

The Seductions of Scala

Dean Wampler

dean@deanwampler.com

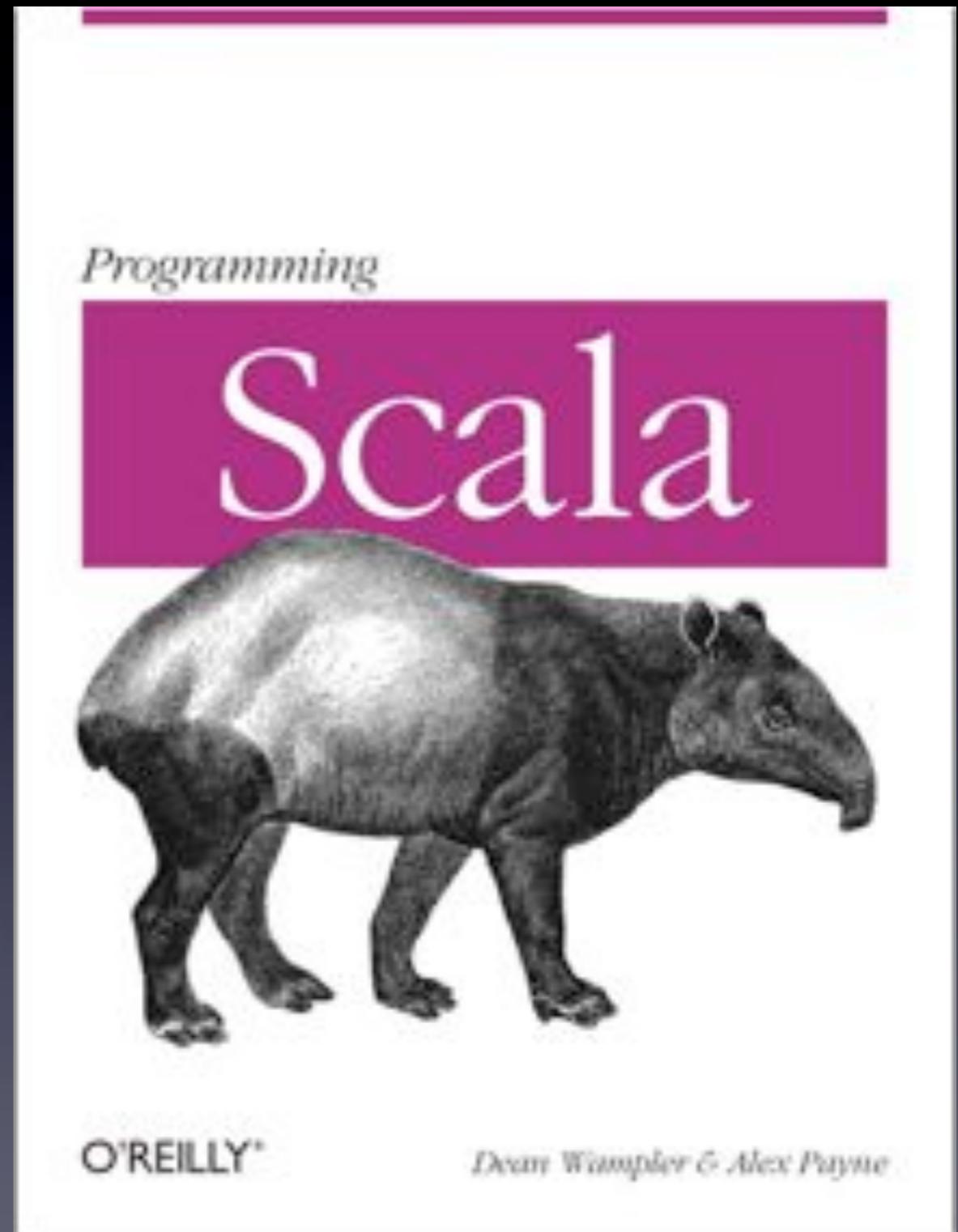
@deanwampler

polyglotprogramming.com/papers

<shameless-plug/>

Co-author,
*Programming
Scala*

programmingscala.com



Outline

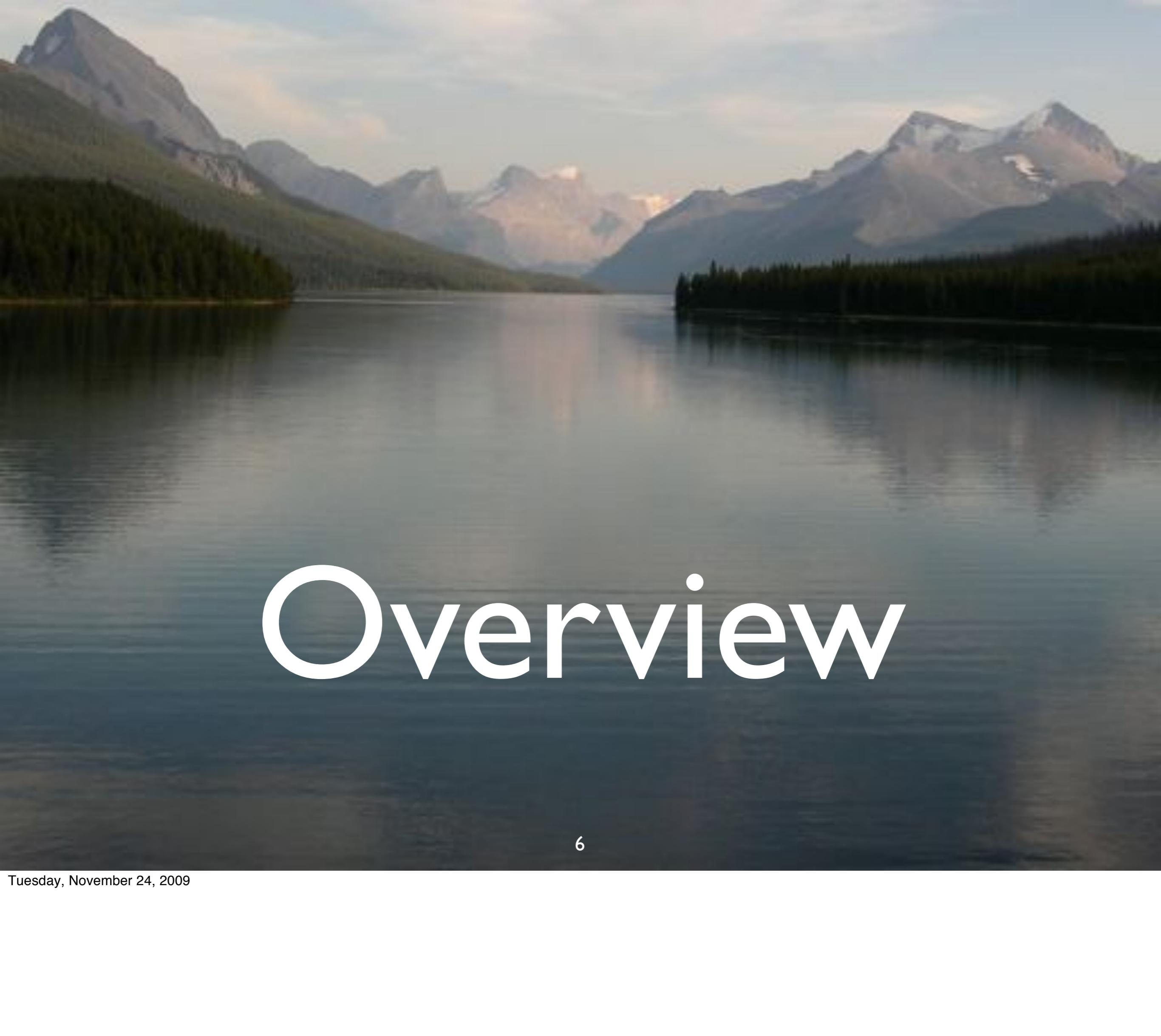
- Overview.
- Scala as an *OO* language.
- Scala as a *functional* language.
- *DSLs* in Scala.
- Recap.

Install Java & Scala for the Exercises

- [java.com/en/download/
manual.jsp](http://java.com/en/download/manual.jsp)
- scala-lang.org/downloads
 - v2.7.5 or better
 - v2.7.7 recommended

“Scaladocs”

- [http://www.scala-lang.org/
docu/files/api/index.html](http://www.scala-lang.org/docu/files/api/index.html)

A wide-angle photograph of a mountainous landscape. In the foreground, there is a calm lake reflecting the surrounding environment. On the left side of the lake, a dense forest of coniferous trees is visible. The background is dominated by a range of mountains, their peaks partially obscured by a hazy, warm-colored sky, likely from a sunset or sunrise. The overall atmosphere is serene and natural.

Overview

Why do we need a new language?

7

Tuesday, November 24, 2009

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...

#|

We need
Functional

Programming

...

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

#2

We need a better
Object Model

...

10

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

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.

Appealing if you like:

- *Rigor.*
- Deeply thought-through
principles.
- *Static* typing.

Not appealing if you find

- Rigor is *tedious*.
- Dynamic languages are *easier*.

Does dynamic typing *scale?*

Performance and application evolution

I think *performance*
issues will go away.

18

Tuesday, November 24, 2009

A lot of clever people are working on performance optimizations for dynamic languages (e.g., the JRuby team, the various Smalltalk and JavaScript VM's)

Harder question:

Should applications with *long lives* be written in *static* or *dynamic* languages?

20

Tuesday, November 24, 2009

It's not clear to me. The little decisions imposed by a static language "may" yield a better design in the long run. However, team quality and effective architecture decisions are ultimately the most important for success, using either kind of language.

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 sky filled with soft, pastel-colored clouds. The lighting suggests either sunrise or sunset, casting a warm glow over the entire scene.

Everything is an Object

21

Tuesday, November 24, 2009

There are no primitives and even "functions" are actually objects.

Succinct Code

```
$ scala  
Welcome to Scala version 2.7.7 ...
```

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

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

Method Names

Almost any
character allowed

pseudo operator overloading.

Pseudo operator overloading

"hello" + "world"

same as

"hello".+("world")

“Infix” operator notation

Not limited to “operators”

```
“one”.compareTo(“two”)  
// => -5
```

```
“one” compareTo “two”  
// => -5
```

Equivalent

Type Inference

Department of Redundancy Department

Typing in Java

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

Typing in Java

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

Tedious, redundant, and noisy!

Type *inferencing* in Scala

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

Only say things once

Read-only “variable”

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

No () needed

name:type syntax

“parameterized types”
use “[...], not <...>”

... and optional semicolons!

31

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

```
val persons2  
= new HashMap[String, Person]
```

type “annotation”
not needed

Initialize with a value; no type annotation required

```
val name = "Dean Wampler"
```

```
var count = 0
```

Read-write variable

Succinct Types

34

Tuesday, November 24, 2009

Type inference makes code more succinct. So does some other syntax features that give Scala code the succinct flavor of code in dynamic languages.

```
class Person {  
    private String firstName;  
    private String lastName;  
    private int age;  
  
    public Person(String firstName, String lastName, int age){  
        this.firstName = firstName;  
        this.lastName = lastName;  
        this.age = age;  
    }  
  
    public void String getFirstName() {return this.firstName;}  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public void String getLastname() {return this.lastName;}  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public void int getAge() {return this.age;}  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

Typical Java

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

Typical Scala!

*Class body is the
“primary” constructor*

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

*Makes the arg a field
with accessors*

Parameter list for c'tor

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

```
class Person(  
    fn: String, ...) {  
    private var fName:String = fn  
  
    def firstName:String = {  
        fName  
    }  
    def firstName_=(  
        fn: String):Unit = {  
        fName = fn  
    }  
    ...
```

What the compiler generates

```

class Person(
    fn: String, ...) {
    private var fName:String = fn
}

def firstName:String = {
    fName
}

def firstName_=(fn: String):Unit = {
    fName = fn
}

```

The code is annotated with several callout boxes:

- A box labeled "field" with an arrow pointing to the private field `fName`.
- A box labeled "no ()!" with an arrow pointing to the empty parentheses in the getter `firstName`.
- A box labeled "no ‘return’" with an arrow pointing to the lack of a `return` statement in the getter `firstName`.
- A box labeled "p.firstName = ‘...’" with an arrow pointing to the parameterless setter `firstName_=`.
- A box labeled "like ‘void’" with an arrow pointing to the return type `Unit` in the setter `firstName_=`.

...

39

Tuesday, November 24, 2009

Note that the c'tor parameters and private fields can't have the same names as the methods. Note that the reader has no parentheses. Reader/writer don't start with "get/set". "Unit" is roughly equivalent to "void" in Java. Why don't reader/writer methods start with "get/set"? We'll see in a few slides.

```
class Person(  
    fn: String, ...) {  
    private var fName: String = fn
```

```
def firstName: String = {  
    fName
```

```
}
```

```
def firstName = (  
    fn: String): Unit = {  
    fName = fn
```

```
}
```

```
...
```

We can write this more succinctly

```
class Person(  
    fn: String, ...) {  
    private var fName = fn
```

```
    def firstName = {}  
    fName  
}  
  
    def firstName_=(  
        fn: String) = {}  
        fName = fn  
}  
...
```

The return types can be inferred...

```
class Person(  
    fn: String, ...) {  
    private var fName = fn
```

```
    def firstName = fName
```

```
    def firstName_=(  
        fn: String) = fName = fn
```

```
...
```

... methods w/ one-expression don't need {}

Why not the
JavaBeans
getter/setter
convention?

```
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...

Tuples

46

Tuesday, November 24, 2009

A useful data type we want to introduce now...

```
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)
```

Literal syntax

```
val pair = (123, "Dean W.")  
// of type Tuple2[Int, String]
```

Exercise

Create an Address class.



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**.

50

Primary and Secondary Constructors

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

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

*pass parameters to
parent constructor*

User-defined Operators

55

```
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”

```
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”

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

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

}
```

“Operator overloading”

Sugar for “unary minus”

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

*required when overriding
concrete method*

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

“Operator overloading”

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

60

Exercise

Create a Rational Number class.



A Rational Class

- Create an *immutable* Rational Number 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

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,             ← selective  
        Reader => JReader,  
        Writer => JWriter,  
        _}                            ↑  
    ...                            ← suppress  
}
```

everything else...

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?

68

Tuesday, November 24, 2009

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?

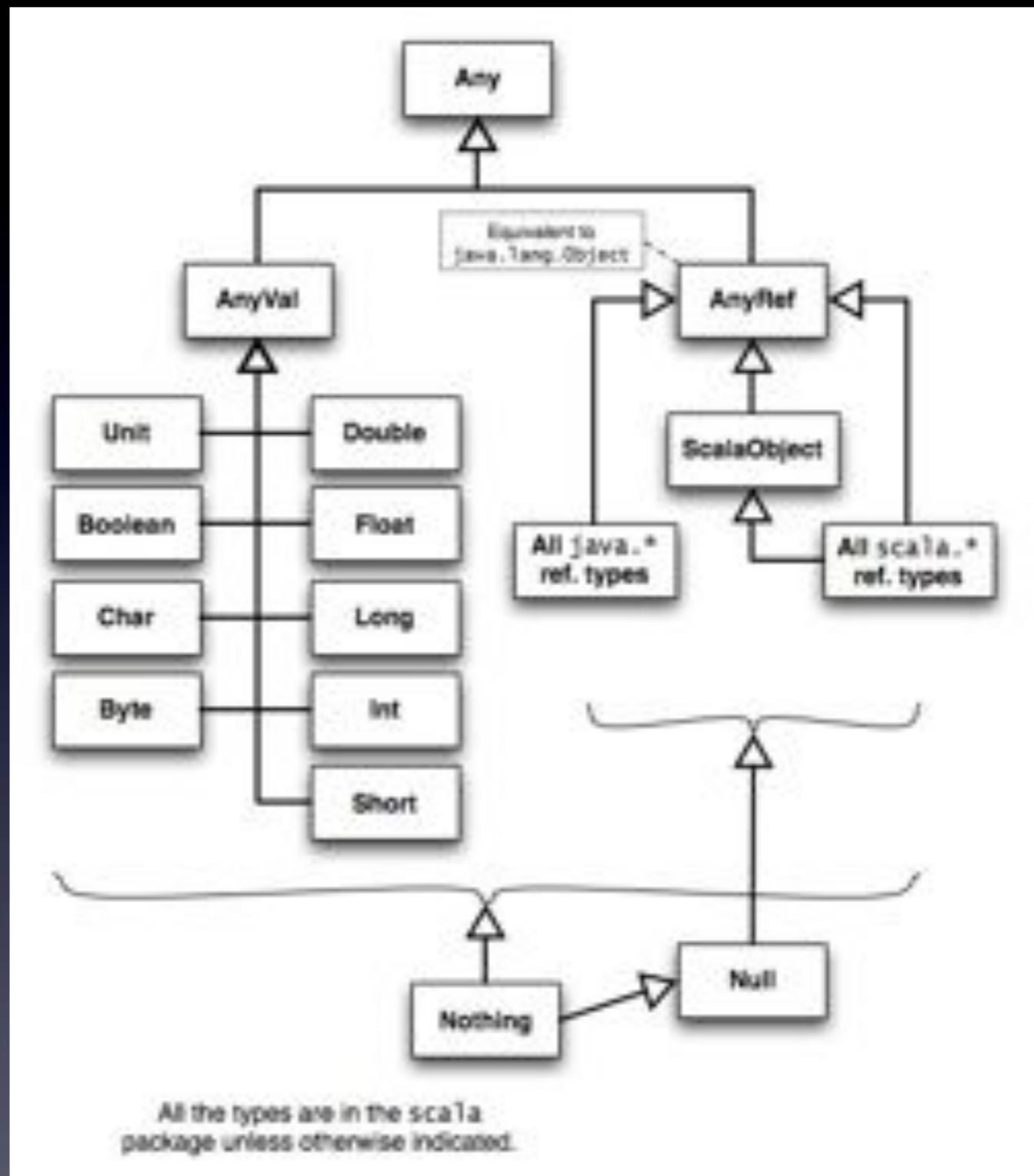
*File and type
names
don't have to
match*

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

The Type Hierarchy

71

Root of the Type Hierarchy



72

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

“AnyRef” is equivalent to Java’s Object

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

Scala “encourages” using immutables

<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

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

Type “variance
annotations”

What Does [+A] Mean?

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

```
new List[String]
```

is a subclass of

```
new List[AnyRef]
```

Covariant

Example

```
def printClass(l: List[AnyRef]) =  
{  
    l.foreach(x =>  
        println(x.getClass))  
}
```

```
printClass("a" :: "b" :: Nil)  
printClass(1 :: 2 :: Nil)
```

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

What Does [-A] Mean?

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

Function2[AnyRef, AnyRef, String]

is a subclass of

Function2[String, String, AnyRef]

Contravariant

By the way...

(AnyRef ,AnyRef) \Rightarrow String

is equivalent to

Function2[AnyRef ,AnyRef ,String]

*“Function Literal”
syntax*

Example

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

Function argument

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

*Now using
Function literal...*

Function literal (value)

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*

What about the return?

```
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*

86

Recap: What [-A] Means

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

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

is a subclass of

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

Contravariant

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...

...Java has
call-site
inheritance.

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.

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

Exercise

Experiment with mapper.



Mapper

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

Implicit Conversions

You write:

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

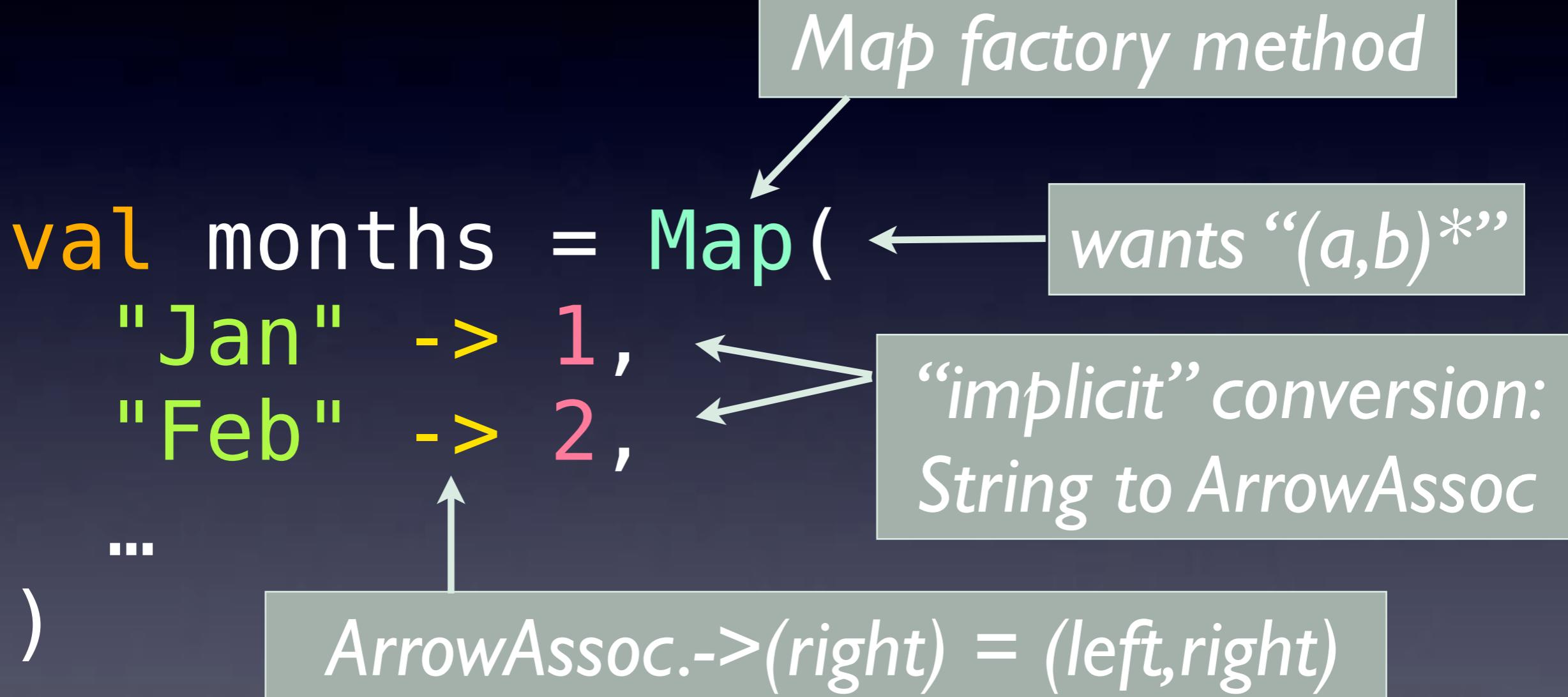
Part of the language grammar??

98

Tuesday, November 24, 2009

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



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

ArrowAssoc

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

...
*The “->”
method*

*Returns a
tuple “(x,y)”*

101

Implicit Conversions

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

102

Exercise

Implicit conversion methods.



103

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).

Traits

Composable Units of Behavior

107

Java

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

Java's object model

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

Traits

Like interfaces with
implementations,

Traits

... or like

*abstract classes +
multiple inheritance
(if you prefer).*

III

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*

115

```
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

116

Tuesday, November 24, 2009

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

117

Tuesday, November 24, 2009

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*.

Stackable Traits

119

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
}
```

120

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

123

```
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

Filtering occurred before logging

Example use

124

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

128

Method Lookup Order

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

Simpler cases, only...

129

Note: traits can't
have constructors.

Must initialize fields other ways.

130

Tuesday, November 24, 2009

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 are a powerful
mixin-based
composition
mechanism!

Exercise

Traits as Mixins



| 32

Traits

- Define a `QueueMultiplier` trait that puts the 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
}
```

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 sky filled with soft, warm-colored clouds. The overall atmosphere is serene and natural.

Everything is a Function (?)

135

Objects as Functions

136

Recall

```
trait Function1[-A,+R]  
  
def mapper(l: List[AnyRef],  
          f: (AnyRef) => Any) = {  
    l.map(f(_))  
}  
mapper("a" :: "b" :: Nil,  
      (x: AnyRef) => x.getClass)
```

Function “objects”

How can we
use an object
like a *function*:
 $f(x)$?

138

How the Sausage is...

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

*No method body:
=> abstract*

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

140

Tuesday, November 24, 2009

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...

What the Compiler Does

(x:AnyRef) => x.getClass

What you write.

```
new Function1[-A,+R] {  
    def apply(a:A) = x.getClass  
}
```

What the compiler generates

An anonymous class

|4|

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

becomes:

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

Functional Programming

143

Tuesday, November 24, 2009

We've danced around the idea of FP, now let's see what it really means.

What is Functional Programming?

Don't we already write “functions”?

$y = \sin(x)$

Based on *Mathematics*

|45

Tuesday, November 24, 2009

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

$$y = \sin(x)$$

Setting x fixes y

\therefore variables are *immutable*

20 + = | ??

We never *modify*
the 20 “object”

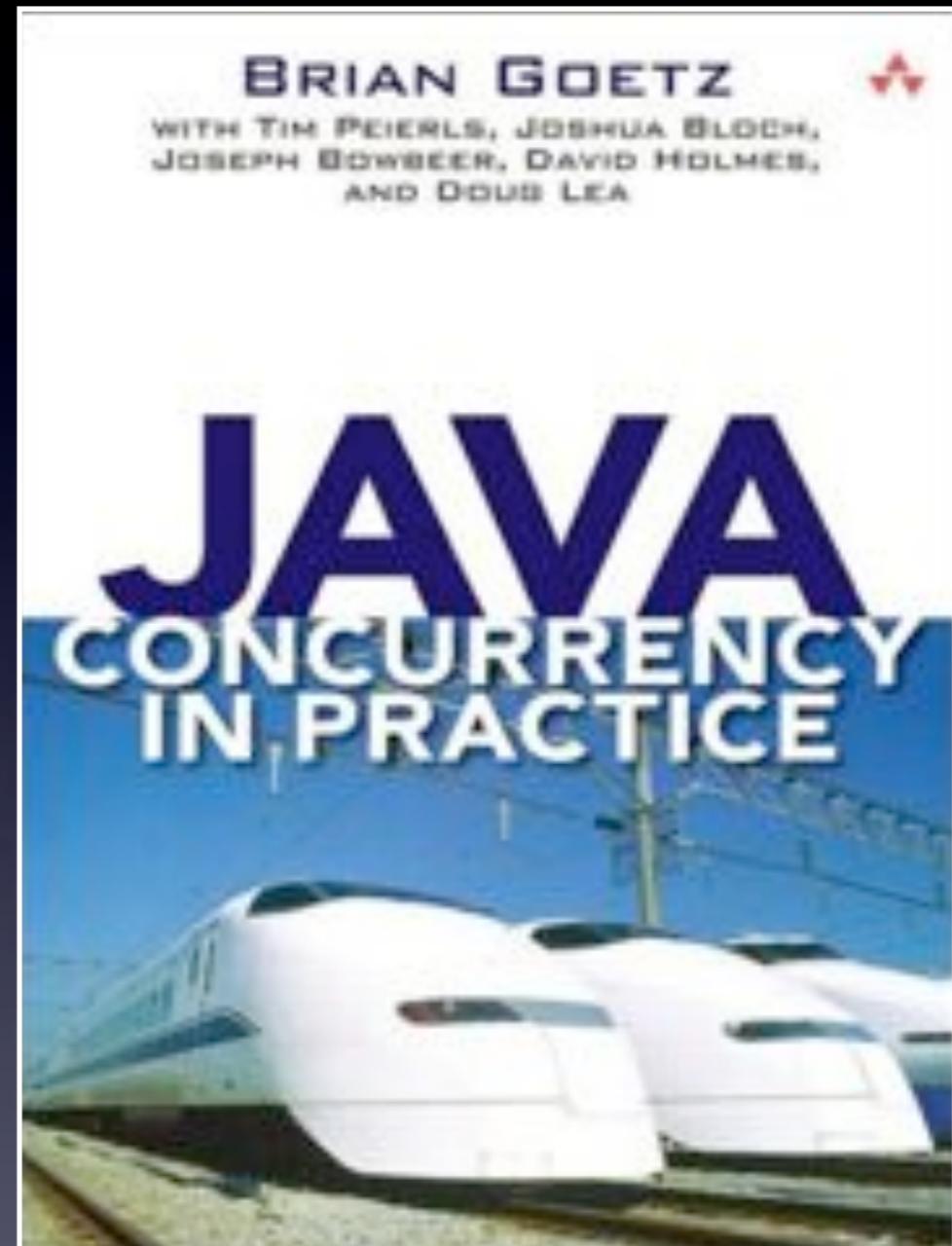
Concurrency

No *mutable state*

∴ *nothing to synchronize*

When you share mutable state...

Hic sunt dracones
(Here be dragons)



Immutable data

- *Good*
 - *Safer concurrency.*
 - *Safer to share with clients.*
- *Bad*
 - *Copying overhead.*

150

Tuesday, November 24, 2009

You don't worry about handing an immutable object to a client, because it won't get modified (messed up).

Downside: increased overhead of copying and potential stack overflows for recursion (which becomes necessary in some cases when you can't modify loop counters, for example).

$$y = \sin(x)$$

Functions don't

change state

∴ *side-effect* free

Side-effect free functions

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

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

*Compose functions of
other functions*

∴ first-class citizens

Fibonacci Sequence

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

154

Tuesday, November 24, 2009

A familiar example of a recursive mathematical function.

Notice the declarative nature; we're not telling you how to calculate, per se. We're just telling you the relationships between values.

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

155

Tuesday, November 24, 2009

If you can't increment a loop counter (because it wouldn't be immutable!), then one way to do iteration is recursion.

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

156

Tuesday, November 24, 2009

Because math is inherently declarative, rather than imperative, FP tries to copy this quality as well. It yields very succinct code and it gives the implementer (language runtime, API implementer, etc.) more freedom to optimize the implementation.

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

$$y = \exp(pi^*x)$$

Properties of Types
are very important

Contrasting Approaches

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

Functional vs. Object- Oriented Programming

159

Scala's Thesis: FP *Complements* OOP

Despite surface contradictions...

160

Tuesday, November 24, 2009

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.

Objects are *functions*
(or can be).

Functions are *objects*.

Observation:

Any object graph
decomposes into
values and *collections*.

Collections:

Construct, iterate,
manage them
functionally.

Values:

Carefully specify
the *properties*
of value *types*.

What are the *properties* of

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

Prefer *immutable objects.*

166

Tuesday, November 24, 2009

Immutability makes concurrency far easier to implement reliably. It also makes equals and hashCode more reliable, etc.

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

Functions should be first class.

For composability

168

Tuesday, November 24, 2009

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.

Companion Objects

169

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

The diagram illustrates the relationship between a Scala class and its companion object. A green box labeled "Singleton" has an arrow pointing to the word "Complex" inside a pink-bordered box. Another green box labeled "Factory" method has an arrow pointing to the "apply" method definition. A third green box labeled "Companions" is located at the bottom right.

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

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

We had “new Complex(...)” before

171

apply can be an
instance method, too.

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

“function object”

173

Tuesday, November 24, 2009

Not just companion objects can define “apply”. usually, an “apply” instance method is not necessarily used as a factory (but can be...).

If any object
is followed by a
parameter list,
apply is called

Case Classes

175

Recall:

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

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

This pattern is so common...

Equivalent:

```
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(k->v,...)?

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

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

0 to N tuples

Exercise

Case Classes



180

Case Classes

- Convert your previous `Rational` class to a **case class**.
- Create objects using `Rational.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

185

Tuesday, November 24, 2009

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

Ist, More About Lists

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

The diagram illustrates the definition of the variable `list`. It shows the code `val list = 1 :: 2 :: 3 :: 4 :: 5 :: Nil`. A pink rounded rectangle highlights the `::` operator between 3 and 4. An arrow points from the word "cons" to this highlighted operator, indicating its meaning. Another arrow points from the text "empty list" to the `Nil` symbol at the end of the list.

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

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

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

Wildcard: match
any type

Return type
required

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

pattern match

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

2nd, a Look at Option

sealed abstract class

Option[+A] { ... }

case final class

Some[+A](val a: A)

extends Option[A] { ... }

case object None

extends Option[Nothing] { ... }

“sealed” keyword

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

```
case final class Some[+A](val a: A)
  extends Option[A] {...}

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

only need one

```
val map = Map(  
  "c" -> Complex(1,2),  
  "t" -> (1.1,2.2))
```

```
println(map.get("c"))  
println(map.get("t"))  
println(map.get("a"))
```

```
// => Some(Complex(1.0,2.0))  
// => Some((1.1,2.2))  
// => None
```

```
trait Map[A,+B] {  
  ...  
  def get(key: A): Option[B]  
  ...  
}
```

Option
is better than
null.

194

Tuesday, November 24, 2009

When the type is “Option”, it tells the reader explicitly that there may be no value, unlike null. It forces the user to extract the value from the “Some”, when there is a value, as in our “match” example previously.

Back to
Pattern Matching...

```
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)
```

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

```
list foreach { _ match {  
  case Complex(1,i) => NOT needed  
  case Complex(r,i) => // #2  
  case (3,2,_ ) => // #3  
  case (a,b,c) => // #4  
  case Nil => // #5  
}} //=> #1, #2, #4, #3 (no Nil)
```

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

```
list foreach {  
  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)
```

```

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

list foreach {
  case Complex(1,i) => println("i="+i)
  case Complex(r,i) => println(r+", "+i)
  case (3,2,_ ) => println("(3,2,?)")
  case (a,b,c) =>
    println(("("+a+"," +b+"," +c+")"))
  case Nil => // #5 (nothing)
}

=> i=2.0
=> 1.1,2.2
=> (1,2,3)
=> (3,2,?)

```

```
val map = Map(  
  "c" -> Complex(1,2),  
  "t" -> (1.1,2.2))
```

```
List("c","t","a") foreach {k =>  
  println(map.get(k) match {  
    case Some((a,b)) => a+":" + b  
    case Some(x) => x  
    case None => "None: " + k  
  })}  
// => (1.0, 2.0)  
// => 1.1:2.2  
// => None: a
```

*Pattern matching
with Options*

200

Tuesday, November 24, 2009

Map stores all values in a Some, so that it can return a None if you specify a key with no value. Note how the API emphasizes the possibility of “nothing” being there, vs. risking a “null”. The pattern matching makes it easy to get the wrapped value, too. Note we can match on Some wrapping a specific type or Some wrapping anything.

How does this work?

The *unapply* method.

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

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

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

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

Unapply is also
generated for a
case class.

```
case class  
  Complex(real: Double,  
          imag: Double)  
{  
  def +(...) = ...  
  def -(...) = ...  
  def unary_- = ...  
  override def toString() = ...  
}
```

No companion object *Complex* needed!

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`.

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

Pattern matching
is used in *FP*
like *polymorphism*
is used in *OOP*.

Exercise

Pattern matching



209

Pattern Matching

- Using your **Rational** case class, complete the code on the next page to match separately on **Rationals** with 1.0 as the denominator, all other **Rationals**, and all other cases.
- Call **println** with a separate message for each.
- Extra: extract the **Some** values.

```
val l = List(  
    Rational(2, 2),  
    Rational(3, 1),  
    Some("hello!"), (1, 2, 3), None)  
  
l foreach {  
    // case statements  
}
```

Currying

212

Tuesday, November 24, 2009

Not just a feature of Asian cuisine...

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

Named after Haskell Curry

213

Tuesday, November 24, 2009

Haskell Curry was a pioneer in mathematical Combinatory Logic.

```
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.

214

Tuesday, November 24, 2009

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.

```
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

216

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)
```

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??

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 Numberator {

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

}
```

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

_ like * in Java

“singleton” class == 1 object

```
object Numberator {
```

loop “control”

two parameters

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

function taking line # and line

like “void”

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

```
object Numberator {
```

two parameters

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

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

```
object Numberator {
```

two parameters lists

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

Why 2 Param. Lists?

```
// Print with line numbers.  
import Numberator.loop ← import  
loop (new File("...")) {} ← 1st param.:  
  (n, line) => a file  
    format("%3d: %s\n", n, line)  
}
```

2nd parameter: a “function literal”

```
object Numberator {  
    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...

229

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

recursive

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

Finishing Numberator...

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*

232

Exercise

A whileTrue loop



233

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 Parameters

```
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*

Recursion

239

Recall

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

Tail recursion

240

Contrast with
non-tail call
recursion...

Fibonacci Sequence

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

242

Tuesday, November 24, 2009

A familiar example of a recursive mathematical function.

Notice the declarative nature; we're not telling you how to calculate, per se. We're just telling you the relationships between values.

```

object Fib {
  def apply(n: Int): Int =
    n match {
      case 0 => 0
      case 1 => 1
      case _ => Fib(n-1)+Fib(n-2)
    }
}

```

default: matches anything

“+” after Fib calls

Not tail recursive

243

Tuesday, November 24, 2009

Using an object is convenient. It is thread-safe and stateless, so we don't really need instances.
Can you write a tail-recursive implementation of Fib.apply?

Fib is side-effect free.
But, the
implementation
could cache results...

*Contrast external API “model”
with internal implementation.*

244

Tuesday, November 24, 2009

The expected external API is that this is a purely side-effect function, with all the benefits they provide. However, the implementer might do hidden side-effects, like caching previously requested numbers. The implementer has to preserve the external semantics...

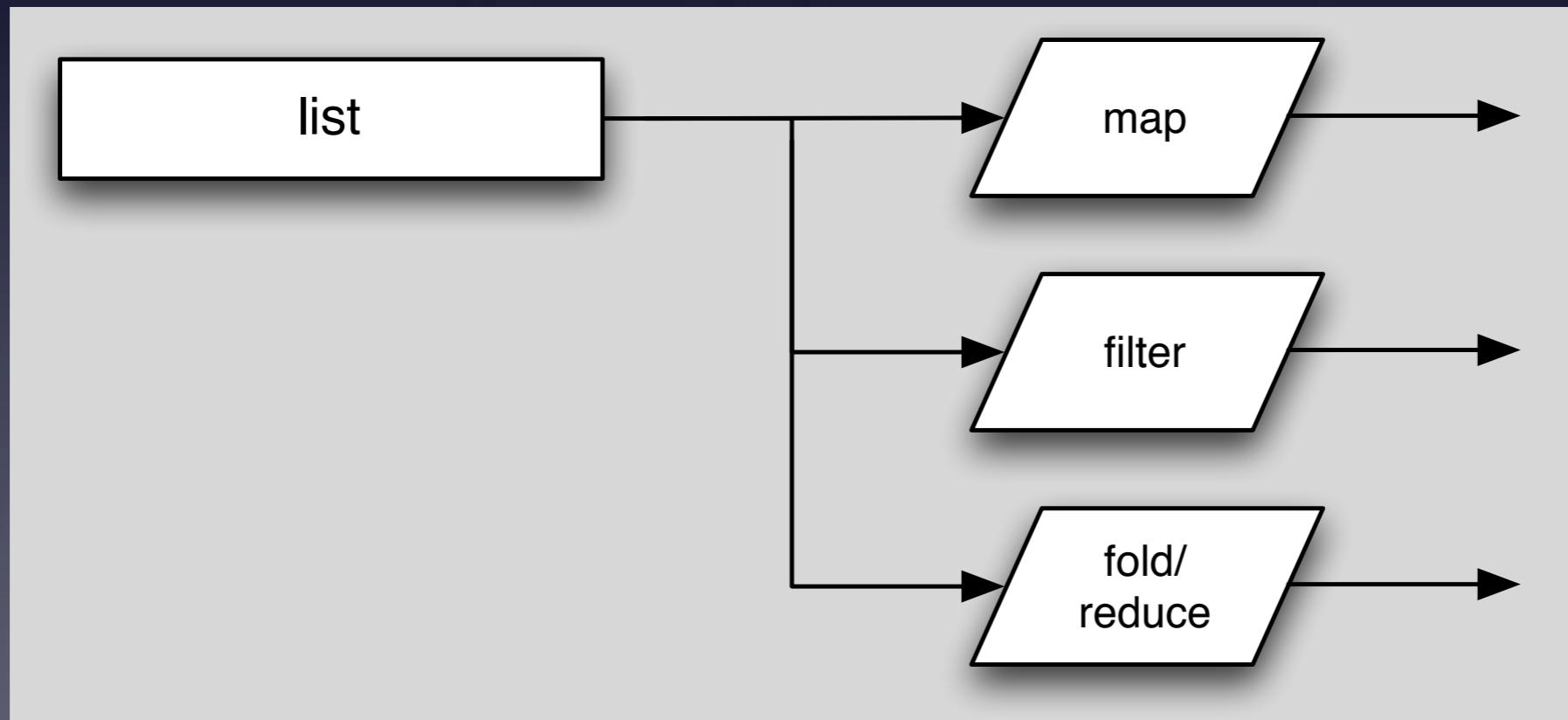
More on Avoiding *Mutable* State...

Chaining collection
operations that
handle iteration
for us.

Avoiding loop counters

246

Classic Operations on *Functional Data Types*

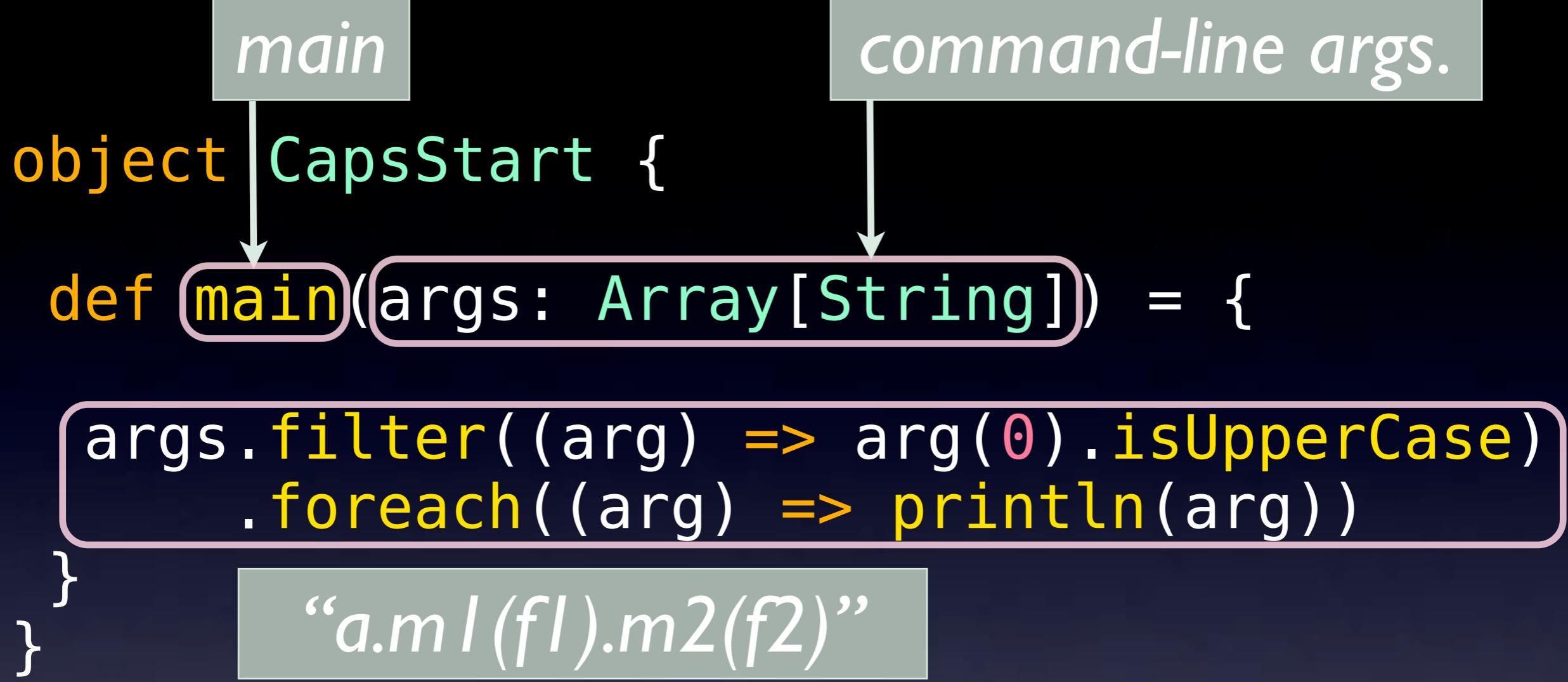


247

```
object CapsStart {  
  def main(args: Array[String]) = {  
    args.filter((arg) => arg(0).isUpperCase)  
      .foreach((arg) => println(arg))  
  }  
}
```

*Application to filter out strings that
don't start with a capital letter.*

248



```

// $ scalac CapsStart.scala
// $ scala -cp . CapsStart aB Ab AB ab
// Ab
// AB

```

Exercise

Experiment with collection operations



250

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)

252

```

object CapsStartFor {

  def main(args: Array[String]) = {
    for (
      i <- 0 until args.length;
      arg = args(i);
      if (arg(0).isUpperCase)
    )
      println(arg)
  }
}

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

```

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

Tuesday, November 24, 2009

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 ()  
            i <- 0 until args.length;  
            arg = args(i);  
            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 {  
            i <- 0 until args.length  
            arg = args(i)  
            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 {
        i <- 0 until args.length
        arg = args(i)
        if (arg(0).isUpperCase)
    }
        yield arg
    println(capList)
}
}

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

```

“println” is outside loop

“yield” to create a list

Exercise

Creating a list of tuples using a
for comprehension



List (i,j,k) Tuples Where:

- $i \geq j \geq k$
- $(i + j + k) \% 3 == 0$

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

```
val list = for {  
    ??  
} yield ((i, j, k))  
println(list)  
  
// => RangeG((1,1,1), (2,2,2),  
(3,2,1), (3,3,3), ...
```

Robust Concurrency: Actors

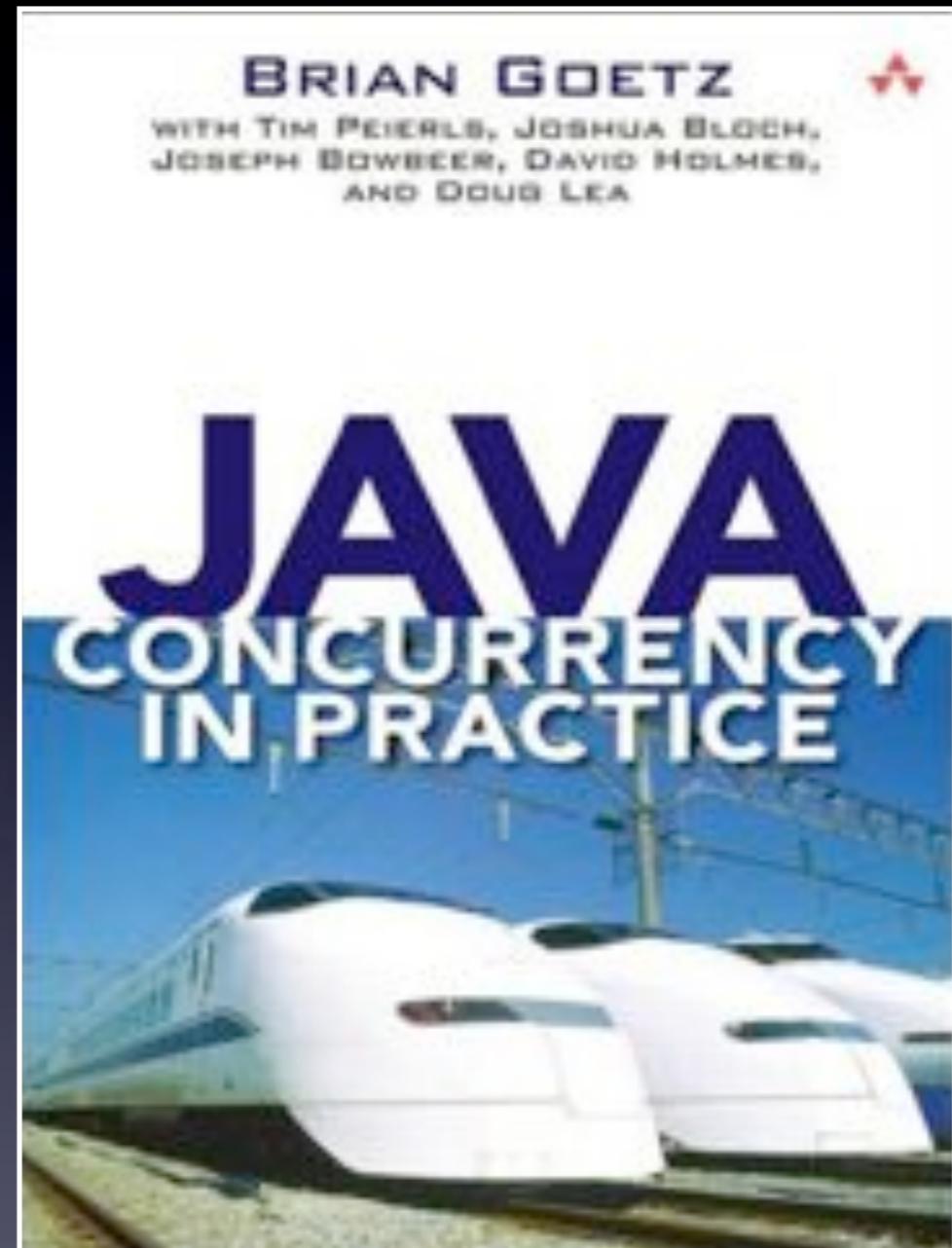
260

Tuesday, November 24, 2009

One abstraction (one of several good ones...) for concurrency above the level of multithreaded programming.

When you share mutable state...

Hic sunt dracones
(Here be dragons)



Hard!

Actor Model

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

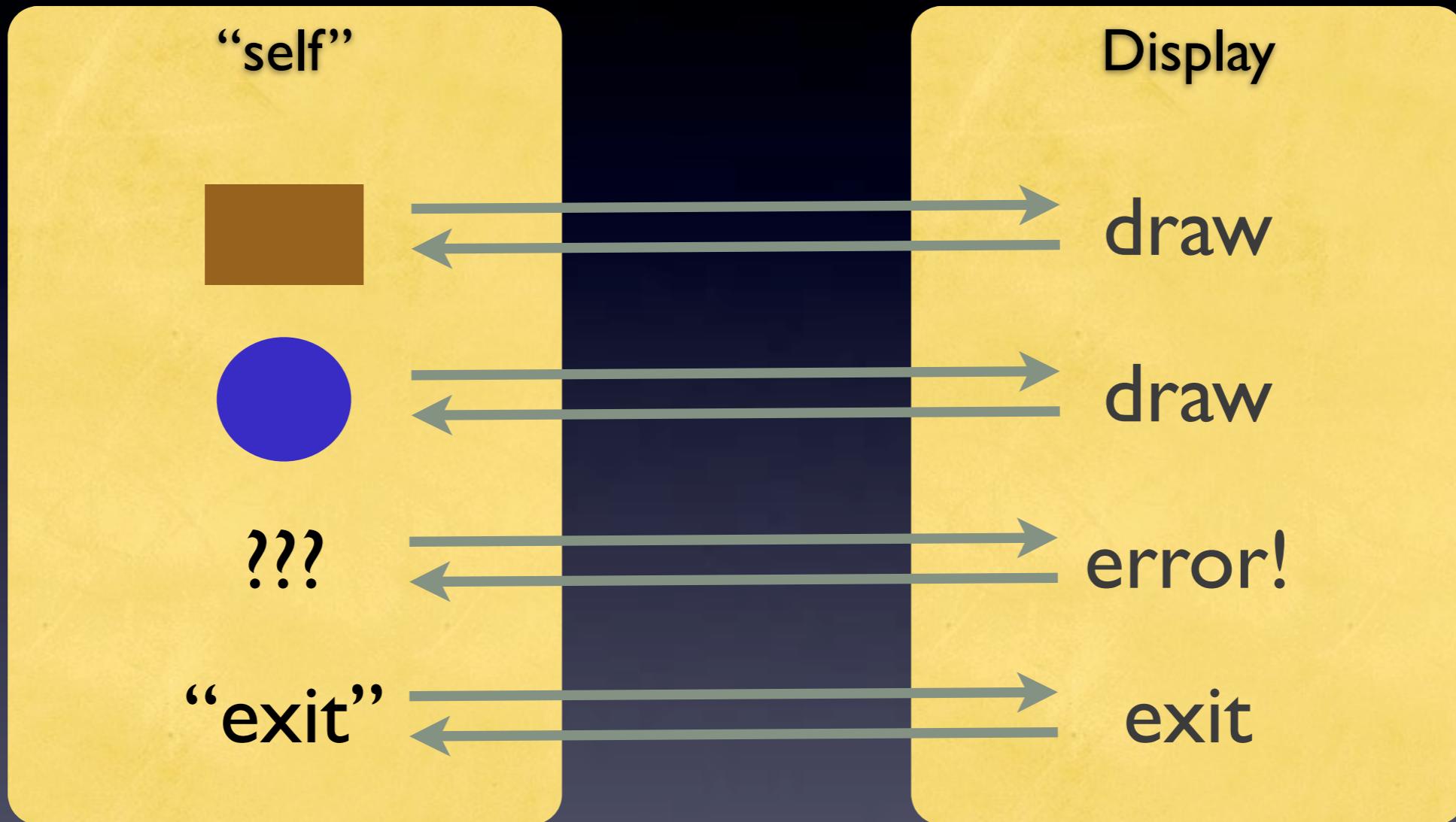
Actor Model

- First developed in the 70's-80's by Hewitt, Agha, Hoare, etc.
- Made “famous” by *Erlang*.

Scala's Actor Model

- Patterned after Erlang's.
- Allows *shared, mutable state*.
 - But *discouraged*.

2 Actors:



```
package shapes
```

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

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

Hierarchy of geometric shapes

Tuesday, November 24, 2009

“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

267

Tuesday, November 24, 2009

“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

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

Actor for drawing shapes

269

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

Actor library

new Actor

loop indefinitely

receive and handle each message

Actor for drawing shapes

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

```

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*

```

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

```
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

```
import shapes._  
import scala.actors.Actor._  
  
def sendAndReceive(msg: Any) = {  
    ShapeDrawingActor ! msg  
  
    self.receive {  
        case reply => println(reply)  
    }  
}
```

script to try it out

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

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

send message...

... and wait for a reply

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

script to try it out

```
...
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!
```

```
...
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() ←————— polymorphism  
    sender ! "drawn"  
}  
  
case ...  
case ...
```

pattern matching

polymorphism

A powerful combination!

Exercise

Working with Actors.



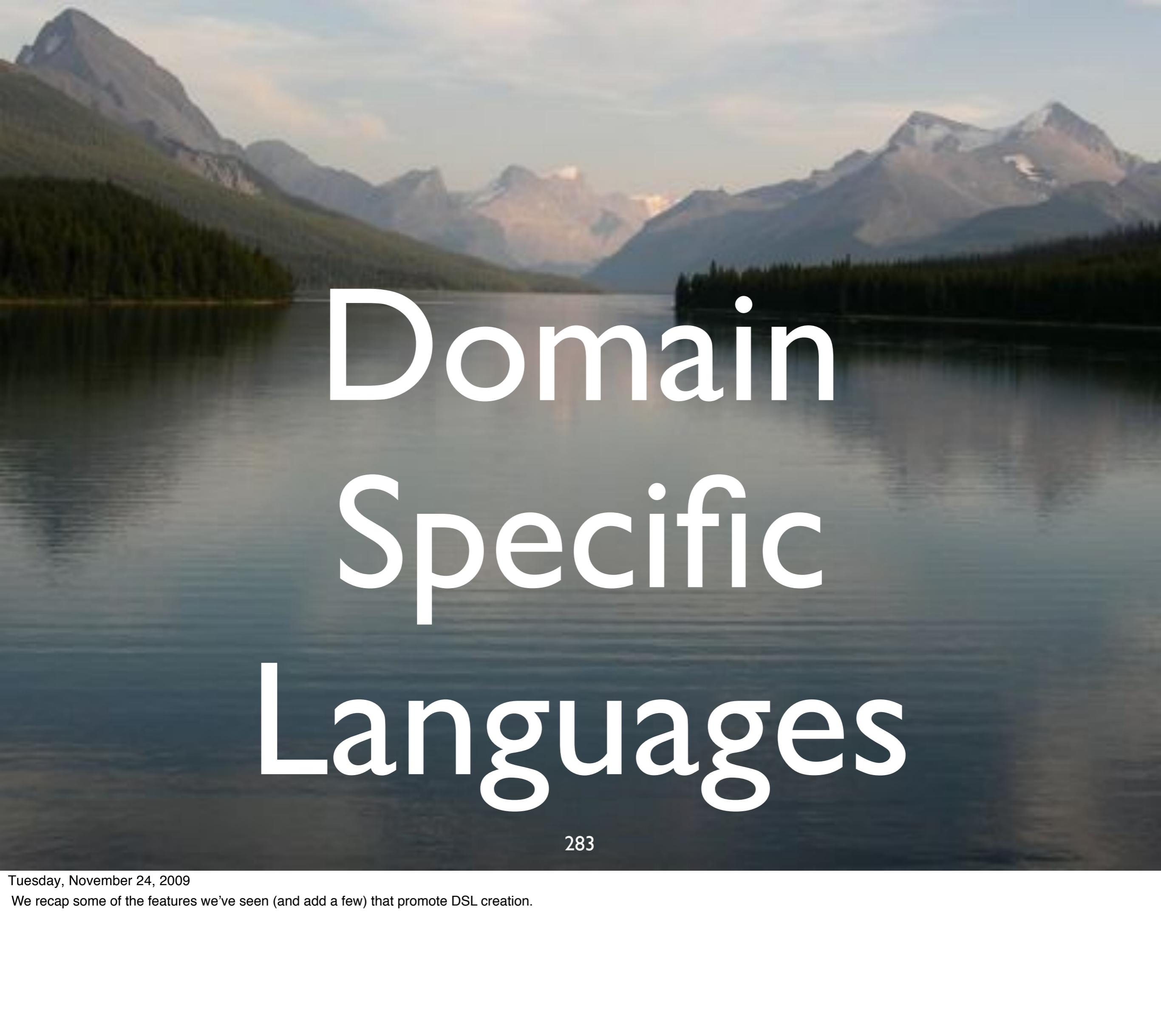
280

Enhance the Actor Example

- Change `ShapeDrawingActor` to reply with a modified version of the received shape.
- Change the “script” to loop forever, receiving modified shapes from `ShapeDrawingActor` and sending new shapes back to it.

Enhance the Actor Example

- Put in a **sleep** or other delay to slow it down.
- Change the shape hierarchy.
- Add more “control” messages.



Domain Specific Languages

283

Tuesday, November 24, 2009

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

Internal DSLs

284

Features for Building *Internal* DSLs

- Infix operation notation.
- Implicit type converters.
- User-defined “controls”.
- First-class functions.

Infix operator notation

"hello" + "world"

same as

"hello".+("world")

Implicit Type Converters

1 hour fromNow

What I want

*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 as on previous
slides. 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  
}
```

“by-name”
parameter

Called w/out
parentheses

External DSLs

293

Features for Building *External* DSLs

- Parser Combinator Library

Consider this DSL

```
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

300

Recap



301

Scala is...

302

a better Java and C#,

303

*object-oriented
and
functional,*

*succinct,
elegant,
yet
powerful.*

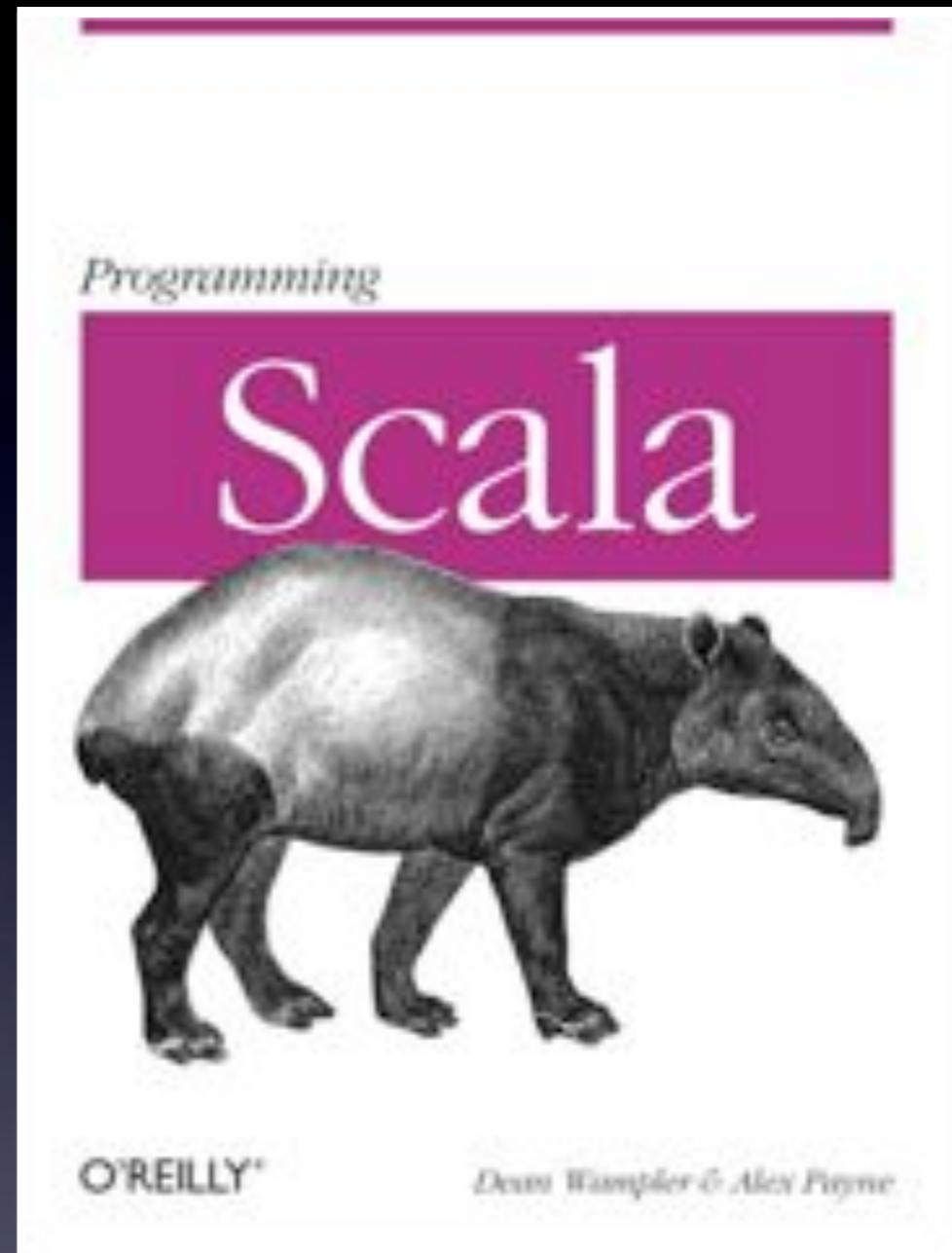
305

Thanks!

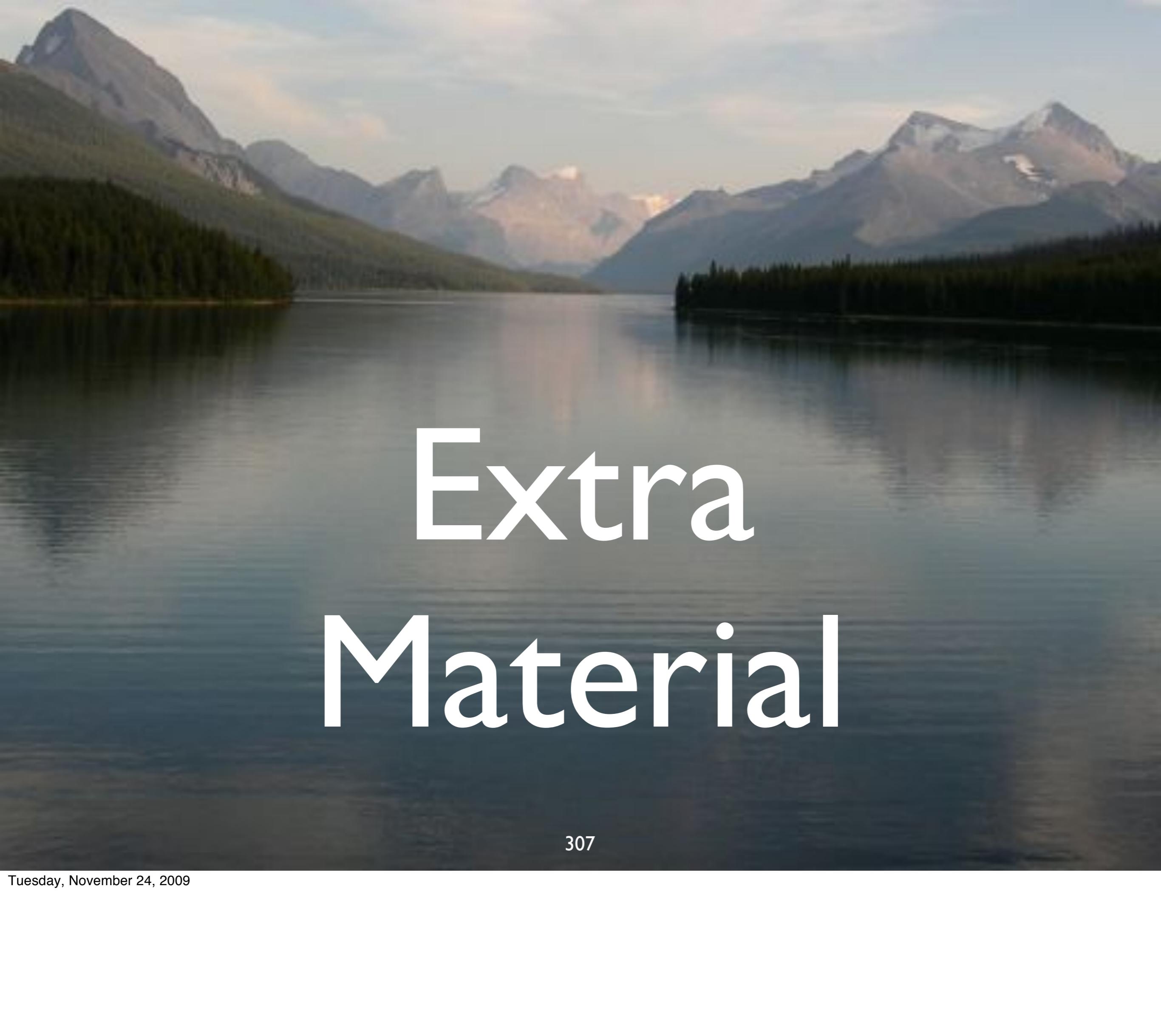
dean@deanwampler.com

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

programmingscala.com
polyglotprogramming.com/talks



306



Extra Material

307

Structural Types

308

A type-safe form of *duck typing.*

309

Tuesday, November 24, 2009

Duck typing is the term used in dynamically-typed languages like Ruby for the ability to simply call a method on an object and not care how the call is resolved.

```
trait Subject {  
    type Observer = {  
        def update(subject: AnyRef)  
    }  
}
```

Any type that has this method.

```
private var observers = List[Observer]()  
  
def addObserver(observer: Observer) =  
    observers ::= observer  
  
def notifyObservers =  
    observers foreach (_.update(this))  
}
```

Observer Pattern

310

Tuesday, November 24, 2009

I don't declare an Observer Trait (as you would in Java), but allow any type to be used as an observer, as long as it has the "update" method defined.
Note that we can declare "types", like member fields and methods.

“structural”
type

```
trait Subject {  
    type Observer = {  
        def update(subject: AnyRef)  
    }  
}
```

```
private var observers = List[Observer]()  
  
def addObserver(observer: Observer) =  
    observers ::= observer  
  
def notifyObservers =  
    observers foreach (_.update(this))  
}
```

Observer Pattern

311

Tuesday, November 24, 2009

I don't declare an Observer Trait (as you would in Java), but allow any type to be used as an observer, as long as it has the “update” method defined.
Note that we can declare “types”, like member fields and methods.

Components with Self Types

312

Build a simple Twitter client...

313

```
trait Tweeter {  
    def tweet(message: String)  
}
```

```
trait TwitterClientUIComponent{  
    val ui: TwitterClientUI  
  
    abstract class TwitterClientUI(  
        val client: Tweeter) {  
        def sendTweet(  
            msg:String) = client.tweet(msg)  
        def showTweet(msg: String): Unit  
    }  
}
```

```
// “Business” tier  
  
trait TwitterServiceComponent{  
    val service: TwitterService  
  
    trait TwitterService {  
        def sendTweet(msg:String): Boolean  
    }  
}
```

// Wiring it all together

```
trait TwitterClientComponent{
  self: TwitterClientComponent with
    TwitterServiceComponent =>
  val client: TwitterClient

  class TwitterClient(val user: String)
    extends Tweeter {
    def tweet(message: String) = {
      if (service.sendTweet(message)) {
        ui.showTweet(message)
      }
    }
  }
}
```



Scala on the VMs

317

Scala must generate
valid JVM/.NET
byte code.

```
public class Complex
  extends java.lang.Object
  implements scala.ScalaObject {
  private final double real; // read-only
  private final double imag; // read-only
  public double imag();
  public double real();
  public Complex(double, double);
  public Complex $plus(Complex); // "+"
  public Complex $minus(Complex); // "-"
  public Complex unary_$minus(); // "-C"
  ...
}
```

“javap -private Complex”

319

Tuesday, November 24, 2009

Decompiling Scala-generated byte code to see how it maps to valid names, etc.
Note that “Double” is mapped to primitive “double” and “\$plus” is used for “+”, etc.

```

public class Complex           uses Java primitives
  extends java.lang.Object
  implements scala.ScalaObject {
  private final double real; // read-only
  private final double imag; // read-only
  public double imag();
  public double real();
  public Complex(double, double);
  public Complex $plus(Complex); // "+"
  public Complex $minus(Complex); // "-"
  public Complex unary_$minus(); // "-C"
  ...
}

```

“operator” encoding

“javap -private Complex”

320

Recall: *Functions*
are *instances* of
Function N ,
where N is the *arity*.

Example

(i: Int, s: String) => Double

is an instance of

Function2[Int, String, Double]

It's easy
for Scala to *call* Java
and (mostly)
vice-versa.

323

Tuesday, November 24, 2009

Of the new JVM languages, Scala has the easiest integration with Java, with the possible exception of Groovy.

Scala vs. other languages

324

Scala vs. other JVM languages.

[http://blog.objectmentor.com/articles/2009/01/15/
adopting-new-jvm-languages-in-the-enterprise](http://blog.objectmentor.com/articles/2009/01/15/adopting-new-jvm-languages-in-the-enterprise)

Scala vs. JRuby

- ✓ Closeness to Java's *object model*.
- ✓ Calling from Java.
- ✓ Concurrency.

326

Tuesday, November 24, 2009

Check mark means Scala has the edge.

Invoking Java from Scala or JRuby is equally easy, but it's much easier to invoke Scala from Java than JRuby from Java. Ruby has a poor concurrency story, although there are Ruby actor libraries.

Scala vs. JRuby

- *Scripting.*
- Dynamic typing *hotness.*
- *Ruby on Rails.*

327

Tuesday, November 24, 2009

Minus sign means JRuby has the edge.

Scala has limited scripting support, but Ruby is much better.

There are times when dynamic languages are hard to beat, like interpreting code on the fly, self-modifying code, etc.

More or less
the same for
Groovy, Jython, ...

328

Tuesday, November 24, 2009

It's more or less the same comparison for other dynamically-typed "scripting" languages on the JVM.

Scala vs. Clojure

- ✓ Closeness to Java's *object model*.
- ✓ Easier for Java team to *understand*.

329

Tuesday, November 24, 2009

Clojure actually doesn't support OOP, except in an indirect way. It emphasizes FP. As a lisp, and because of it's innovative features, it will be harder for a typical Java team to learn and use.

Scala vs. Clojure

- Innovative approaches to:
 - Concurrency.
 - *Mutable* data.

330

Tuesday, November 24, 2009

But clojure is in many ways the most innovative language around, with pioneering ideas for handling concurrency and “principled” mutation of data.

Scala vs. Erlang

331

Tuesday, November 24, 2009

Erlang isn't a JVM language...

Scala vs. Erlang

✓ General purpose, all-in-one language.

- Extreme:
 - concurrency.
 - reliability.

332

Tuesday, November 24, 2009

Finally, outside the JVM, Erlang is outstanding for particular kinds of problems, especially when extremely high concurrency and reliability are needed. But Scala is a better general purpose language, most suitable when you can't create separate apps., some in Erlang, some in