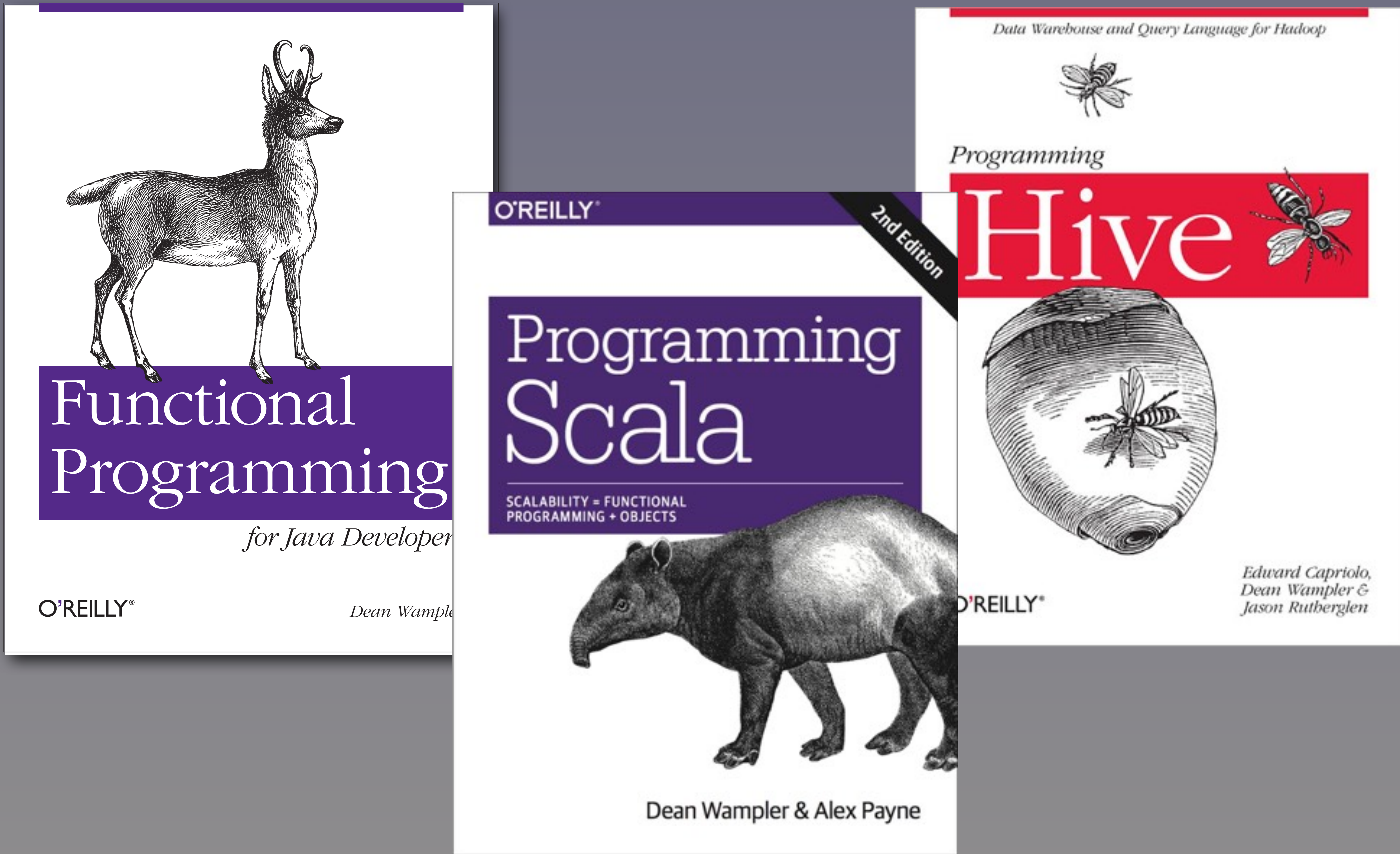
The Seductions of Scala

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

July 23, 2015

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/>

Thursday, July 23, 15

My books.

Why did we *need* a *new* language?

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*

...

Thursday, July 23, 15

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...

Thursday, July 23, 15

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...

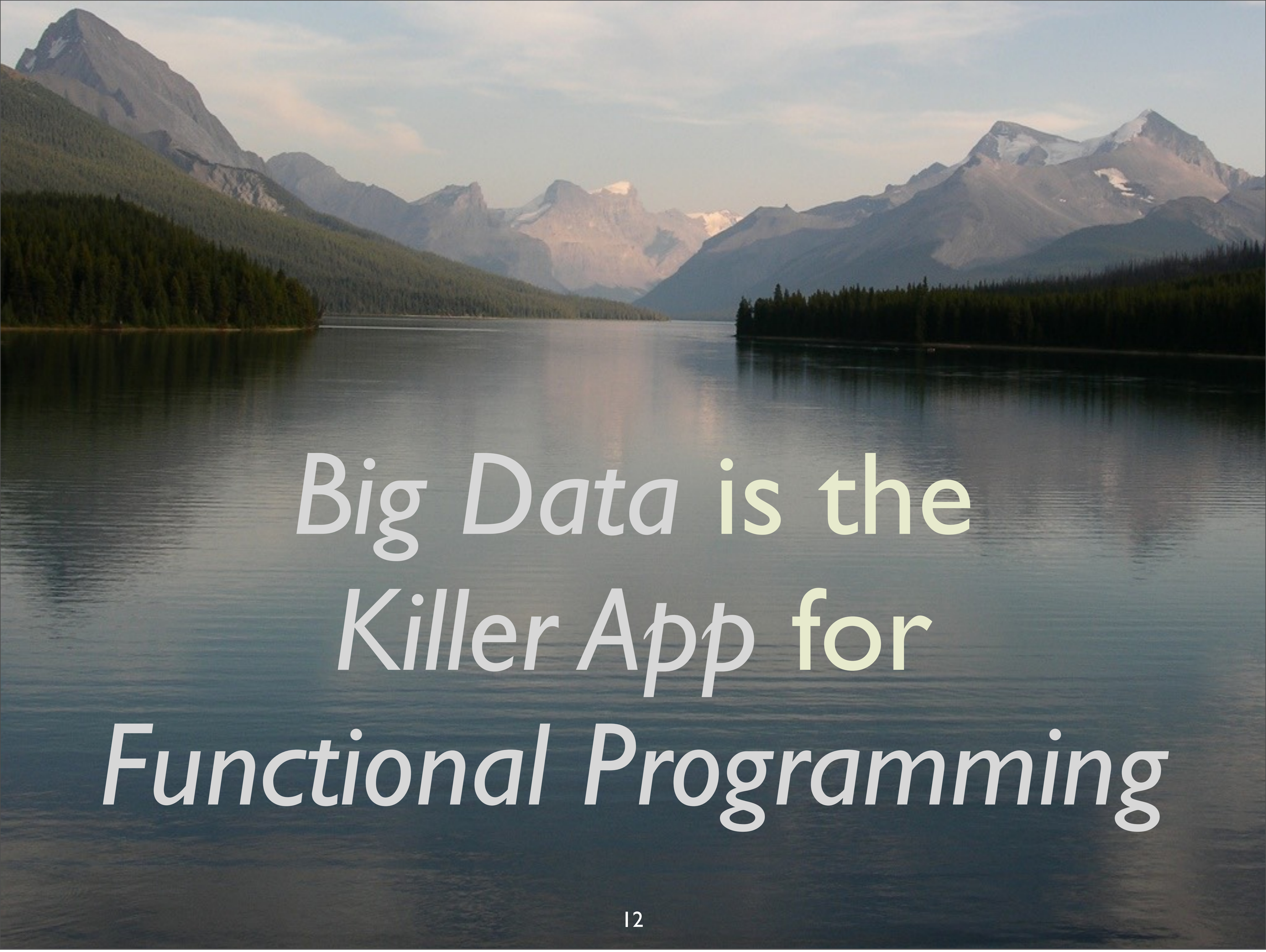
- A JVM language.
- Functional and object oriented.
- Statically typed.
- An improved Java.

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.

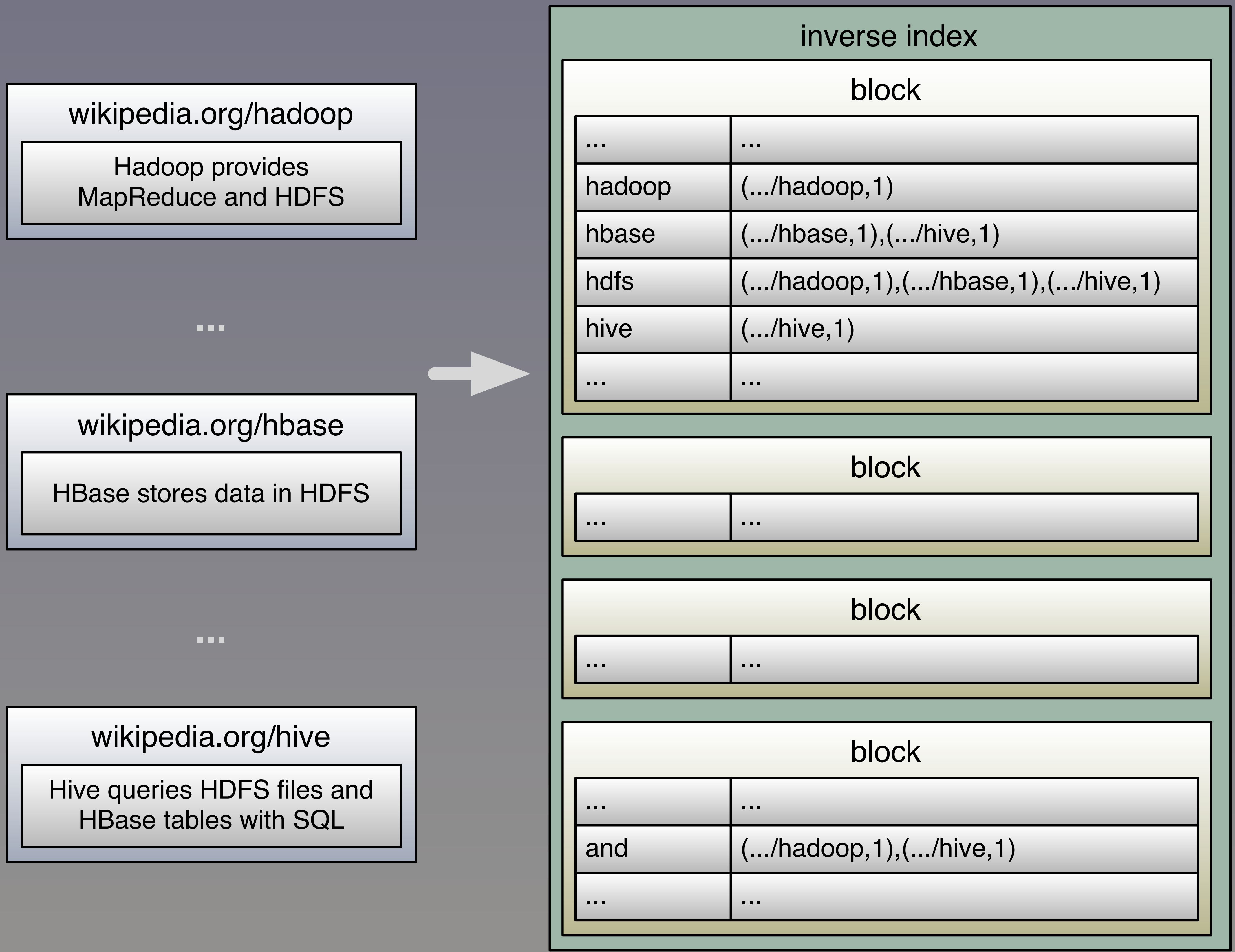


Thursday, July 23, 15

This talk has evolved a lot in the ~6 years I've been giving it. Right now, the most compelling argument for Scala is that it's the best language we have for writing Big Data applications (e.g., for Hadoop clusters), as exemplified by several tools...

So for motivation, here's a teaser of where we're headed.

Inverted Index



Thursday, July 23, 15

What we want to do, read a corpus, in this case Wikipedia pages, where we use the URL as the doc Id as the key and the contents as the value. We will tokenize into words, and “invert” the index so the words are keys and the values are a list of “tuples”, “(id1, count1), (id2, count2), ...”, for the corresponding word.

Spark: Inverted Index

```
...
sparkContext.textFile("/path/to/input")
.map { line =>
  val array = line.split(",", 2)
  (array(0), array(1)) // (id, text)
}.flatMap {
  case (id, text) => tokenize(text).map(
    word => ((word, id), 1))
}.reduceByKey {
  (count1, count2) => count1 + count2
}.map {
  case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```


Thursday, July 23, 15

The inverted index algorithm in Spark, which ingests an identifier for a document (e.g., web page) and the contents of it, then “inverts” that data outputting each word in the texts globally and all the documents where it’s found, plus its count in each document.

I won’t explain the details now, but they should be easier to understand once we’re through. The gist of the algorithm is this: Using the entry point, a SparkContext instance, we load some text files, then map over each line of text to split it on the first comma, returning a two-element “tuple” with the first and second elements from the Java array that results from splitting. They are the document id and the text. Next we “flat map” over those tuples. While map is transforms one input to one output (1-to-1), flatMap transforms one input to 0-to-many outputs. Here we use it tokenize text into worlds and outputs a new 2-tuple, where the first element is itself a 2-tuple (word, id) and a “seed” count of 1 is the second element in the outer tuple. reduceByKey is an optimized groupBy, using the (word,id) keys, when we just need to “reduce” the rest of the outer tuples. In this case, we add the “1”s. Now we have a total count of (word,id) pairs. The next step converts those tuples to (word, (id,n)) tuples, after which we group over the words, and now we have the final result, which we write out.

Let’s go back and build up to understanding what we just saw. First, objects can behave like “functions”, (although not all objects do this).

```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** *an immutable field*

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

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

method

class body is the "primary" constructor

Thursday, July 23, 15

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

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

returns and semicolons inferred

Thursday, July 23, 15

No need for return keywords or semicolons (in most cases).

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

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

val error = new Logger(ERROR)

...
error("Network error.")
```


Thursday, July 23, 15

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

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

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

apply *is called*

```
…
error("Network error.")
```


Thursday, July 23, 15

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.

Thursday, July 23, 15

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!

More on *Methods*

A singleton object

```
object Error {

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

Like a static method

apply *is called*

```
…
Error("Network error.")
```


Thursday, July 23, 15

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

Infix Operator Notation

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

is actually just

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


Thursday, July 23, 15

Note the “infix operator notation”; x.m(y) ==> x m y. Why? Scala lets you use more characters for method names, like math symbols, so this syntax looks more natural.

Java Primitives?

A side note on Java primitives

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.*

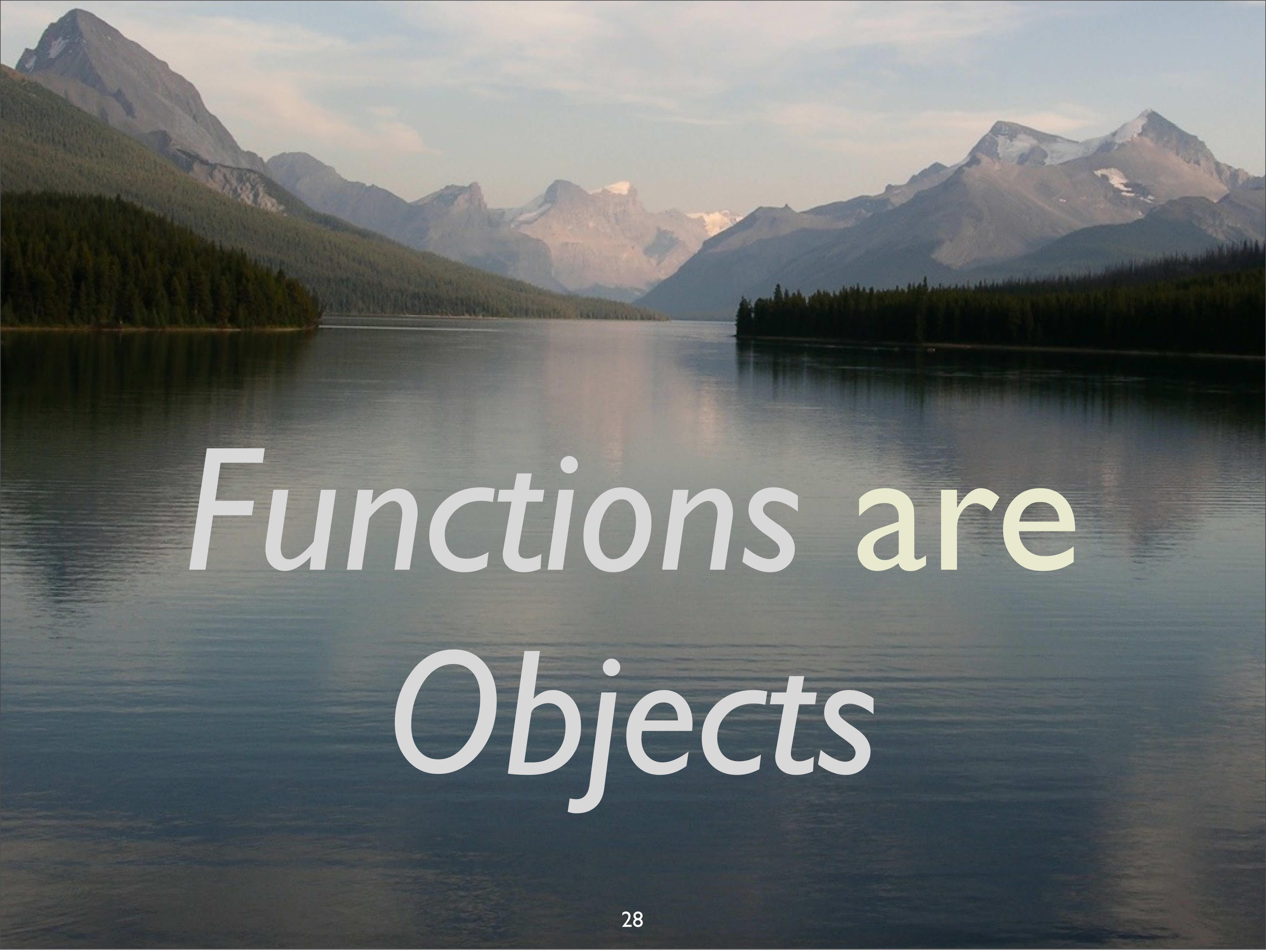
```
val l = List.empty[Int]
```

An empty list of Ints.

```
Java? ... List<Int>
```


Thursday, July 23, 15

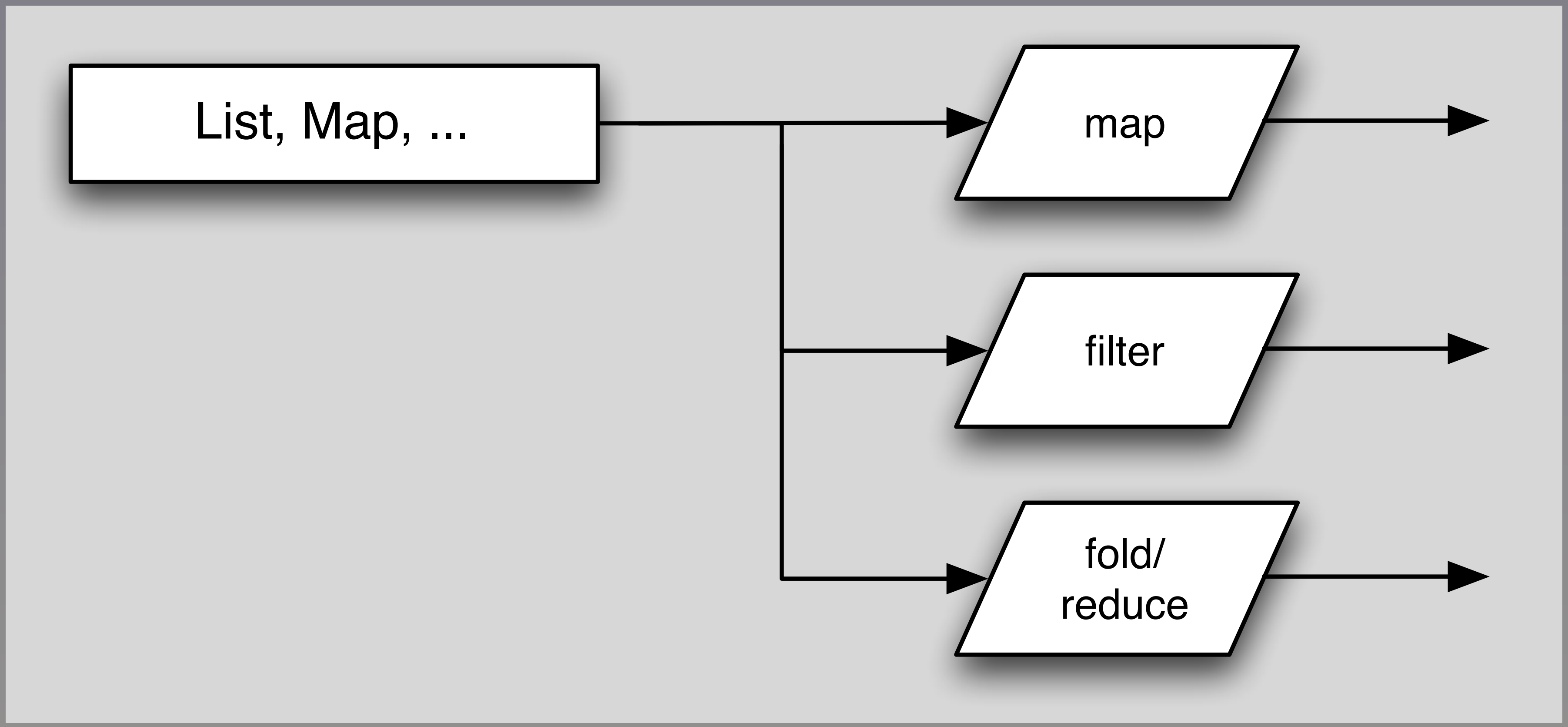
You don't have to explicitly box primitives; the compiler will optimize these objects to primitives (with some issues involving collections...)
Note the syntax for parameterizing the type of List, [...] instead of <...>.



Thursday, July 23, 15

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!

Classic Operations on *Container* Types



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

Seq.apply

```
val seq1 = Seq("a", "b")
// seq1: List[String] = List(a,b)

val seq2 = seq1.map {
  s => s.toUpperCase
}
// seq2: List[String] = List(A,B)
```


The comment is what the Scala intepreter shell would echo back to you.
Let’s map a list of strings with lower-case letters to a corresponding list of uppercase strings.

map called on seq1 *(dropping the ".")*

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

```
seq1 map {
  s => s.toUpperCase
}
```

"function literal"

function argument list

function body

Thursday, July 23, 15

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

```
seq1 map {
  s => s.toUpperCase
}
```

inferred type

```
seq1 map {
  (s:String) => s.toUpperCase
}
```

Explicit type

Thursday, July 23, 15

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)
```


Thursday, July 23, 15

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...

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

list foreach (println)
// the same as:
list foreach println
```

"Point-free" style

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

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

Declaration of map

The function argument

map's return type

Thursday, July 23, 15

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 a Java 8 interface

"contravariant", "covariant" typing

```
trait Function1[-A,+R] {
  def apply(a:A): R
  …
}
```

No method body, therefore it is abstract

Thursday, July 23, 15

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

```scala
s => s.toUpperCase
```

What you write.

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

What the compiler generates

An anonymous class

Thursday, July 23, 15

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

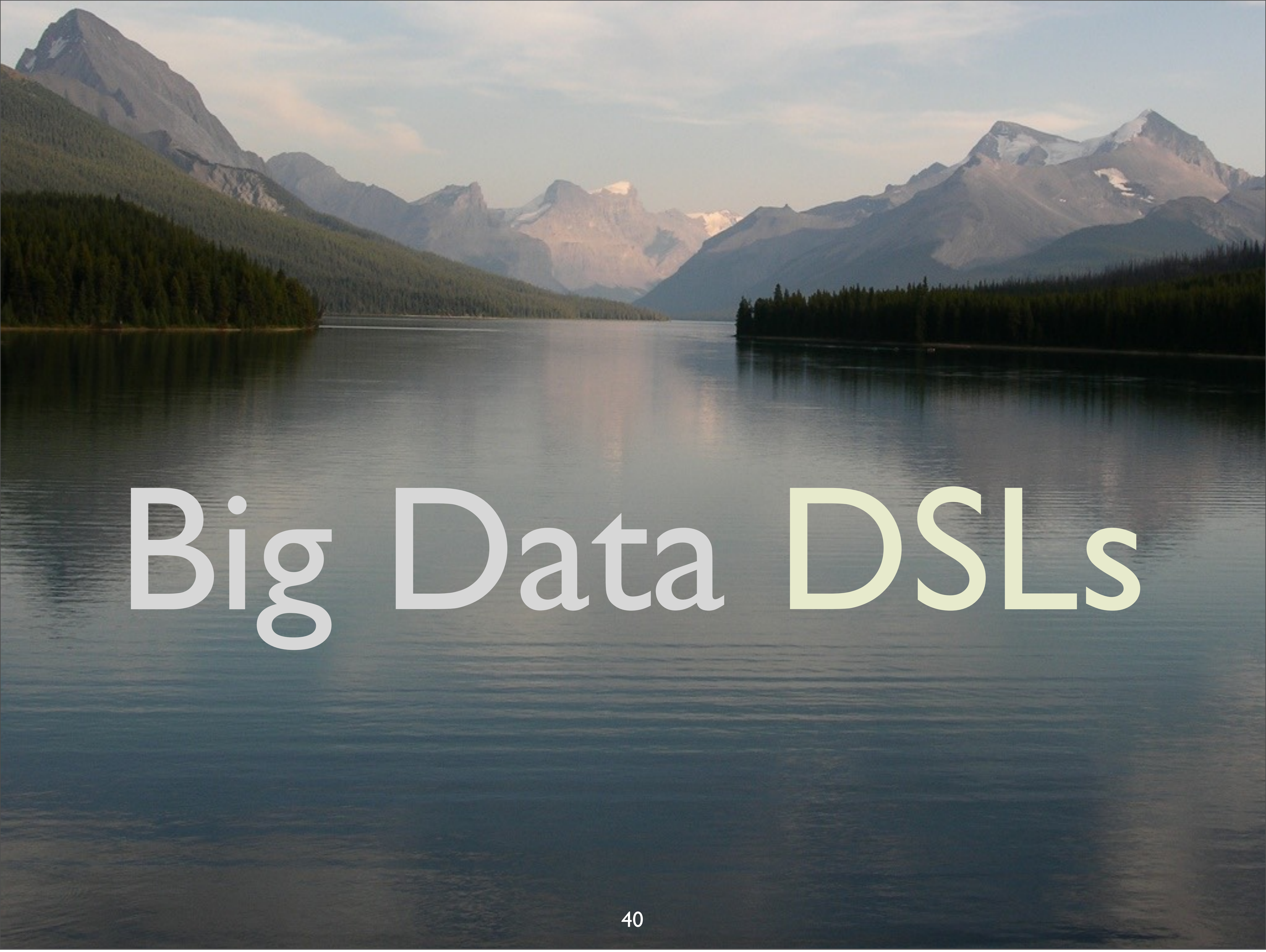
```
val seq1 = Seq("a", "b")
// seq1: List[String] = List(a,b)

val seq2 = seq1.map {
  s => s.toUpperCase
}
// seq2: List[String] = List(A,B)
```

Function "object"

Thursday, July 23, 15

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.



Thursday, July 23, 15

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...

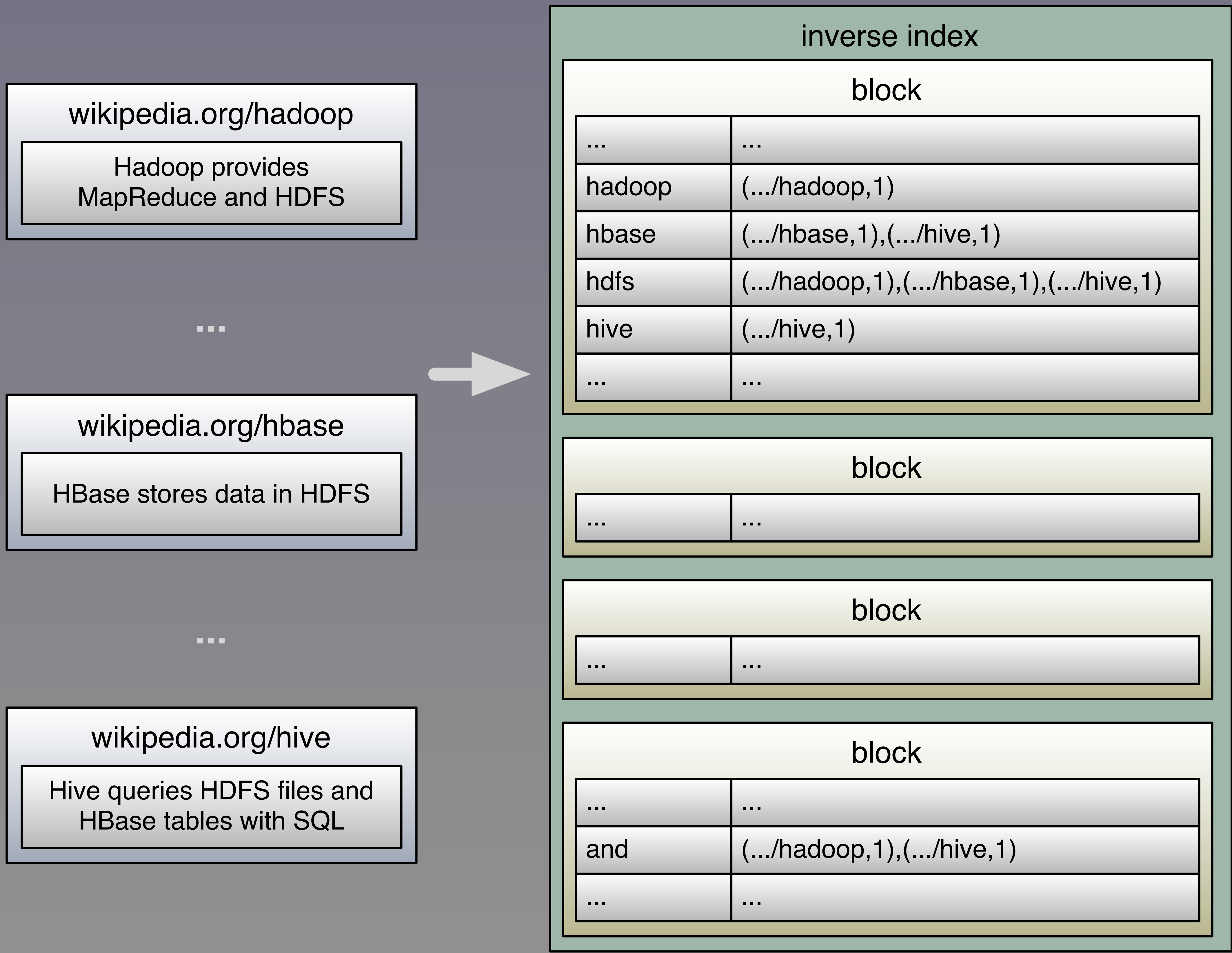
Spark: Replacing MapReduce in Hadoop

- Written In Scala
- API based on Scala collections API
- http://spark.apache.org

I found MapReduce incredibly frustrating when I started doing Hadoop. It's a very limited programming model, with poor performance, and a terrible API in Hadoop, specifically.

Let's revisit the *Inverted Index* algorithm.

Inverted Index



What we want to do, read a corpus, in this case Wikipedia pages, where we use the URL as the doc Id as the key and the contents as the value. We will tokenize into words, and "invert" the index so the words are keys and the values are a list of "tuples", "(id1, count1), (id2, count2), ...", for the corresponding word.

```scala
import org.apache.spark.SparkContext

object InvertedIndex {
 def main(args: Array[String]) = {
  val sparkContext = new SparkContext(...)
  sparkContext.textFile("/path/to/input")
  ...
  sparkContext.stop()
 }
}
```

What we had before...

Thursday, July 23, 15

Now a full program for the Inverted Index. Note that the stuff in “main” could also be run interactively or as a script in a version of the Scala console called the Spark Shell, even on a cluster!!

```scala
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id), 1))
}.reduceByKey {
 (count1, count2) => count1 + count2
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```


This is a SINGLE expression, although you could break up the steps and assign them to variables. Let's walk through it.

```
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split
 (array(0), array(1)) /
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id)
}.reduceByKey {
 (count1, count2) => c
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```

Load data from files, e.g., in HDFS

Each step returns a new RDD: Resilient Distributed Dataset

Thursday, July 23, 15

RDDs are distributed data sets. Your data is partitioned and each partition is computed over in a separate JVM process, giving you parallelism.

```
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tol
   word => ((word, id)
}.reduceByKey {
 (count1, count2) => co
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```

Split each line on the 1st ",". Return a 2-tuple holding the resulting "id" and "text"

Thursday, July 23, 15

Note the elegant (a,b,c,d) tuple syntax.

```
(1, “two”, 3.14)
```

What you write.

```
new Tuple3[Int,String,Double] {
  private val first  = 1
  private val second = “two”
  private val third  = 3.14
  def _1:Int      = first
  def _2:String = second
  def _3:Double = third
}
```

What the compiler generates (sort of)

Thursday, July 23, 15

Note the elegant (a,b,c,d) tuple syntax.

```
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id), 1))
}.reduceByKey {
 (count1, count2) => co
}.map {
 case ((word, id), n) =
}.groupByKey
// (w1, list((id11,n11
.saveAsTextFile("/path/
```

Tokenize the text into words, return a 2-tuple with another 2-tuple (word, id) and a seed count of 1

Thursday, July 23, 15

Note the elegant (a,b,c,d) tuple syntax.

Pattern Matching

```
val string = value match {
  case 1 => “one”
  case “two” => “two”
  case d: Double => “double”
  case ((a,b),c) =>
        s“3-tuple: $a,$b,$c”
  case unknown => “unknown”
}
```


Power tool for determining the type, matching on specific values or general positions and for the latter, assigning a variable to the matched elements, so it’s also very convenient for “de-structuring” values. Note that the s”...” is an interpolated string where the variables referenced with “$a” etc will be filled in.

```scala
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id), 1))
}.reduceByKey {
 (count1, count2) => count1 + count2
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```


Thursday, July 23, 15

Note the elegant (a,b,c,d) tuple syntax.

```
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id), 1))
}.reduceByKey {
 (count1, count2) => count1 + count2
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11
.saveAsTextFile("/path/
```

Like group by followed but the function is applied to add the 1s.

Thursday, July 23, 15

0-to-many output. Pattern matching is used to extract the tuple elements into named variables...

```scala
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id), 1))
}.reduceByKey {
 (count1, count2) => count1 + count2
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```

Now make the "word" the key and group over words.

0-to-many output. Pattern matching is used to extract the tuple elements into named variables...

```
sparkContext.textFile("/path/to/input")
.map { line =>
 val array = line.split(",", 2)
 (array(0), array(1)) // (id, text)
}.flatMap {
 case (id, text) => tokenize(text).map(
   word => ((word, id), 1))
}.reduceByKey {
 (count1, count2) => count1 + count2
}.map {
 case ((word, id), n) => (word, (id, n))
}.groupByKey
// (w1, list((id11,n11), (id12,n12), ...))
.saveAsTextFile("/path/to/output")
```

Save to the file system

0-to-many output. Pattern matching is used to extract the tuple elements into named variables...

For more on Spark see my workshop:

[*github.com/deanwampler/spark-workshop*](https://github.com/deanwampler/spark-workshop)

Thursday, July 23, 15

https://github.com/deanwampler/spark-workshop

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] {…}
```


Thursday, July 23, 15

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?

Thursday, July 23, 15

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:

```
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)

Thursday, July 23, 15

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

Thursday, July 23, 15

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 makes the value argument a field (val keyword not required).
- equals, hashCode, toString.

Case Classes

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

- *singleton object* with a *factory* apply method
- pattern matching support.
- ...

Scala's Object Model: *Traits*

Composable Units of Behavior

Thursday, July 23, 15

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 { … }
```

Better conceptually...

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

Inspired Java 8 interfaces; add method implementations and state...

Java 8 interfaces don’t support state (i.e., what will become fields in a class that mixes in the trait).

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

Thursday, July 23, 15

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.


Recap

Scala is...

a better Java,

object-oriented
and
functional,

succinct,
elegant,
and
powerful.

Typesafe

Questions?

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

July 23, 2015

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

Lists and Maps

Lists

List.apply()

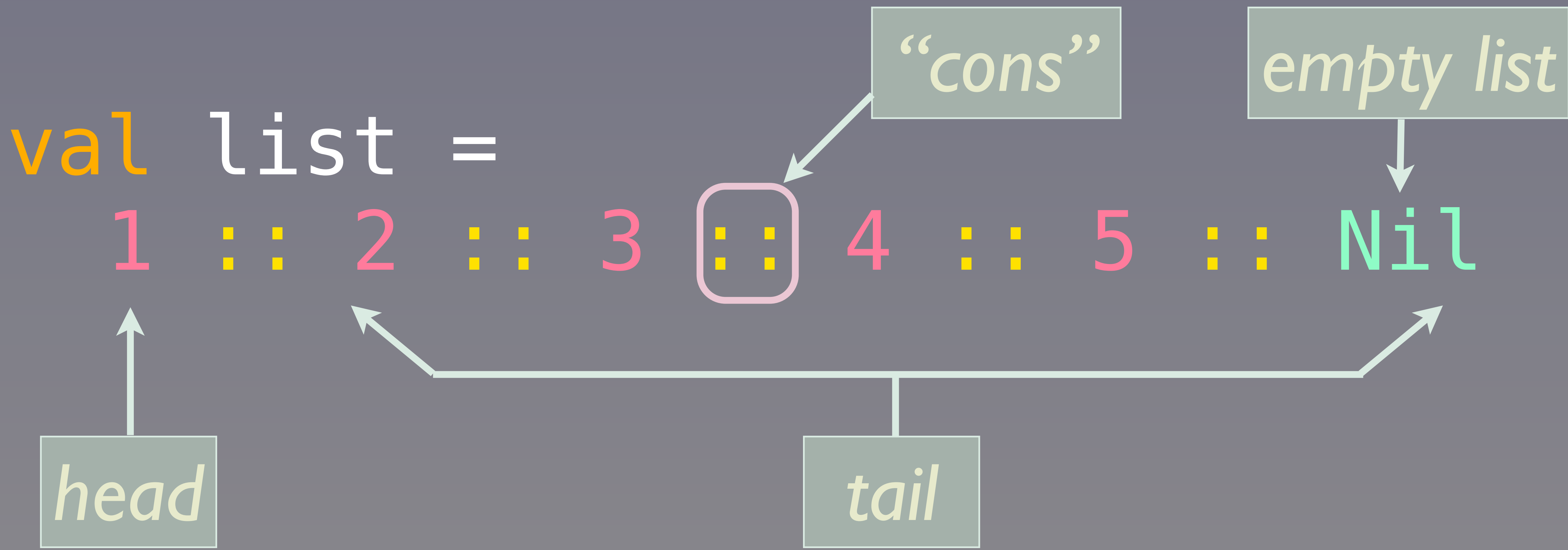
```
val list = List(1, 2, 3, 4, 5)
```

The same as this "list literal" syntax:

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


Thursday, July 23, 15

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.



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!

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

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

Method names can contain almost any character.

Thursday, July 23, 15

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!

“::” 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 ")".

Infix operator notation.

Infix Operator Notation

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

is actually just

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


Thursday, July 23, 15

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>
```


Thursday, July 23, 15

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)
```

Thursday, July 23, 15

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...

Thursday, July 23, 15

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*

Thursday, July 23, 15

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!

Thursday, July 23, 15

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)
}
```


Thursday, July 23, 15

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.

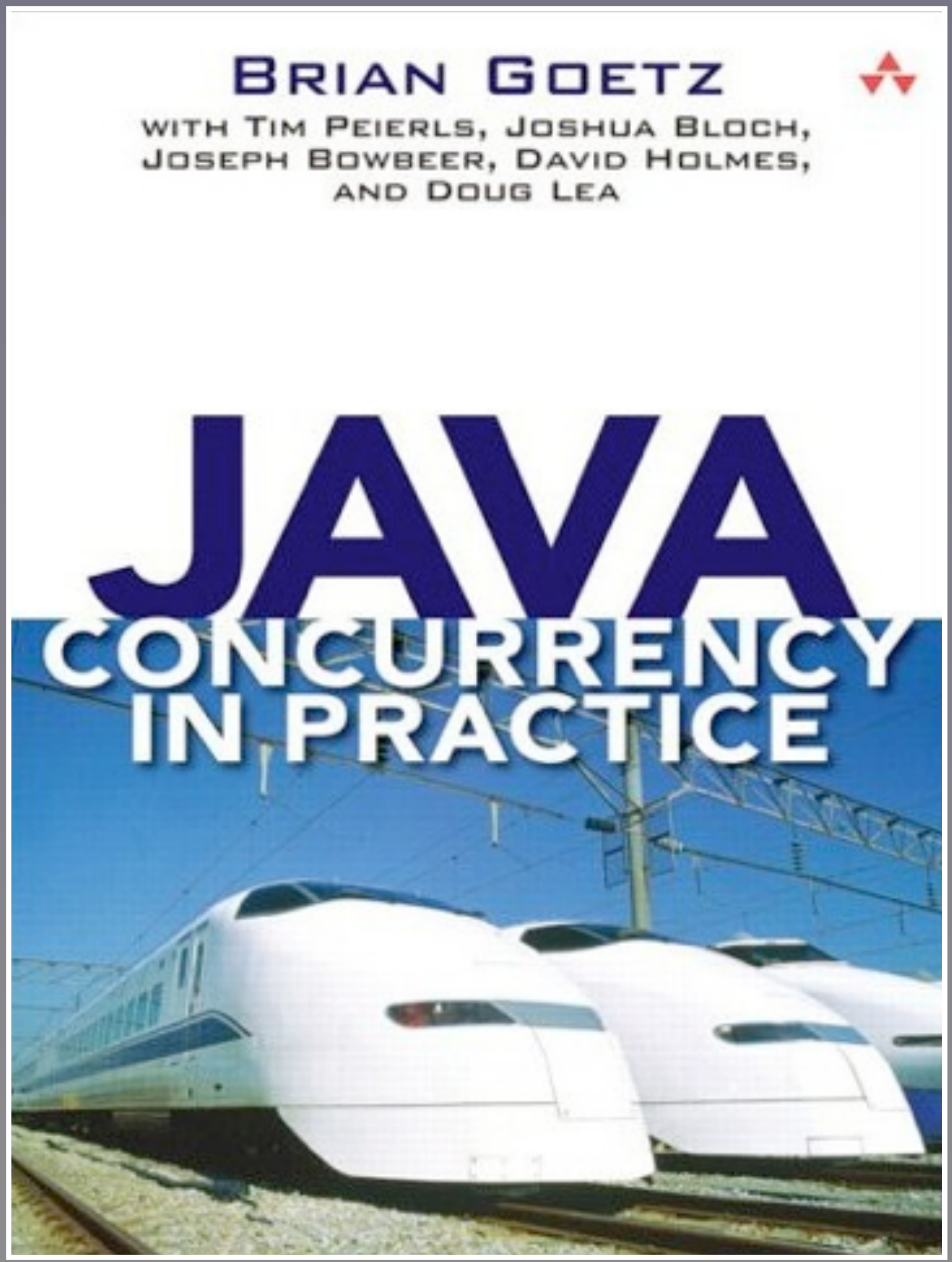
Thursday, July 23, 15

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)



Thursday, July 23, 15

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!

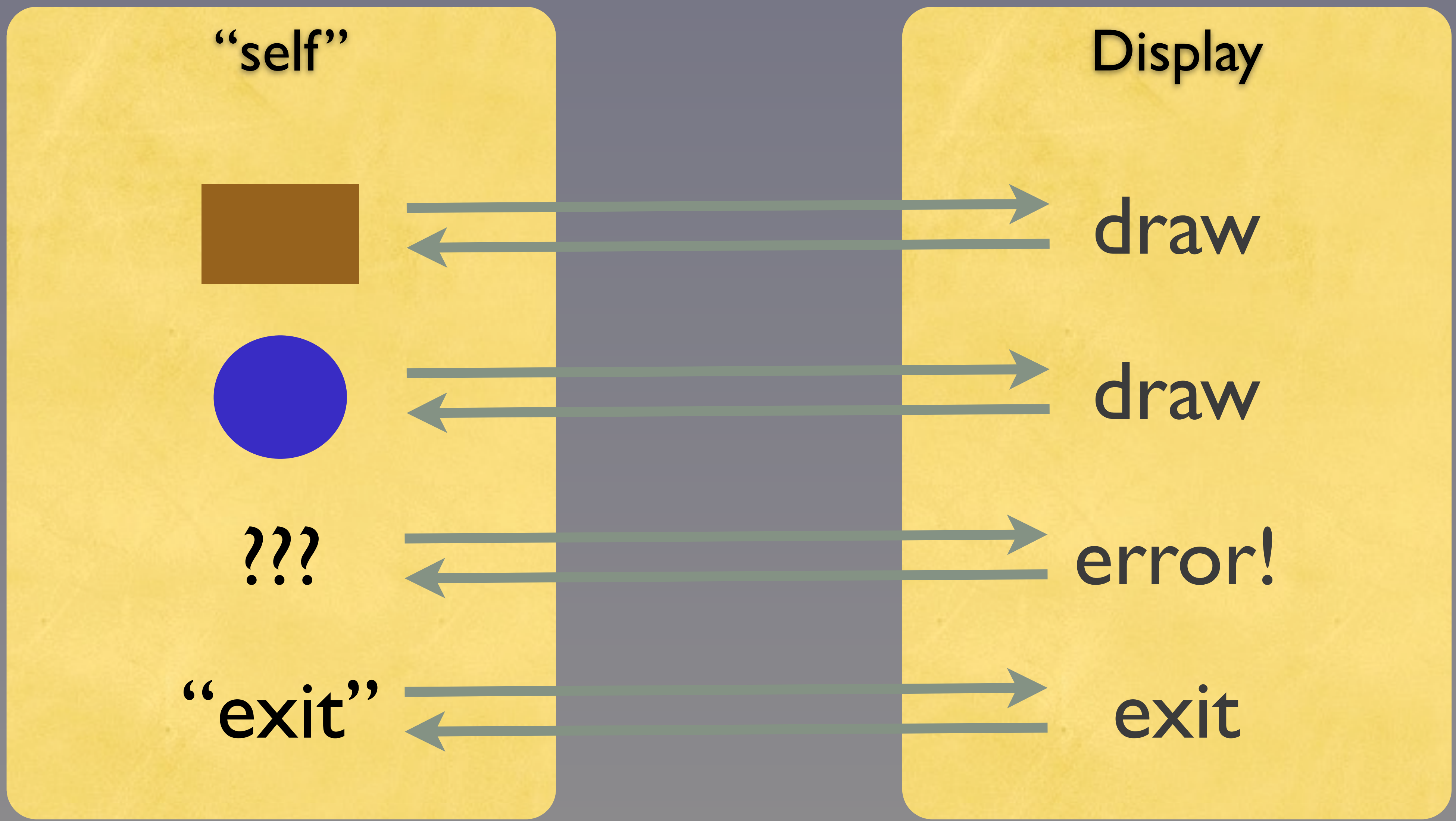
Akka

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

Thursday, July 23, 15

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

2 Actors:



Thursday, July 23, 15

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

Thursday, July 23, 15

"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

Thursday, July 23, 15

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)”.

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

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

Use the Akka Actor library

Actor

receive and handle each message

Actor for drawing shapes

Thursday, July 23, 15

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)
}
```


Thursday, July 23, 15

"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.

```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)
}
```

pattern matching

Thursday, July 23, 15

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*

Thursday, July 23, 15

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

Thursday, July 23, 15

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

Thursday, July 23, 15

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 {
  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

Thursday, July 23, 15

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 = {
    ...
  }
}
```


Thursday, July 23, 15

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.

```scala
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

Thursday, July 23, 15

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.

Thursday, July 23, 15

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

Thursday, July 23, 15

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

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)
  }
}
```


Thursday, July 23, 15

(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??

Thursday, July 23, 15

(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

Thursday, July 23, 15

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

Thursday, July 23, 15

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

Thursday, July 23, 15

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*

Thursday, July 23, 15

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*

Thursday, July 23, 15

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??

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)
}
```


Thursday, July 23, 15

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) =
  {…}
}
```


Thursday, July 23, 15

Here’s the code that implements loop...

_ like * in Java

"singleton" class == 1 object

loop "control"

two parameters

```
import java.io._

object Loop {

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

like "void"

function taking line # and line

Thursday, July 23, 15

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

Thursday, July 23, 15

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

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

Thursday, July 23, 15

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…

Thursday, July 23, 15

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)
    }
  }
}
```

f and reader visible from outer scope

recursive

Finishing Numberator...

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

Thursday, July 23, 15

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")
```


Thursday, July 23, 15

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.