

The Seductions of Scala

Dean Wampler

dean@deanwampler.com

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

polyglotprogramming.com/talks

April 3, 2011



|

Wednesday, November 28, 12

The online version contains more material. You can also find this talk and the code used for many of the examples at github.com/deanwampler/Presentations/tree/master/SeductionsOfScala

Copyright © 2010–2013, Dean Wampler. Some Rights Reserved – All use of the photographs and image backgrounds are by written permission only. The content is free to reuse, but attribution is requested.

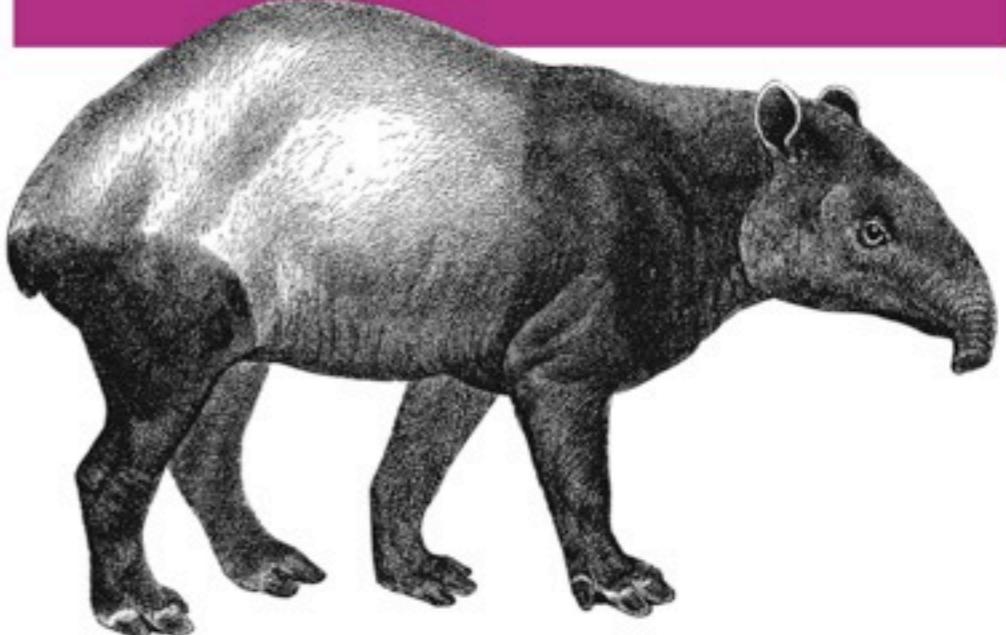
<shameless-plug/>

Co-author,
Programming
Scala

programmingscala.com

Programming

Scala



O'REILLY®

Dean Wampler & Alex Payne

Why do we need a new language?

3

Wednesday, November 28, 12

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.

#I We need *Functional* Programming

...

... for concurrency.
... for concise code.
... for correctness.

#2

We need a better
Object Model

...

... for composability.
... for scalable designs.

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

Despite surface contradictions...

But we want to
keep our *investment*
in *Java/C#*.

Scala is...

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

Martin Odersky

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

11

Wednesday, November 28, 12

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.

Useful References

- scala-lang.org
- programmingscala.com
- geekontheloose.com/wp-content/uploads/2010/02/Scala_Cheatsheet.pdf

A wide-angle photograph of a serene lake nestled in a mountainous region. The foreground is dominated by the dark, calm water of the lake. In the middle ground, two small, densely forested islands are visible, one on each side of the frame. The background features a majestic range of mountains, their peaks partially obscured by a hazy, warm-toned sky. The overall atmosphere is peaceful and natural.

Everything can
be a *Function*

14

Wednesday, November 28, 12

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

Objects as Functions

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

makes “level” a field

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

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

method

*class body is the
“primary” constructor*

```
class Logger(val level:Level) {  
  
    def apply(message: String) = {  
        // pass to Log4J...  
        Log4J.log(level, message)  
    }  
}  
  
val error = new Logger(ERROR)  
  
...  
error("Network error.")
```

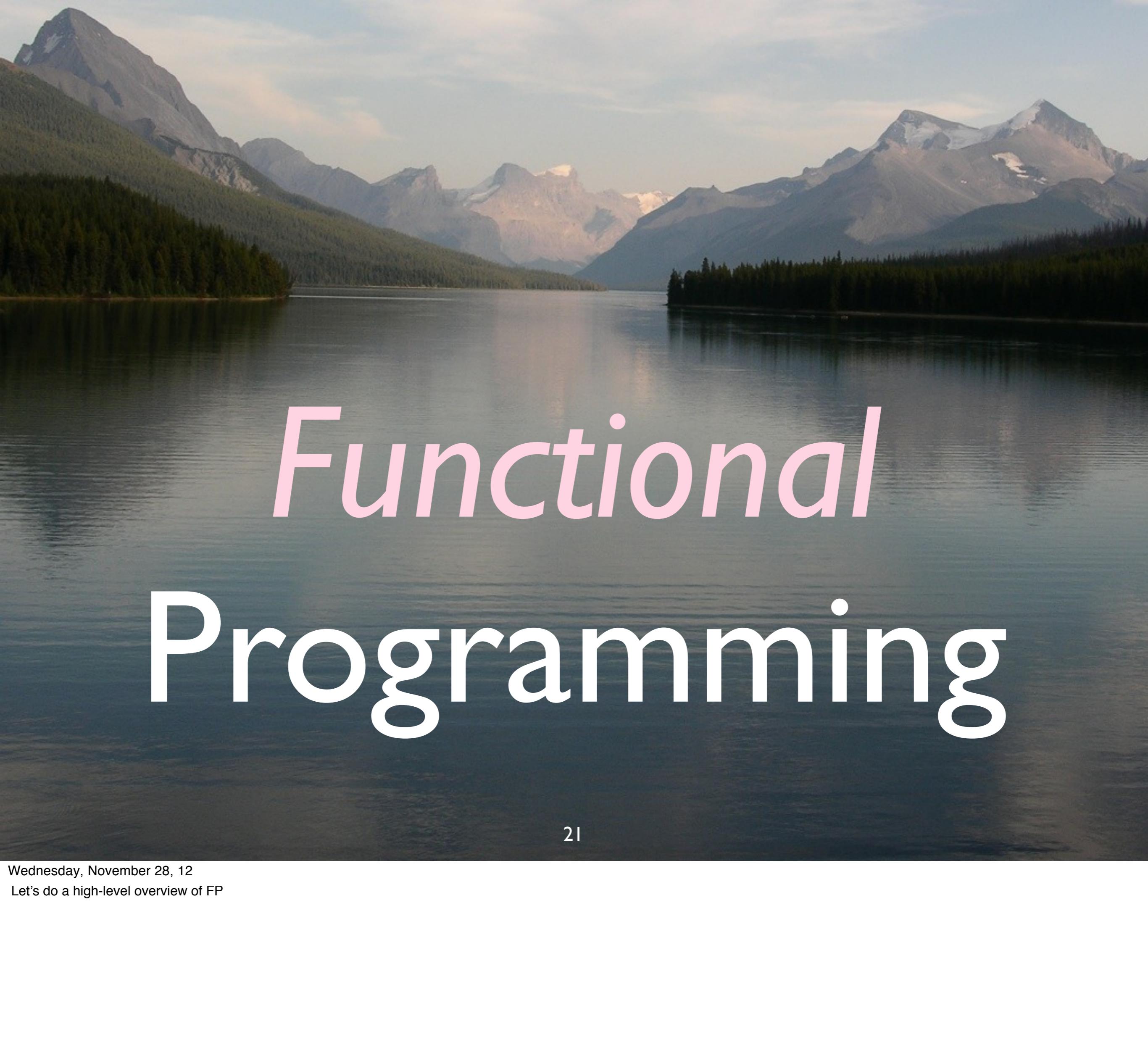
```
class Logger(val level:Level) {  
  
    def apply(message: String) = {  
        // pass to Log4J...  
        Log4J.log(level, message)  
    }  
}
```

apply is called

“function object”

...
error("Network error.")

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

The background of the slide features a serene landscape of a lake nestled among majestic mountains. The sky is a soft, warm orange and yellow, suggesting either sunrise or sunset. The water of the lake is calm, reflecting the surrounding environment. In the foreground, there's a dark, semi-transparent overlay where the title text is placed.

Functional Programming

21

Wednesday, November 28, 12

Let's do a high-level overview of FP

What is *Functional* Programming?

Don't we already write “functions”?

$y = \sin(x)$

Based on *Mathematics*

$$y = \sin(x)$$

Setting x fixes y

\therefore variables are *immutable*

$$20 + = | ??$$

We never modify
the 20 “object”

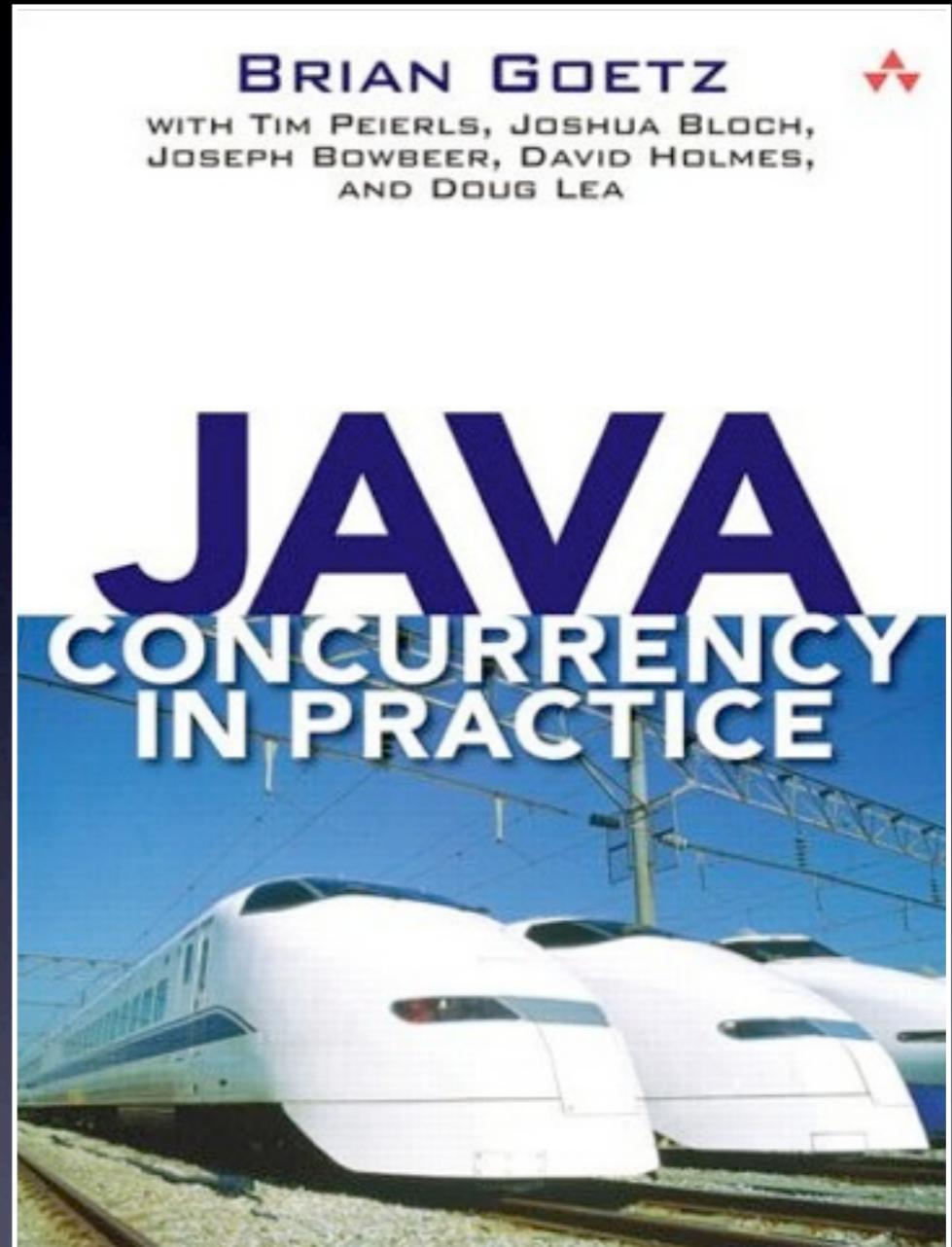
Concurrency

No *mutable* state

\therefore *nothing* to synchronize

When you share mutable state...

Hic sunt dracones
(Here be dragons)



$$y = \sin(x)$$

Functions don't
change state
∴ *side-effect* free

Other benefits of Side-effect free functions

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

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

Compose functions of
other functions

\therefore first-class citizens

Immutable values

*Safer concurrency;
nothing to synchronize!*

Immutable values

Safer to share with
clients;
nothing to screw up!

Immutable values

*But making copies has
high overhead.*

Immutable values

∴ use structural sharing!

Recursion

35

Wednesday, November 28, 12

Recursion is a core concept in FP. In pure FP, where loop counters would be forbidding, recursion is the main way to do iteration.

Fibonacci Sequence

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n > 1 \end{cases}$$

Fibonacci Sequence

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n > 1 \end{cases}$$

It is Recursive

Fibonacci Sequence

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n > 1 \end{cases}$$

It is also declarative, not imperative

Typing?

39

Wednesday, November 28, 12

Some FP languages really push static typing...

$$\tan(\Theta) = \sin(\Theta)/\cos(\Theta)$$
$$y = \exp(pi^*x)$$

*Properties of Types
are very important*

Contrasting Approaches

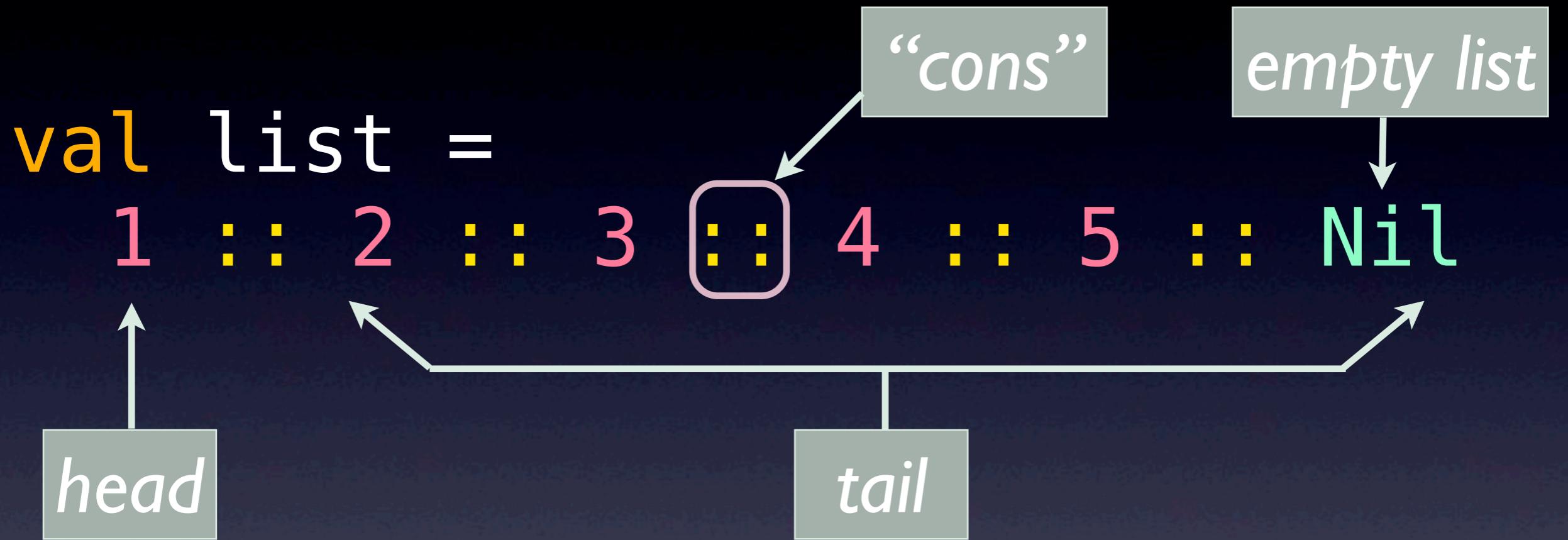
- FP
 - Get the *types right*
- OOP
- Flesh out correctness with *TDD*.

Lists

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

The same as this “list literal” syntax:

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



Baked into the Grammar?

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

No, just method calls!

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

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

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

Method names can contain almost any character. Exclusions include ., () [] Characters like _ # can't be used alone.

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

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

Any method ending in “::” binds to the right!

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

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

If a method takes one argument, you can drop the “.” and the parentheses, “(“ and “)”.

Infix Operator Notation

"hello" + "world"

is actually just

"hello".+("world")

Similar mini-DSLs
have been defined
for other types.

Also in many third-party libraries.

Oh, and Maps

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

Oh, and Maps

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

“baked” into the
language grammar?

No, just method calls...

Oh, and Maps

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

*What we like
to write:*

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

*What Map.apply()
actually wants:*

Oh, and Maps

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

*What we like
to write:*

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

*What Map.apply()
actually wants:*

*More succinct
syntax for Tuples*

We need to get from this,

"name" -> "Dean"

to this,

Tuple2("name", "Dean")

There is no String-> method!

Implicit Conversions

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

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

v2.10 Implicit Classes

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

implicit

Put the keyword on the class and the compiler generates the implicit method for you

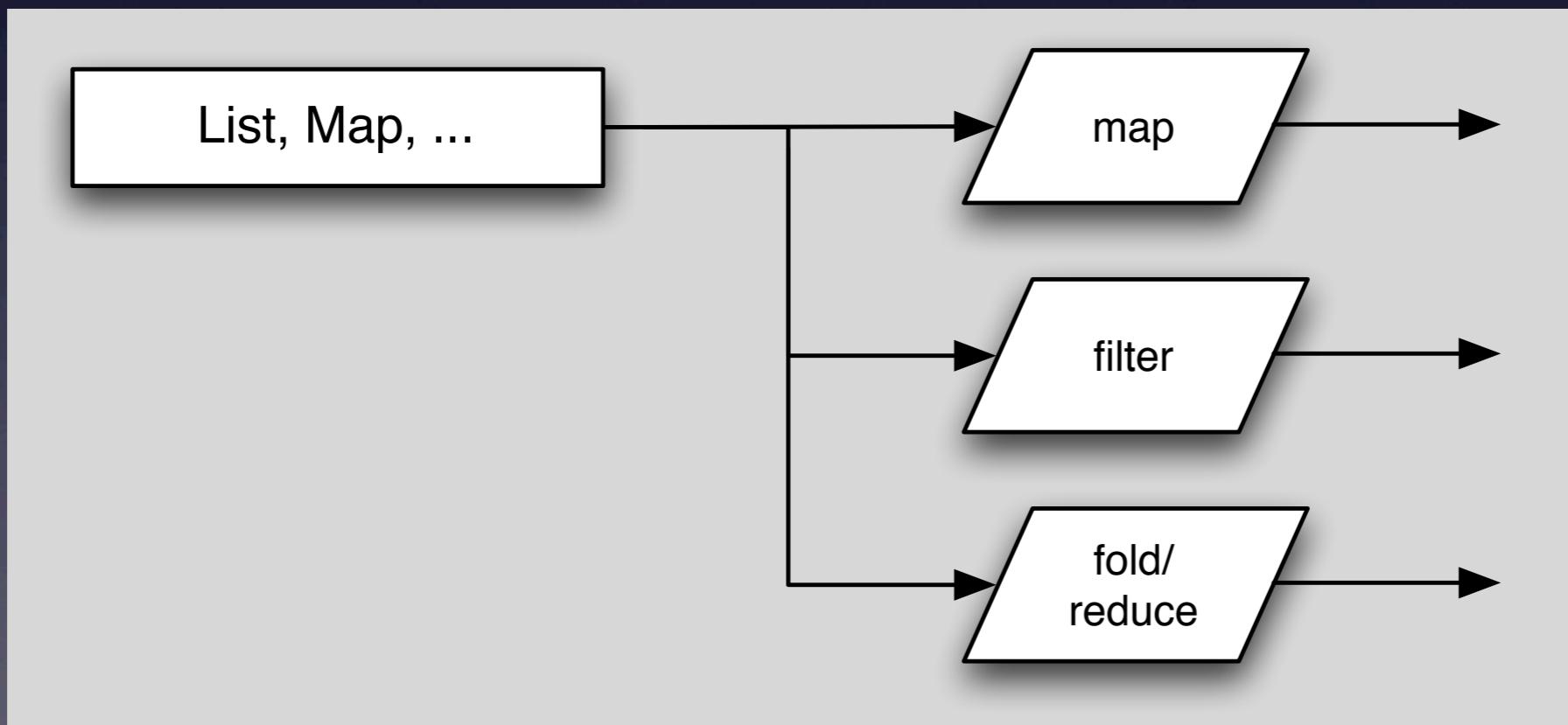
Back to Maps

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

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

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

Classic Operations on “Container” Types



Exercise

Implicit conversion methods.



Add a **toXML** method to Map

```
val map = Map(  
    "name" -> "Dean",  
    "age"  -> 39)  
println(map.toXML)  
// <map>  
//   <name>Dean</name>  
//   <age>39</age>  
// </map>
```

Exercise

Objects as functions.



Convert toXML to an Object

- Refer back to the early slides where we made a `Logger` object with `apply`.
- Create a `ToXML` object that implements the same `toXML` logic.

```
class ToXML {  
  def apply(map: Map... ) = {...}  
}  
  
val toXML = new ToXML  
println(toXML(map))
```

62

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

Everything is an Object

63

Wednesday, November 28, 12

While an object can be a function, every “bare” function is actually an object, both because this is part of the “theme” of scala’s unification of OOP and FP, but practically, because the JVM requires everything to be an object!

Int, Double, etc.
are true objects.

But they are compiled to primitives.

Functions as Objects

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

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

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

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

argument to *map*
(using "{} vs. "()")

list map {
 s => s.toUpperCase}

}

function
argument list

function body

"function literal"

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


inferred type


```

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


Explicit type


```

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

How the Sausage Is Made

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

Parameterized type

Declaration of `map`

The function argument

`map`'s return type

The diagram illustrates the annotations for the List class and its map method. It shows the class definition with annotations for the parameterized type, the declaration of the map method, the function argument, and the return type.

How the Sausage Is Made

like an “abstract” class

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

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

```
  ...  
}
```

No method body:
=> abstract

“contravariant”,
“covariant” typing

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

Recap

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

{...} ok, instead of (...)

Function “object”

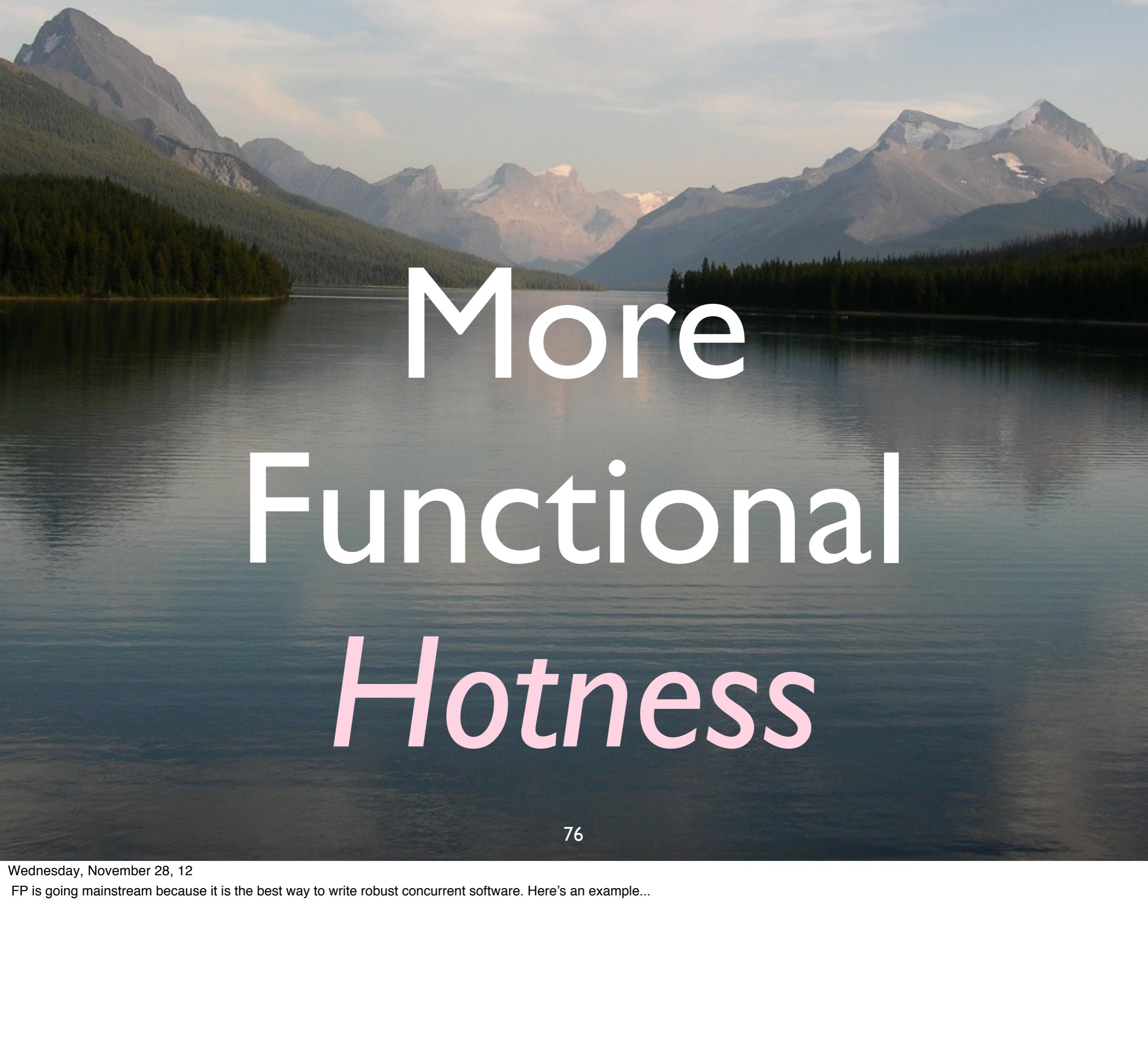
Since *functions*
are *objects*,
they could have
mutable state.

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

75

Wednesday, November 28, 12

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

A scenic landscape featuring a calm lake in the foreground, framed by a dense forest of evergreen trees. In the background, a range of majestic mountains is visible under a clear sky, with the warm light of sunset casting a golden glow over the scene.

More Functional *Hotness*

76

Wednesday, November 28, 12

FP is 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] {...}

An Algebraic Data Type

77

Wednesday, November 28, 12

I am omitting MANY details. You can't instantiate Option, which is an abstraction for a container/collection with 0 or 1 item. If you have one, it is in a Some, which must be a class, since it has an instance field, the item. However, None, used when there are 0 items, can be a singleton object, because it has no state! Note that type parameter for the parent Option. In the type system, Nothing is a subclass of all other types, so it substitutes for instances of all other types. This combined with a property called covariant subtyping means that you could write "val x: Option[String] = None" and it would type correctly, as None (and Option[Nothing]) is a subtype of Option[String]. Note that Options[+T] is only covariant in T because of the "+" in front of the T.

Also, Option is an algebraic data type, and now you know the scala idiom for defining one.

```
// Java style (schematic)
class Map[K, V] {
    def get(key: K): V = {
        return value || null;
    }
}
```

```
// Scala style
class Map[K, V] {
    def get(key: K): Option[V] = {
        return Some(value) || None;
    }
}
```

Which is the better API?

In Use:

```
val m = Literal syntax for map creation  
Map("one" -> 1, "two" -> 2)
```

```
...  
val n = m.get("four") match {  
  case Some(i) => i  
  case None      => 0 // default  
}
```

Use pattern matching to extract the value (or not)

Option Details: sealed

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

*All children must be defined
in the same file*

Case Classes

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

- Provides factory method, pattern matching, equals, toString, etc.
- Makes “value” a field without **val** keyword.

Object

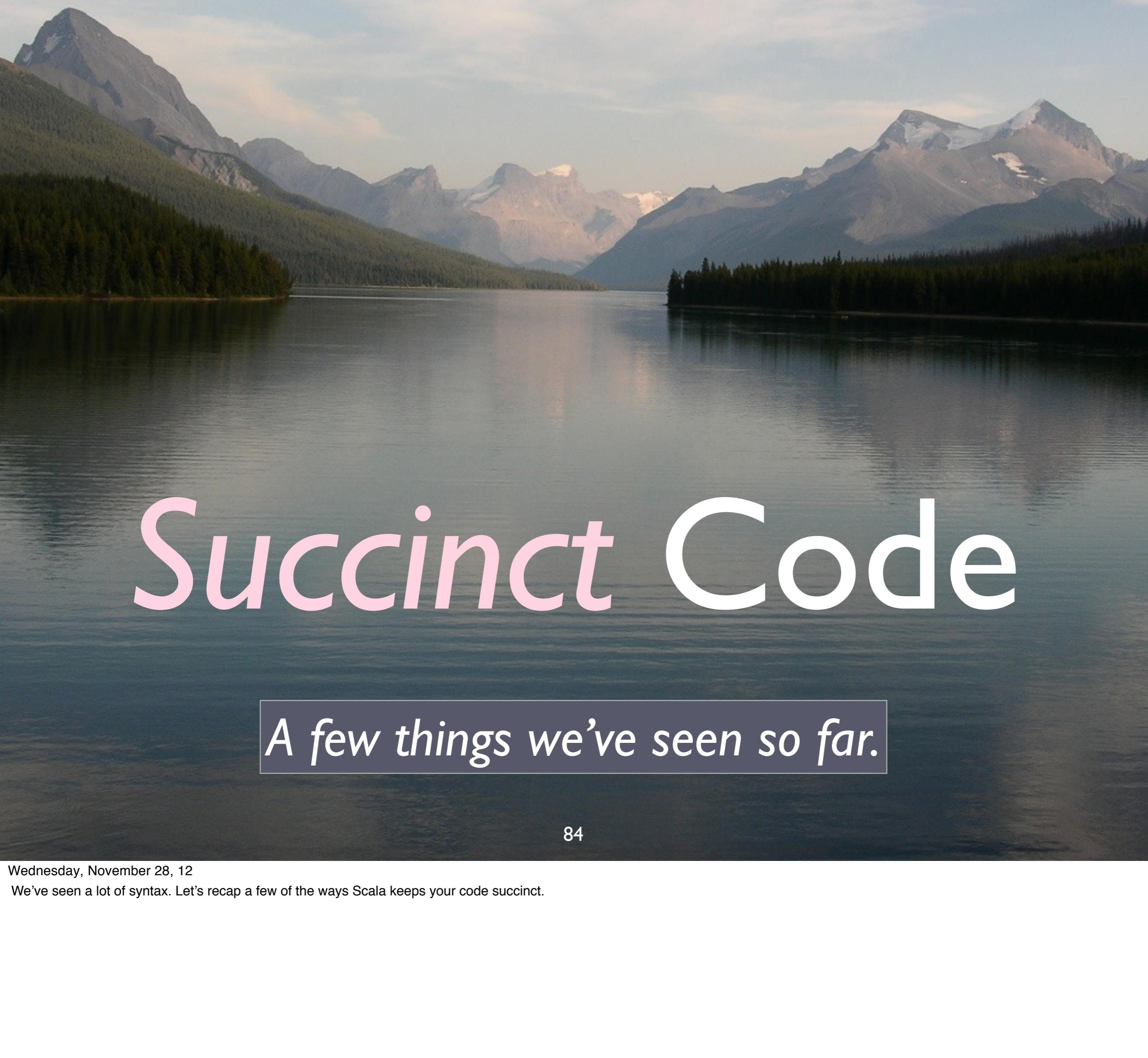
```
case object None  
extends Option[Nothing] {...}
```

A “singleton”. *Only one instance will exist.*

Nothing

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

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

The background of the slide is a photograph of a serene mountain lake. In the foreground, the dark water of the lake reflects the surrounding environment. On either side of the lake, there are forested hills. In the distance, a range of majestic mountains rises against a clear sky. The lighting suggests it might be either sunrise or sunset, with warm, golden light illuminating the peaks and the sky.

Succinct Code

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

Infix Operator Notation

"hello" + "world"

same as

"hello".+("world")

Type Inference

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

VS.

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

Type Inference

```
// Java
```

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

VS.

```
// Scala
```

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

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

↑
no () needed.
Semicolons inferred.

User-defined Factory Methods

```
val words =  
  List("Scala", "is", "fun!")
```

no **new** needed.
Can return a subtype!

```

class Person {
    private String firstName;
    private String lastName;
    private int    age;

    public Person(String firstName, String lastName, int age){
        this.firstName = firstName;
        this.lastName  = lastName;
        this.age       = age;
    }

    public void String getFirstName() {return this.firstName;}
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void String getLastname() {return this.lastName;}
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void int getAge() {return this.age;}
    public void setAge(int age) {
        this.age = age;
    }
}

```

Typical Java

90

Wednesday, November 28, 12

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!

*Class body is the
“primary” constructor*

Parameter list for c’tor

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

*Makes the arg a field
with accessors*

*No class body {...}.
nothing else needed!*

Actually, not exactly the same:

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

*Doesn't follow the
JavaBean convention.*

These *are* function calls

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

```
class Person(...) {  
    def firstName = fName  
}
```

or

```
class Person(...) {  
    var firstName = fn  
}
```

*Uniform Access
Principle*

```
val person = new Person(...)  
println(person.firstName)
```

Clients don't care which...

```
class Person(...) {  
    @scala.reflect.BeanProperty  
    var firstName = fn  
  
    ...  
}
```

If you need JavaBeans getters and setters...

Even Better...

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

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

97

Wednesday, November 28, 12

Case classes also get equals, hashCode, toString, and the “factory” method we’ve mentioned (e.g., for List and Map) that, by default, creates an instance of the type without the “new”. We’ll see more about case classes later.

Tuples

Fixed size, mixed types

```
class MyMap[A, B] {  
  def firstKeyValue: ??  
}
```

What should it return??

...

```
val key_val = map.firstKeyValue
```

```
class MyMap[A, B] {  
    def firstKeyValue: Tuple2[A, B]  
}  
...  
  
val key_val = map.firstKeyValue  
println(key_val._1 + "," +  
       key_val._2)
```

```
// type: Tuple2[Int, String]
val pair1 = Tuple2(123, "Dean")
// Same as:
val pair2 = (123, "Dean")
```



Literal syntax

Using the *REPL*
(even with Java)
is great for
experimenting:

102

```
$ scala  
Welcome to Scala version 2.8.1 ...
```

```
scala> "hello" + "world"  
res0: java.lang.String = helloworld
```

```
scala> "hello".+("world")  
res1: java.lang.String = helloworld
```

Exercise

Create an Address class.

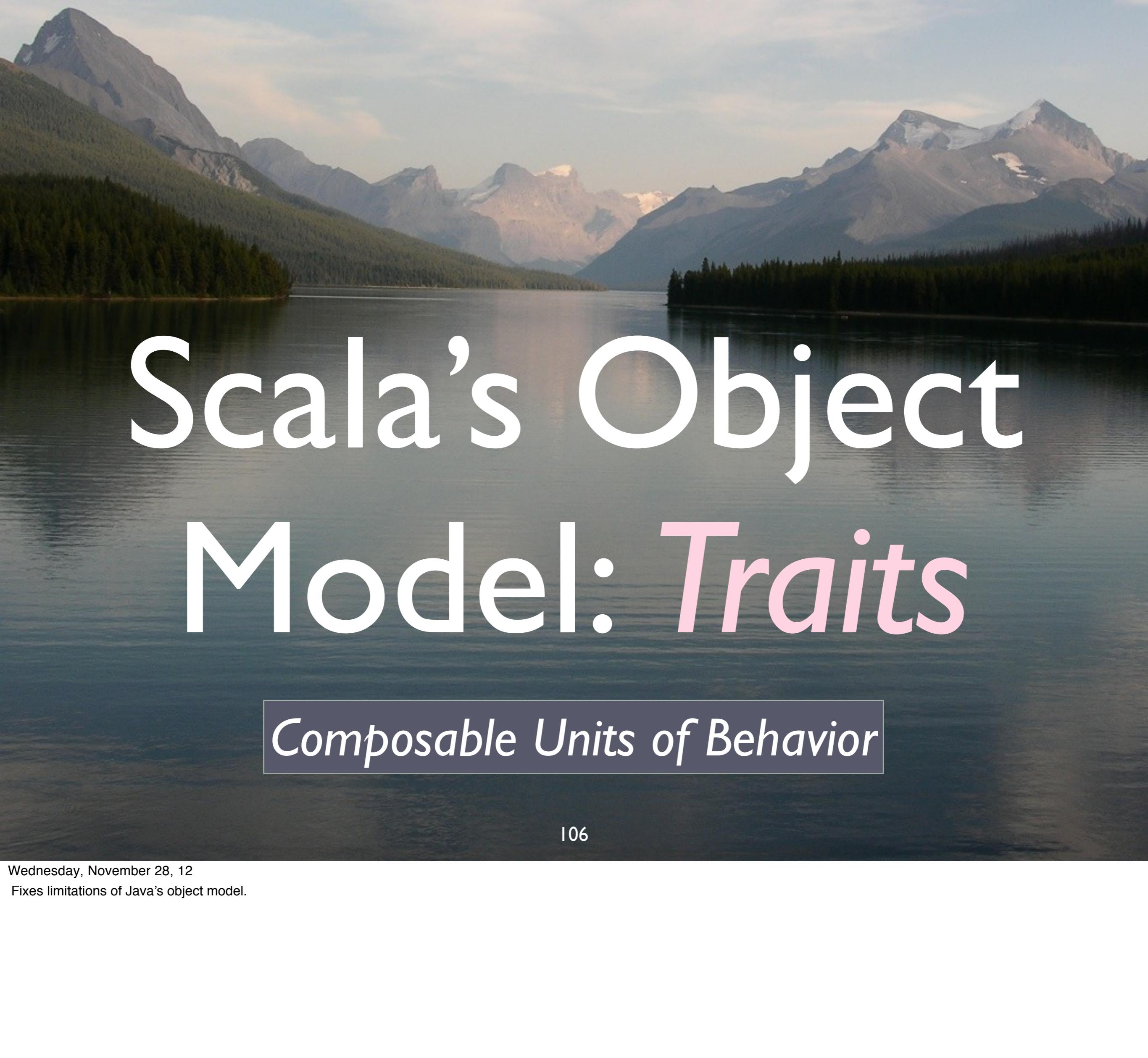


104

An Address Class

- Create an Address class
 - with *mutable* number, street, city, etc. fields.
 - Create instances, read-write fields, etc.
 - Make it *immutable*.
 - Create instances, read fields, etc.
 - Bonus: add **toString**.

105

The background of the slide features a wide-angle photograph of a mountainous landscape. In the foreground, there is a calm lake reflecting the light. The middle ground shows a dense forest of evergreen trees. In the background, a range of mountains is visible, with some peaks showing signs of snow or ice. The sky is overcast with soft, warm light from the setting sun.

Scala's Object Model: *Traits*

Composable Units of Behavior

106

Wednesday, November 28, 12

Fixes limitations of Java's object model.

Mixin Composition

107

We would like to
compose *objects*
from *mixins*.

108

Java: What to Do?

```
class Server  
  extends Logger { ... }
```

“Server is a Logger”?

```
class Server  
  implements Logger { ... }
```

Logger isn’t an interface!

Java's object model

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

Traits

Like interfaces with
implementations,

III

Traits

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

Logger as a Mixin:

```
trait Logger {  
    val level: Level // abstract  
  
    def log(message: String) = {  
        Log4J.log(level, message)  
    }  
}
```

Traits don't have constructors, but you can still define fields.

Logger as a Mixin:

```
trait Logger {  
    val level: Level // abstract  
    ...  
}  
  
val server =  
    new Server(...) with Logger {  
        val level = ERROR  
    }  
server.log("Internet down!!")
```

mixed in Logging

abstract member defined

Modifying Existing Behavior

Example

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

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

Log put

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

Log put

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

What is “super” bound to??

```
class StandardQueue[T]
    extends Queue[T] {
import ...ArrayBuffer
private val ab =
    new ArrayBuffer[T]
def put(t: T) = ab += t
def get() = ab.remove(0)
...
}
```

Concrete (boring) implementation

119

Wednesday, November 28, 12

Our concrete class. We import `scala.collection.mutable.ArrayBuffer` wherever we want, in this case, right where 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
println(sq.get())   // #2
// => put(10)      (on #1)
// => 10            (on #2)
```

Example use

120

Wednesday, November 28, 12

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

121

Wednesday, November 28, 12

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.

Stackable Traits

122

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
}
```

123

Wednesday, November 28, 12

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

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

Defines “veto”

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

loop from -2 to 2

Example use

126

Wednesday, November 28, 12

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

```
for (i <- -2 to 2) {  
    sq.put(i)  
}  
  
println(sq)
```

loop from -2 to 2

```
// => -2 rejected!  
// => -1 rejected!  
// => put(0)  
// => put(1)  
// => put(2)  
// => StandardQueue: ...
```

*Filtering occurred
before logging*

Example use

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)  
}
```

```
// => put(-2)  
// => -2 rejected!  
// => put(-1)  
// => -1 rejected!
```

```
// => put(0)  
// => put(1)  
// => put(2)
```

*logging comes
before filtering!*

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

“Linearization” algorithm

131

Wednesday, November 28, 12

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...

132

Note: traits can't have constructors.

Must initialize fields other ways.

133

Wednesday, November 28, 12

If the fields have a natural default value, use it. There is also the concept of abstract fields, like abstract methods - they are declared, but not defined (not supported in Java), but we won't cover that feature.

Traits Recap

134

Like Aspect-Oriented Programming?

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

*Traits let us compose
disjoint behaviors
and modify existing
behaviors.*

Exercise

Traits as Mixins



137

Traits

- Define a `QueueDoubling` trait that puts each new element *twice*.
- Use it with the other traits in different orderings. Do the results make sense to you?

What's the output?

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

Companion Objects

140

```
class Complex(val real: Double,  
             val imag: Double)  
{...}
```

The diagram illustrates the relationship between a class and its companion object. A box labeled "Singleton" has an arrow pointing to the class name "Complex". Another box labeled "Factory" method has an arrow pointing to the "apply" method in the companion object. The code shown is:

```
object Complex{  
    def apply(r:Double, i:Double) =  
        new Complex(r, i)}
```

```
...  
}
```

“Companions”

141

Wednesday, November 28, 12

Companion class and object: must have same name and be in the same file.
Companion object's “apply” method is usually used as a “factory”.

```
var c1 = Complex(1.1, 1.1)
val c2 = Complex(2.2, 2.2)
```

Calls “Complex.apply(...)”

142

Wednesday, November 28, 12

Must use “override” modifier when overriding a concrete method.

Case Classes

143

Recall:

```
class Complex(val real: Double,  
             val imag: Double)  
{...}
```

```
object Complex{  
    def apply(r:Double, i:Double) =  
        new Complex(r, i)  
}
```

This pattern for apply is so common...

Case class

```
case class Complex(  
    real: Double, imag: Double)  
{...}
```

With the **case** keyword, you get:

- *Arguments to primary constructor become fields.*
 - The **val** keyword is not required.
 - Object *equals*, *hashCode*, *toString*.
 - Based on the fields.
 - Companion object w/ factory **apply**.

Remember Map?

```
val map = Map(  
    k1 -> v1,  
    k2 -> v2)
```

```
object Map {  
    def apply[A, B](elems: (A, B) *)  
    ...  
}
```

0 to N Tuple2s

Exercise

Case Classes



| 48

Case Classes

- Convert your previous `Address` class to a **case class**.
- Create objects using `Address.apply`.
- Would a second **apply** method be useful?
- Play with the generated `toString`, `equals`, `hashCode` and field accessor methods.

Case classes
are very convenient
for “structural”
classes.

However
avoid inheriting
one case class
from another.

“`equals`”, “`hashCode`” don’t work properly

There is also an
unapply method...

*... and why is the keyword
called **case**?*

Pattern Matching

153

Wednesday, November 28, 12

Like conditionals/switch statements in imperative languages, but on steroids.

Recall this List syntax:

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

“cons”

empty list

The diagram illustrates the list construction syntax. It shows the definition of a variable 'list' with its value being a list of integers. The list is built using the 'cons' operator (indicated by the '::' symbol) to prepend each integer to the head of the remaining list. The final element is 'Nil', which is labeled as the 'empty list'. A pink box highlights the '::' operator, and arrows point from the words 'cons' and 'empty list' to their respective parts in the code.

Print a List

```
def lprint(l: List[_]): Unit =  
l match {  
  
  case head :: tail =>  
    print(head + ", ")  
    lprint(tail)  
  case Nil => // do nothing  
}
```

*Wildcard: match
any type*

```
def lprint(l: List[_]): Unit =  
l match {  
    case head :: tail =>  
        print(head + ", ")  
        lprint(tail)  
    case Nil => //nothing  
}
```

pattern match

Note recursive call

Return type
required

```
def lprint(l: List[_]): Unit =  
l match {  
  
  case head :: tail =>  
    print(head + ", ")  
    lprint(tail)  
  case Nil => //nothing  
}
```

```
val list =  
  1 :: 2 :: 3 :: 4 :: 5 :: Nil  
  
lprint(list)  
  
// => 1, 2, 3, 4, 5,
```

More Pattern Matching...

159

```
val list =  
  Complex(1,2) ::  
  Complex(1.1,2.2) ::  
  (1,2,3) :: (3,2,10) :: Nil
```

```
list foreach { _ match {  
  case Complex(1,i) => // #1  
  case Complex(r,i) => // #2  
  case (3,2,_ ) => // #3  
  case (a,b,c) => // #4  
  case Nil => // #5  
}} //=> #1, #2, #4, #3 (no Nil)
```

The extractors used
for `Complex` and
`Tuple3` are called
`unapply`.

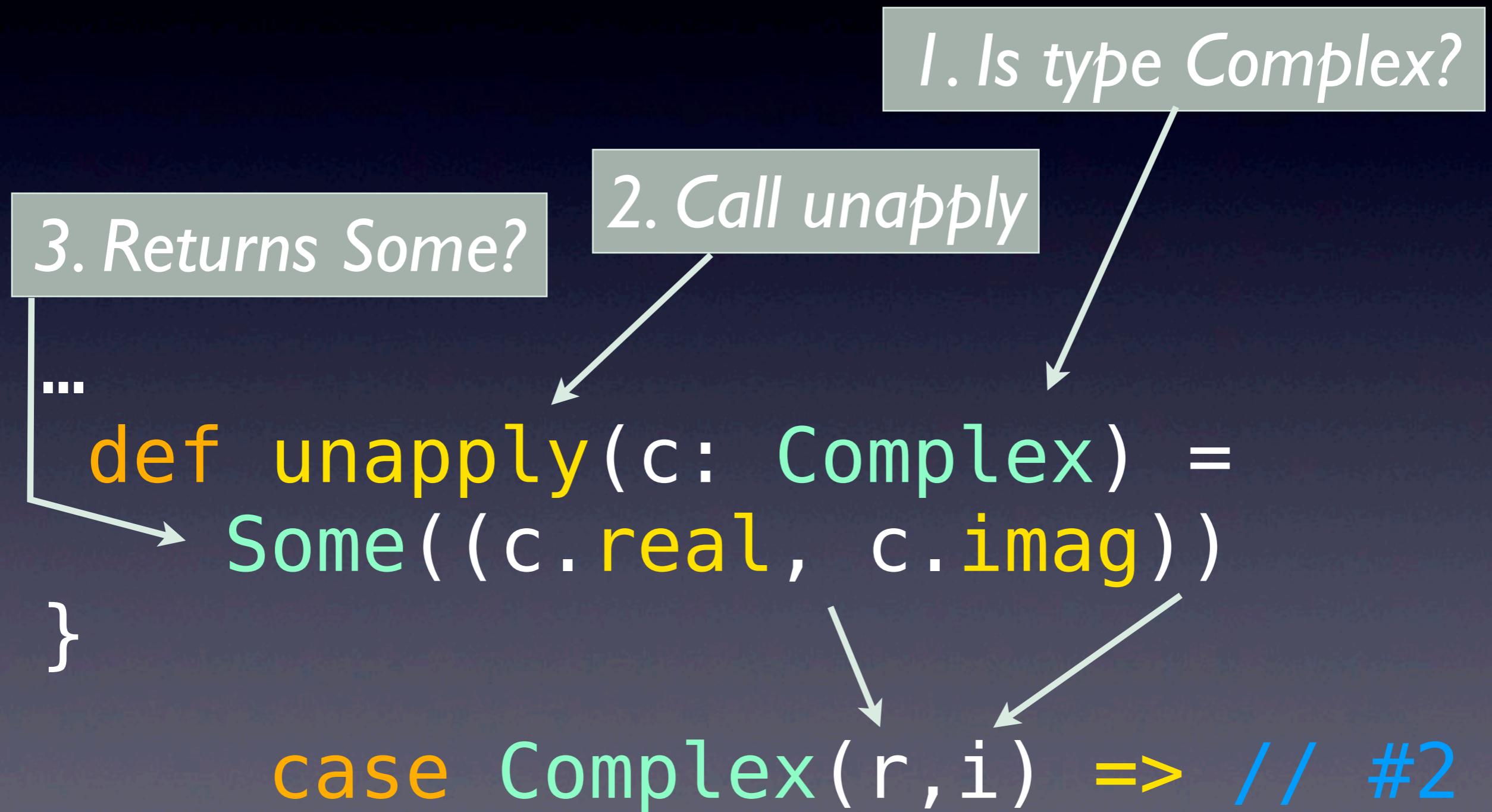
```
class Complex(val real: Double,  
             val imag: Double)  
{...}
```

```
object Complex{  
    def apply(...) = {...}  
    def unapply(c: Complex) =  
        Some((c.real, c.imag))  
}
```

```
class Complex(val real: Double,  
             val imag: Double)  
{...}
```

```
object Complex{  
    def apply(...) = {...}  
    def unapply(c: Complex) =  
        Some((c.real, c.imag))  
    }  
  
    case Complex(r, i) => // #2
```

Algorithm



164

Wednesday, November 28, 12

This is how `unapply` is used. First, the runtime checks the type of the object being matched. If it is a `Complex` instance (step 1), then `unapply` is called (step 3). `Unapply` can return `None` or `Some(...)`, where the `Some` contains a tuple of the values that `unapply` wants to expose (not necessarily all the fields!). If `None` is returned, then the runtime tries another match. Only if `Some` is returned is the match successful.

Unapply is
generated for
case classes.

You rarely write your own!

Aside:

NOT needed

```
list foreach {  
    _ match {  
        case Complex(1,1) => // #1  
        case Complex(r,i) => // #2  
        case (3,2,_ ) => // #3  
        case (a,b,c) => // #4  
        case Nil => // #5  
    } }  
}
```

Aside:

```
list foreach {  
    case Complex(1,i) => // #1  
    case Complex(r,i) => // #2  
    case (3,2,_) => // #3  
    case (a,b,c) => // #4  
    case Nil => // #5  
}
```

Can pass a PartialFunction

A `PartialFunction` is *partial* in that it doesn't apply for the whole range (possible inputs).

A `{...}` of **case**
statements is a
literal syntax for a
PartialFunction.

With the **case** keyword, you get (updated):

- *Arguments to primary constructor become fields.*
 - The **val** keyword not required.
 - Object *equals*, *hashCode*, *toString*.
 - Based on the fields.
 - Companion object w/ factory *apply* and *unapply*.

170

The keyword is
named **case** because
it facilitates
pattern matching.

Pattern matching
plays a
fundamental role in *FP*
analogous to
polymorphism in *OOP*.

Exercise

Pattern matching



173

Pattern Matching

- Using your `Address` case class, complete the code on the next page to match separately on `Addresses` with “Chicago” as the city, `Addresses` with any city, and all other cases.
 - Call `println` with a separate message for each.
 - Extra: extract the `Some` values.

```
val l = List(  
    Address(1, "2nd St.",  
            "Seattle", "WA", 98000),  
    Address(3, "4th Ave.",  
            "Chicago", "IL", 60600),  
    Some("hello!"), (1,2,3), None)
```

```
l foreach {  
    // case statements  
}
```

Primary and Secondary Constructors

176

Wednesday, November 28, 12

Because you sometimes need more than one c'tor.

primary constructor

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

secondary constructor

```
def this( ←  
    fName: String,  
    lName: String) =  
    this(fName, lName, 0)  
}
```

Parent and Child Classes

178

```
class Employee(  
    fName: String,  
    lName: String,  
    age: Int,  
    val job: Job) extends Person(  
    fName, lName, age)  
{...}
```

*pass parameters to
parent constructor*

Currying

180

Wednesday, November 28, 12

Not just a feature of Asian cuisine...

Currying:
Applying a
function's arguments
one at a time.

Named after Haskell Curry

```
def mod(m: Int)(n: Int) = n % m
```

2 param. lists

```
def mod3 = mod(3) - “_” required
```

```
for (i <- 0 to 3)
  println(mod(3)(i) + " == " +
    mod3(i) + " ?")
```

```
// => 0 == 0 ?
// => 1 == 1 ?
// => 2 == 2 ?
// => 0 == 0 ?
```

Currying methods.

182

Wednesday, November 28, 12

The “_” (our familiar wildcard) is required to indicate the remaining arguments are not yet applied. One “_” stands in for all the rest of the arguments.

Each parameter you want to “curry” separately has to be in its own parameter list.

“mod3” is a new function with “m” already set to “3”.

```
val mod = (m:Int,n:Int) => n%m
```

```
val modc = mod.curry
val mod3 = modc(3) // no '_'
```

```
for (i <- 0 to 3)
  println(mod(3,i) + ", " +
           modc(3)(i) + ", " +
           mod3(i))
// => 0, 0, 0
// => 1, 1, 1
// => 2, 2, 2
// => 0, 0, 0
```

Currying functions.

183

```
val mod = (m:Int, n:Int) => n%m
mod: (Int, Int) => Int = <...>
```

```
val modc = mod.curry
modc: (Int) => (Int) => Int = ...
```

```
val mod3 = modc(3)
mod3: (Int) => Int = <function>
```

Blue: interpreter output

184

Wednesday, November 28, 12

I'm showing in blue what the scala interpreter returns for each definition. Note the type signatures. The first and third should be straightforward to understand now, but what about the 2nd?

What does this mean?

```
val modc = mod.curry  
modc: (Int) => (Int) => Int
```

- Modc is a function that takes an Int arg
- ... and returns a function that takes an Int and returns an Int.

What does this mean?

```
val modc = mod.curry  
modc: (Int) => (Int) => Int
```

- It binds left to right.

```
modc: (Int) => ((Int) => Int)
```

User-defined Operators

187

```
case class Complex(  
    val real: Double,  
    val imag: Double) {  
  def +(that: Complex) =  
    new Complex(real+that.real,  
                imag+that.imag)  
  
  def -(that: Complex) =  
    new Complex(real-that.real,  
                imag-that.imag)}
```

...

“Operator overloading”

188

```
case class Complex(  
    val real: Double,  
    val imag: Double) {  
  def +(that: Complex) =  
    new Complex(real+that.real,  
                imag+that.imag)}
```

“operators”

```
def -(that: Complex) =  
  new Complex(real-that.real,  
              imag-that.imag)
```

...

“*Operator overloading*”

189

```
...
def unary_- =
    new Complex(-real, imag)

override def toString() =
    "(" + real +
    ", " + imag + ")"

}
```

“Operator overloading”

190

Sugar for “unary minus”

```
...  
def unary_- =  
    new Complex(-real, imag)
```

required when overriding
concrete method

```
override def toString() =  
    "(" + real +  
    ", " + imag + ")"  
}
```

“Operator overloading”

```
var c1 = Complex(1.1, 1.1)
val c2 = Complex(2.2, 2.2)
```

```
c1 + c2    //=> (3.3, 3.3)
c1 += c2   // same as c1 = c1+c2
c1 - c2    //=> (-1.1, -1.1)
-c1        //=> (-1.1, 1.1)
```

Example usage

192

Wednesday, November 28, 12

In use... Note that the compiler automatically converts `c1 += c2` into `c1 = c1 + c2`.

Exercise

Create a Rational Number class.



193

A Rational Class

- Create an *immutable* Rational Number case class.
 - Used to represent division of two integers.
 - $\text{Rational}(n, d) = n/d$
- Add methods for `+`, `-`, `toString`.
- Test with examples.
- Create instances, read fields, etc.

Packages and Imports

195

Packages and Imports: Java Style

```
package com.megacorp.util  
import scala.actors.Actor  
import java.io.File  
...  
class Person ...
```

Packages and Imports: Alternative Style

```
package com.megacorp.util {  
    import scala.actors._  
    import java.io.{File,  
        Reader => JReader,  
        Writer => _,  
        _ }  
    ...  
}
```

namespace style

```
package com.megacorp.util {  
    import scala.actors._           ← all types  
    import java.io.{File,  
        Reader => JReader,  
        Writer => _},  
    _  
    ...  
}
```

everything else...

suppress

“rename”

selective

all types

198

Wednesday, November 28, 12

Besides normal java-style package declarations, you can nest namespace” style declarations, like C++/C# namespaces or Ruby modules.
Imports can pick things to import, “rename” them (to avoid name collisions), and explicit suppress some things.

Declarations In the Namespace

```
package com.megacorp.util {  
    ...  
    class StringUtils {...}  
}  
package com.megacorp.model {  
    ...  
    class Person {...}  
}
```

Even in the same file!

File names and Directories?

200

Wednesday, November 28, 12

If you can use the namespace construct, what does that mean for a required directory layout, if any? And what about file names matching type names?

File and *type*
names
don't have to
match

201

Wednesday, November 28, 12

Scala removes Java's restriction. You can still follow it if you want.

*Directory and
package names
don't have to
match*

202

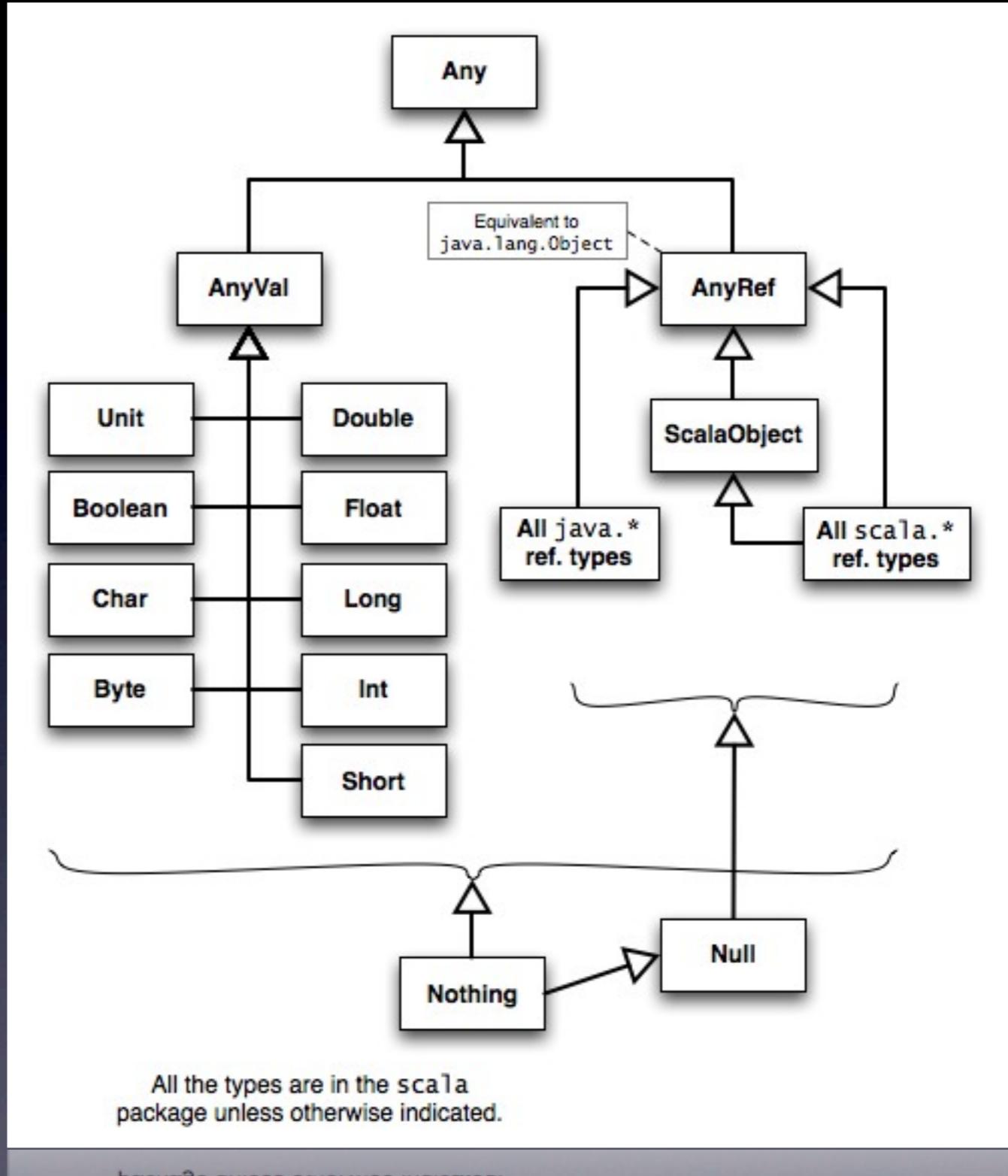
Wednesday, November 28, 12

Scala removes Java's restriction. You can still follow it if you want.

The Type Hierarchy

203

Root of the Type Hierarchy



Any	Root type
AnyVal	Parent of <i>Value</i> types: Unit, Int, Boolean, ...
AnyRef	Parent of <i>Ref.</i> types: List, Actor, ...
Nothing	Child of <i>all other types</i> . <i>No instances!</i>

“AnyRef” is equivalent to Java’s Object

205

Wednesday, November 28, 12

Scala doesn’t have primitives (“everything is an object”). Instead, Scala has immutable “value” types that represent the primitives. These types can’t be subclassed and they can’t be instantiated with new. Rather, you use literals (e.g., “1”) and operations on values.

<code>List[+A]</code>	Immutable, pervasive <i>functional</i> type
<code>Set[+A]</code>	Immutable and mutable versions
<code>Map[+A]</code>	Immutable and mutable versions

Scala “encourages” using immutables

206

Wednesday, November 28, 12

Scala “encourages” using immutable collections, List, Set, and Map are already “imported”. Mutable alternative collections have to be imported explicitly. Collections are parameterized (like Java generics). The “+” means “A or subclass of A”

<code>TupleN[A1 , ... , AN]</code>	literal <code>(x1, x2, x3, ...)</code>
<code>Option[A]</code>	<code>Some(a)</code> or <code>None</code>
<code>FunctionN[-A1 , ... , -AN , +R]</code>	Function literals

Option helps avoid nulls

207

Wednesday, November 28, 12

Tuples are very convenient for ad hoc, fixed sized groups of things.

Option we'll explore later. For now, think of it as a better alternative than allowing something to "null". Rather, it's "optional".

What do the [+A] & [-A] mean?

Type “variance
annotations”

208

Wednesday, November 28, 12

The + and - are type variance annotations that describe how the parameterized types behave under subclassing relative to the subclassing relationship of the corresponding type parameters.

What Does [+A] Mean?

```
class List[+A] {...}
```

```
new List[String]
```

is a subclass of

```
new List[AnyRef]
```

Covariant

209

Example

```
def printClass(l: List[AnyRef])  
= {  
    l.foreach(x =>  
        println(x.getClass))  
}  
  
printClass("a" :: "b" :: Nil)  
printClass(1 :: 2 :: Nil)
```

210

Wednesday, November 28, 12

We'll describe this literal syntax for lists later.

The function expects a list of reference type objects. It calls "getClass" on each, which is not supported by Any or AnyVal. So, if we pass it a list of Strings, the function works fine and is none the wiser...

By the way...

```
val l = "a" :: "b" :: Nil  
l.foreach(x =>  
    println(x.getClass))
```

// Try these variants:

```
l.foreach(println(_.getClass))
```

Fails!

```
l.map(_.getClass).foreach(  
    println)
```

Works

Another variation...

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

```
l map _.getClass foreach println  
l map (_.getClass) foreach println
```

2nd one works

212

Wednesday, November 28, 12

Normally, you can chain “pipe-like” methods without the “.” and parentheses. In this case, we have to write (`_.getClass`) to disambiguate the meaning of “`_`”.

What Does [-A] Mean?

trait Function2[-A1, -A2, +R]

Function2[AnyRef, AnyRef, String]

is a subclass of

Function2[String, String, AnyRef]

Contravariant in the arg. types.

Covariant in the return types.

Wednesday, November 28, 12

For now, think of traits as abstract classes...

The - means the subclassing behavior is “contravariant”, i.e., the subclassing of the parameterized type “goes in the opposite direction” as the subclassing of the type parameter(s).

Recall that:

(AnyRef , AnyRef) => String

is equivalent to

Function2[AnyRef , AnyRef , String]

“Function Literal”
syntax

Contravariant Arguments

```
def mapper(l: List[AnyRef],  
          f: (AnyRef) => Any) = {  
    l.map(f(_))  
}
```

Function argument

```
mapper("a" :: "b" :: Nil,  
       (x: AnyRef) => x.getClass)
```

*Now using
Function ...*

Function literal (value)

215

Example

```
def mapper(l: List[AnyRef],  
          f: (AnyRef) => Any) = {  
    l.map(f(_))  
}
```

```
mapper("a" :: "b" :: Nil,  
(x:String) => x.toUpperCase)
```

Error!

Example

```
def mapper(l: List[AnyRef],  
          f: (AnyRef) => Any) = {  
    l.map(f(_))  
}
```

```
mapper("a" :: "b" :: Nil,  
(x:Any) => x.asInstanceOf[Int])
```

Works!

*Why arguments must
be contravariant*

Covariant Returns

```
def mapper(l: List[AnyRef],  
          f: (AnyRef) => Any) = {  
    l.map(f(_))  
}
```

```
mapper("a" :: "b" :: Nil,  
(x: Any) => x.asInstanceOf[Int])
```

Works!

*Boolean (and String) are
subclasses of Any*

218

Recap: [-A,+R]

```
class Function2[-A1,-A2,+R]
```

```
new Function2[AnyRef,AnyRef,  
String]
```

is a subclass of

```
new Function2[String,String,  
AnyRef]
```

Aside: Can we make this work?

```
def mapper(l: List[AnyRef],  
          f: (AnyRef) => Any) = {  
    l.map(f(_))  
}  
  
mapper("a" :: "b" :: Nil,  
(x:String) => x.toUpperCase)
```

Yes, with this change:

```
def mapper[T](l: List[T],  
  f: (T) => Any) = {  
  l.map(f(_))  
}
```

```
mapper("a" :: "b" :: Nil,  
(x:String) => x.toUpperCase)
```

Works!

Scala has *declaration-site* inheritance...

222

Wednesday, November 28, 12

In scala, you specify the parameterized type inheritance at the declaration site. That means it's up to the API designer to decide on the correct subtyping behavior, as long as it's consistent with the language rules; Scala doesn't allow covariant function argument types and contravariant return types!

...Java has
call-site
inheritance.

223

Wednesday, November 28, 12

In Java, users of the parameterized type inheritance specify the allowed inheritance. I.e., the burden is on the user of the API to get it right.

In Java:

```
class Function2<A1,A2,R>
```

```
... f = new Function2<  
? super String,  
? super String,  
? extends String>
```

*Variance defined
at the call site*

In Scala, the
type designer
specifies the
right behavior.

225

Wednesday, November 28, 12

In Scala, an experienced type designer can specify the correct behavior, relieving the user of that burden. Also, the compiler will enforce certain kinds of correctness.

In Java, the
user has to do
the *right* thing.

226

Wednesday, November 28, 12

In Java, users have to understand what's allowed and not make mistakes.

Exercise

Experiment with mapper.



227

Mapper

- Experiment with calling **mapper** using different lists and functions.
- Experiment with the implementation of **mapper**. How would you implement a **filter**?

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??

230

Make your own controls

// Print with line numbers.

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

Make your own controls

// Print with line numbers.

```
control?           File to loop through  
loop (new File(...)) {}  
(n, line) => ← Arguments passed to...
```

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

```
}
```

what do for each line

How do we do this?

Output on itself:

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

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

```
object Loop {
```

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

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

— 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”

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

```
object Loop {
```

two parameters

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

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

```
object Loop {
```

two parameters lists

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

Why 2 Param. Lists?

```
// Print with line numbers.  
import Loop.loop  
  
loop (new File(".")) {}  
(n, line) =>  
    format("%3d: %s\n", n, line)  
}  
  
2nd parameter: a “function literal”
```

import

*1st param.:
a file*

2nd parameter: a “function literal”

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

nested method

Finishing Numberator...

239

Wednesday, November 28, 12

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...

240

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

Pure functional “looping” technique

241

It is *Tail* Recursive

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

Scala optimizes tail recursion into loops

Exercise

A whileTrue loop



243

I want to write:

```
var i = 0
whileTrue(i < 10) {
    println(i)
    i += 1
}
```

Implement whileTrue

- Here is the declaration:

```
def whileTrue(  
    condition: => Boolean)(  
    block: => Unit): Unit
```

- Each argument is a *by-name* parameter.
- Use recursion.

By-name vs. by-value

```
def method(bool: => Boolean){  
    if (bool)  
        ...  
    ...  
}
```

*Called w/out
parentheses*

**“by-name”
parameter**

```
def method2(  
    byvalue: (Int) => Boolean){  
    ...  
}
```

“by-value” parameter

Why use *by-name* parameters?
They are evaluated
each time they are
referenced.

They aren't eval'ed before passing to whileTrue.

I want to write:

```
var i = 0
whileTrue(i < 10) {
    println(i)
    i += 1
}
```

*Evaluated
inside
whileTrue*

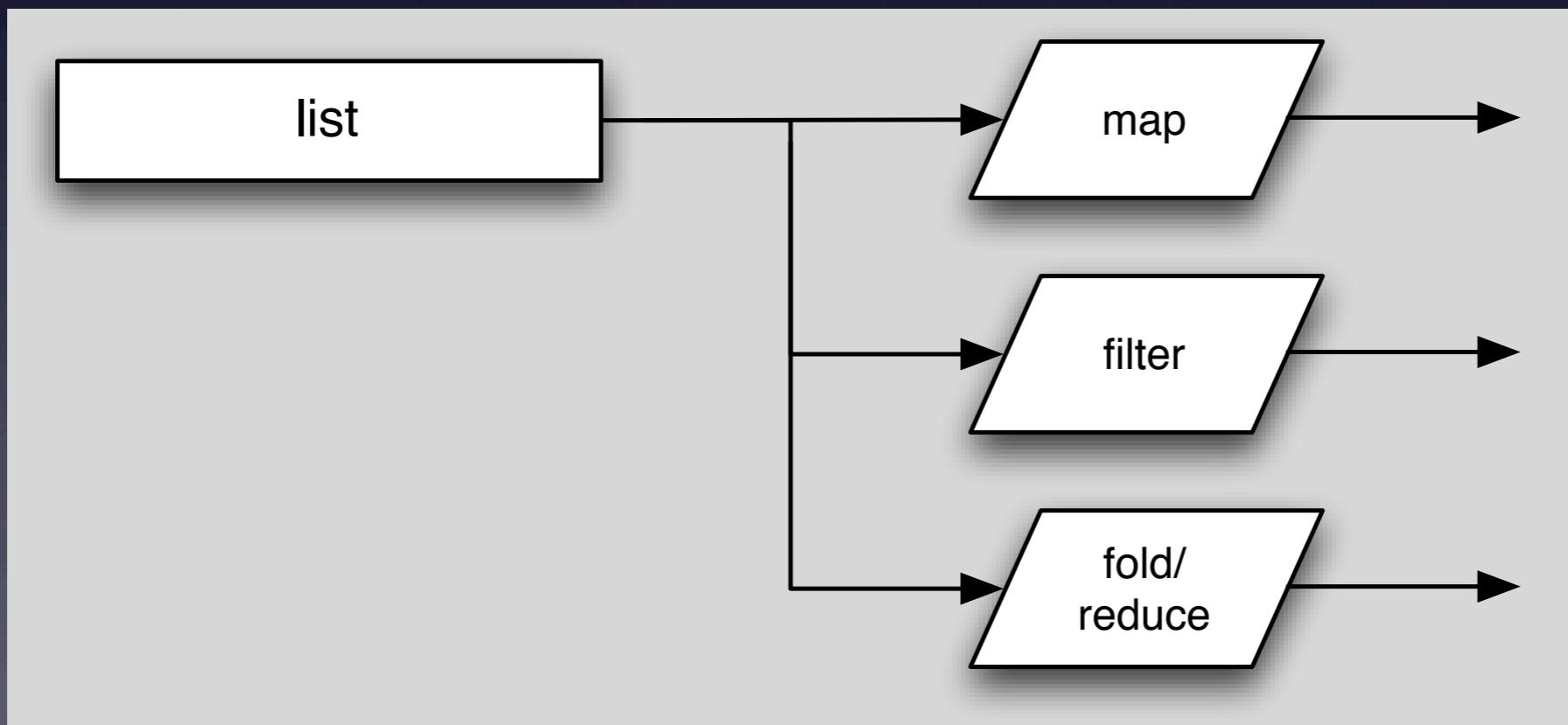
Avoiding Mutable State, Revisited

Chaining collection
operations that
handle iteration
for us.

Avoiding loop counters

250

Recall the Classic Operations on *Functional* *Data Types*



```
object Main {
```

You put *main* in an object

```
def main(args:Array[String])={
```

```
    args.map(s => s.toInt)
```

```
.toSet
```

```
.toList
```

```
:sortWith((x,y) => x < y)
```

```
:foreach(i => println(i))
```

```
}
```

```
}
```

What does this do?

```
$ scalac Main.scala  
$ scala -cp . Main 1 3 4 2 3 4 5 7 6 2 1 3  
1  
2  
3  
4  
5  
6  
7
```

Application to sort integers, print out unique values.

253

```
object Main {  
  
    def main(args:Array[String])={  
  
        args.map(s => s.toInt)  
        .toSet  
        .toList  
        .sortWith((x,y) => x < y)  
        .foreach(i => println(i))  
    }  
}
```

*Some of these “placeholder”
variables are unnecessary.*

```
object Main {  
  
    def main(args:Array[String])={  
  
        args.map(_.toInt)  
            .toSet  
            .toList  
            .sortWith(_ < _)  
            .foreach(println(_))  
    }  
}
```

“_” is a placeholder

```
// Group by the 1st few
// Fibonacci numbers
object GroupByFibs {
  def main(args:Array[String])={
    args.map(_.toInt)
      .toSet
      .toList
      .sortWith(_ < _)
      .foreach(println(_))
  }
}
```

“_” is a placeholder

Exercise

Experiment with collection operations



257

Collection Operations

- Experiment with this program.
 - Pass different functions to **filter** and **foreach**.
 - Try different combinations of functions like **map**, **foldLeft/foldRight**, **reduceLeft/reduceRight**, as well as **filter** and **foreach**.

For Loops (Comprehensions)

259

```
object CapsStartFor {  
  
    def main(args: Array[String]) = {  
        for (  
            arg <- args;  
            if (arg(0).isUpper)  
        )  
            println(arg)  
    }  
}
```

```
// $ scalac CapsStartFor.scala  
// $ scala -cp . CapsStartFor aB Ab AB ab  
// Ab  
// AB
```

*“For” can have an arbitrary number
of generators, conditions, assignments*

Wednesday, November 28, 12

The “println(arg)” is the “body” of the for loop. Because there is only one statement, we don’t need “{...}”. Note the “for (stmt1; stmt2; ...)” syntax. Unlike Java, you’re not required to have 3 statements, each with a specific “role”.

```
object CapsStartFor {  
  
    def main(args: Array[String]) = {  
        for (arg <- args;  
             if (arg(0).isUpperCase))  
        )  
            println(arg)  
    }  
}
```

```
// $ scalac CapsStartFor.scala  
// $ scala -cp . CapsStartFor aB Ab AB ab  
// Ab  
// AB
```

```
object CapsStartFor {  
  
    def main(args: Array[String]) = {  
        for {  
            arg <- args  
            if (arg(0).isUpperCase)  
        }  
            println(arg)  
    }  
  
    // $ scalac CapsStartFor.scala  
    // $ scala -cp . CapsStartFor aB Ab AB ab  
    // Ab  
    // AB
```

Replaced “(...)” with “{...}”, dropped “;”

```

object CapsStartList {

  def main(args: Array[String]) = {
    val capList = for {
      arg <- args
      if (arg(0).isUpperCase)
    }
    yield arg
  }
  println(capList)
}

```

“println” is outside loop

```

// $ scalac CapsStartList.scala
// $ scala -cp . CapsStartList aB Ab AB ab
// List(Ab, AB)

```

“yield” to create a list

263

Wednesday, November 28, 12

Yield returns each “arg” that makes it through the conditions, etc., forming the List “capList”. We then print the list. That statement is outside the loop.

Pattern Matching:

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

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

No “if” statement

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

264

Wednesday, November 28, 12

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.

“Monadic” Behaviors

```
val l = List(  
  Some("a"), None, Some("b"),  
  None, Some("c"))  
for {  
  x <- l  
  s <- x  
} yield s  
// List(a, b, c)
```

“Iterate” over the Option[T], skip None!

Note how Lists and Options can be treated uniformly

265

Wednesday, November 28, 12

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.

Exercise

Creating a list of tuples using a
for comprehension



List (i,j,k) Tuples Where:

- i goes from 1 to 10
- $i \geq j \geq k$
- $(i + j + k) \% 3 == 0$

```
for {  
    ??  
}  
    format("(%d,%d,%d)", i, j, k)
```

Variation

```
val list = for {  
    ??  
} yield ((i, j, k))  
println(list)
```

The background of the slide features a wide-angle photograph of a mountainous landscape. In the foreground, there is a calm lake reflecting the surrounding environment. On either side of the lake, there are forested hills. In the distance, a range of majestic mountains with snow-capped peaks rises against a clear sky. The lighting suggests it might be either sunrise or sunset, with warm, golden light illuminating the peaks.

Actor Concurrency

269

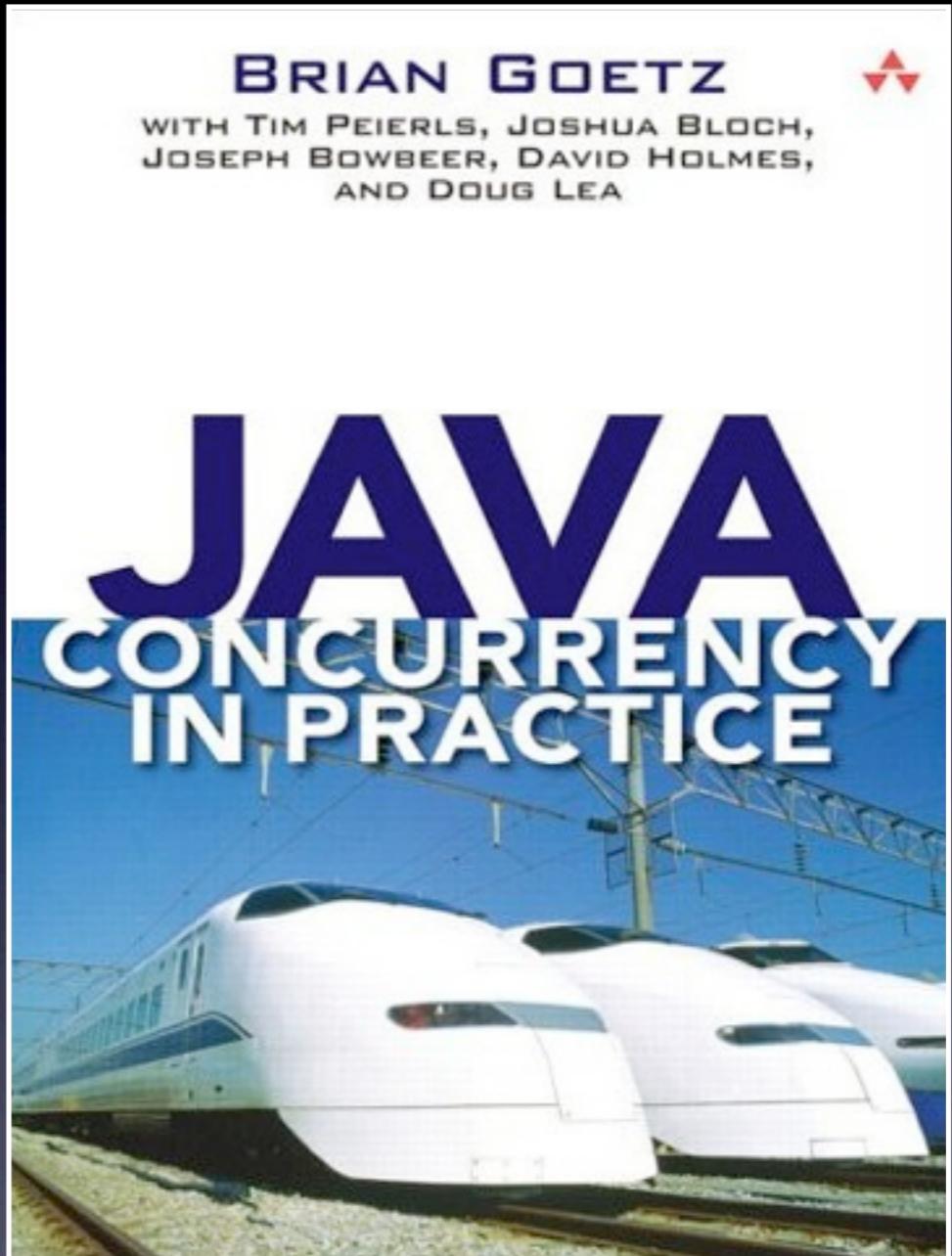
Wednesday, November 28, 12

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)



270

Wednesday, November 28, 12

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

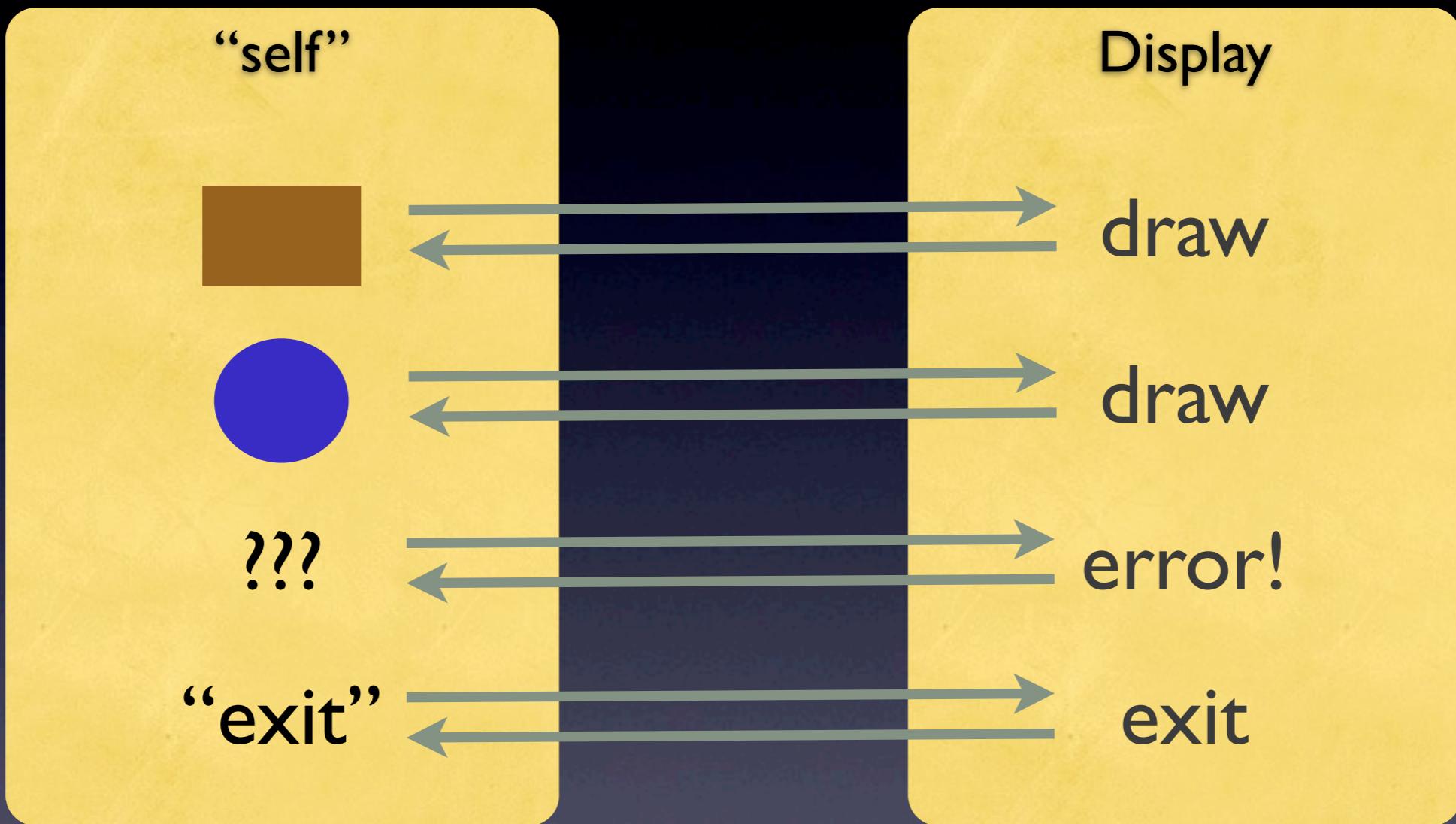
Actor Model

- 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.

2 Actors:



```
package shapes
```

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

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

Hierarchy of geometric shapes

274

Wednesday, November 28, 12

“Case” classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape. The “case” keyword makes the arguments “vals” by default, adds factory, equals, etc. methods. Great for “structural” objects.

(Case classes automatically get generated equals, hashCode, toString, so-called “apply” factory methods - so you don’t need “new” - and so-called “unapply” methods used for pattern matching.)

NOTE: The full source for this example is at <https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala/code-examples/actor>.

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

*concrete “draw”
methods*

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

```
package shapes  
import akka.actor._
```

*Use the “Akka”
Actor library*

```
class ShapeDrawingActor extends Actor {  
    def receive = {  
        ...  
    }  
}
```

*receive and handle
each message*

Actor for drawing shapes

276

Wednesday, November 28, 12

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 I prefer over Scala's native actors.

Receive
method

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

Actor for drawing shapes

277

Wednesday, November 28, 12

“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()
        self.reply("Shape drawn")
    case "exit" =>
        println("-> exiting...")
        self.reply("good bye!")
    case x => // default
        println("-> Error: " + x)
        self.reply("Unknown: " + x)
}

```

*pattern
matching*

Actor for drawing shapes

278

Wednesday, November 28, 12

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 treat as an unexpected error.

```

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

```

*draw shape
& send reply*

done

unrecognized message

279

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

Altogether

```
import shapes._  
import akka.actor._  
import akka.actor.Actor._
```

a “singleton” type to hold main

```
object Driver {  
  def main(args: Array[String]) = {  
    val driver = actorOf[Driver]  
    driver.start  
    driver ! "go!"  
  }  
}  
class Driver ...
```

*! is the message
send “operator”*

driver to try it out

Its “companion” object Driver
was on the previous slide.

```
...  
class Driver extends Actor {  
    val drawer =  
        actorOf[ShapeDrawingActor]  
    drawer.start  
    def receive = {  
        ...  
    }  
}
```

driver to try it out

```

def receive = {
  case "go!" => ← sent by main
    drawer ! Circle(Point(...), ...)
    drawer ! Rectangle(...)
    drawer ! 3.14159
    drawer ! "exit"
  case "good bye!" => ← sent by drawer
    println("<- cleaning up...")
    drawer.stop; self.stop
  case other =>
    println("<- " + other)
}

```

driver to try it out

```
case "go!" =>
  drawer ! Circle(Point(...),...)
  drawer ! Rectangle(...)
  drawer ! 3.14159
  drawer ! "exit"
```

```
// run Driver.main (see github README.md)
-> drawing: Circle(Point(0.0,0.0),1.0)
-> drawing: Rectangle(Point(0.0,0.0),
2.0,5.0)
-> Error: 3.14159
-> exiting...
<- Shape drawn.
<- Shape drawn.
<- Unknown: 3.14159
<- cleaning up...
```

“<-” and “->” messages
may be interleaved!!

284

Wednesday, November 28, 12

NOTE: The full source for this example is at <https://github.com/deanwampler/Presentations/tree/master/SeductionsOfScala/code-examples/actor>. See the README.md for instructions.

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!!

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

*Functional-style
pattern matching*

*Object-
oriented-style
polymorphism*

*“Switch” statements
are not evil!*

Exercise

Variations on an Actor theme.



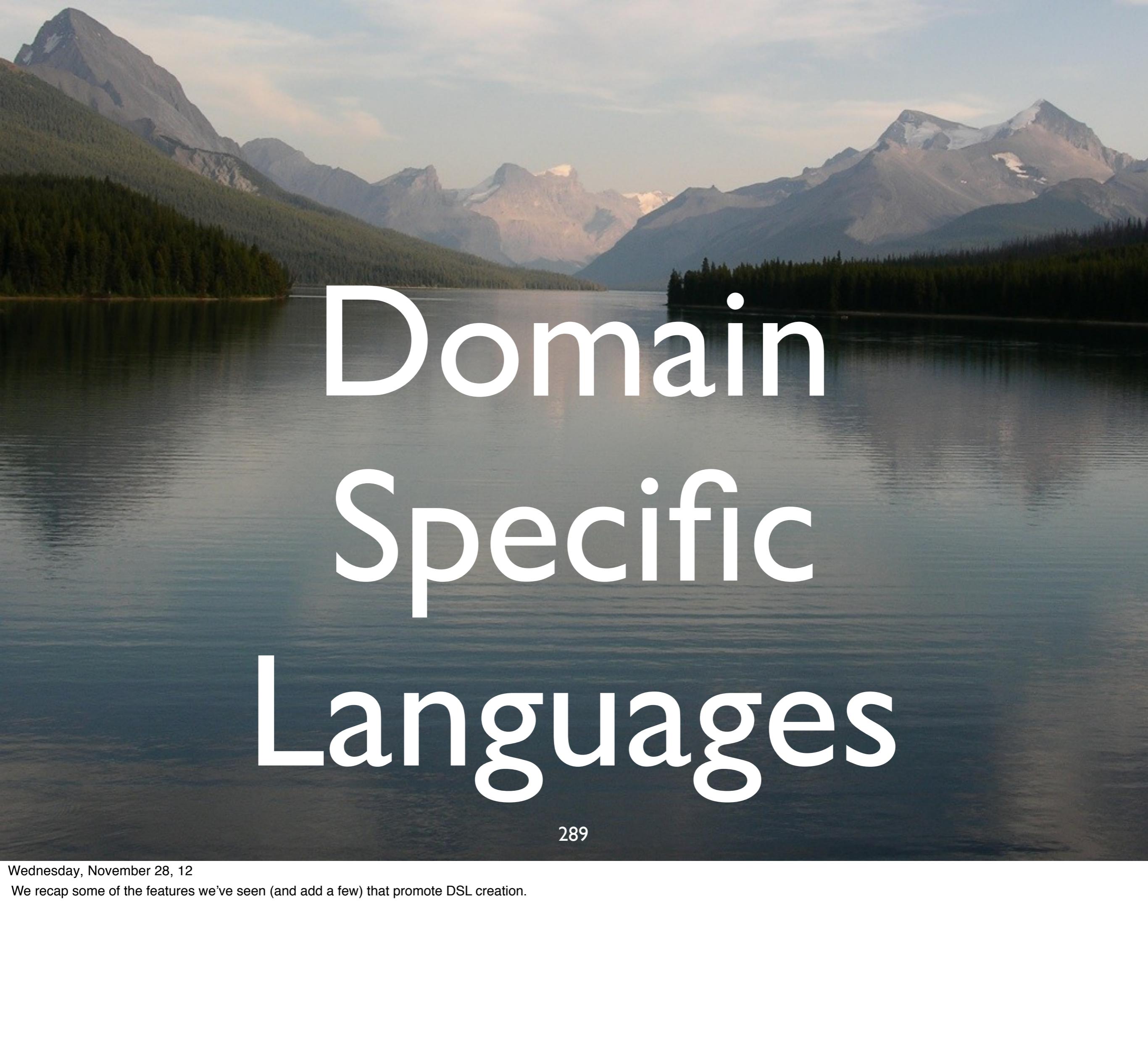
286

Refine Features

- Add more geometric shapes
- Add other messages for new behaviors.
 - Redraw all shapes drawn so far.
 - Clear the “screen”.
 - ...

What happens when...

- You replace the ! (asynchronous) calls with !! (synchronous) calls in the driver?
- Look at <http://doc.akka.io/actors-scala>.
- Find the !! method description.
- Modify the driver **receive** method.

A scenic landscape featuring a calm lake in the foreground, framed by dense evergreen forests on the left and right. In the background, a range of majestic mountains is visible under a clear blue sky.

Domain Specific Languages

289

Wednesday, November 28, 12

We recap some of the features we've seen (and add a few) that promote DSL creation.

Internal DSLs

290

Features for Building *Internal* DSLs

- Infix operation notation.
- Implicit type conversions.
- First-class functions.

Infix operator notation

"hello" + "world"

same as

"hello".+("world")

Infix operator notation

1 hour fromNow

Instead of

1.hour(fromNow)

Implicit Type Converters

1 hour fromNow

Need the “hour”
method on Int

Implicit Type Converters

```
class Hour (val howMany:Int){  
    def hour(when: Int) =  
        when + howMany  
}  
  
object Hour {  
    def fromNow = ...  
    implicit def int2Hour(i:Int) =  
        new Hour(i)  
}
```

*The current time
in hours...*

Implicit Type Converters

Must import...

```
import Hour._
```

```
1 hour fromNow
```

fromNow()

passed to hour.

int2Hour called,
returning Hour(1).

Hour.hour(...)
called.

User Defined Controls

```
repeat (10 times) {  
    checkForUpdates()  
}
```

What I want

“times” is a “bubble” (throwaway) word.
Implement with an implicit.
It should just return the same Int value.

User Defined Controls

```
repeat (10 times) {  
    checkForUpdates()  
}
```

```
object Repeater {  
    def repeat(i:Int)(f: => Unit) =  
        for (j <- 1 to i) f  
}
```

User Defined Controls

```
repeat (10 times) {  
    checkForUpdates()  
}
```

```
object Repeater {  
    def repeat(i:Int)(f: => Unit) =  
        for (j <- 1 to i) f  
}
```

TWO
argument lists

User Defined Controls

```
repeat (10 times) {  
    checkForUpdates()  
}
```

```
object Repeater {  
    def repeat(i:Int)(f: => Unit) =  
        for (j <- 1 to i) f  
    }  
}
```

“by-name”
parameter

“Called” w/out
parentheses

External DSLs

301

Features for Building *External* DSLs

- Parser Combinator Library

302

Wednesday, November 28, 12

For building external DSLs, i.e., those with a unique grammar and parser, Scala's library has an internal DSL for building parser combinator libraries.

Consider this Grammar

```
repeat 10 times {  
    say "hello"  
}
```

BNF grammar

```
repeat = "repeat" n "times" block;  
n      = wholeNumber;  
block  = "{" lines "}" ;  
lines  = { line }; ← repetition  
line   = "say" message;  
message = stringLiteral;
```

Translating to Scala

```
def repeat = "repeat" ~> n <~  
    "times" ~ block  
def n      = wholeNumber  
def block  = "{" ~> lines <~ "}"  
def lines  = rep(line)  
def line   = "say" ~> message  
def message = stringLiteral
```

```
import
scala.util.parsing.combinator._

object RepeatParser extends
JavaTokenParsers {

  var count = 0 // set by "n"
  def repeat = "repeat" ~> n <~
    "times" ~ block
  def n      = wholeNumber
  def block  = "{" ~> lines <~ "}"
  def lines  = rep(line)
  def line   = "say" ~> message
  def message = stringLiteral
}
```

```

def repeat = "repeat" ~> n <~
  "times" ~ block
def n      = wholeNumber ^^
  {reps => count = reps.toInt}
def block  = "{" ~> lines <~ "}"
def lines  = rep(line)
def line   = "say" ~> message ^^
  {msg => for (i <- 1 to count)
    println(msg)}
def message = stringLiteral

```

1) save count, 2) print message count times.

In Action...

```
val input =  
  """repeat 10 times {  
    say "hello"  
}"""  
  
RepeatParser.parseAll(  
  RepeatParser.repeat, input)
```

In Action...

“hello”
“hello”

The output

Implicit Conversions

Alternative slides

You write:

```
val months = Map(  
  "Jan" -> 1,  
  "Feb" -> 2,  
  ...  
)
```

Part of the language grammar??

310

Wednesday, November 28, 12

This is a nice literal syntax for initializing a map, reminiscent of Ruby's syntax for hashes (maps). Is this an “ad hoc” feature of the language grammar? No! We can invent a “domain-specific language” like this using regular methods and “implicit conversions”.

How this works

```
val months = Map( "Jan" -> 1, "Feb" -> 2, ... )
```

Map factory method

wants “(a,b)*”

“implicit” conversion:
String to ArrowAssoc

ArrowAssoc.->(right) = (left,right)

The *implicit* conversion

```
implicit def any2ArrowAssoc[A] (x: A): ArrowAssoc[A] =  
    new ArrowAssoc(x)
```

keyword: parser will use this method for type conversion

parameterized method

Imported automatically, inside *Predef*

312

Wednesday, November 28, 12

The implicit keyword marks this method as a “candidate” for doing a type conversion, as long as it is in scope when needed. In this case, it always is in scope because it’s defined in “Predef”, a set of imports and definitions the parser automatically includes.

ArrowAssoc

```
class ArrowAssoc[A](val x: A) {  
  def -> [B](y:B): Tuple2[A,B] =  
    Tuple2(x, y)  
  ...  
}
```

*The “->”
method*

*Returns a
tuple “(x,y)”*

Implicit Conversions

- *Good*
 - Help create *elegant API's*.
 - Promote *DSL* creation.
- *Bad*
 - Can cause *mystifying* behavior.

314



Recap

315

Scala is...

316

Wednesday, November 28, 12

a better
Java and C#,

*object-oriented
and
functional,*

*succinct,
elegant,
and
powerful.*

Some General Lessons:

320

*Objects can be functions.
Functions are objects.*

Observation:

Any object graph
decomposes into
values and collections.

322

Wednesday, November 28, 12

No matter how complex your objects, ultimately they are composed of simpler instances of types and collections of things, which are best manipulated with FP.

Collections:

Construct, iterate,
and manage them
functionally.

323

Wednesday, November 28, 12

We'll see what this means shortly.

Values:

Prefer
immutable
values.

324

Wednesday, November 28, 12

What are allowed for numbers, strings, ..., a person's age, street address, ...

Values:

Carefully specify
the properties
of value types.

325

Wednesday, November 28, 12

What are allowed for numbers, strings, ..., a person's age, street address, ...

What are the *properties* of

- Names?
- Account balances?
- Street addresses?
- Financial instruments?

326

Wednesday, November 28, 12

What are allowed for numbers, strings, ..., a person's age, street address, ...

If you require
mutable objects,
specify *states* and
state transitions
carefully.

Prefer *functions* and *mixins* (traits) for composition.

328

Wednesday, November 28, 12

First-class functions make code far more composable and eliminate vast quantities of boilerplate that takes time to develop, test, and maintain.
We've already seen first-class functions.

Thanks!

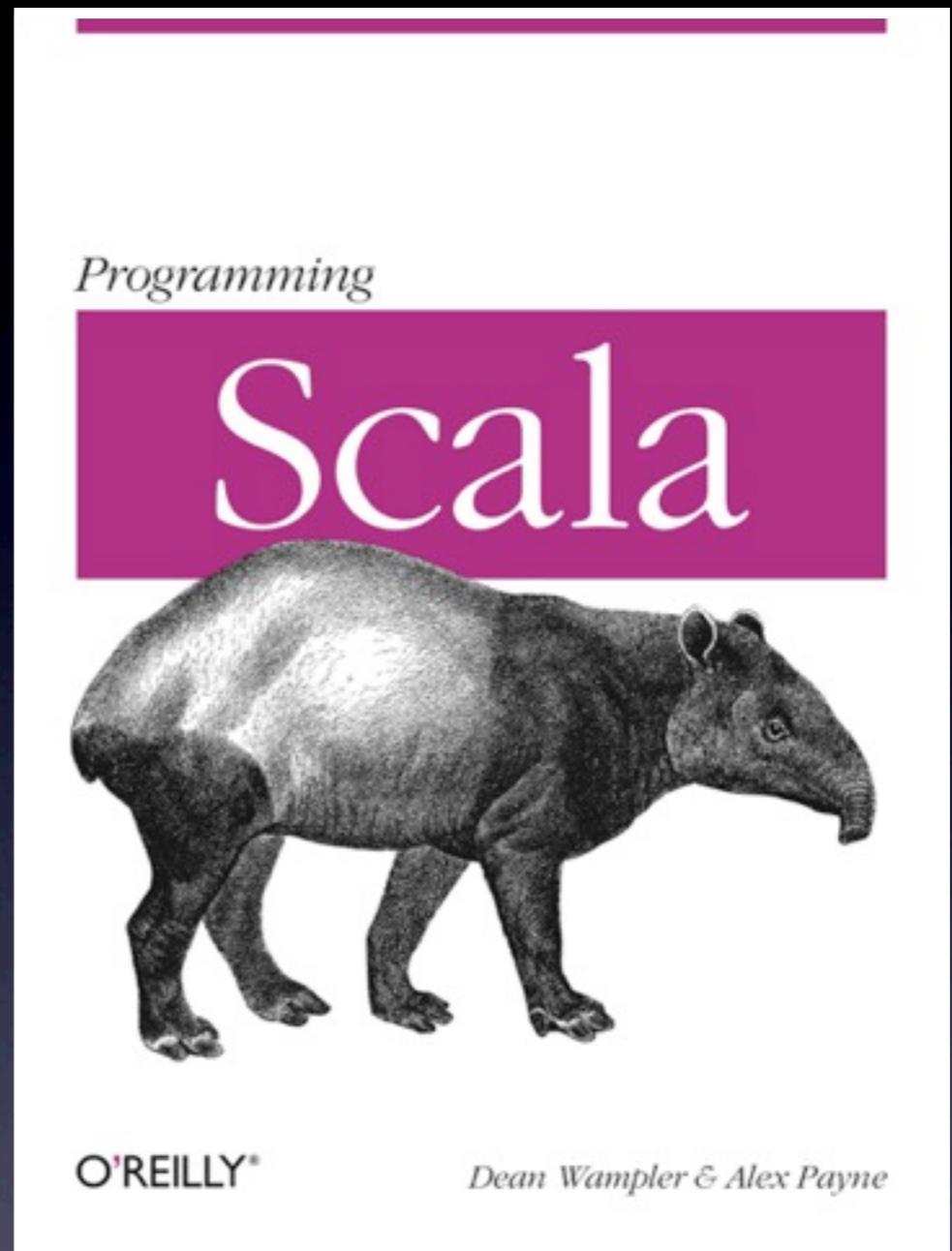
dean@deanwampler.com
@deanwampler

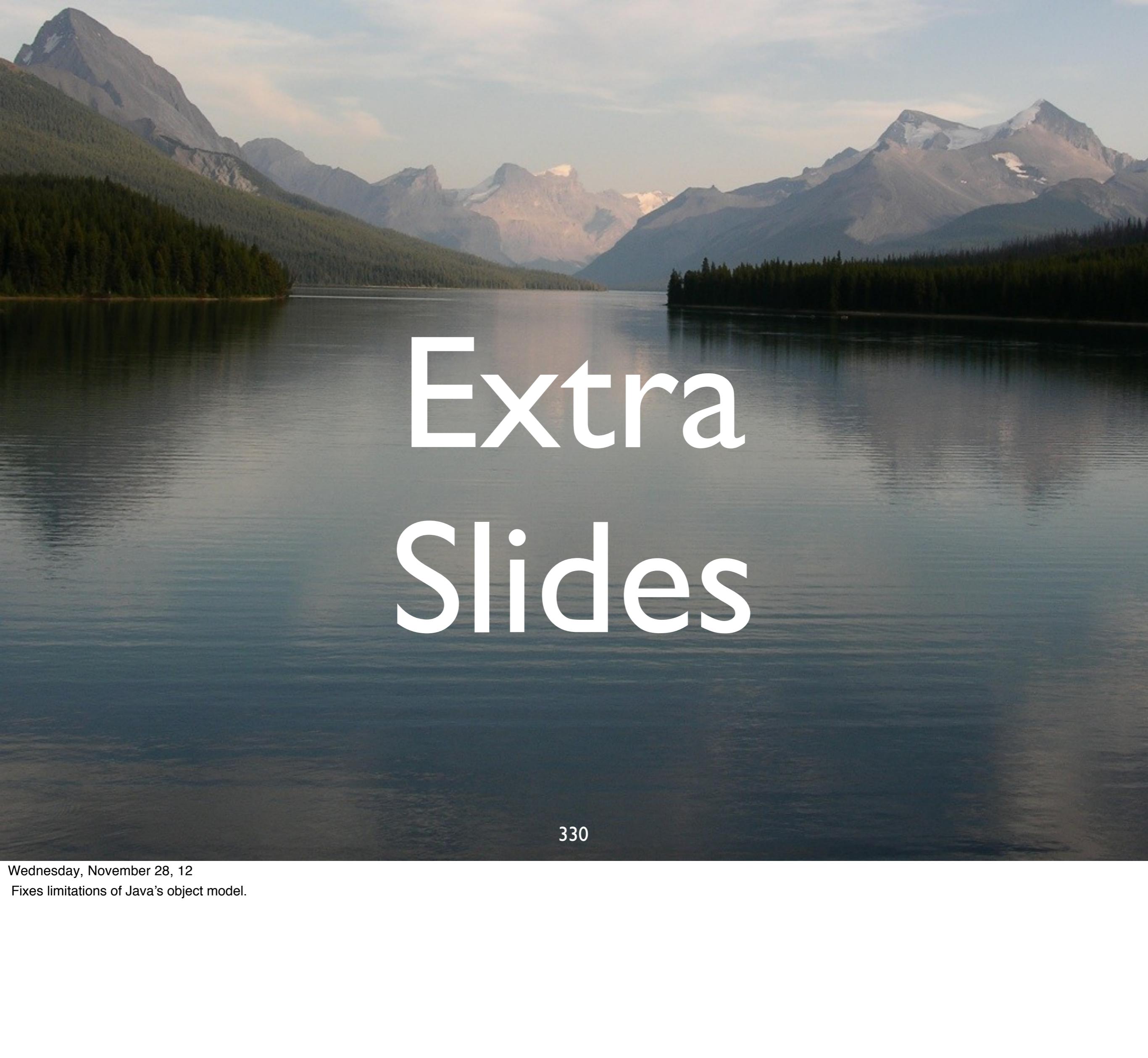
polyglotprogramming.com/talks
programmingscala.com

thinkbiganalytics.com



329



A wide-angle photograph of a mountainous landscape. In the foreground, a calm lake reflects the surrounding environment. On the left, a dense forest of evergreen trees covers a hillside. The background is dominated by a range of mountains, their peaks partially obscured by a hazy sky. The lighting suggests either sunrise or sunset, casting a warm glow on the scene.

Extra Slides

330

Wednesday, November 28, 12

Fixes limitations of Java's object model.

Exercise

Implicit conversion methods.



331

Another way to create a List

```
val list = List("Jan", "Feb", ...)
```

same as

```
val list = "Jan" :: "Feb" ::  
... :: Nil
```

Consider This Example

```
def process[A,B,C](  
    t: Tuple3 [A,B,C] ) = {  
    println(t._1)  
    println(t._2)  
    println(t._3)  
}  
process((1,2,3))  
process(List(4,5,6)) // fails
```

How about?

Converter: List to Tuple

- Write an **implicit** method to convert a 3-element **List** to a **Tuple3**.
- Use it to successfully invoke:
`process(List(4,5,6))`
- Hint:
 - `list(n)` returns the n^{th} element (0-based).