The Seductions of Scala

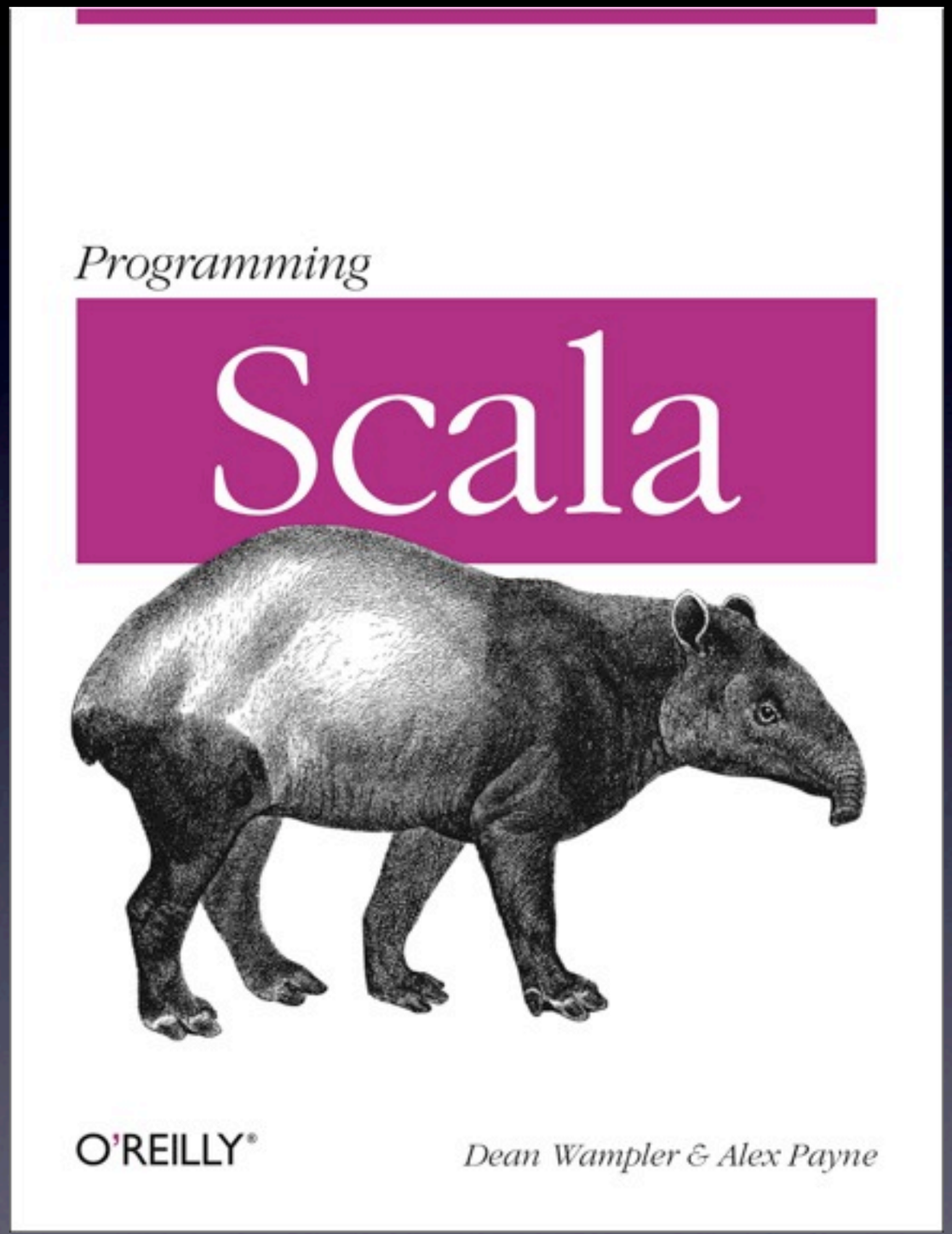
Dean Wampler

dean@deanwampler.com
@deanwampler
polyglotprogramming.com/talks
programmingscala.com

<*shameless-plug*/>

Co-author,
Programming Scala

programmingscala.com



Tuesday, July 20, 2010

Available now from oreilly.com, Amazon, etc.

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

Tuesday, July 20, 2010

I picked Scala in late 2007 to learn because I wanted to learn a functional language. Scala appealed because it runs on the JVM and interoperates with Java...

#1

We need

Functional

Programming

. . .

Tuesday, July 20, 2010

First reason, we need the benefits of FP.

... for *concurrency*.

... for *concise* code.

... for *correctness*.

#2
We need a better
Object Model

• • •

... for *composability*.

... for *scalable designs*.

Scala's *Thesis:* Functional Programming *Complements* Object-Oriented Programming

Despite surface contradictions...

Tuesday, July 20, 2010

We think of objects as mutable and methods as state-modifying, which is often appropriate. But using functional idioms reduces bugs and often simplifies code. You can make your objects immutable, too!

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.

Tuesday, July 20, 2010

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.


Everything can
be a Function

Objects

as

Functions

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

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

makes "level" an immutable field

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

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

method

class body is the "primary" constructor

Tuesday, July 20, 2010

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

name : Type

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

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


Tuesday, July 20, 2010

Note how variables are declared, “name: Type”.

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

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

val error = new Logger(ERROR)

…
error(“Network error.”)
```

error’s type inferred

Tuesday, July 20, 2010

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 logger system
    log(level, message)
  }
}

val ew Logge(ERROR)

…
error(“Network error.”)
```

apply is called

“function object”

Tuesday, July 20, 2010

Adding a parameterized arg. list after an object causes the compiler to invoke the object’s “apply” method.

Put an arg list after any object, *apply* is called.

Tuesday, July 20, 2010

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...


Everything is
an Object

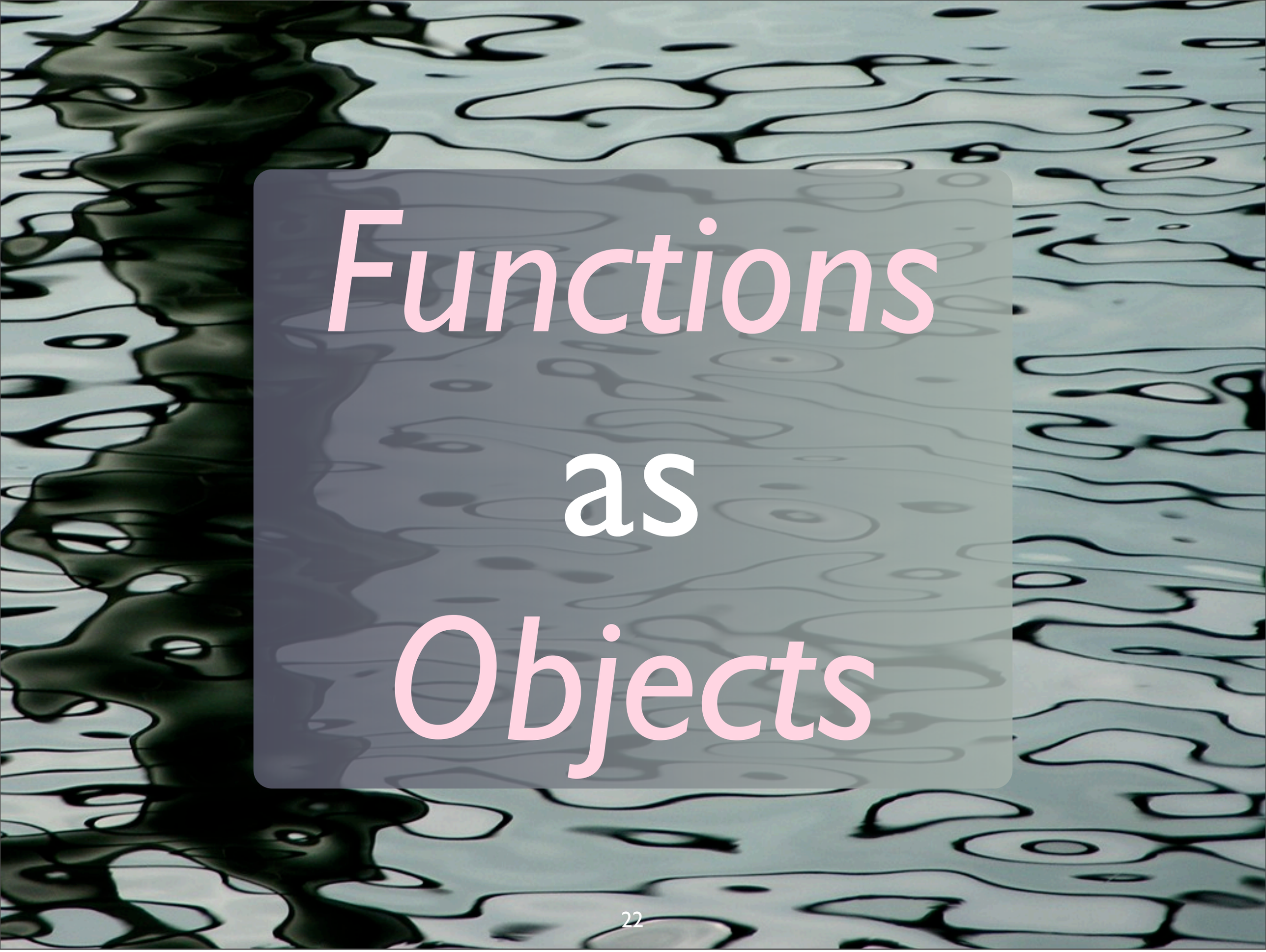
Int, Double, etc.

are true *objects*.

But they are compiled to primitives.

Tuesday, July 20, 2010

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...


Functions
as
Objects

First, About Lists

no new

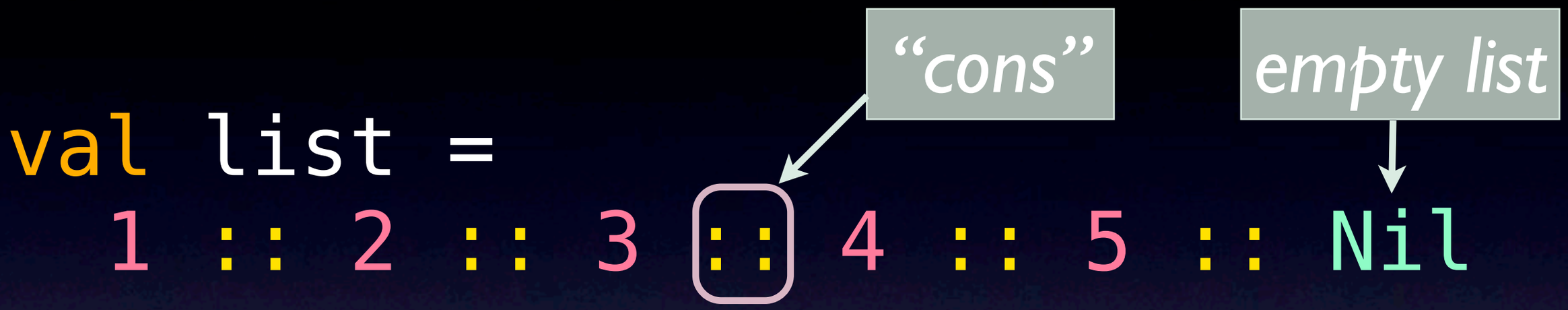
```
val list = List(1, 2, 3, 4, 5)
```

The same as this "list literal" syntax:

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


Tuesday, July 20, 2010

We build up a literal list with the "::" cons operator to prepend elements, starting with an empty list, the Nil "object". "::" binds to the right, so the second form shown is equivalent to the first. Note the "operator notation"; x.m(y) ==> x m y



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

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

“Operator” Notation

```
“hello” + “world”
```

is actually just

```
“hello”.+(“world”)
```


Tuesday, July 20, 2010

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

Similarly

```
“hello” compareTo “world”
```

is actually just

```
“hello”.compareTo(“world”)
```


Tuesday, July 20, 2010

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

Oh, and Maps

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


Maps also have a literal syntax. 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...

Tuesday, July 20, 2010

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

Oh, and Maps

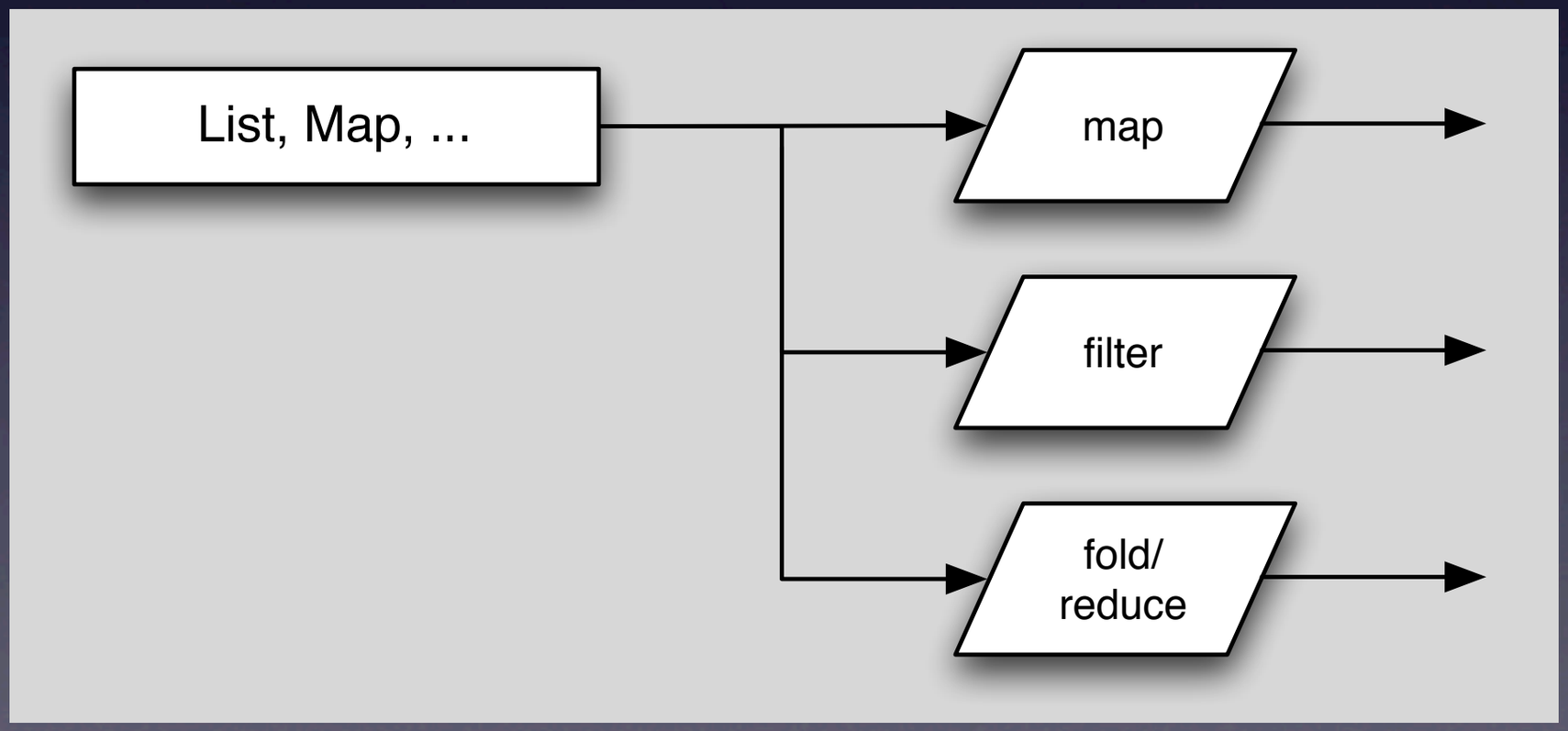
```
val map = Map(
  “name” -> “Dean”,
  “age”  -> 39)
```

An “implicit conversion” converts a string to a type that has the -> method.

Tuesday, July 20, 2010

We can’t discuss implicit conversions here, but see the extra slides which discuss how implicits work with and example of they might be used in a DSL.

Classic Operations on *Functional Data Types*



Collections like List and have a set of common operations that can be used on them.

Back to *functions* as *objects*...

```
val list = “a” :: “b” :: Nil

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

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


Tuesday, July 20, 2010

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

map called on list

map argument list

"function literal"

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

function argument list

function body

No () and ; needed!

Tuesday, July 20, 2010

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.

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

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

Explicit type

Tuesday, July 20, 2010

We've used type inference, but here's how we could be more explicit about the argument list to the function literal.

So far,
we've used
type inference
a lot...

How the Sausage Is Made

Parameterized type (like <A> in Java)

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

Declaration of map

The function argument

Tuesday, July 20, 2010

Here's the declaration of List's map method (lots of details omitted…).

How the Sausage Is Made

like an "abstract" class

```
trait Function1[A,R]
  extends AnyRef {
```

Java's "Object"

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

No method body: => abstract

Tuesday, July 20, 2010

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 important details…) The trait defines an abstract method "apply".
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
```

becomes:

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

No "return" needed

Compiler generates an anonymous class

Tuesday, July 20, 2010

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.

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

using a function value

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

list map {s => f(s)}
```

Alternative

Tuesday, July 20, 2010

Back to where we started. Note again that we can use “{…}” instead of “(…)” for the argument list (i.e., the single function) to map.

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

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


Tuesday, July 20, 2010

Back to where we started. Note again that we can use “{…}” instead of “(…)” for the argument list (i.e., the single function) to map.

Tuesday, July 20, 2010

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

Infix Operator Notation

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

same as

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

Great for DSLs!

Tuesday, July 20, 2010

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

Type *Inference*

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

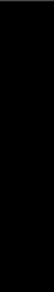
vs.

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


Tuesday, July 20, 2010

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

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



no () needed.
Semicolons inferred.

Tuesday, July 20, 2010

Other things aren't needed...

User-defined *Factory* Methods

```
val persons = Map (
  “dean” -> deanPerson,
  “alex” -> alexPerson)
```

no new needed.
Returns an appropriate subtype.

Factory methods on the cheap...

```
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

Tuesday, July 20, 2010

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

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

Typical Scala!

Tuesday, July 20, 2010

Scala is much more succinct. It eliminates a lot of boilerplate.

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!

Tuesday, July 20, 2010

Scala is much more succinct. It eliminates a lot of boilerplate.

Actually, not *exactly* the same:

```
val person = new Person(“dean”,…)
val fn = person.firstName
person.firstName = “Bubba”
// Not:
// val fn = person.getFirstName
// person.setFirstName(“Bubba”)
```

Doesn’t follow the JavaBean convention.

Tuesday, July 20, 2010

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!

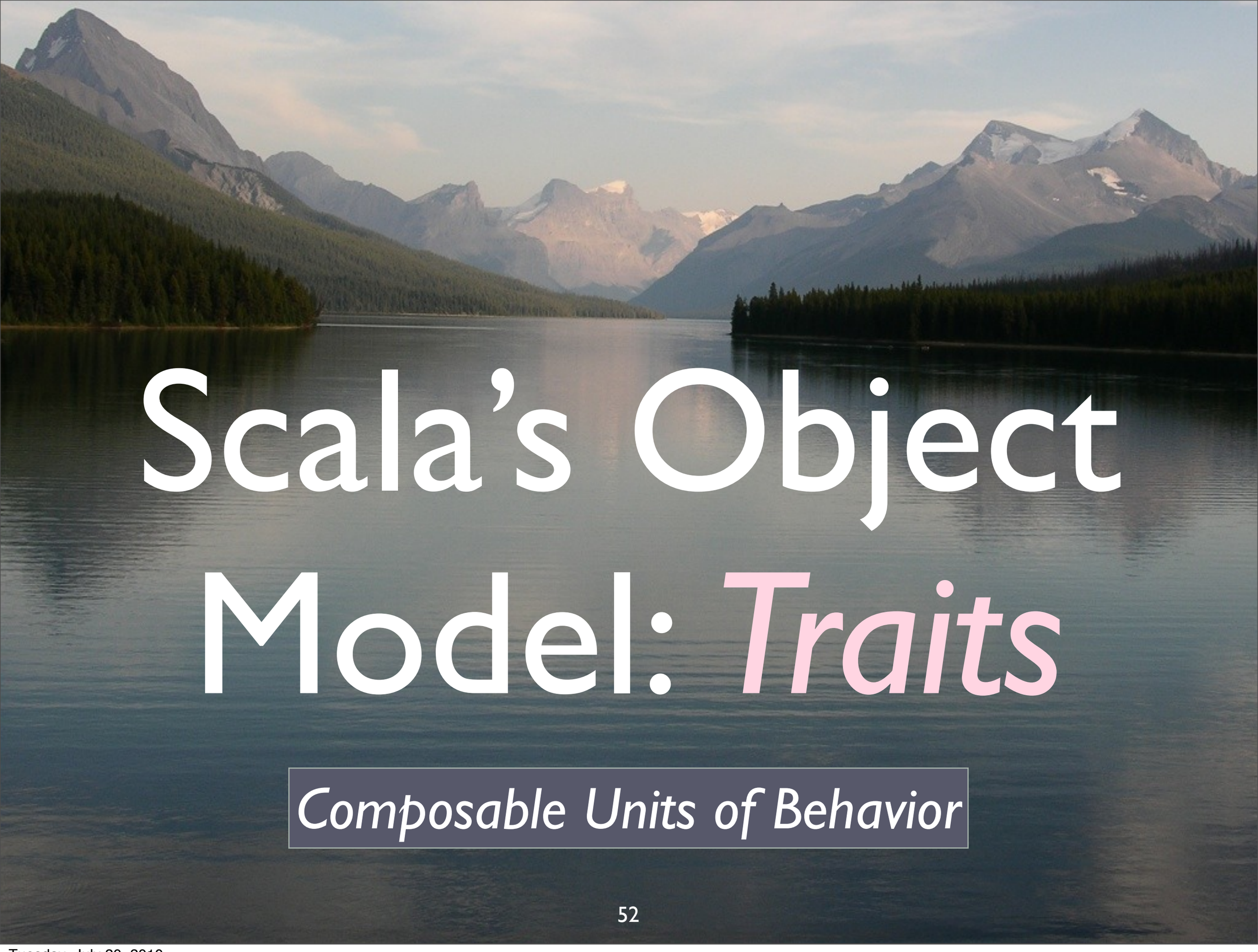
However, these *are* function calls:

```
class Person(fn: String, …) {
  // init val
  private var _firstName = fn

  def firstName = _firstName

  def firstName_=(fn: String) =
    _firstName = fn
}
```

Uniform Access Principle

Tuesday, July 20, 2010

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!



Tuesday, July 20, 2010

Fixes limitations of Java's object model.

Java

```
class Queue
  extends Collection
  implements Logging, Filtering
{ … }
```


Tuesday, July 20, 2010

Made-up example Java type.

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,

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

Traits

… or like

abstract classes +

multiple inheritance

(if you prefer).

… and another way.

Example

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

A pure abstraction (in this case...)

Log put and get

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


Tuesday, July 20, 2010

(“get” is similar.) “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 and get

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

What is "super" bound to??

Tuesday, July 20, 2010

(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".

```
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

Tuesday, July 20, 2010

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...

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

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

Example use

Tuesday, July 20, 2010

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

Example use

Tuesday, July 20, 2010

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

Like Aspect-Oriented Programming?

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

Tuesday, July 20, 2010

If you know AspectJ or Spring AOP, traits make it easy to implement "advice", but there is no join point language for querying over the set of all possible join points, like a real AOP framework provides.


Stackable
Traits

Filter put

```
trait QueueFiltering[T]
 extends Queue[T] {
  abstract override def put(
    t: T) = {
   if (veto(t))
     println(t+" rejected!")
   else
     super.put(t)
  }
  def veto(t: T): Boolean
}
```


Tuesday, July 20, 2010

Like QueueLogging, this trait can veto potential puts. Implementers/subclasses decide what “veto” means.

Filter put

```
trait QueueFiltering[T]
  extends Queue[T] {
  abstract override def put(
      t: T) = {
    if (veto(t))
      println(t+" rejected!")
    else
      super.put(t)
  }
  def veto(t: T): Boolean
}
```

"Veto" puts

Unlike QueueLogging, this trait can veto potential puts. Implementers/subclasses decide what "veto" means.

```
val sq = new StandardQueue[Int]
       with QueueLogging[Int]
       with QueueFiltering[Int] {
  def veto(t: Int) = t < 0
}
```

Defines "veto"

Anonymous Class

We instantiate StandardQueue AND mixin both traits. Note that we have to define veto for our current needs, in this case to prevent putting negative integers.

```
for (i <- -2 to 2) {
    sq.put(i)
}
println(sq)
// => -2 rejected!
// => -1 rejected!
// => put(0)
// => put(1)
// => put(2)
// => 0, 1, 2
```

loop from -2 to 2

Filtering occurred before logging

Example use

Tuesday, July 20, 2010

The filter trait's "put" is invoked before the logging trait's put.

What if we
reverse the *order*
of the Traits?

```
val sq = new StandardQueue[Int]
       with QueueFiltering[Int]
       with QueueLogging[Int]     {
  def veto(t: Int) = t < 0
}
```

Order switched

```
for (i <- -2 to 2) {
    sq.put(i)
}
println(sq)
// => put(-2)
// => -2 rejected!
// => put(-1)
// => -1 rejected!
// => put(0)
// => put(1)
// => put(2)
// => 0, 1, 2
```

logging comes before filtering!

Tuesday, July 20, 2010

Now, the logger trait's "put" is invoked before the filtering trait's put.

Loosely speaking,
the *precedence*
goes *right* to *left*.

"Linearization" algorithm

Tuesday, July 20, 2010

Method lookup algorithm is called “linearization”. For complex object graphs, it's a bit more complicated than "right to left".

Method Lookup Order

- Defined in object's *type*?
- Defined in *mixed-in traits*, *right to left*?
- Defined in *superclass*?

Simpler cases, only...

Tuesday, July 20, 2010

Can be more complex for complex hierarchies (but not that much more complex…). "Defined" also included "overridden".

Traits are also powerful for *mixin composition.*

Logger, revisited:

```
trait Logger {
 def log(level: Level,
         message: String) = {
  Log.log(level, message)
 }
}
val dean =
   new Person(…) extends Logger
dean.log(ERROR, “Bozo alert!!”)
```

mixed in Logging

Tuesday, July 20, 2010

I changed some details compared to our original Logger example, e.g., no “level” field. Mix in Logger.

Tuesday, July 20, 2010

Fixes limitations of Java's object model.

Building Our Own Controls

Exploiting First-Class Functions

Recall *infix* operator notation:

```
1 + 2          // => 3
1.+(2)         // => 3
```

also the same as

```
1 + {2}
```

Why is this useful??

Tuesday, July 20, 2010

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

Make your own *controls*

```
// Print with line numbers.

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

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


Tuesday, July 20, 2010

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) =>

  printf("%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:     printf("%3d: %s\n", …
8: }
```

```
import java.io._

object Loop {

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


Tuesday, July 20, 2010

Here’s the code that implements loop...

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

_ like * in Java

"singleton" class == 1 object

```
object Loop {
```

loop "control"

two parameters

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

function taking line # and line

like "void"

Tuesday, July 20, 2010

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

Tuesday, July 20, 2010

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

two parameters lists

Tuesday, July 20, 2010

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) =>
    printf("%3d: %s\n", n, line)
}
```

import new method

1st param.: a file

2nd parameter: a "function literal"

Tuesday, July 20, 2010

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(n:Int) = {…}
    doLoop(1)
  }
}
```

nested method

Finishing Loop.loop…

Tuesday, July 20, 2010

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 Loop.loop…

Tuesday, July 20, 2010

Here is the nested method, doLoop.

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)
}
```

Scala optimizes tail recursion into loops

Tuesday, July 20, 2010

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

Recap: Make a DSL

```
// Print with line numbers.
import Loop.loop

loop (new File("…")) {
  (n, line) =>

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


Tuesday, July 20, 2010

We've used some syntactic sugar (infix operator notation, substituting {…} for (…)) and higher-order functions to build a tiny DSL. Other features supporting DSLs include implicits

Tuesday, July 20, 2010

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

Avoiding Nulls

```
abstract class Option[T] {…}

case class Some[T](t: T)
  extends Option[T] {…}

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

Child of all other types

Tuesday, July 20, 2010

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](t: T)
```

Provides factory, pattern matching, equals, toString, and other goodies.

Tuesday, July 20, 2010

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].

```
class Map1[K, V] {
 def get(key: K): V = {
   return v;      // if found
   return null; // if not found
 }
}
class Map2[K, V] {
  def get(key: K): Option[V] = {
    return Some(v); // if found
    return None; // if not found
  }
}
```

Which is the better API?

Tuesday, July 20, 2010

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

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

Tuesday, July 20, 2010

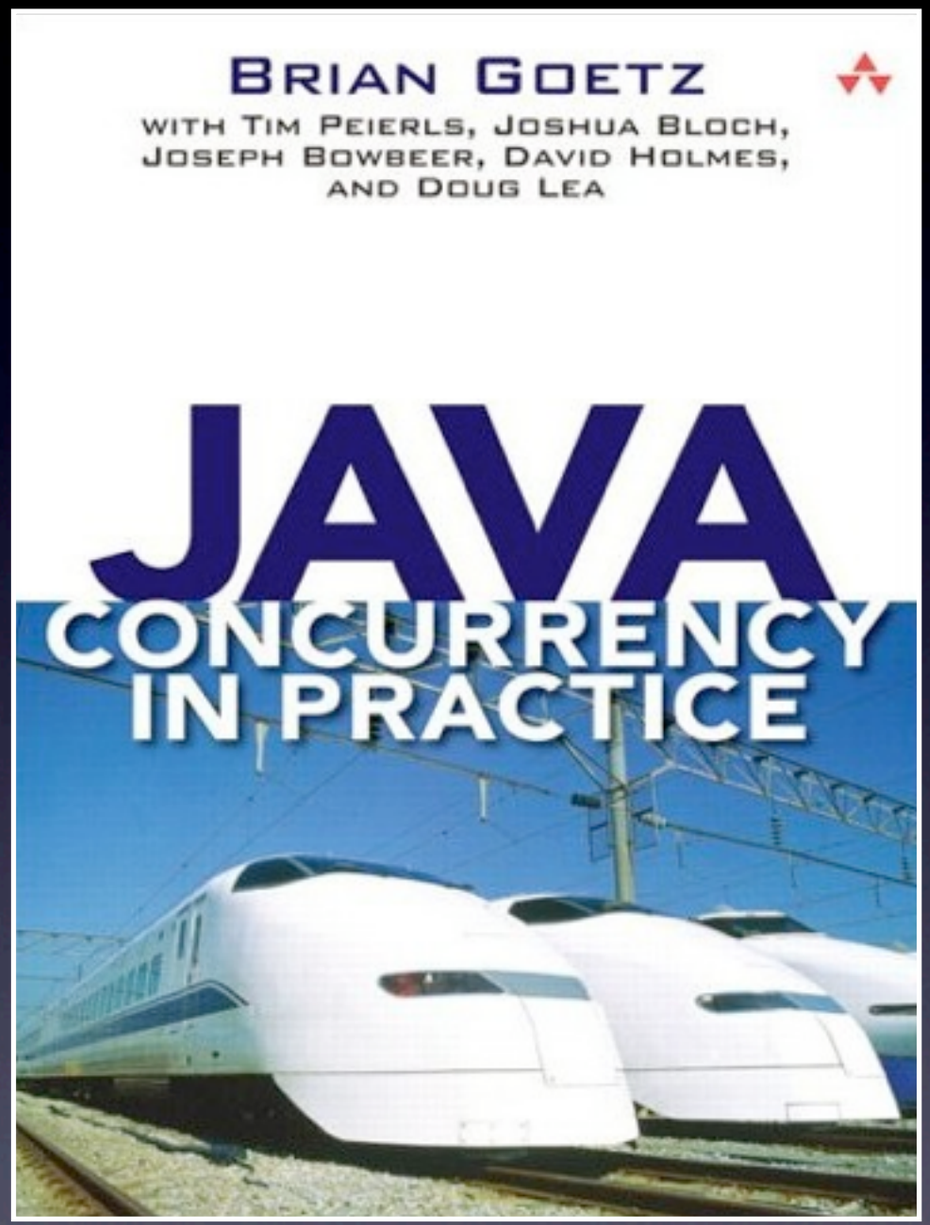
We’re using the type system and pattern matching built into case classes to discriminate elements in the list. No conditional statements required.
This is just the tip of the iceberg of what “for comprehensions” can do and not only with Options, but other containers, too.

Tuesday, July 20, 2010

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

When you share *mutable* state...

Hic sunt dracones
(Here be dragons)



Hard!

Tuesday, July 20, 2010

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

Actor Model

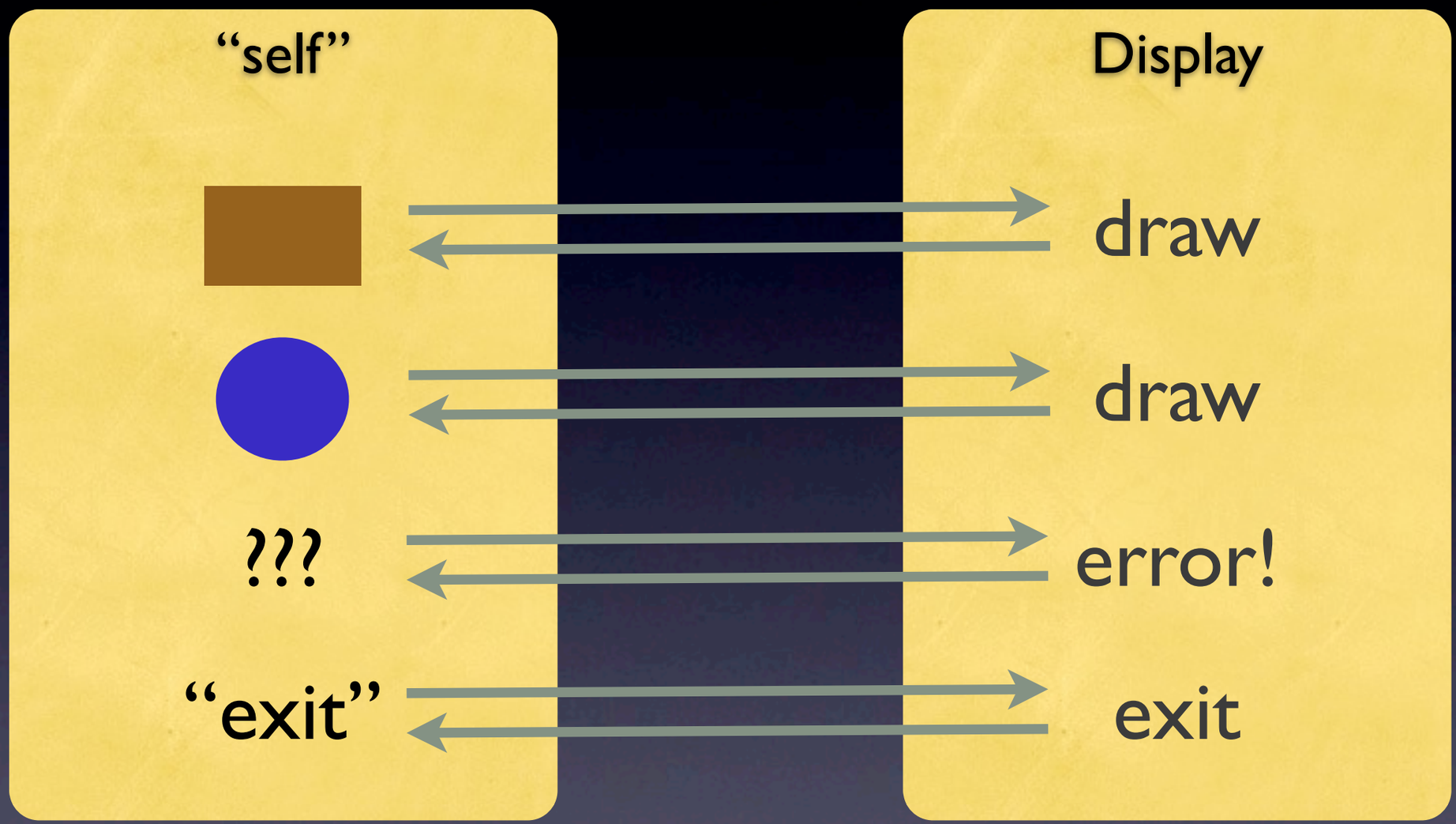
- *Message* passing between autonomous *actors*.
- No *shared* (mutable) *state*.

Actor Model

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

The actor model is not new!!

2 Actors:



Our example

```
package shapes

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

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

Hierarchy of geometric shapes

“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.)

```
package shapes

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

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

abstract "draw" method

Hierarchy of geometric shapes

Tuesday, July 20, 2010

"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.)

```
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() = …
}
```

Hierarchy of geometric shapes

Tuesday, July 20, 2010

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(this.toString)".

```
package shapes
import scala.actors._, Actor._
object ShapeDrawingActor
         extends Actor {
  def act() {
    loop {
      receive {
        ...
      }
    }
  }
}
```

Actor for drawing shapes

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...

```
package shapes
import scala.actors._, Actor._
object ShapeDrawingActor
        extends Actor {
  def act() {
    loop {
      receive {
        ...
      }
    }
  }
}
```

Actor library

"singleton" Actor

loop indefinitely

receive and handle each message

Actor for drawing shapes

Tuesday, July 20, 2010

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...

Receive method

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


Tuesday, July 20, 2010

“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 =>
    s.draw()
    sender ! "drawn"
 case "exit" =>
    println("exiting...")
    sender ! "bye!"
    // exit
 case x =>
    println("Error: " + x)
    sender ! ("Unknown: " + x)
}
```

pattern matching

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.

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

draw shape & send reply

done

unrecognized message

After handling each message, a response is sent to the sender, which we get by calling the “sender” method. The “!” is the message send method (from Erlang).

```
package shapes
import …
object ShapeDrawingActor extends Actor {
  def act() {
    loop {
      receive {
        case s: Shape =>
          s.draw()
          sender ! "drawn"
        case "exit" =>
          println("exiting...")
          sender ! "bye!" //; exit
      case x =>
        println("Error: " + x)
        sender ! ("Unknown: " + x)
} } } }
```

Altogether

Tuesday, July 20, 2010

The whole thing, with “elided” imports and other edits, so it would fit.

```
import shapes._
import scala.actors.Actor._


def sendAndReceive(msg: Any) ={
 ShapeDrawingActor ! msg
```

```
  self.receive {
    case reply => println(reply)
  }
}
```

script to try it out

Tuesday, July 20, 2010

Here's a scala script (precompilation not required) to drive the drawing actor.
Normally, you would not do such synchronous call and response coding...

```
import shapes._
import scala.actors.Actor._
```

helper method

parent of AnyRef, AnyVal

```
def sendAndReceive(msg: Any) ={
 ShapeDrawingActor ! msg
```

send message...

wait for a reply

```
  self.receive {
    case reply => println(reply)
  }
}
```

script to try it out

Tuesday, July 20, 2010

The “!” method sends a message. Then we wait for a reply. In our receive method, we match on any “reply” and just print it.
This script uses the method “self” to get the actor corresponding to “me”.

```
…
ShapeDrawingActor.start()
sendAndReceive(
  Circle(Point(0.0,0.0), 1.0))
sendAndReceive(
  Rectangle(Point(0.0,0.0), 2, 5))
sendAndReceive(3.14159)
sendAndReceive("exit")

// => Circle(Point(0.0,0.0),1.0)
// => drawn.
// => Rectangle(Point(0.0,0.0),2.0,5.0)
// => drawn.
// => Error: 3.14159
// => Unknown message: 3.14159
// => exiting...
// => bye!
```


Tuesday, July 20, 2010

Start the drawing actor, then send it four messages. The blue shows what gets printed (after the “// => “).

```
…
ShapeDrawingActor.start()
sendAndReceive(
  Circle(Point(0.0,0.0), 1.0))
sendAndReceive(
  Rectangle(Point(0.0,0.0), 2, 5))
sendAndReceive(3.14159)
sendAndReceive("exit")

// => Circle(Point(0.0,0.0),1.0)
// => drawn.
// => Rectangle(Point(0.0,0.0),2.0,5.0)
// => drawn.
// => Error: 3.14159
// => Unknown message: 3.14159
// => exiting...
// => bye!
```

```
…
receive {
  case s:Shape =>
    s.draw()
    sender ! "drawn"

  case …
  case …
}
```

Functional style pattern matching

Object-oriented polymorphism

A powerful combination!

Tuesday, July 20, 2010

The power of combining the best features of FP and OOP.

Recap

Scala is...

a *better*

Java and C#,

object-oriented

and

functional,

succinct,

elegant,

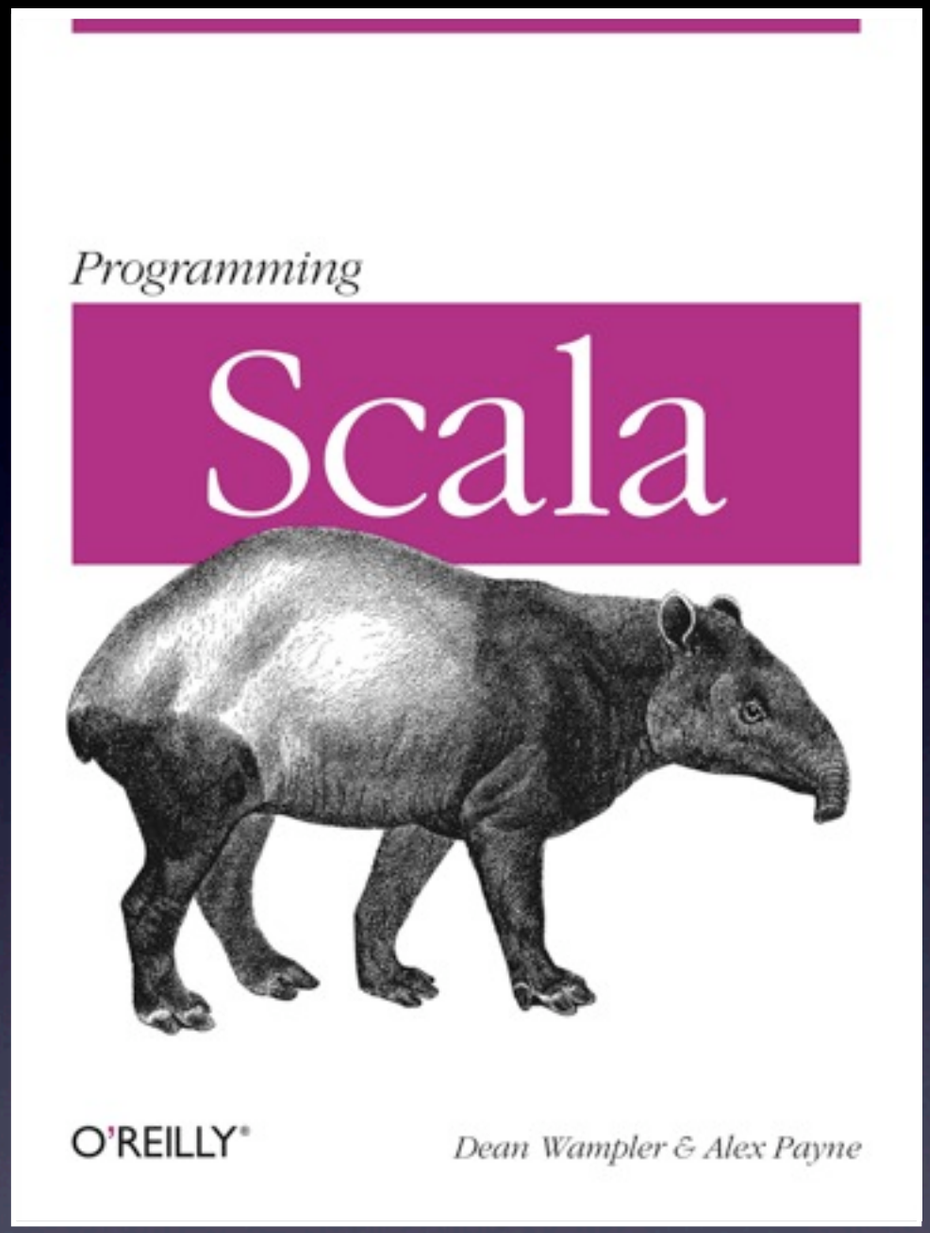
and

powerful.

Thanks!

dean@deanwampler.com
@deanwampler

polyglotprogramming.com/talks
programmingscala.com



Extra Slides


Functional
Programming

What is *Functional Programming*?

Don't we already write "functions"?

$$y = \sin(x)$$

Based on *Mathematics*

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

$$y = \sin(x)$$

Setting *x* fixes *y*

∴ *variables* are *immutable*

Tuesday, July 20, 2010

"Functional Programming" is based on the behavior of mathematical functions.
"Variables" are actually immutable values.

20 += 1 ??

We never *modify*
the 20 "object"

Concurrency

No *mutable state*

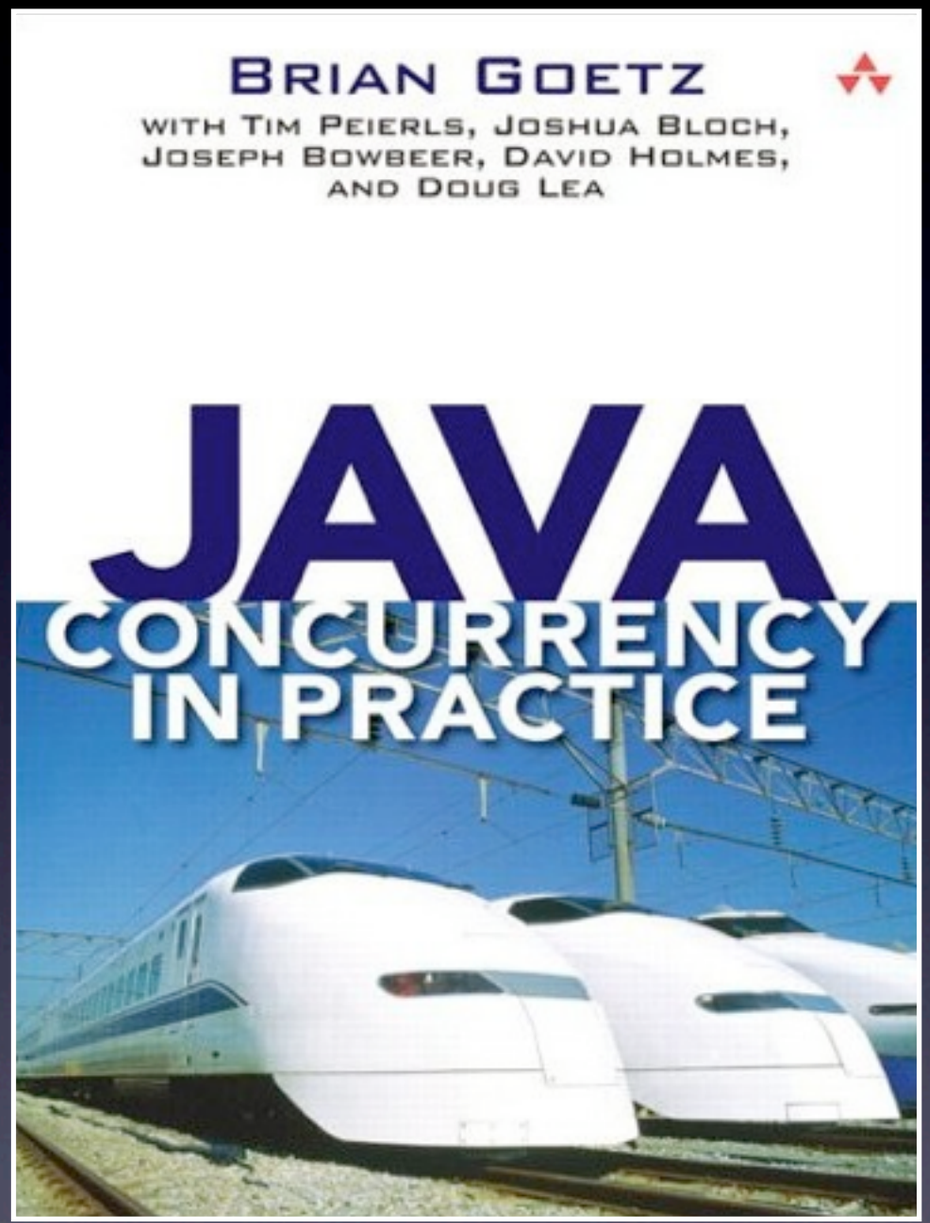
∴ *nothing* to *synchronize*

Tuesday, July 20, 2010

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

When you share *mutable* state...

Hic sunt dracones
(Here be dragons)



$$y = \sin(x)$$

Functions don't

change state

$\therefore$ *side-effect* free

Tuesday, July 20, 2010

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

Side-effect free functions

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

Tuesday, July 20, 2010

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

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

Compose functions of other functions

$\therefore$ *first-class citizens*

Tuesday, July 20, 2010

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

Tuesday, July 20, 2010

Fixes limitations of Java's object model.

"Pimp My Library"

Typesafe "monkey patching"

Tuesday, July 20, 2010

A funny name for a powerful feature, simulating open types (i.e., adding behavior to types) in a type-safe way.

Recall our "Loop"

```
// Print with line numbers.
import Loop.loop

loop (new File("…")) {
  (n, line) =>

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


Tuesday, July 20, 2010

We defined this earlier.

Suppose we want a *loop* method on *File* instead?

Now a method on File?

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


Tuesday, July 20, 2010

What if we would prefer for File to have a loop method instead? Is this possible in Scala?

Ruby

```
class File; …; end
file = File.new …

def file.loop
  n = 0
  while line = self.gets
    yield n, line
    n += 1
  end
end
```

Add a method to the object!

Open Types

Tuesday, July 20, 2010

In languages with open types, like Ruby, we can easily add new methods to types or individual objects. Can’t do that in Scala, right?

Implicits

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

```
object WithLoop {
 implicit def file2WithLoop(
     file: File) =
   new WithLoop (file)
}
```


Tuesday, July 20, 2010

You can't do that in Scala, but "implicits" let you define a conversion from the type you have to a new type that has the method you want. The syntax you use looks as if you have the method on the original type. The compiler finds the best-matching implicit conversion method in scope and applies it for you.
The implicit function will be used by the compiler to convert a File to a WithLoop, then we can call the loop method, which now has only one argument, the function to apply to each line in the file.
Technical note. Because of closures, we don't actually need to make the File a field (val) of the class. Also, it's okay for the object and class to have the same names; they are called "companions".

Using the Implicit

```
// Print with line numbers.
import WithLoop._

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

import required!

Tuesday, July 20, 2010

The compiler sees you try to call loop() on a File object, but that method doesn't exist. So, it looks for another type that has the loop method and an implicit conversion method in scope to convert from File to the other type. It then calls loop on the new object.

Implicits can be used to mimic *Haskell*-like *type classes*.

http://debasishg.blogspot.com/2010/06/scala-implicits-type-classes-here-i.html