

The Seductions of Scala

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

April 3, 2011

1



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

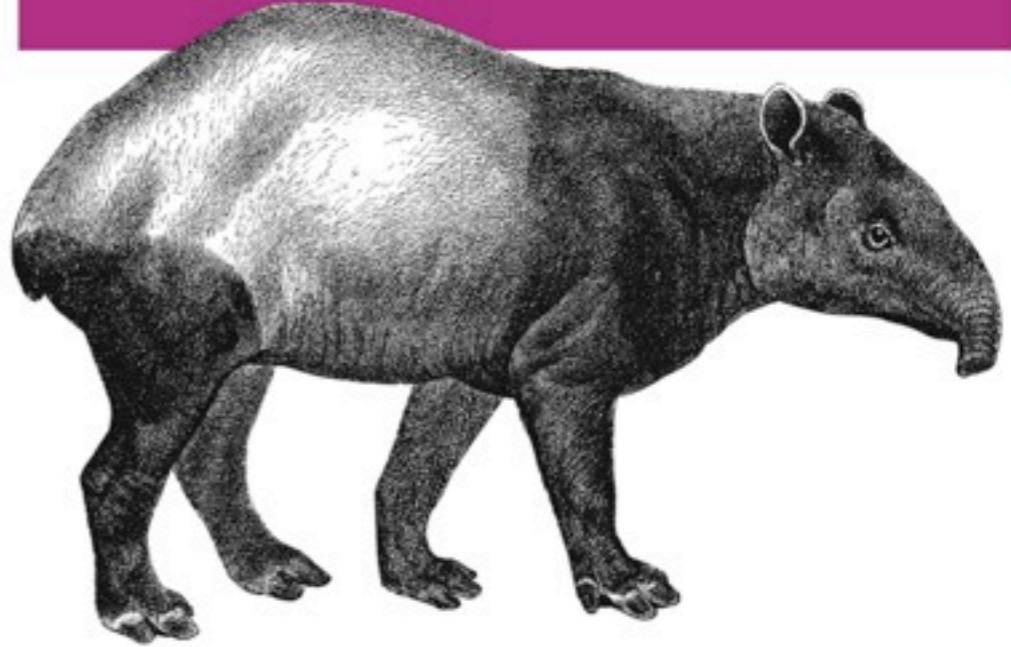
<shameless-plug/>

Co-author,
*Programming
Scala*

programmingscala.com

Programming

Scala



O'REILLY®

Dean Wampler & Alex Payne

Why do we need a new language?

#1

We need *Functional* Programming

• • •

- ... 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/C#.

Scala is...

- A *JVM* and *.NET* language.
- *Functional* and *object oriented*.
- *Statically typed*.
- An *improved Java/C#*.

Martin Odersky

- Helped design java *generics*.
- Co-wrote *GJ* that became *javac* (v1.3+).
- Understands Computer Science *and* Industry.

II

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 background, a range of majestic mountains rises, their peaks partially obscured by a hazy sky. The mountains are covered in dense forests of evergreen trees. The lighting suggests either sunrise or sunset, with warm, golden hues reflecting off the water and the lower slopes of the mountains.

Everything can
be a *Function*

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 arg list
after any object,
apply is called.

A wide-angle photograph of a mountainous landscape. In the foreground, a calm lake reflects the surrounding environment. The middle ground features a dense forest of coniferous trees lining the shore. In the background, a range of majestic mountains rises, their peaks partially obscured by a hazy sky. The lighting suggests either sunrise or sunset, casting a warm glow over the scene.

Everything is an Object

19

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.

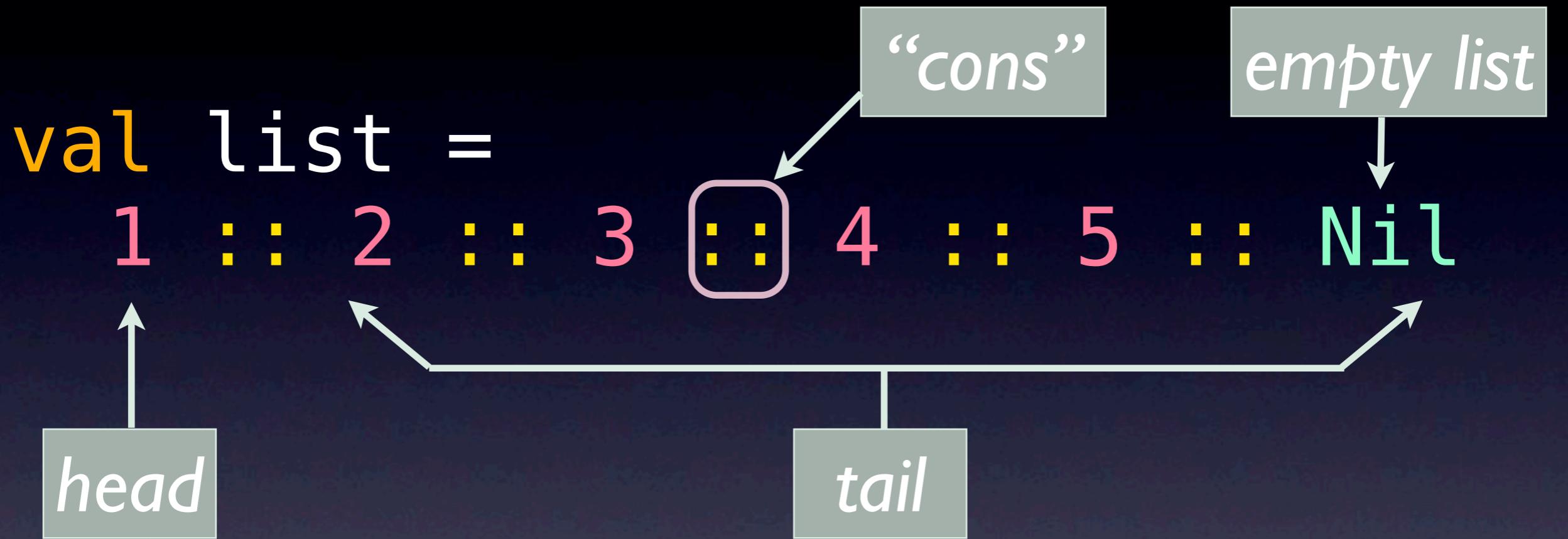
Functions as Objects

First, About Lists

```
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")

Similar mini-DSLs
have been defined
for other types.

Also in many third-party libraries.

Oh, and Maps

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

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...

X

Scala provides mechanisms to define convenient “operators” as methods, without special exceptions baked into the grammar (e.g., strings and “+” in Java).

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()
actually wants:*

x

Oh, and Maps

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

*What we like
to write:*

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

*What Map()
actually wants:*

*More succinct
syntax for Tuples*

x

We need to get from this,

"name" -> "Dean"

to this,

Tuple2("name", "Dean")

There is no String-> method!

x

We won't discuss implicit conversions here, due to time....

Implicit Conversions

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

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

X

String doesn't have `->`, but `ArrowAssoc` does! Also, it's `->` returns a `Tuple2`. So we need to somehow convert our strings used as keys, i.e., on the left-hand side of the `->`, to `ArrowAssoc` object, then call `->` with the value on the right-hand side of the `->` in the Map literals, and then we'll get the `Tuple2` objects we need for the Map factory method.

The trick is to declare a method (or in some contexts, a value) with the keyword "implicit". The compiler will look for those in scope and then call them to convert the object without the desired method (a string and `->` in our case) to an object with the method (`ArrowAssoc`).

Back to Maps

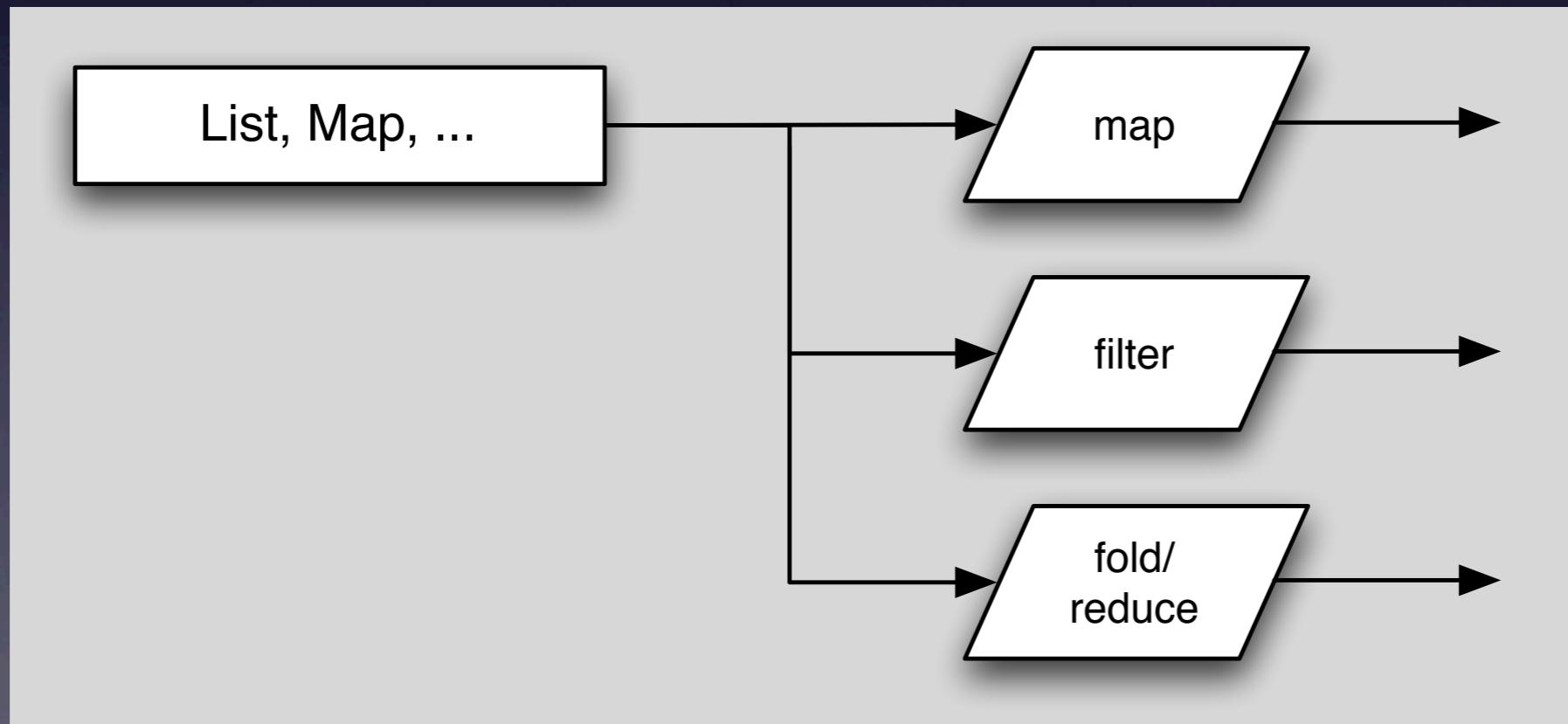
```
val map = Map(  
    "name" -> "Dean",  
    "age"  -> 39)
```

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

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

x

Classic Operations on “Container” Types



Back to
functions as objects...

```
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  
}  
    inferred type
```

A diagram illustrating type inference. A white arrow points from the variable 's' in the code above to a light gray rectangular box containing the text 'inferred type'.

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

A diagram illustrating explicit typing. A white arrow points from the type annotation 'String' in the code above to a light gray rectangular box containing the text '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]  
...  
}
```

The function argument

Parameterized type

Declaration of `map`

map's return type

How the Sausage Is Made

like an “abstract” class

```
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

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

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

*Can use {...}
instead of (...)*

Function “object”

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

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)      x // List("a","b")
```

Our functions can have state! Not the usual thing for FP-style functions, where functions are usually side-effect free, but you have this option. Note that this is like a normal closure in FP.

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

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

41

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 = 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*

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.

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-right. 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 on their summits. The sky is a soft, warm color, suggesting either sunrise or sunset.

Succinct Code

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

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]
```

```
// 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

```
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!*

Actually, not *exactly* the same:

```
val person = new Person("dean",...)  
val fn = person.firstName  
// Not:  
// val fn = person.getFirstName
```

*Doesn't follow the
JavaBean convention.*

x

Note that Scala does not define an argument list for "firstName", so you can call this method as if it were a bare field access. The client doesn't need to know the difference!

However, these are function calls:

```
class Person(fn: String, ...){  
    // init val  
    private var _firstName = fn  
  
    def firstName = _firstName  
  
    def firstName_= (fn: String) =  
        _firstName = fn  
}
```

Uniform Access Principle

x

Note that Scala does not define an argument list for "firstName", so you can call this method as if it were a bare field access. The client doesn't need to know, nor care, if it is making a bare field access or going through a method.

Even Better...

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

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

A scenic landscape featuring a calm lake in the foreground, framed by a dense forest. In the background, a range of majestic mountains is visible under a clear sky.

Scala's Object Model: *Traits*

Composable Units of Behavior

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

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??

x

Make your own *controls*

```
// Print with line numbers.
```

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

X

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) =  
    {...}  
}
```

x

Here's the code that implements loop...

```
import java.io._
```

like * in Java

```
object Loop {
```

loop “control”

two parameters

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

function taking line # and line

like “void”

X

Singleton “objects” replace Java statics (or Ruby class methods and attributes). As written, “loop” takes two parameters, the file to “numberate” and a the function that takes the line number and the corresponding line, does something, and returns Unit. User’s specify what to do through “f”.

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

```
object Loop {
```

two parameters

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

X

The oval highlights the comma separating the two parameters in the list. Watch what we do on the next slide...

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

```
object Loop {
```

two parameters lists

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

X

We convert the single, two parameter list to two, single parameter lists, which is valid syntax.

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”

X

Having two, single-item parameter lists, rather than one, two-item list, is necessary to allow the syntax shown here. The first parameter list is (file), while the second is {function literal}. Note that we have to import the loop method (like a static import in Java). Otherwise, we could write Loop.loop.

```
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...

x

Finishing the implementation, loop creates a buffered reader, then calls a recursive, nested method "doLoop".

```
object Loop {
```

```
...
```

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

recursive

*“f” and “reader” visible
from outer scope*

Finishing Numberator...

x

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

Classic Functional Programming technique

It is *Tail* Recursive

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

x

Scala optimizes tail recursion into loops

A tail recursion - the recursive call is the last thing done in the function (or branch).

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.

Actor Concurrency

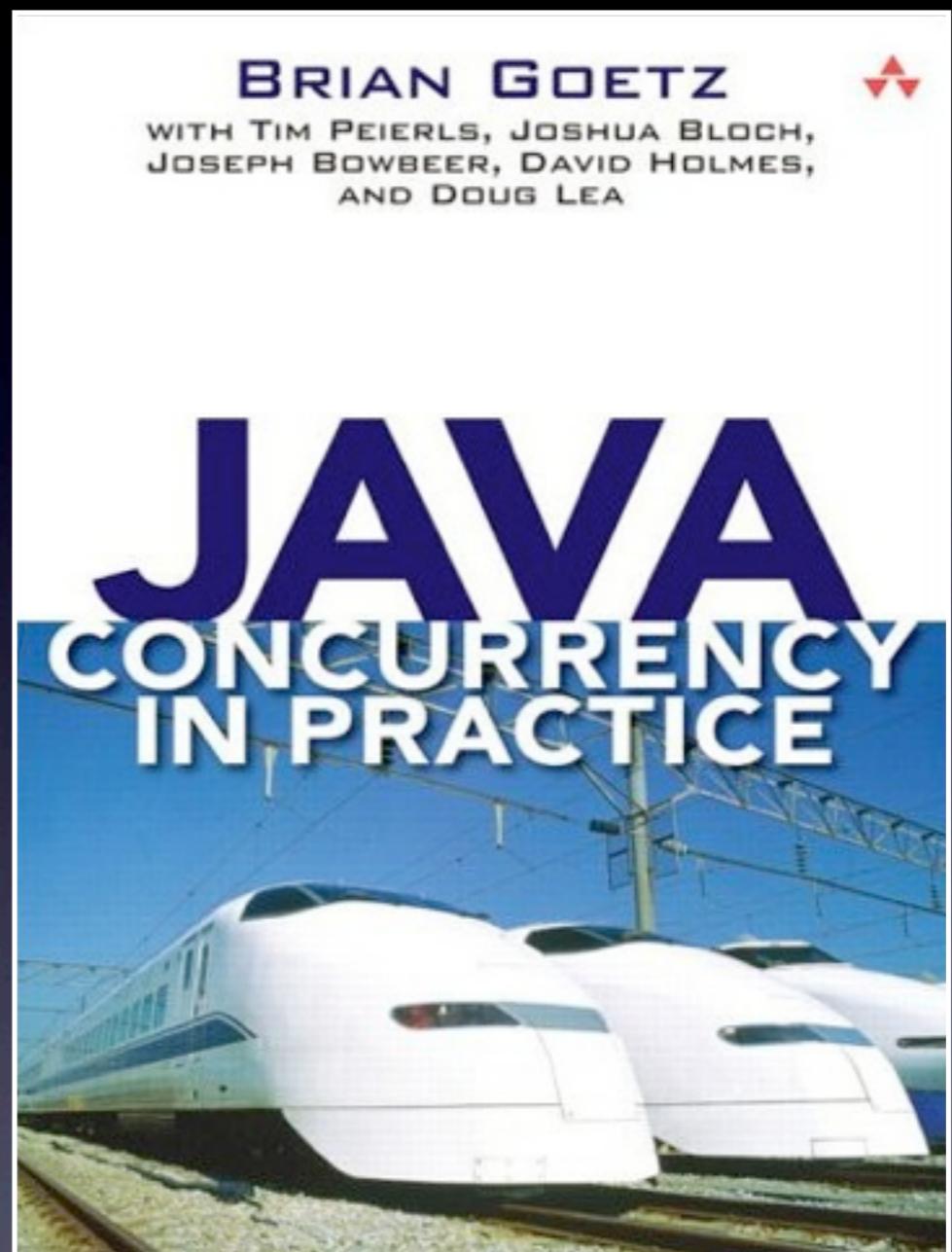
66

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)



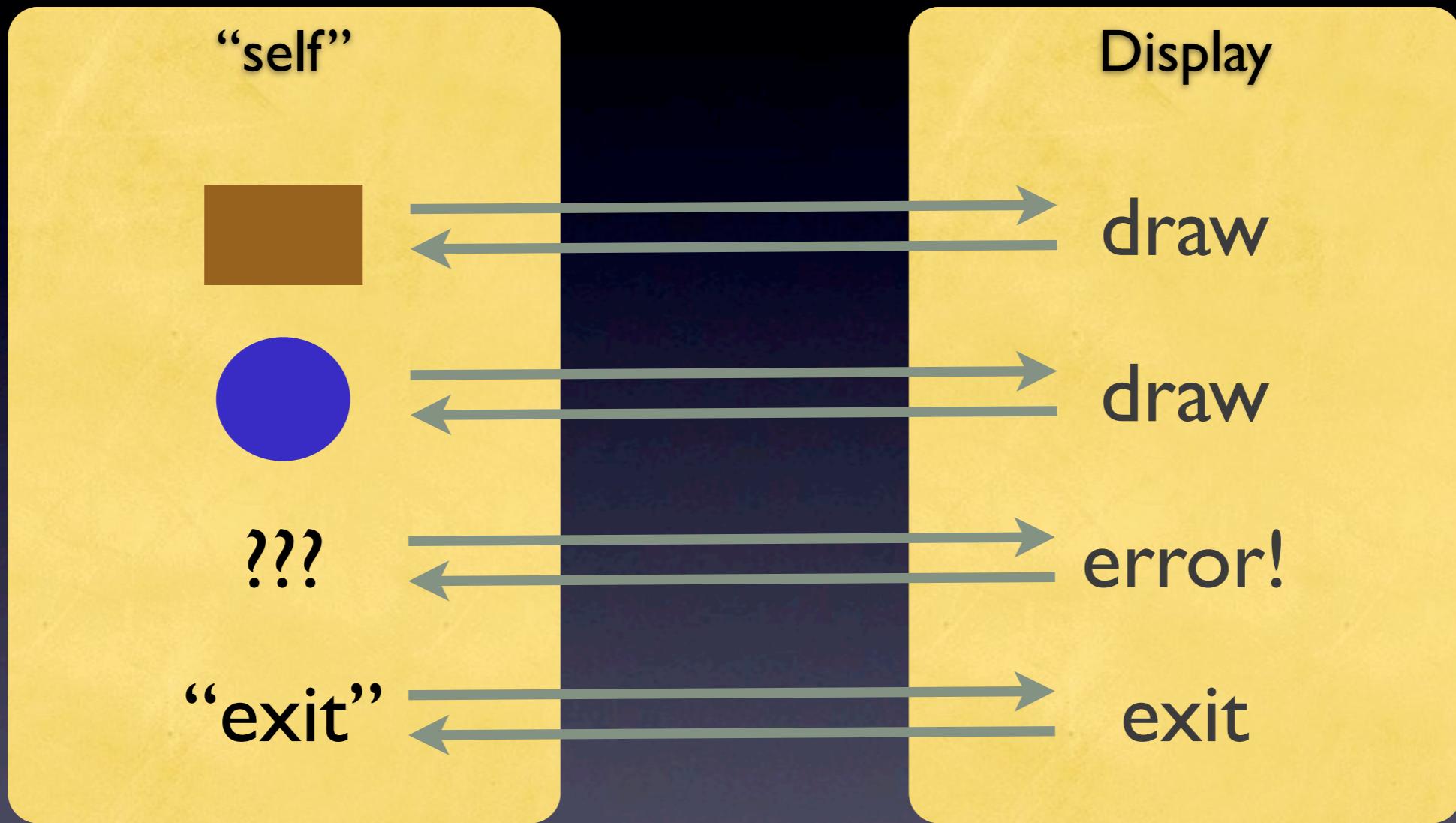
Actor Model

- 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

71

“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

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

```

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!"  
  }  
}  
...  
}
```

! 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!" =>
        drawer ! Circle(Point(...),...)
        drawer ! Rectangle(...)
        drawer ! 3.14159
        drawer ! "exit"
    case "good bye!" =>
        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!!

Functional-style pattern matching

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

*Object-
oriented-style
polymorphism*

A powerful combination!



Recap

Scala is...

a better
Java and C#,

*object-oriented
and
functional,*

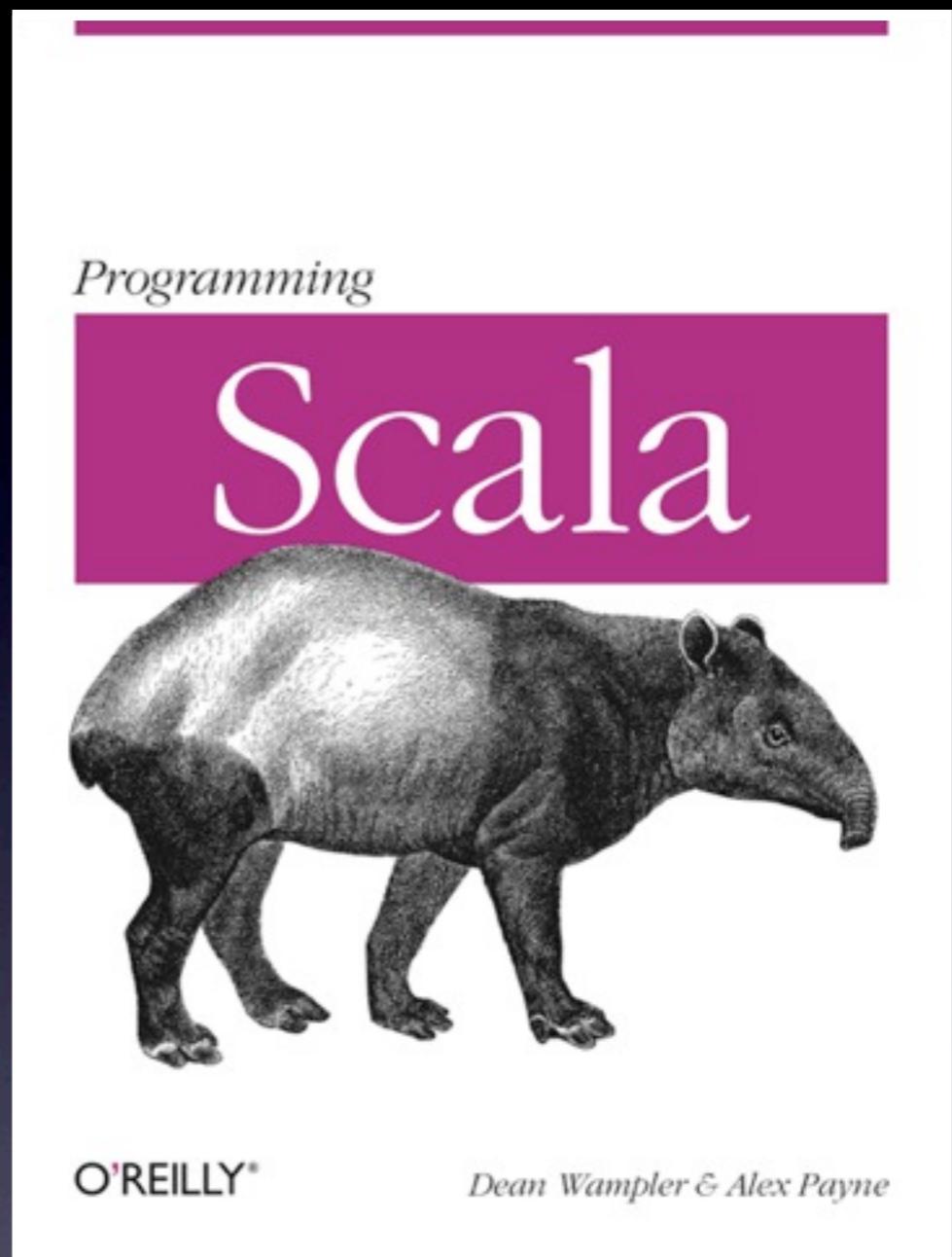
*succinct,
elegant,
and
powerful.*

Thanks!

dean@deanwampler.com
@deanwampler

polyglotprogramming.com/talks
programmingscala.com

thinkbiganalytics.com



A wide-angle photograph of a mountainous landscape. In the foreground is a calm lake with a small, dark island in the center-right. The middle ground shows a range of mountains with dense forests on their lower slopes. The background features a majestic range of mountains under a clear sky.

Extra Slides

Modifying Existing Behavior with Traits

Example

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

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

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

```
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

*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

Like Aspect-Oriented Programming?

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

Traits are a powerful
composition
mechanism!

A wide-angle photograph of a serene lake nestled in a mountain range. The mountains in the background are partially covered in snow, with their peaks catching the light of a setting or rising sun. The water of the lake is calm, reflecting the surrounding green forests and the warm, golden hues of the sky. The overall atmosphere is peaceful and natural.

Functional Programming

What is *Functional* Programming?

Don't we already write “functions”?

$y = \sin(x)$

Based on *Mathematics*

$$y \equiv \sin(x)$$

Setting x fixes y

\therefore variables are *immutable*

`20 += | ??`

We never modify
the 20 “object”

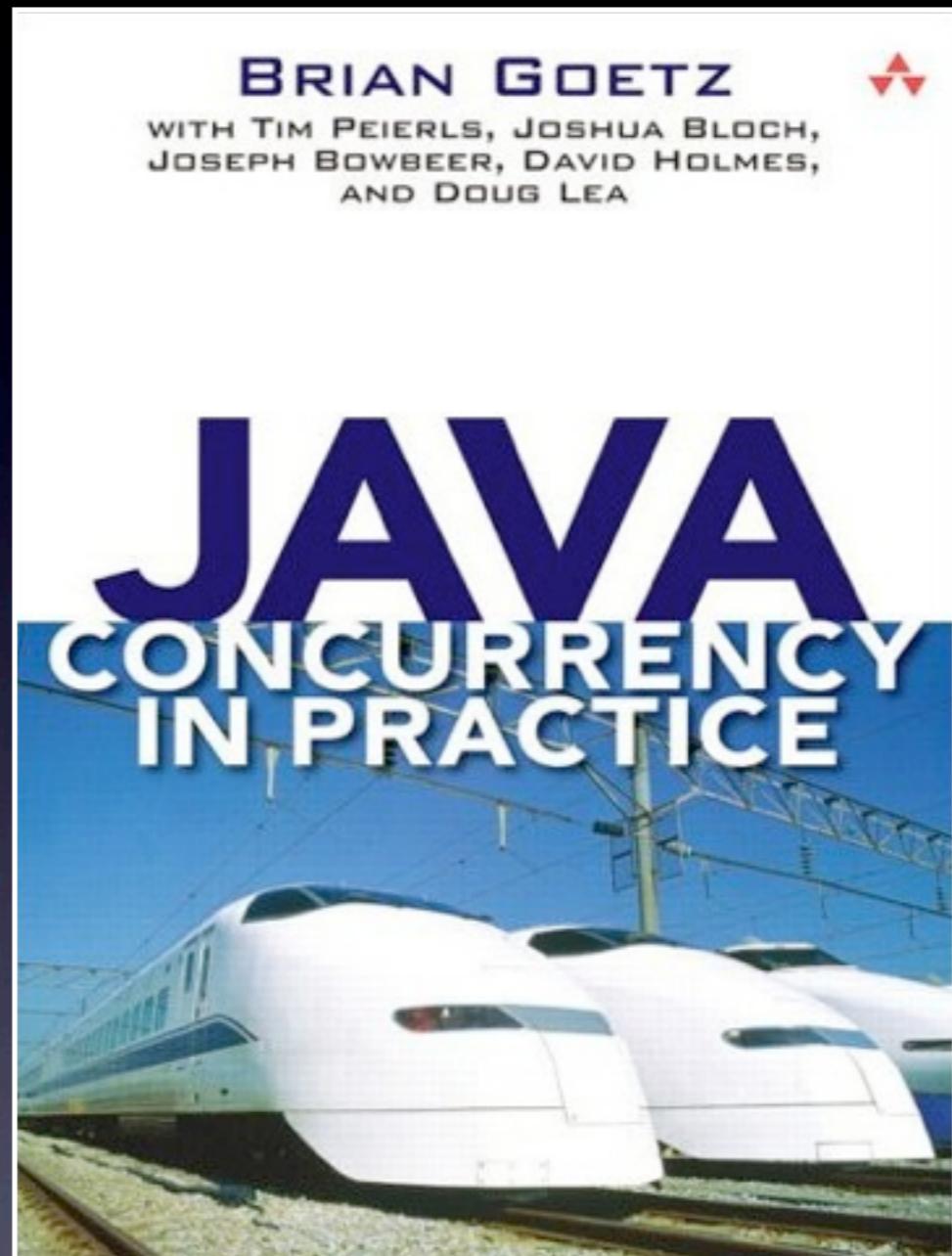
Concurrency

No mutable state

∴ *nothing to synchronize*

When you
share mutable
state...

Hic sunt dracones
(Here be dragons)



$$y = \sin(x)$$

Functions don't
change state
 \therefore *side-effect free*

Side-effect free functions

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

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

*Compose functions of
other functions*

∴ first-class citizens

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.

Even More Functional Hotness

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.