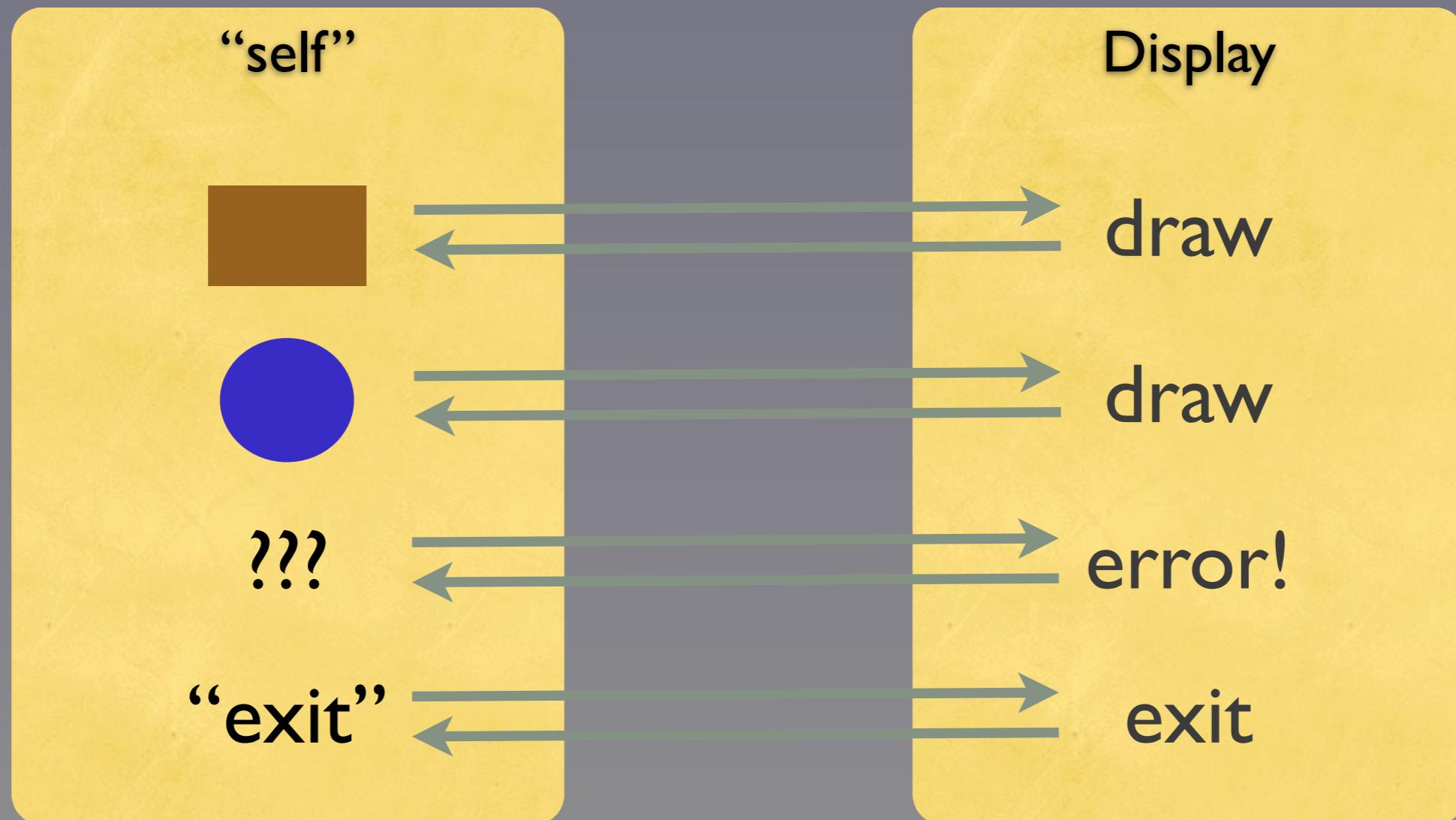
Akka

- Scala's Actor library.
 - Supports supervision for resilience.
 - Supports distribution and clustering.
 - akka.io

Akka

- Also has a complete Java API.
 - akka.io

2 Actors:



```
package shapes
```

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

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

Hierarchy of geometric shapes

Saturday, November 16, 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() = ...  
  }
```

Use the Akka Actor library

```
package shapes
```

```
import akka.actor.Actor
```

Actor

```
class Drawer extends Actor {
```

```
  def receive = {
```

```
    ...
```

*receive and handle
each message*

```
}
```

Actor for drawing shapes

receive
method

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

```

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

```

*pattern
matching*

```

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

```

*draw shape
& send reply*

done

unrecognized message

sender ! sends a reply

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

Altogether

```
import shapes._  
import akka.actor._  
import com.typesafe.config._  
  
object Driver {  
    def main(args:Array[String])={  
        val sys = ActorSystem(...)  
        val driver=sys.actorOf[Driver]  
        val drawer=sys.actorOf[Drawer]  
        driver ! Start(drawer)  
    }  
}  
}  
}
```

Application driver

...

95

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Saturday, November 16, 13

Here's the driver actor. It is declared as an "object" not a class, making it a singleton.

When we start, we send the "go!" message to the Driver actor that is defined on the next slide. This starts the asynchronous message passing.

The "!" is the message send method (stolen from Erlang).

```

import shapes._
import akka.actor._
import com.typesafe.config._

object Driver { Singleton for main
  def main(args:Array[String])={

    val sys = ActorSystem(...)

    val driver=sys.actorOf[Driver]
    val drawer=sys.actorOf[Drawer]

    driver ! Start(drawer) Instantiate actors

  }
}

}

```

Send a message to start the actors

...

Saturday, November 16, 13

Here's the driver actor. It is declared as an "object" not a class, making it a singleton.

When we start, we send the "go!" message to the Driver actor that is defined on the next slide. This starts the asynchronous message passing.

The "!" is the message send method (stolen from Erlang).

Companion class

```
...  
class Driver extends Actor {  
    var drawer: Option[Drawer] =  
        None  
  
    def receive = {  
        ...  
    }  
}
```

```

def receive = {
  case Start(d) => sent by  
driver
    drawer = Some(d)
    d ! Circle(Point(...), ...)
    d ! Rectangle(...)
    d ! 3.14159
    d ! "exit"
  case "good bye!" =>
    sent by  
drawer
    println("<- cleaning up...")
    context.system.shutdown()
  case other =>
    println("<- " + other)
}

```

```
d ! Circle(Point(...),...)  
d ! Rectangle(...)  
d ! 3.14159  
d ! "exit"
```

```
-> 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.*

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

*Functional-style
pattern matching*

*Object-
oriented-style
polymorphism*

*“Switch” statements are
not (necessarily) evil*



Recap

101

Saturday, November 16, 13

Scala is...

102

a better Java,

103

object-oriented
and
functional,

104

succinct,
elegant,
and
powerful.

105

Questions?

Dean Wampler
dean@deanwampler.com
@deanwampler
polyglotprogramming.com/talks

November 19, 2013



106

Saturday, November 16, 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 middle ground, a small, densely forested island is visible in the center of the lake. The background features a majestic range of mountains, their peaks partially obscured by a light, hazy sky. The overall atmosphere is peaceful and natural.

Extra Slides

107

Saturday, November 16, 13

Modifying Existing Behavior with Traits

108

Example

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

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

109

Log put

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

110

Saturday, November 16, 13

(We're ignoring "get"...)
"Super" is not yet bound, because the "super.put(t)" so far could only call the abstract method in Logging, which is not allowed. Therefore, "super" will be bound "later", as we'll see. So, this method is STILL abstract and it's going to override a concrete "put" "real soon now".

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??

!!!

Saturday, November 16, 13

(We're ignoring "get"...)"Super" is not yet bound, because the "super.put(t)" so far could only call the abstract method in Logging, which is not allowed. Therefore, "super" will be bound "later", as we'll see. So, this method is STILL abstract and it's going to override a concrete "put" real soon now".

```
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

112

```
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

113

Saturday, November 16, 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

114

Saturday, November 16, 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.

*Traits are a powerful
composition
mechanism!*

115

Saturday, November 16, 13

Not shown, nesting of traits...

For Comprehensions

116

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.*

Equivalent to this:

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

```
for (o <- l; x <- o) yield x  
// List(a, b, c)
```

*Second clause extracts
from option; Nones
dropped*

118

Saturday, November 16, 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.

Building Our Own Controls

DSLs Using First-Class Functions

Recall Infix Operator Notation:

```
"hello" + "world"  
"hello".+( "world" )
```

also the same as

```
"hello".+{ "world" }
```

Why is using {...} useful??

120

Make your own controls

// Print with line numbers.

```
loop (new File("...")) {  
  (n, line) =>  
  
    format("%3d: %s\n", n, line)  
}
```

121

Saturday, November 16, 13

If I put the "(n, line) =>" on the same line as the "{", it would look like a Ruby block.

Make your own controls

// Print with line numbers.

```
control?           File to loop through  
loop (new File("...")) {}  
(n, line) => ← Arguments passed to...  
  
format("%3d: %s\n", n, line)  
{}               what do for each line  
                           How do we do this?
```

Output on itself:

```
1: // Print with line ...
2:
3:
4: loop(new File("...")) {
5:   (n, line) =>
6:
7:   format("%3d: %s\n", ...
8: }
```

```
import java.io._

object Loop {

    def loop(file: File,
            f: (Int, String) => Unit) = {
        ...
    }
}
```

124

```
import java.io.Object
```

like * in Java

“singleton” class == 1 object

```
object Loop {
```

loop “control”

```
def loop(file: File,
```

two parameters

```
    f: (Int, String) => Unit) =
```

function taking line # and line

```
{...}
```

like “void”

```
loop (new File("...")) {  
    (n, line) => ...  
}
```

```
object Loop {
```

two parameters

```
def loop(file: File,  
        f: (Int, String) => Unit) =  
{ ... }  
}
```

```
loop (new File("...")) {  
    (n, line) => ...  
}
```

```
object Loop {
```

two parameters lists

```
def loop(file: File) ()  
    f: (Int, String) => Unit) =  
{ ... }  
}
```

Why 2 Param. Lists?

```
// Print with line numbers.  
import Loop.loop  
  
loop (new File("...")) {}  
(n, line) =>  
    format("%3d: %s\n", n, line)  
}
```

import

1st param.: a file

2nd parameter: a function literal

```
object Loop {  
    def loop(file: File) (f: (Int, String) => Unit) =  
    {  
        val reader =  
            new BufferedReader(  
                new FileReader(file))  
        def doLoop(i:Int) = {...}  
        doLoop(1)  
    }  
}
```

nested method

Finishing Numberator...

129

```
object Loop {  
  ...  
  def doLoop(n: Int):Unit ={  
    val l = reader.readLine()  
    if (l != null) {  
      f(n, l)  
      doLoop(n+1)  
    }  
  }  
}  
}
```

*f and reader visible
from outer scope*

recursive

Finishing Numberator...

130

doLoop is recursive.
There is no *mutable*
loop counter!

A goal of Functional Programming

131

It is Tail Recursive

```
def doLoop(n: Int):Unit ={  
    ...  
    doLoop(n+1)  
}
```

Scala optimizes tail recursion into loops

132

Functions with Mutable State

133

Since *functions*
are *objects*,
they could have
mutable state.

```

class Counter[A](val inc:Int =1)
  extends Function1[A,A] {
  var count = 0
  def apply(a:A) = {
    count += inc
    a // return input
  }
}
val f = new Counter[String](2)
val l1 = "a" :: "b" :: Nil
val l2 = l1 map {s => f(s)}
println(f.count) // 4
println(l2)      // List("a", "b")

```