# The Seductions of Scala

**Dean Wampler**
dean.wampler@typesafe.com
@deanwampler
polyglotprogramming.com/talks

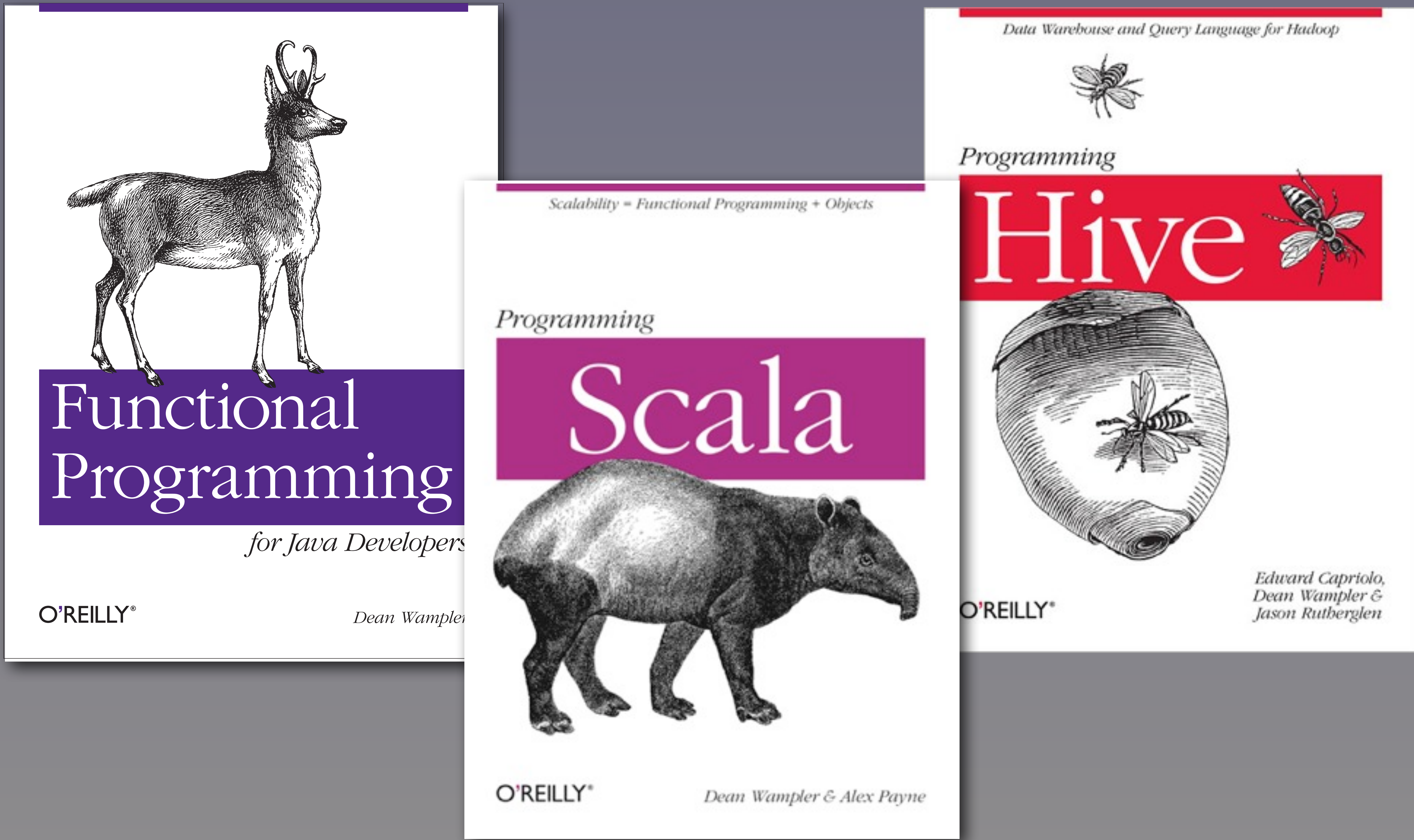
November 19, 2013





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.




<shameless-plug/>





Available now from oreilly.com, Amazon, etc.




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*

...





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.*





Java’s object model (and to a lesser extent, C#‘s) has significant limitations.

# Scala's Thesis: Functional Prog. *complements* Object-Oriented Prog.

*Despite surface contradictions...*





We think of objects as mutable and methods as state-modifying, while FP emphasizes immutability, which reduces bugs and often simplifies code. Objects don't have to be mutable!

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





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

# Scala is...

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





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.





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.




Saturday, November 16, 13

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

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

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





A simple wrapper around your favorite logging library (e.g., Log4J).

*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*



Saturday, November 16, 13

Note how variables are declared, "name: Type".

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

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

val error = new Logger(ERROR)

…
error("Network error.")
```



Saturday, November 16, 13

After creating an instance of Logger, in this case for Error logging, we can “pretend” the object is a function!

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

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

apply *is called*

```scala
…
error("Network error.")
```



Saturday, November 16, 13

Adding a parameterized arg. list after an object causes the compiler to invoke the object's "apply" method.

*“function object”*

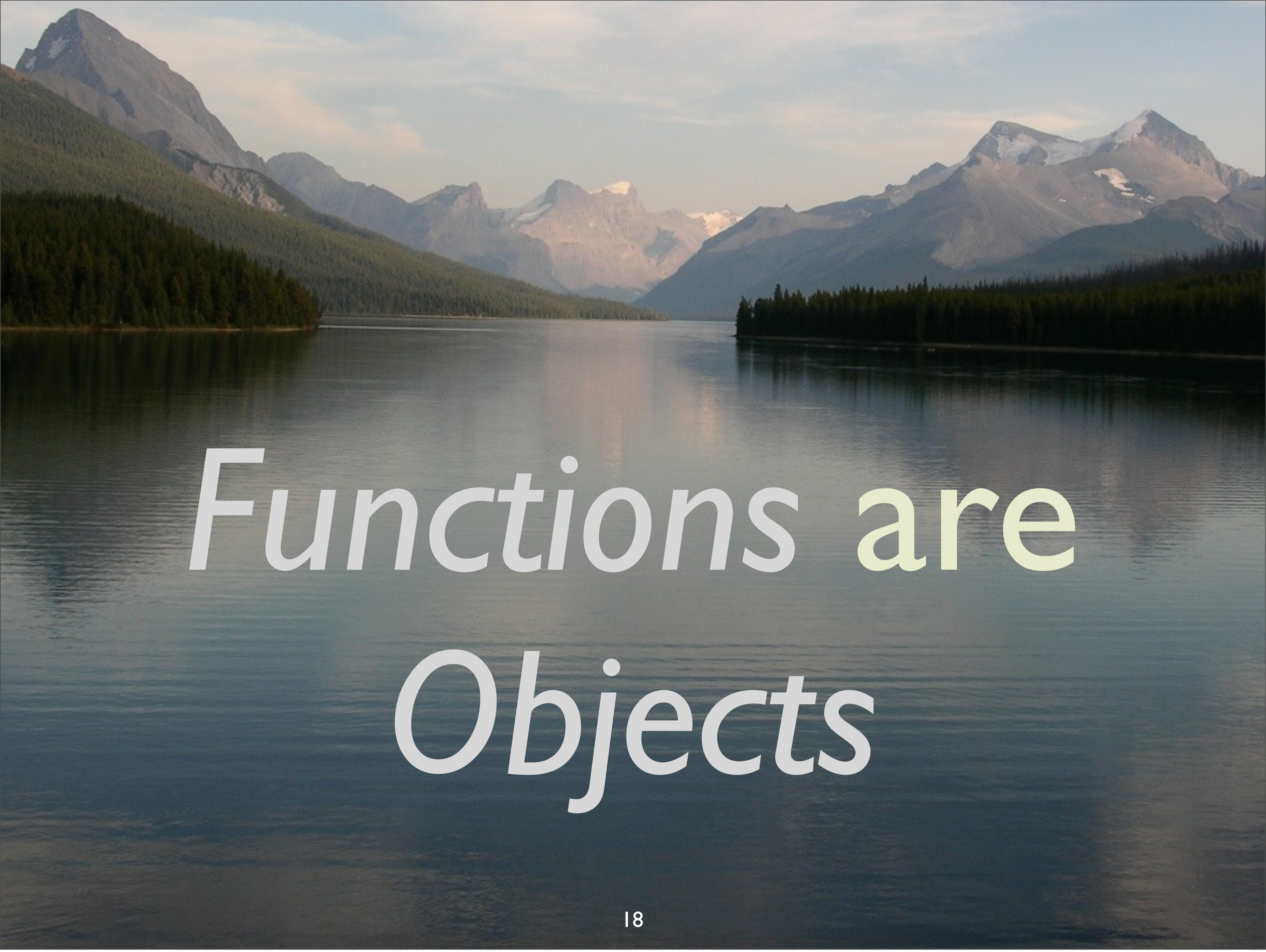
```
…
error("Network error.")
```

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



Saturday, November 16, 13

This is how any object can be a function, if it has an apply method. Note that the signature of the argument list must match the arguments specified. Remember, this is a statically-typed language!




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
```





Why is there no "new"? You can guess what's going on based on what we've already said. There must be some object named "List" with an apply method.
In fact, there is a "singleton" object named List that is a "companion" of the List class. This companion object has an apply method that functions as a factory for creating lists.

“cons”

empty list

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

head

tail





We build up a literal list with the “::” cons operator to prepend elements, starting with an empty list, the Nil “object”.

# Baked into the Grammar?

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

*No, just method calls!*

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





But this isn't something backed into the grammar; we're just making method calls on the List type!

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

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

*Method names can contain almost any character.*



Saturday, November 16, 13

There are some restrictions, like square brackets [ and ], which are reserved for other uses.

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

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

*Any method ending in ":" binds to the right!*



Saturday, November 16, 13

"::" binds to the right, so the second form shown is equivalent to the first.

```scala
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 ")".*



Saturday, November 16, 13

Infix operator notation.

# Infix Operator Notation

```
"hello" + "world"
```

*is actually just*

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



Saturday, November 16, 13

Note the “infix operator notation”; x.m(y) ==> x m y. It’s not just a special case backed into the language grammar (like Java’s special case for string addition). Rather, it’s a general feature of the language you can use for your classes.

*Note:*

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





If you know Java, you might wonder if these integer lists were actually List<Integer>, the boxed type. No. At the syntax level, Scala only has object (reference) types, but it compiles these special cases to primitives automatically.

# This means that *generics just work.*

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

*An empty list of* Ints.

```
Java: ... List<Int>
```





You don't have to explicitly box primitives; the compiler will optimize these objects to primitives (with some issues involving collections...)
Note the syntax for parameterizing the type of List, [...] instead of <...>.

# 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

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

*“baked” into the language grammar?*

*No! Just method calls...*



Saturday, November 16, 13

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

# Maps

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

*What we like to write:*

*What Map.apply() actually wants:*

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

# Maps

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

*What we like to write:*

*What Map.apply() actually wants:*

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

*More succinct syntax for Tuples*



Saturday, November 16, 13

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

*We need to get from this,*

```
"name" -> "Dean"
```

*to this,*

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

*There is no String.-> method!*



Saturday, November 16, 13

We've got two problems:

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)
}
```





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 the class as "implicit". The compiler will look for any implicits in scope and then call them to convert the object without a desired method (a string and -> in our case) to an object with that method (ArrowAssoc). Then the call to -> can proceed, which returns the tuple we need!

# 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))
```





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

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

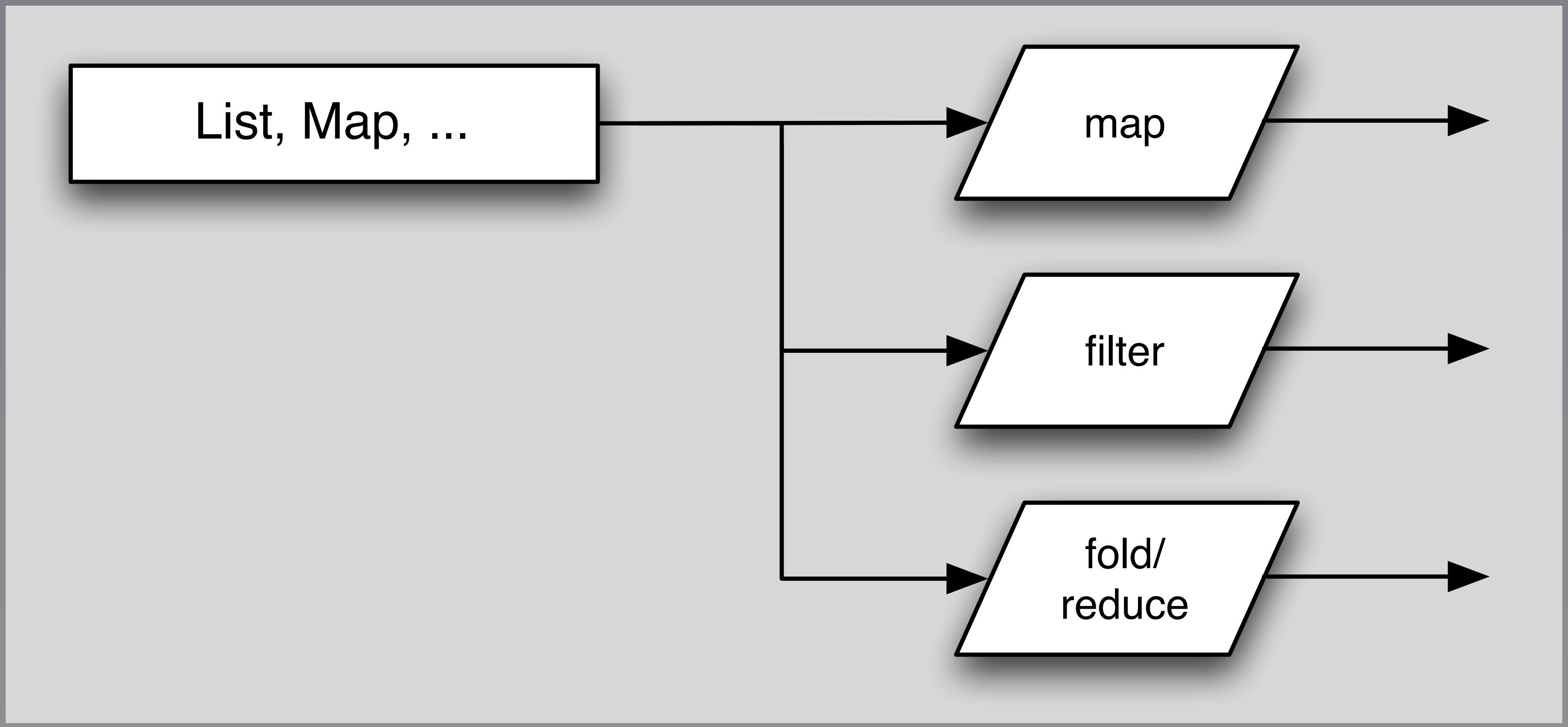




This demonstrates a powerful feature of Scala for constructing embedded/internal DSLs.

# Back to *Functions* as *Objects*

# Classic Operations on *Container* Types






Saturday, November 16, 13

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

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

list map {
  s => s.toUpperCase
}

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



Saturday, November 16, 13

Let’s map a list of strings with lower-case letters to a corresponding list of uppercase strings.

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

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

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

*"function literal"*

*function argument list*

*function body*



Saturday, November 16, 13

Note that the function literal is just the "s => s.toUpperCase". The {…} are used like parentheses around the argument to map, so we get a block-like syntax.

# Typed Arguments

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

*inferred type*

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

*Explicit type*



Saturday, November 16, 13

We've used type inference, but here's how we could be more explicit about the argument list to the function literal. (You'll find some contexts where you have to specify these types.)

# But wait! There's more!

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

*Placeholder*

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



Saturday, November 16, 13

We have this "dummy" variable "s". Can we just eliminate that boilerplate?
I used an informal convention here; if it all fits on one line, just use () instead of {}. In fact, you can use () across lines instead of {}. (There are two special cases where using () vs. {} matters: 1) using case classes, the literal syntax for a special kind of function called a PartialFunction - {} are required, and 2) for comprehensions, - as we'll see.)

# Watch this...

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

list map (println)
// or
list map println
```

*"Point-free" style*



Saturday, November 16, 13

Scala doesn't consistently support point-free style like some languages, but there are cases like this where it's handy; if you have a function that takes a single argument, you can simply pass the function as a value with no reference to explicit variables at all!

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

# How the Sausage Is Made

*Parameterized type*

*Declaration of map*

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

*The function argument*

*map's return type*



Saturday, November 16, 13

Here's the declaration of List's map method (lots of details omitted…). Scala uses [...] for parameterized types, so you can use "<" and ">" for method names!
Note that explicitly show the return type from map (List[B]). In our previous examples, we inferred the return type. However, Scala requires types to be specified on all method arguments!

# 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*



Saturday, November 16, 13

We look at the actual implementation of Function1 (or any FunctionN). Note that the scaladocs have links to the actual source listings. (We're omitting some details…) The trait declares an abstract method "apply" (i.e., it doesn't also *define* the method.) Traits are a special kind of abstract class/interface definition, that promote "mixin composition". (We won't have time to discuss…)

# 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*





You use the function literal syntax and the compiler instantiates an anonymous class using the corresponding FunctionN trait, with a concrete definition of apply provided by your function literal.

# Functions *are* Objects

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

list map {
  s => s.toUpperCase
}

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

*Function "object"*



Saturday, November 16, 13

Back to where we started. Note again that we can use "{…}" instead of "(…)" for the argument list (i.e., the single function) to map. Why, to get a nice block-like syntax.


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/





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.

```scala
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





Homework: Find the Hadoop Java API equivalent implementation.

```scala
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 -> 'word) {
        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.s
      }.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.*

```scala
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](http://polyglotprogramming.com/papers/ScaldingForHadoop.pdf)*





http://polyglotprogramming.com/papers/ScaldingForHadoop.pdf




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] {…}
```



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 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?*



Saturday, November 16, 13

Returning Option tells the user that “there may not be a value” and forces proper handling, thereby drastically reducing sloppy code leading to NullPointerExceptions.

# In Use:

```scala
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)*



Saturday, November 16, 13

Here's idiomatic scala for how to use Options. Our map if of type Map[String,Int]. We match on the Option[V] returned by map.get. If Some(i), we use the integer value I. If there is no value for the key, we use 0 as the default. Note: Option has a short-hand method for this idiom:  m.getOrElse("four", 0).

# Option Details: sealed

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

*All children must be defined in the same file*



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





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.*



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

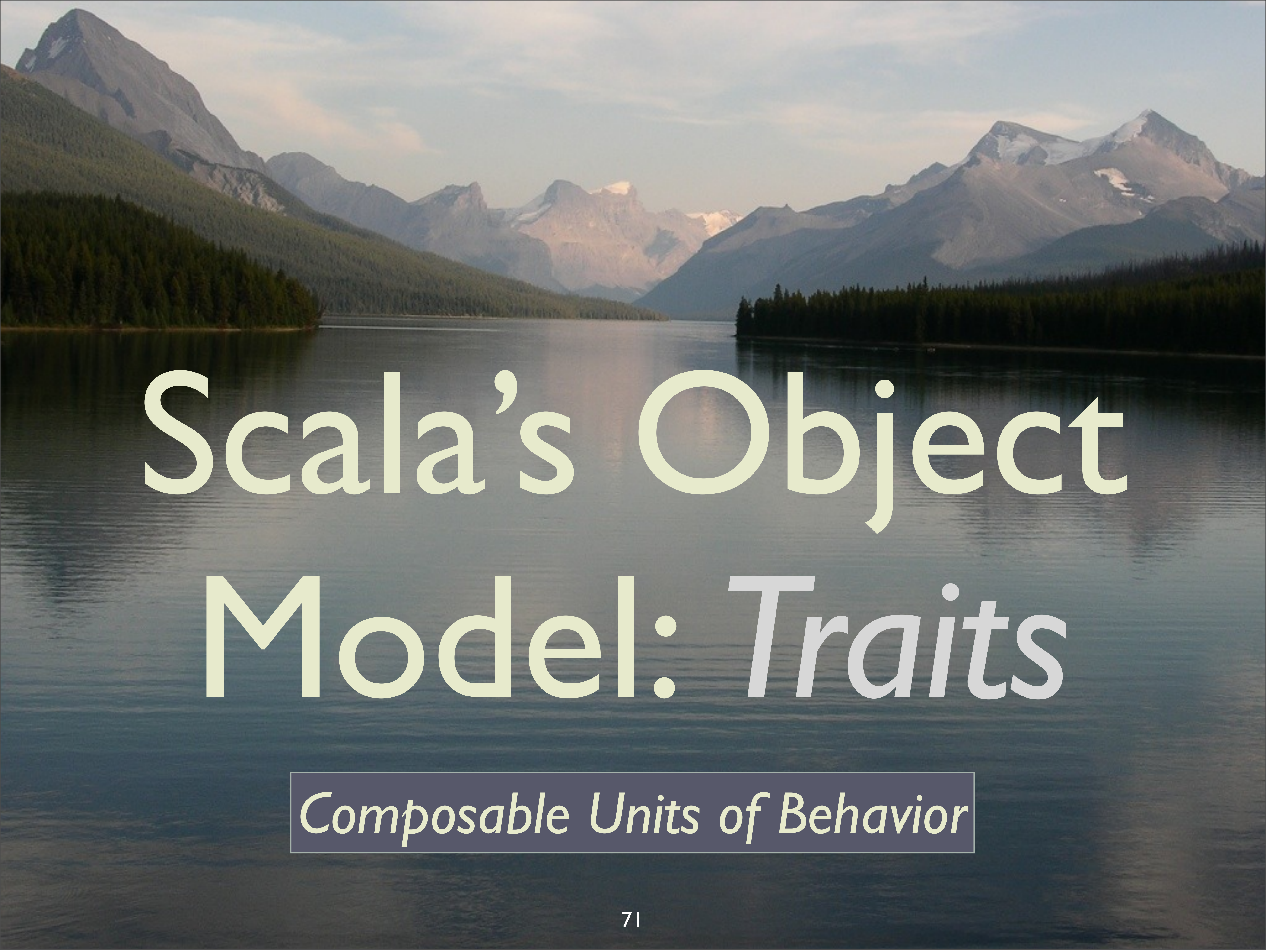
```
case object None
  extends Option[Nothing] {…}
```

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





None is 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].




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!*



Saturday, November 16, 13

Made-up example Java type. The "is a" relationship makes no sense, but the Logger we implemented earlier isn't an interface either.

# Java's object model

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





Chances are, the “logging” and “filtering” behaviors are reusable, yet Java provides no built-in way to “mix-in” reusable implementations. Ad hoc mechanisms must be used.

# Traits

Like interfaces with implementations or...





One way to compare traits to what you know...

# Traits

… like

*abstract classes + multiple inheritance*

(if you prefer).



Saturday, November 16, 13

… and another way.

It’s not an unconstrained form of multiple inheritance like C++. There is no “diamond of death” problem.

# 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.*





I changed some details compared to our original Logger example. Traits don't have constructor argument lists (for various technical reasons), but we can define fields for them, as shown. Here, I make the field abstract, which means that any class that mixes in the trait will have to define "level".

# 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



Saturday, November 16, 13

Note that could have declared a type, say "class ServerWithLogger(…) extends Server(...) with Logger {…}, but if you only need one instance, we can just do it "on the fly!" Note that the level is defined as a body for this object, much the same way you define an anonymous inner class and define its abstract members.

# Like Java 8 Interfaces?

✓ *Default methods*

- Can define method bodies.

X *Fields*

- J8 fields remain *static final*, not *instance* fields.





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.



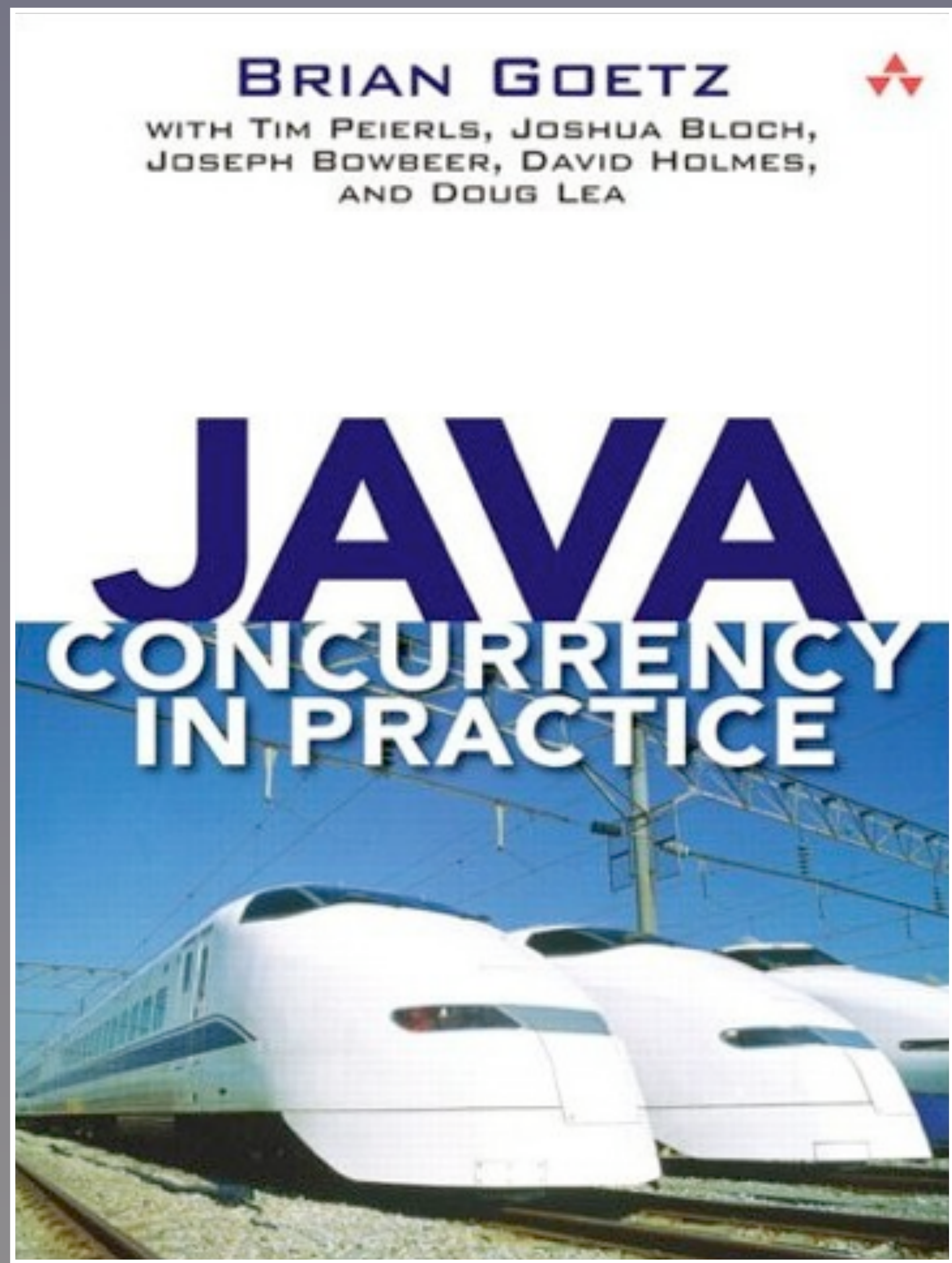

Saturday, November 16, 13

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)*






Saturday, November 16, 13

It's very hard to do multithreaded programming robustly. We need higher levels of abstraction, like Actors.

# Actor Model

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





Each actor might mutate state itself, but the goal is to limit mutations to just a single actor, which is thread safe. All other actors send messages to this actor to invoke a mutation or read the state.

# Actor Model

- First developed in the 70's by Hewitt, Agha, Hoare, *etc.*
- Made "famous" by *Erlang*.





The actor model is not new!!

# Akka

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





The Distributed Programming framework for Scala, which also support Java!

# Akka

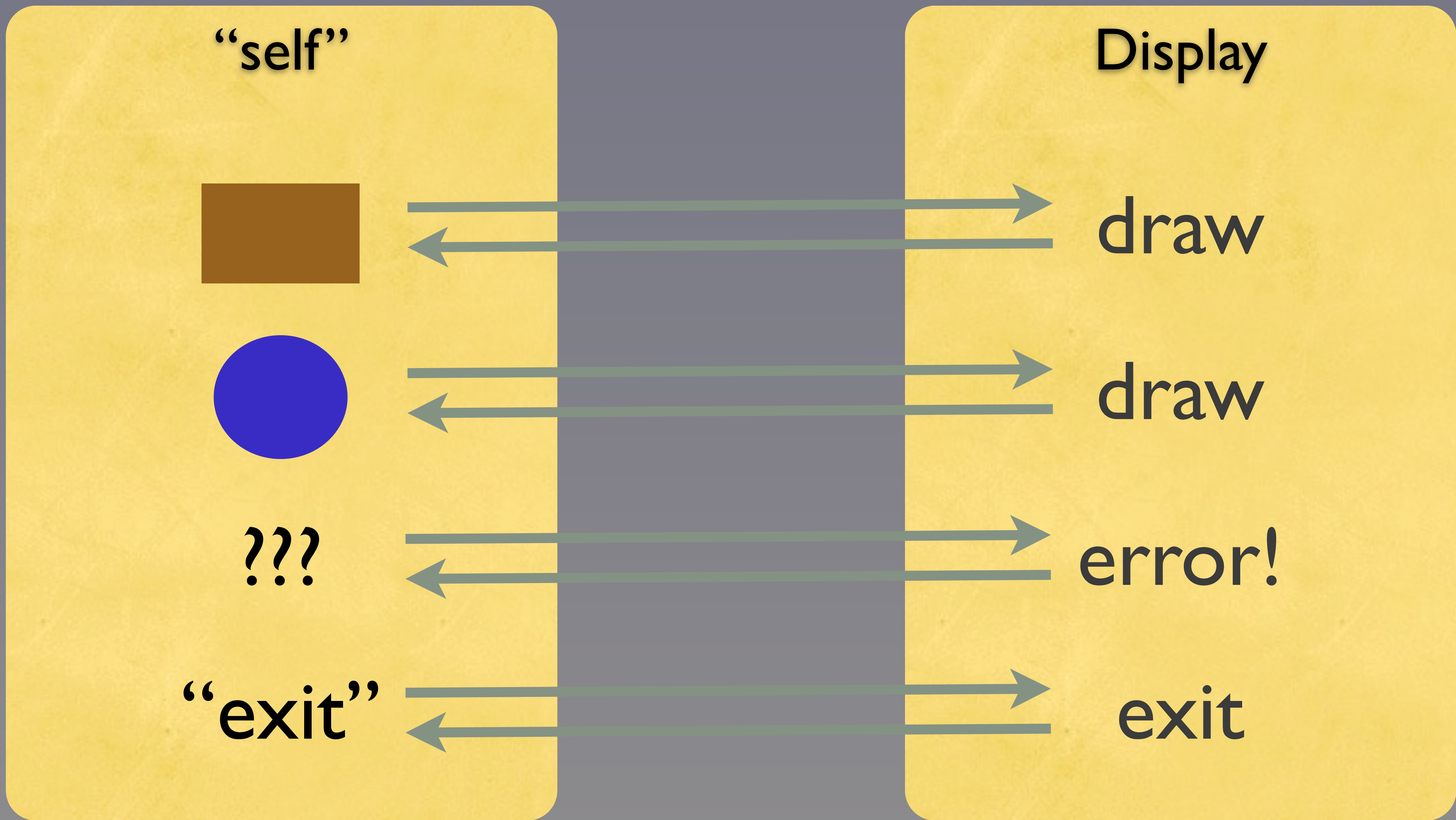
- Also has a complete Java API.
  - akka.io





The Distributed Programming framework for Scala, which also support Java!

# 2 Actors:






Saturday, November 16, 13

Our example. An actor for drawing geometric shapes and another actor that drives it.

```
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.

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

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

*concrete draw methods*





Saturday, November 16, 13

Case classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape.
For our example, the draw methods will just do “println(“drawing: “+this.toString)”.

Use the Akka Actor library

```scala
package shapes
import akka.actor.Actor

class Drawer extends Actor {
  def receive = {
    …
  }
}
```

Actor

receive and handle each message

Actor for drawing shapes





Saturday, November 16, 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 has now replaced Scala's original actors library. So, some of the basic actor classes are part of Scala's library, but we'll use the full Akka distibution.

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)
}
```





Saturday, November 16, 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()
    sender ! ("Shape drawn
  case "exit" =>
    println("-> exiting...")
    sender ! ("good bye!")
  case x =>                  // default
    println("-> Error: " + x)
    sender ! ("Unknown: " + x)
}
```

*pattern matching*





Saturday, November 16, 13

Each pattern is tested and the first match “wins”. The messages we expect are a Shape object, the “exit” string or anything else. Hence, the last “case” is a “default” that catches anything, we we treat as an unexpected error.

```scala
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*





Saturday, November 16, 13

After handling each message, a reply is sent to the sender, using “self” to get the handle to our actor “nature”.

```scala
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





Saturday, November 16, 13

Even compressed on a presentation slide, there isn't a lot of code!

```scala
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*





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).

```scala
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)
 }
}
…
```

*Singleton for main*

*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 = {
    …
  }
}
```





Saturday, November 16, 13

Here's the driver actor "companion class" for the object on the previous slide that held main.
Normally, you would not do such synchronous call and response coding, if avoidable, as it defeats the purpose of using actors for concurrency.

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

*sent by driver*

*sent by drawer*





Saturday, November 16, 13

Here's the driver actor, a scala script (precompilation not required) to drive the drawing actor.
Normally, you would not do such synchronous call and response coding, if avoidable, as it defeats the purpose of using actors for concurrency.

```
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.*





Saturday, November 16, 13

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!!

```
…
// 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*





Saturday, November 16, 13

The power of combining the best features of FP (pattern matching and "destructuring") and OOP (polymorphic behavior).


Recap

# Scala is...

# a better Java,

object-oriented

and

functional,

succinct,
elegant,
and
powerful.


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.


Extra Slides

# Modifying Existing Behavior with Traits

# Example

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

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





A very simple abstraction for a Queue.

# Log put

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





(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 so. 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 so. So, this method is STILL abstract and it's going to override a concrete "put" "real soon now".

```scala
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*





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

```scala
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*



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*





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!*





Not shown, nesting of traits...

# For Comprehensions

# For “Comprehensions”

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

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

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

*No if statement*



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.

# 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*





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??*



Saturday, November 16, 13

Syntactic sugar: obj.operation(arg) == obj operation arg

# Make your own controls

```
// Print with line numbers.


loop (new File("…")) {
  (n, line) =>

  format("%3d: %s\n", n, line)
}
```



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.

loop (new File("…")) {
  (n, line) =>

  format("%3d: %s\n", n, line)
}
```

*control?*

*File to loop through*

*Arguments passed to…*

*what do for each line*

*How do we do this?*

# Output on itself:

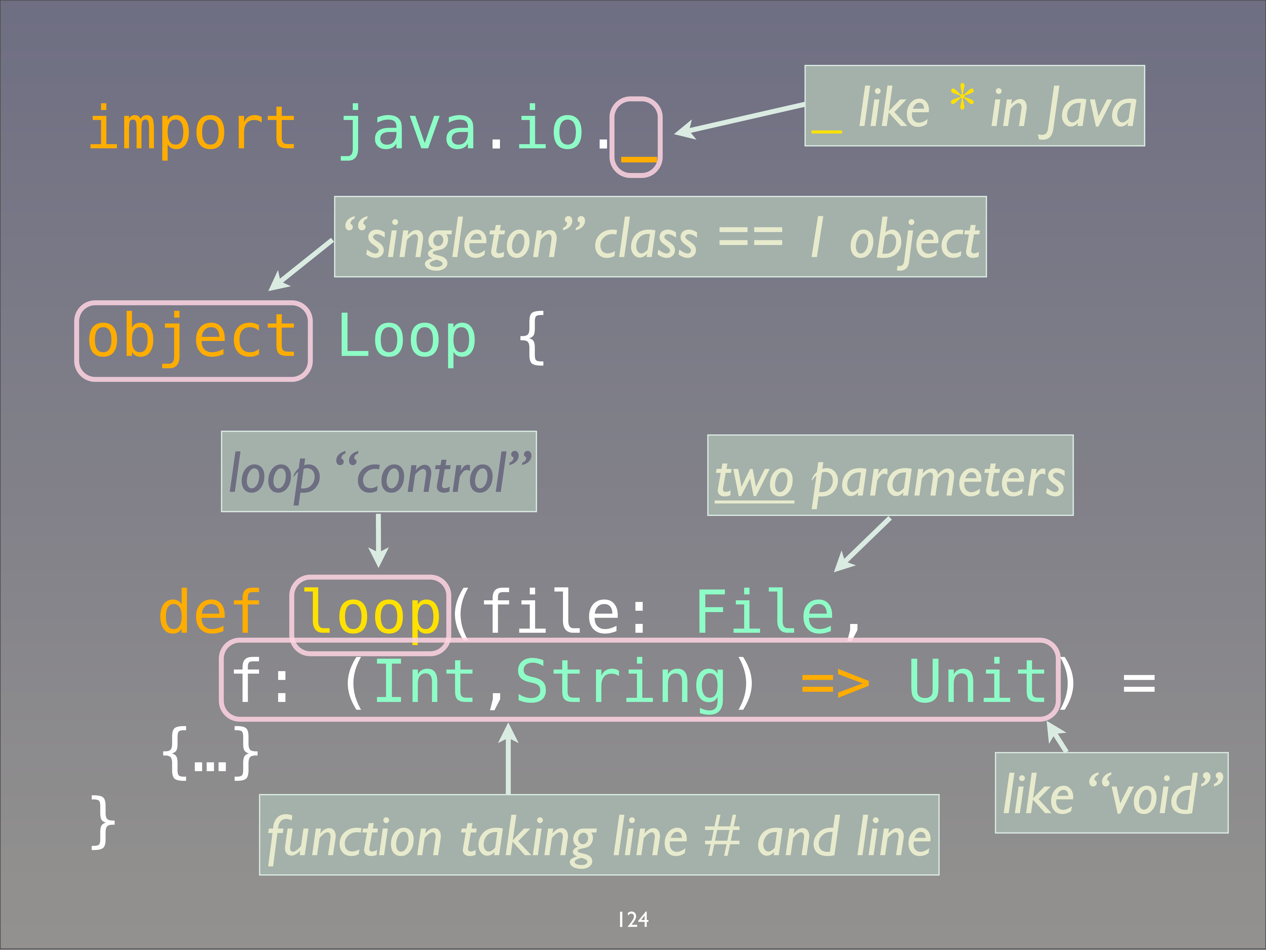
```
1: // Print with line …
2:
3:
4: loop(new File("…")) {
5:   (n, line) =>
6:
7:   format("%3d: %s\n", …
8: }
```

```scala
import java.io._

object Loop {

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



Saturday, November 16, 13

Here's the code that implements loop...






Saturday, November 16, 13

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 {

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

*two parameters*



Saturday, November 16, 13

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

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

```scala
object Loop {

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

*two parameters lists*



Saturday, November 16, 13

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*





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.

```scala
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...*



Saturday, November 16, 13

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...*



Saturday, November 16, 13

Here is the nested method, doLoop.

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

*A goal of Functional Programming*

# It is *Tail Recursive*

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

*Scala optimizes tail recursion into loops*



Saturday, November 16, 13

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

# Functions with Mutable State

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

```scala
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")
```





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.