

The Seductions of Scala

Dean Wampler

dean@concurrentthought.com

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

polyglotprogramming.com/talks

April 26, 2013



|

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

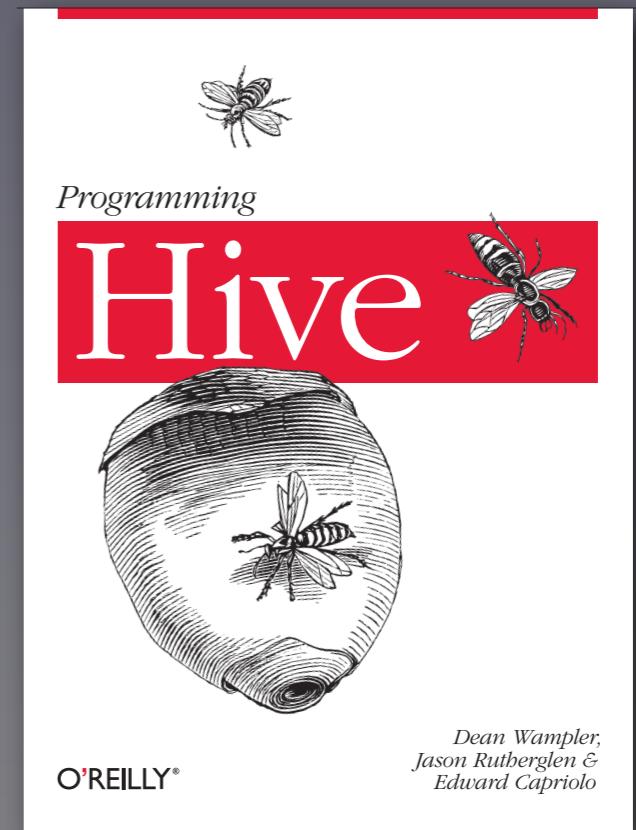
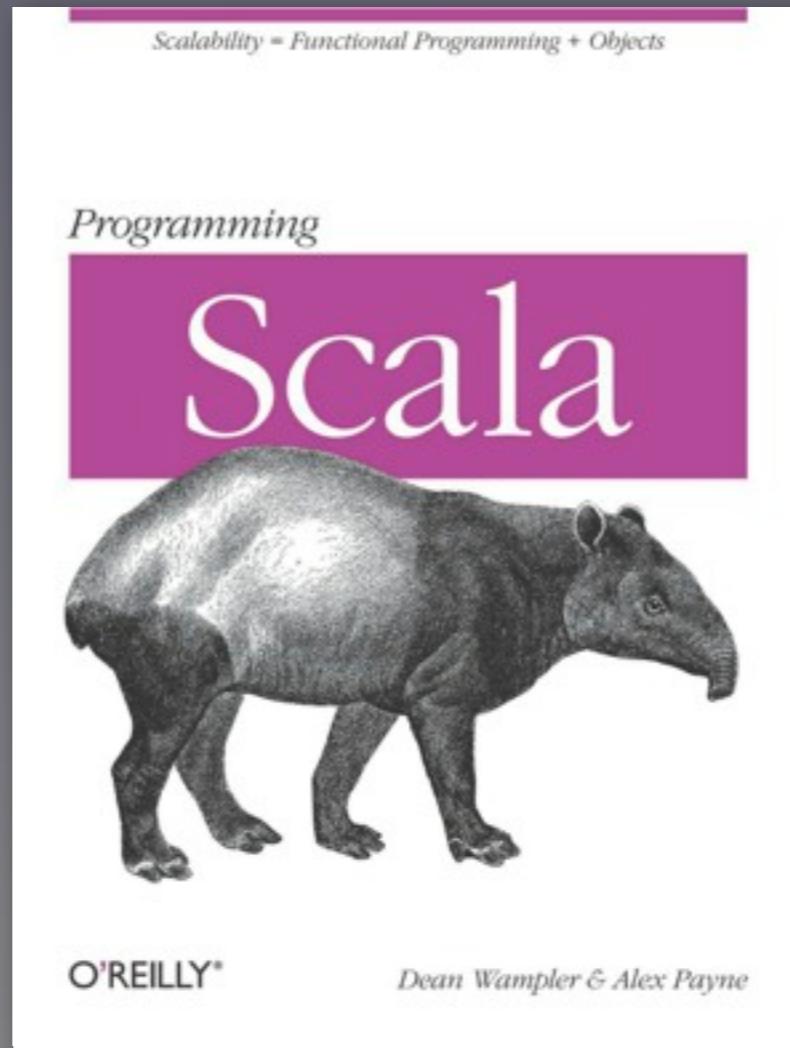
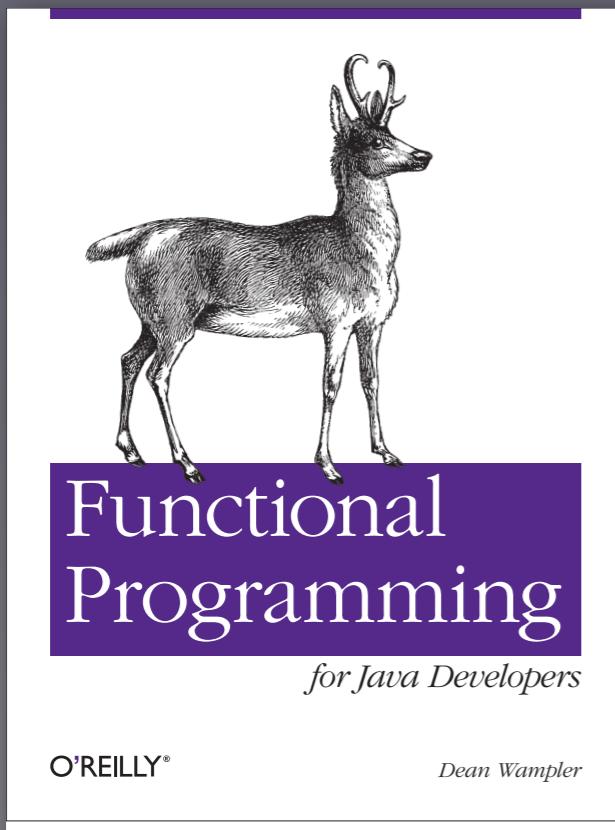
Thursday, April 25, 13

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

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.

<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

<shameless-plug/>



O'Reilly Author.

2

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Available now from oreilly.com, Amazon, etc.

You can read Programming Scala free online: Follow the links in programmingscala.com

<shameless-plug/>

I'm offering
Scala and Big Data
consulting, mentoring,
and training.

dean@concurrentthought.com

Why do we need a new language?

Thursday, April 25, 13

I picked Scala to learn in 2007 because I wanted to learn a functional language. Scala appealed because it runs on the JVM and interoperates with Java. In the end, I was seduced by its power and flexibility.

#1

We need Functional Programming

• • •

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

#2

We need a better
Object Model

...

... for composability.

... for scalable designs.

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

Despite surface contradictions...

#3

We need better
Productivity.

But we want to
keep our investment
in Java.

For practical reasons.

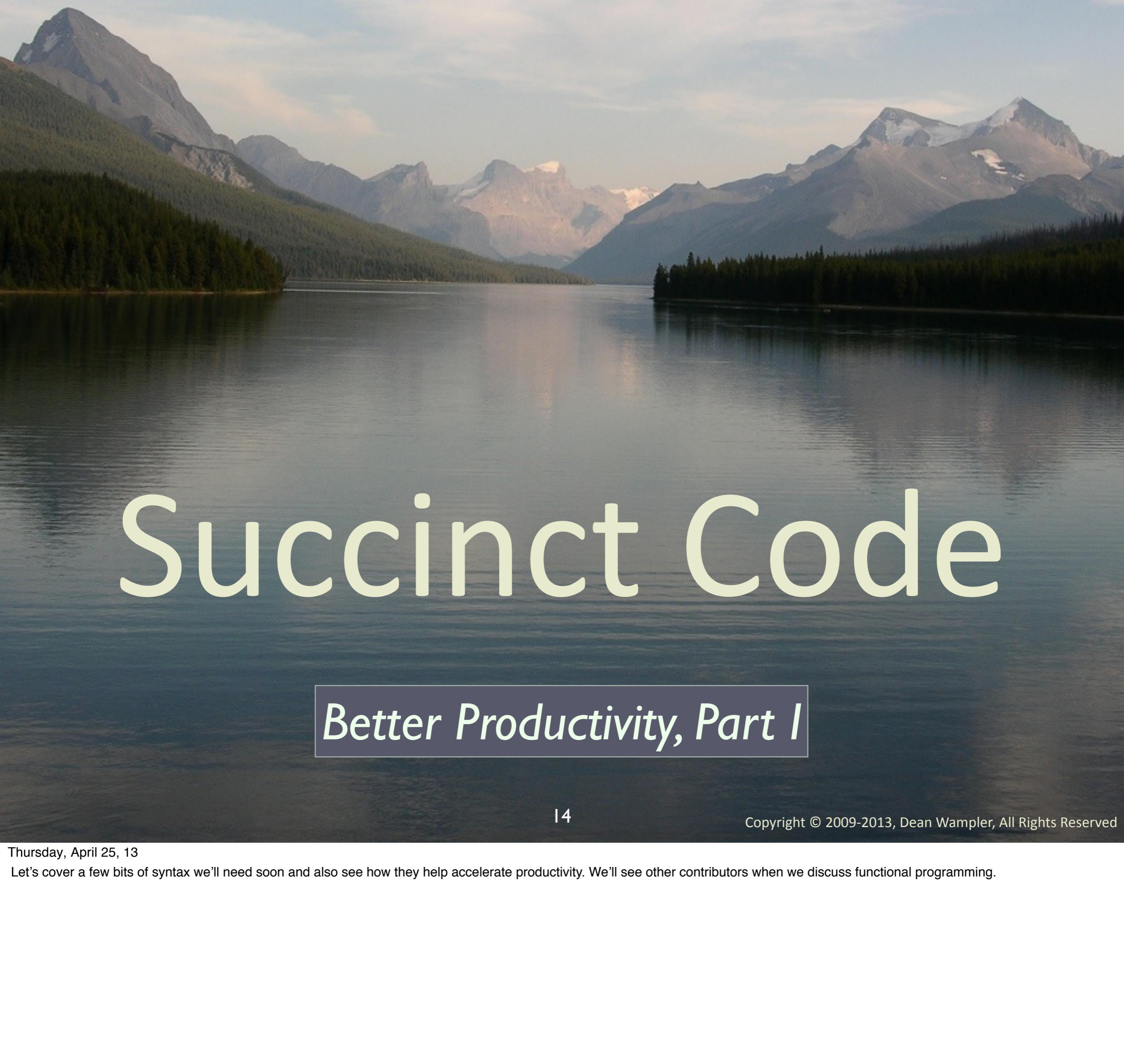
Scala is...

- A JVM language.
- Functional and object oriented.
- Statically typed.

There was an active effort to port Scala to .NET, but it seems to have stagnated. F# is the logical choice on .NET...

Martin Odersky

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

The background of the slide features 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 lower slopes. The background consists of more mountain peaks, some with snow on their summits, under a clear blue sky.

Succinct Code

Better Productivity, Part I

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]
```

*Type parameters
shown with [...].*

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

*no () required.
Semicolons inferred.*

```

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
(for starters...).*

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  
    // getter & setter methods:  
    def firstName = _firstName  
    def firstName_=(fn: String) =  
        _firstName = fn  
}
```

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

or

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

...

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

*Uniform Access
Principle*

Clients don't care which...

Exercise

Declaring classes.
ex1-classes.scala



Even Better: Case Classes

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

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

*Each arg
automatically a val.*

*No class body {...},
but a lot more
methods are created!*

```
case class Person(  
  firstName: String,  
  lastName: String,  
  age: Int) {  
  def toString...  
  def equals...  
  def hashCode...  
  ... pattern-matching  
}  
...  
}
```

*Generated
by the
compiler*

*We'll explore
this later...*

...

```
case object Person {  
    def apply(  
        firstName: String,  
        lastName: String,  
        age: Int) = new Person(...)  
}
```

*And a companion
object with factory
methods.*

User-defined Factory Methods

*no **new** needed.*

Can return a subtype!

```
val bob =  
  Person("Bob", "Smith", 29)
```

*which calls **apply** on the **Person** object.*

```
val bob =  
  Person.apply("Bob", ...)
```

Exercise

Case classes.
ex2-case-classes.scala



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 background, a range of majestic mountains rises, their peaks partially obscured by a light, hazy sky. The mountains are covered with dense forests of evergreen trees, particularly visible along the shoreline. The overall atmosphere is peaceful and natural.

Everything can be a Function

Objects as Functions

34

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Everything is already an object, following JVM conventions, but they can also be treated like functions, if they follow a simple protocol, which we've actually already seen...

```
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”

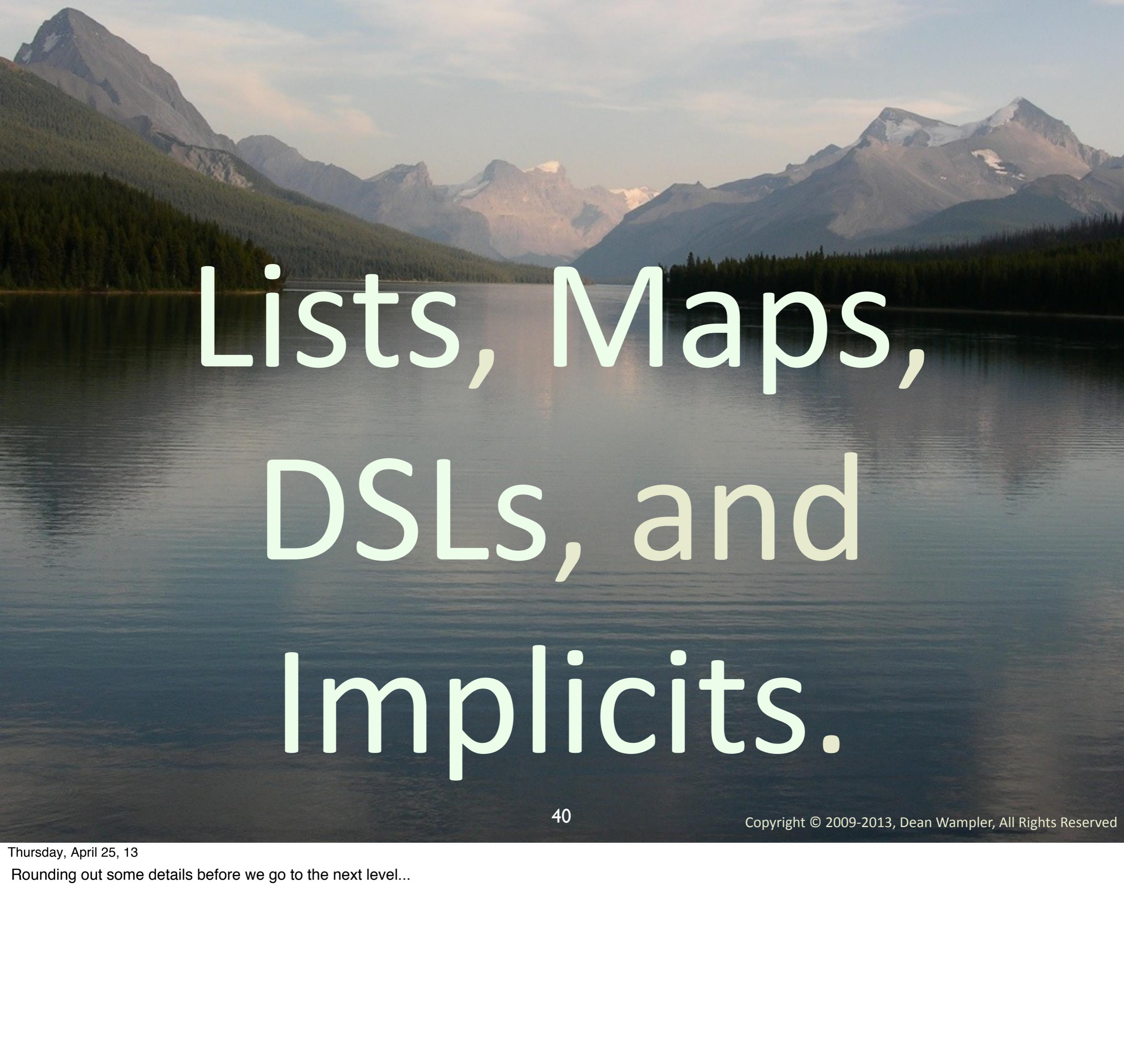
...
error("Network error.")

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

Thursday, April 25, 13

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

In fact, Scala's built in anonymous functions work the same way. The compiler translates the function literal syntax (which we'll see soon) into a Function object and uses the literal as the body of the object's apply method body.

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. The middle ground is filled with a dense forest of evergreen trees. In the background, a range of majestic mountains rises, their peaks partially obscured by a light-colored, overcast sky.

Lists, Maps, DSLs, and Implicits.

40

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Rounding out some details before we go to the next level...

Lists as DSLs

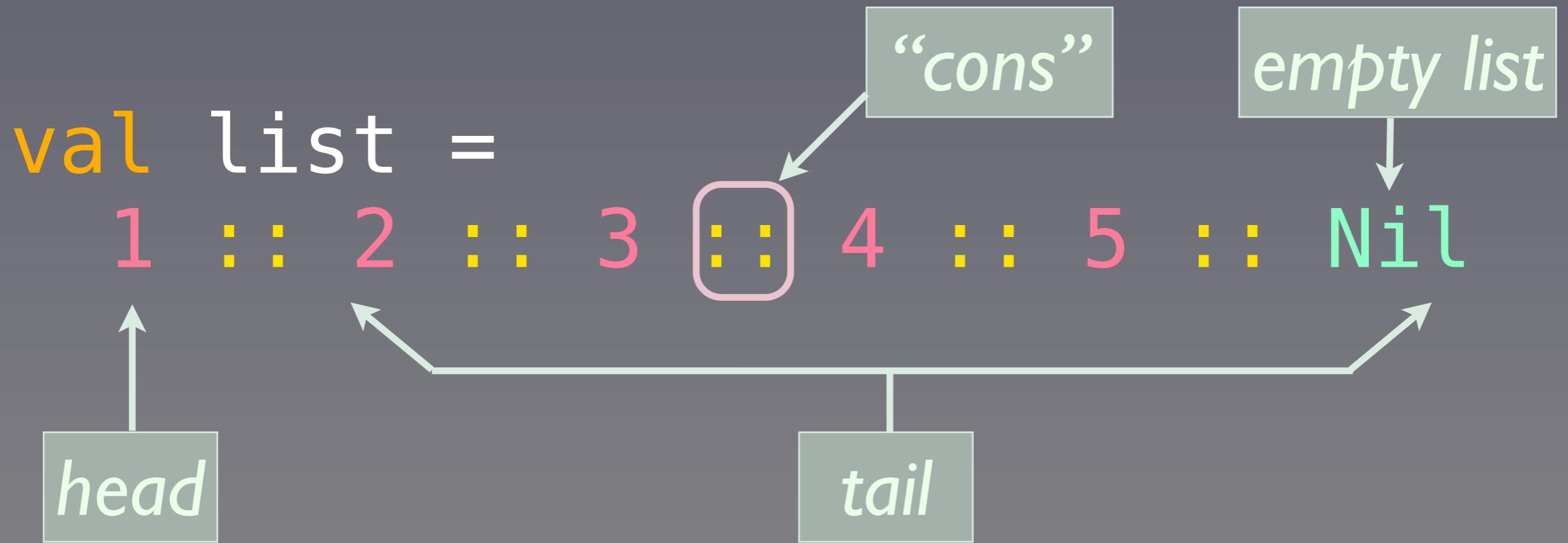
Domain Specific Languages

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 built-ins...

Also in many third-party libraries.

Oh, and Maps

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

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Maps also have a literal syntax, which should look familiar to you Ruby programmers ;) Is this a special case in the language grammar?

Oh, and Maps

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

“baked” into the
language grammar?

No, just method calls...

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*

Implicits to the Rescue

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

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

*Put the **implicit** 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))
```

Exercise

Implicit conversations.
ex3-implicits1.scala
or
ex3-implicits2.scala



Two Exercises -

Choose One:

- Implement doubles with units:
 - feet vs. meters.
- Add a **toXML** method to **Map**.

Units: Feet vs. Meters

```
val feets = 10.feet + 10.meters
println(feets)
// 42.8084 feet
```

Add toXML to Map

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

The background of the slide features a serene landscape of a lake nestled among majestic mountains. The sky is filled with soft, warm colors from the setting sun, creating a peaceful atmosphere.

Functional Programming

63

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Let's do a high-level overview of FP

What is Functional Programming?

In a few minutes...

$y = \sin(x)$

Based on Mathematics

$$y = \sin(x)$$

Setting x fixes y

\therefore variables are immutable

$20 + = 1 ??$

We never modify
the 20 “object”

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

Compose functions of
other functions

∴ functions are first-class

$y = \sin(x)$

No side effects.

Referential Transparency

Side-effect free functions: other benefits

- Easy to reason about behavior.
- Easy to invoke concurrently.

Side-effect free functions: other benefits

- Easy to invoke anywhere.
- Encourage immutable objects.

Side-effect free functions: other benefits

- These functions are also called pure.

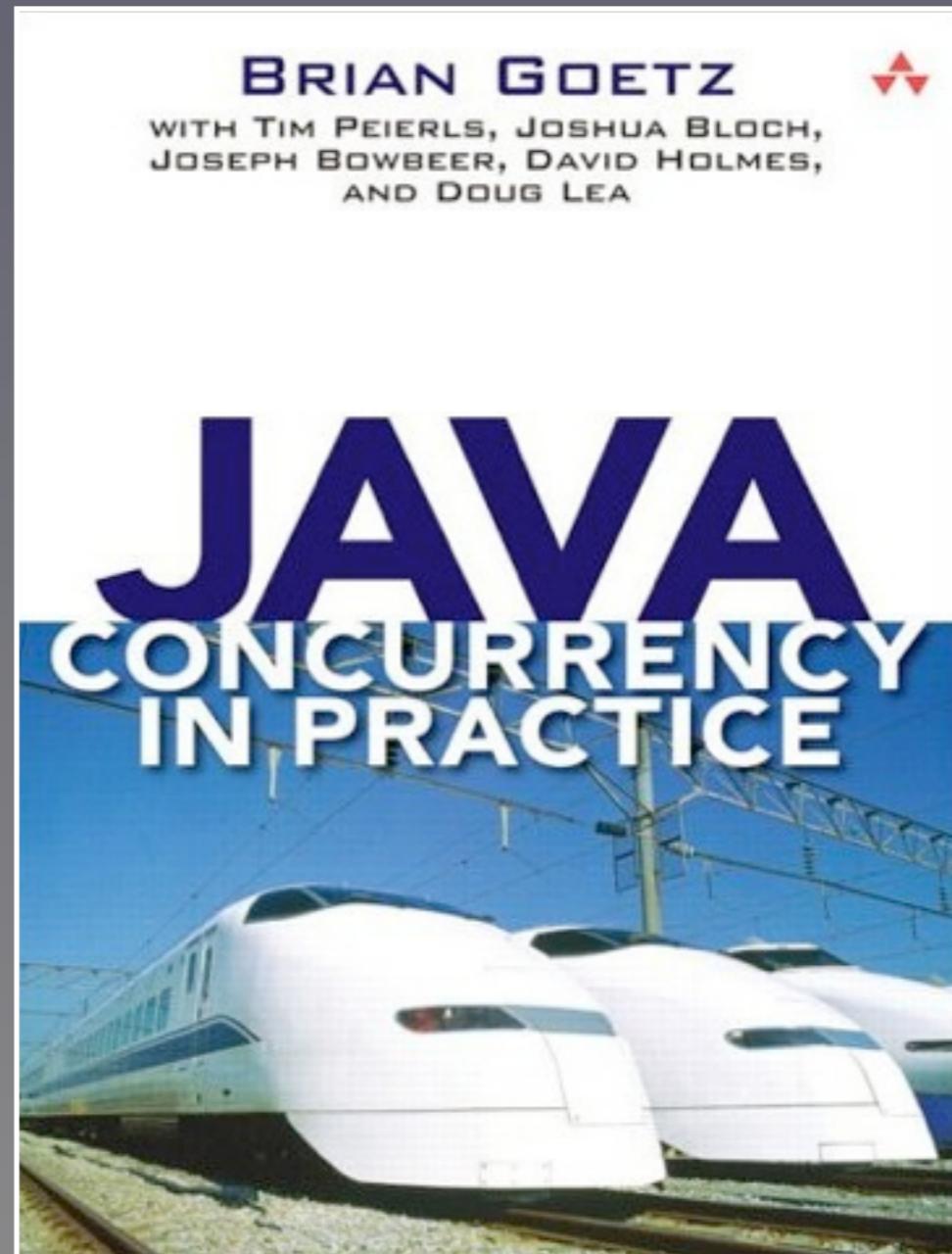
Concurrency

No mutable state

∴ nothing to synchronize!

When you
share
mutable
state...

Hic sunt dracones
(Here be dragons)



“Big Data”

Both FP and data analysis are based on Mathematics.

∴ Data is a killer app for FP!

Recursion

Thursday, April 25, 13

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

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

Note the 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

What about stack overflow? What about function call overhead?

Prefer tail recursions.
Scala implements the
tail call optimization.

Fibonacci is not tail recursive.

What about factorial?

```
def fact(i: Int): Long =  
  if (i == 1) 1L  
  else i*fact(i-1)
```

Wait! Is this tail recursive?

No.

```
def fact(i: Int): Long = {
  def f(ans: Long, i: Int): Long =
    if (i == 1) ans
    else f(ans*i, i-1)

  f(1, i)
}
```

```

def fact(i: Int): Long = {
  def f(ans: Long, i: Int): Long =
    if (i == 1) ans
    else f(ans*i, i-1)
}

```

Nested function

f(1, i)

Call f

*Recursion in
the tail position*

The diagram illustrates the execution flow. A call to *f(1, i)* is shown on the left, with an arrow pointing upwards to the nested function definition *f(ans*i, i-1)* on the right. This indicates that the call *f(1, i)* is the caller of the nested function.

Homework

Look up

Left vs. Right Recursion

Hint: The 1st fact was right recursive, the 2nd was left.

Typing?

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

Properties of Types
are very important

We'll see more of this later...

Contrasting Approaches

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

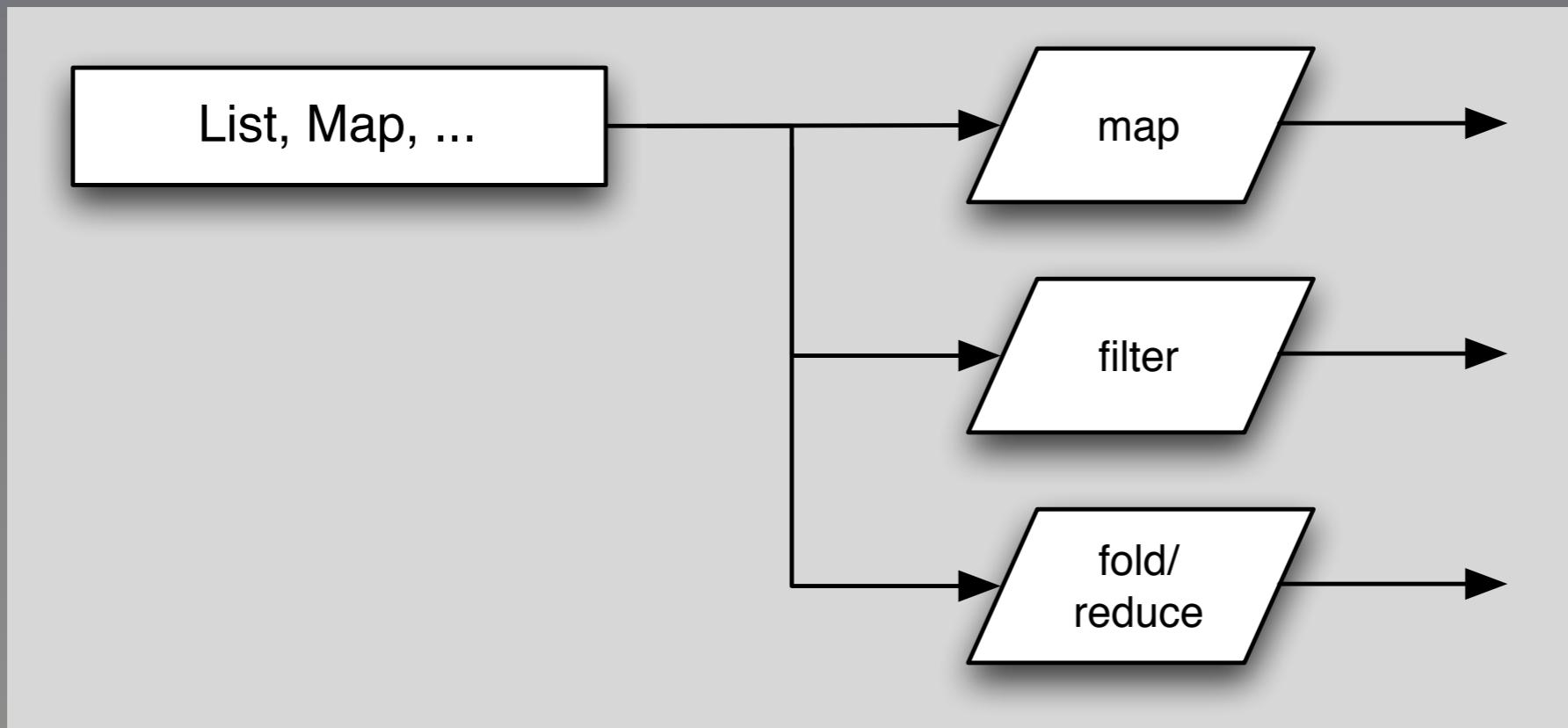
Exercise

Collection Methods

ex4-collections.scala



Classic Operations on Collection Types



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 lower slopes. The background features a majestic range of mountains under a clear sky.

Everything is 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.
use “{“ or “(“

list **map** {
 S => S.toUpperCase}

function argument list

“*function literal*”
a.k.a.
“*anonymous function*”

```
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

How the Sausage Is Made

like an abstract class



“contravariant”,
“covariant” typing



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

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

```
...  
}
```

*No method body:
=> abstract*

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 { ← {...} ok, instead of (...)  
    s => s.toUpperCase  
}  
  
// => "A" :: "B" :: Nil
```

Function “object”

Since functions
are objects,
they could have
mutable state.

```

case class memo(f:Int => Long) {
  val cache =
    ...mutable.Map[Int,Long]
  def apply(i:Int) = {
    if /* in cache? */
      // return cached value
    else {
      // call f(i), save in cache,
      // and return
    }
  }
}

val f2 = memo(fact)
println(f2(5)) // => 120

```

103

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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. We'll replace the comments with real code shortly...

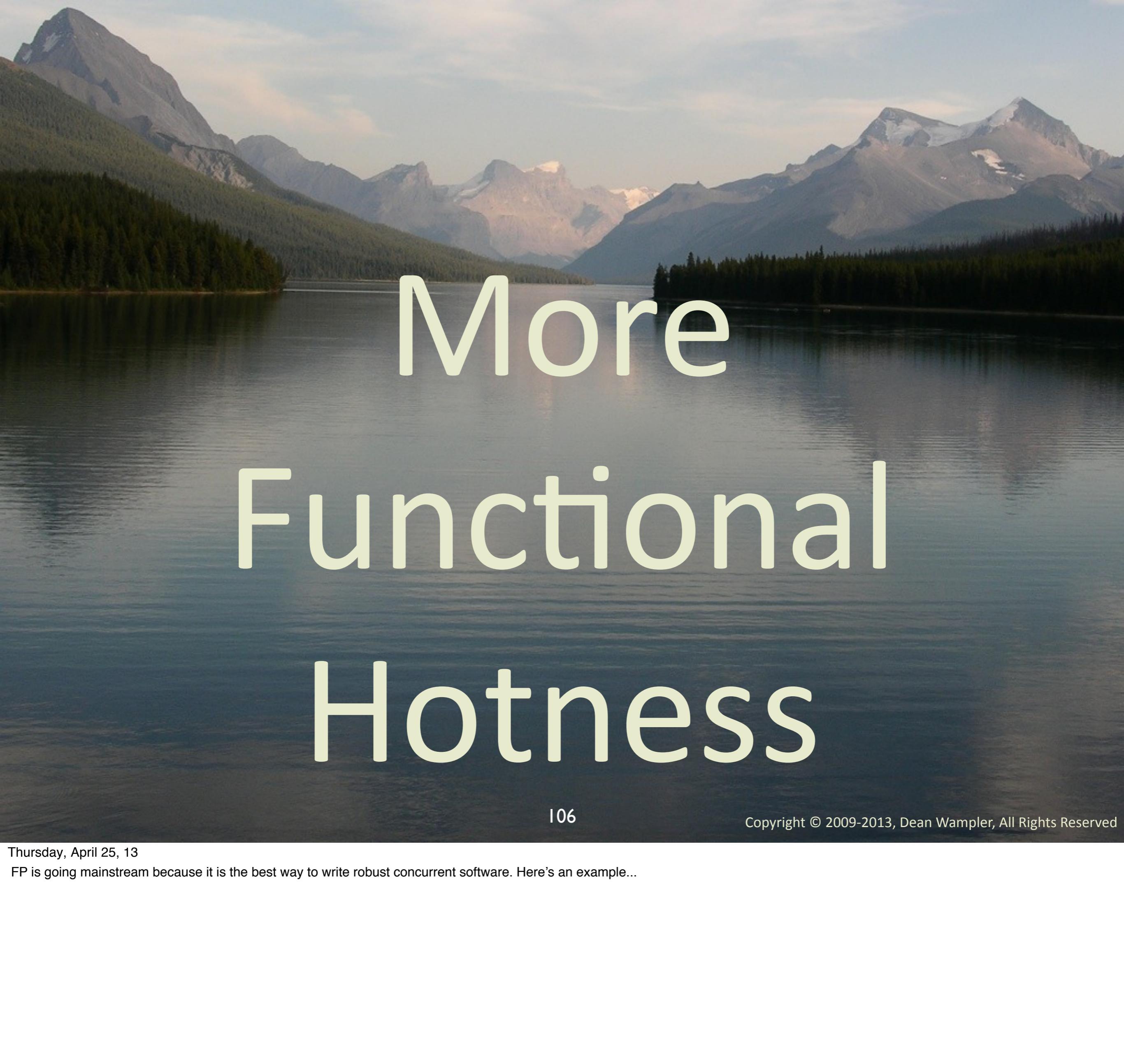
This is an example
of modular,
composable
functions in action!

Exercise

A Tree “Abstract Data Type”

ex5-trees.scala



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, their peaks partially obscured by a soft, warm glow from the setting sun. The sky is a mix of pale blue and orange, creating a peaceful and inspiring atmosphere.

More Functional Hotness

106

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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

107

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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

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

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

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

Which is the better API?

In Use:

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

...

```
m.get("two") match {  
  case Some(i) => println(i)  
  case None      => // do nothing  
}
```

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*

With sealed, this is valid:

```
m.get("two") match {
  case Some(i) => println(i)
  case None      => // do nothing
}
```

You OOP training says this is evil!!

Covariant Type

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

+T means T or any subtype of T

Case Classes

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

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

Case 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.

Remember our
memo
“function”?

```
case class memo(f:Int => Long) {  
    val cache =  
        ...mutable.Map[Int,Long]  
    def apply(i:Int) =  
        cache.get(i) match {  
            case Some(l) => l  
            case None =>  
                val l=f(i); cache.put(i,l); l  
        }  
}
```

```
val f2 = memo(fact)  
println(f2(5)) // => 120
```

117

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Here is the full implementation of memo, using pattern matching.

For Comprehensions

118

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

a.k.a. list comprehensions in Python and sequence comprehensions in other functional languages

Remember This?

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

...

```
m.get("two") match {  
  case Some(i) => println(i)  
  case None      => // do nothing  
}
```

Let's write it more concisely

Remember This?

```
val m =  
  Map("one" -> 1, "two" -> 2)
```

```
...  
for (i <- m.get("two"))  
  println(i)
```

Pattern match; nothing happens for None.

No if statement

```
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

```
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

Pattern Matching:

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

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

*Pattern match again;
only take the **Somes**.*

Monadic Behavior

```
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!*

Lists, Options, and Maps can be treated uniformly!

So, is there ever
a need for explicit
match statements?

“Error” Handling

```
val m =  
  Map("one" -> 1, "two" -> 2)  
...  
m.get("two") match {  
  case Some(i) => println(i)  
  case None      => println("!!")  
}
```

If you need to handle the `None` cases

Exercise

Creating a list of tuples
using a for
comprehension
`ex6-for-comprehensions.scala`



Ranges

```
for {  
    i <- 0 to 2      // inclusive  
    j <- 0 until i // exclusive  
} yield ((i,j)) // yield tuple  
// List((1,0), (2,0), (2,1))
```

List (i,j,k) Tuples Where:

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

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



Scala's Object Model: Traits

Composable Units of Behavior

132

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Fixes limitations of Java's object model.

Mixin Composition

133

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

We would like to
compose objects
from mixins.

Java: What to Do?

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

“*Server is a Logger*”?

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

Logger isn’t an interface!

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).

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

141

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Example

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

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

Log put Calls

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

Log put Calls

```
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

```
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

*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

Stackable Traits

148

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Filter put Calls

```
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
}
```

Filter put Calls

```
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

```
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

What if we
reverse the order
of the Trait
declarations?

```
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

Method Lookup Order

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

Simpler cases, only...

Note: traits can't have constructors.

Must initialize fields other ways.

Exercise

Traits

ex7-traits.scala



QueueDoubling Trait

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

- A trait that puts each new element twice.

A wide-angle photograph of a mountainous landscape. In the foreground, there is a calm lake reflecting the surrounding environment. The middle ground features a dense forest of coniferous trees lining the shore of the lake. In the background, a range of majestic mountains rises, their peaks partially obscured by a hazy sky. The lighting suggests either sunrise or sunset, casting a warm glow over the scene.

Modular Composition

162

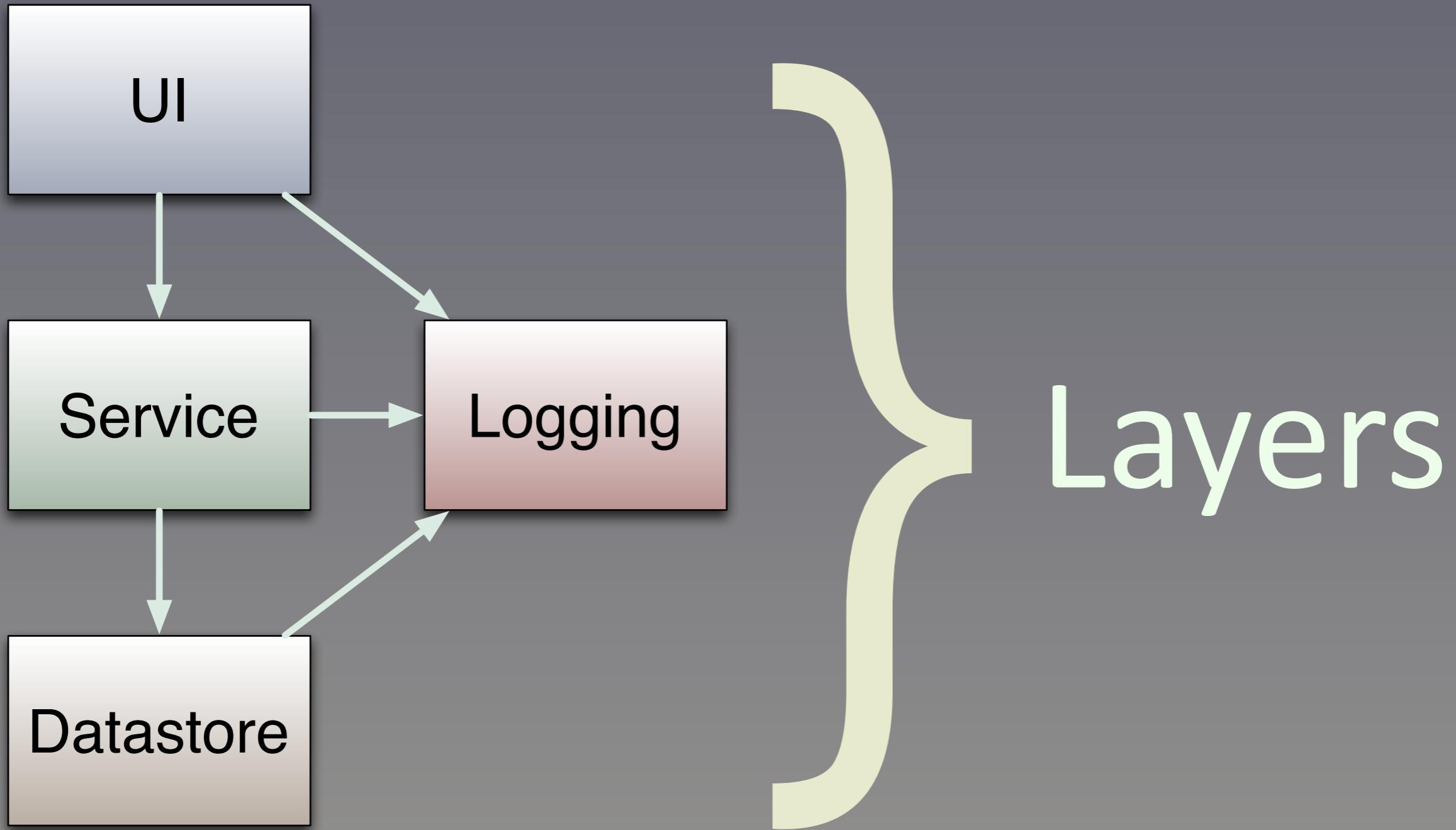
Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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

Traits as modules: The Cake Pattern

Like a Layer Cake



Preliminaries...

```
case class Event(evntStr:String)
case class Query(event:Event)
case class Result(result:String)
trait Logging {
  def log(message:String)
}
```

An abstract method

UI Trait

```
trait UI {  
    this: Service with Logging =>  
    def userEvent(e:Event) = {  
        log("UI: " + e)  
        val result = handleEvent(e)  
        display(result)  
    }  
    protected def display(  
        result:Result): Unit  
}
```

```

trait UI {
  this: Service with Logging =>
  def userEvent(e:Event) = {
    log("UI: " + e)
    val result = handleEvent(e)
    display(result)
  }
  protected def display(
    result:Result): Unit
}

```

Self type annotation

Service method

Like void

Self Type Annotations

```
trait UI {  
  this: Service with Logging =>
```

Functionally similar to:

```
trait UI  
  extends Service with Logging {
```

Rather than hard-wire module
dependencies, defer it...

Self Type Annotations

```
trait UI {  
  foo: Foo =>
```

Use a different name than
this to easily reference
Foo members.

Service Trait

```
trait Service {  
    this: Datastore with Logging =>  
    def handleEvent(e: Event) = {  
        log("Service: " + e)  
        query(Query(e))  
    }  
}
```

Datastore method

Datastore Trait

```
trait Datastore {  
    this: Logging =>  
    def query(q:Query) = {  
        log("DS: " + q)  
        doQuery(q)  
    }  
    protected def doQuery(  
        q:Query): Result  
}
```

Abstract
method

Now, wire the traits
together to build
the app...

```
val app = new App
  with UI with Service
  with Datastore with Logging {
  def log(message:String) =
    println("Log: " + message)
  def display(r:Result) =
    println("Result: " + r)
  def doQuery(q:Query) = {
    println("Query: " + q)
    Result("Success!!")
  }
}
```

Built-in type for
simple Apps

```
val app = new App
with UI with Service
with Datastore with Logging {
  def log(message:String) =
    println("Log: " + message)
  def display(r:Result) =
    println("Result: " + r)
  def doQuery(q:Query) = {
    println("Query: " + q)
    Result("Success!!")
}
```

```
val app = new App
  with UI with Service
  with Datastore with Logging {
    def log(message:String) =
      println("Log: " + message)
    def display(r:Result) =
      println("Result: " + r)
    def doQuery(q:Query) = {
      println("Query: " + q)
      Result("Success!!")
    }
}
```

Wire traits
together

```

val app = new App
with UI with Service
with Datastore with Logging {
  def log(message:String) =
    println("Log: " + message)
  def display(r:Result) =
    println("Result: " + r)
  def doQuery(q:Query) = {
    println("Query: " + q)
    Result("Success!!")
  }
}

```

Concrete
methods

Or do
this in
derived
traits

Run it!

```
val app = new App ...
app.userEvent(Event("search!"))
```

Log: UI: Event(search!)

Log: Service: Event(search!)

Log: DS: Query(Event(search!))

Query: Query(Event(search!))

Result: Result(Success!!)

Actual
implementations
tend to be more
sophisticated...

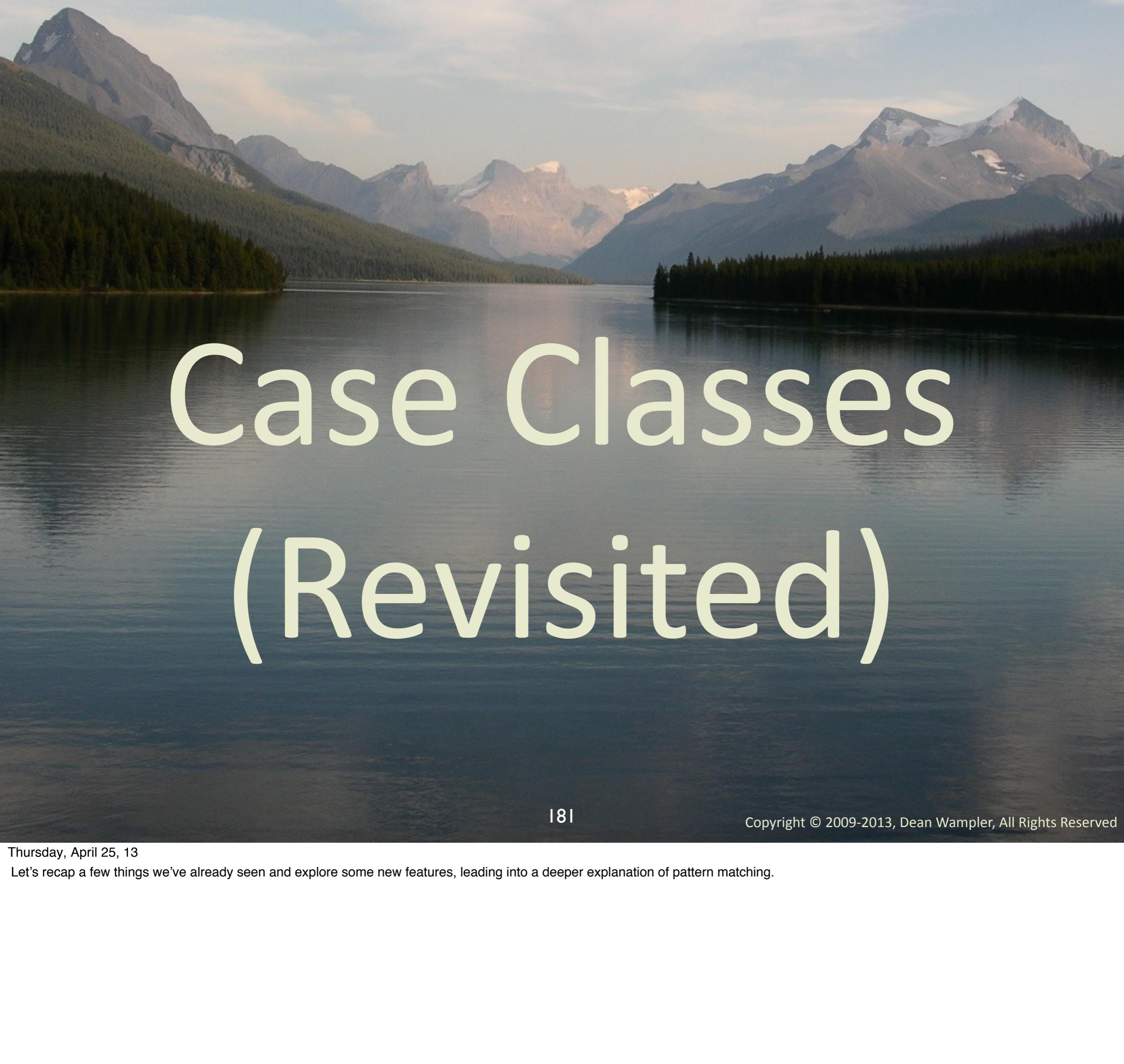
Traits Recap

179

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Traits let us compose
disjoint behaviors
and modify existing
behaviors.

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, a small, densely forested island is visible in the center. The background features a range of majestic mountains with their peaks partially covered in snow. The sky is a clear, pale blue with a few wispy clouds.

Case Classes (Revisited)

181

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Let's recap a few things we've already seen and explore some new features, leading into a deeper explanation of pattern matching.

Companion Objects

182

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Let's explore case classes a bit more, which automatically create "companion objects". You can explicitly create them yourself for non-case classes.

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

The diagram illustrates the relationship between a class and its companion object. A green box labeled "Singleton" has an arrow pointing to the class definition. Another green box labeled "Factory" method has an arrow pointing to the companion object's apply method. The code itself is written in a stylized orange and green color scheme.

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

“Companions”

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

Calls Complex.apply(...)

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 immutable fields.
 - The `val` keyword is not required.
- `equals`, `hashCode`, `toString`:
 - Based on the field values.
 - 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

Case classes
are very convenient for
“structural” types.

Note:
Inheriting one case
class from another is
disallowed.

equals, hashCode don't work properly

There is also an
unapply method...

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

A wide-angle photograph of a mountainous landscape. In the foreground, there's a calm lake reflecting the surrounding environment. On the left, a dense forest of evergreen trees lines the shore. 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.

Pattern Matching

192

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

A very powerful technique in functional languages that also appears on first glance to be contradictory to OOP “best practices”, as we’ll see.

Recall this List syntax:

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



Print a List

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

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

The “lprint” (list print) function takes a list

*Wildcard: match
any type*

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

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

MOAR Pattern Matching...

198

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

```

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 unapply methods.

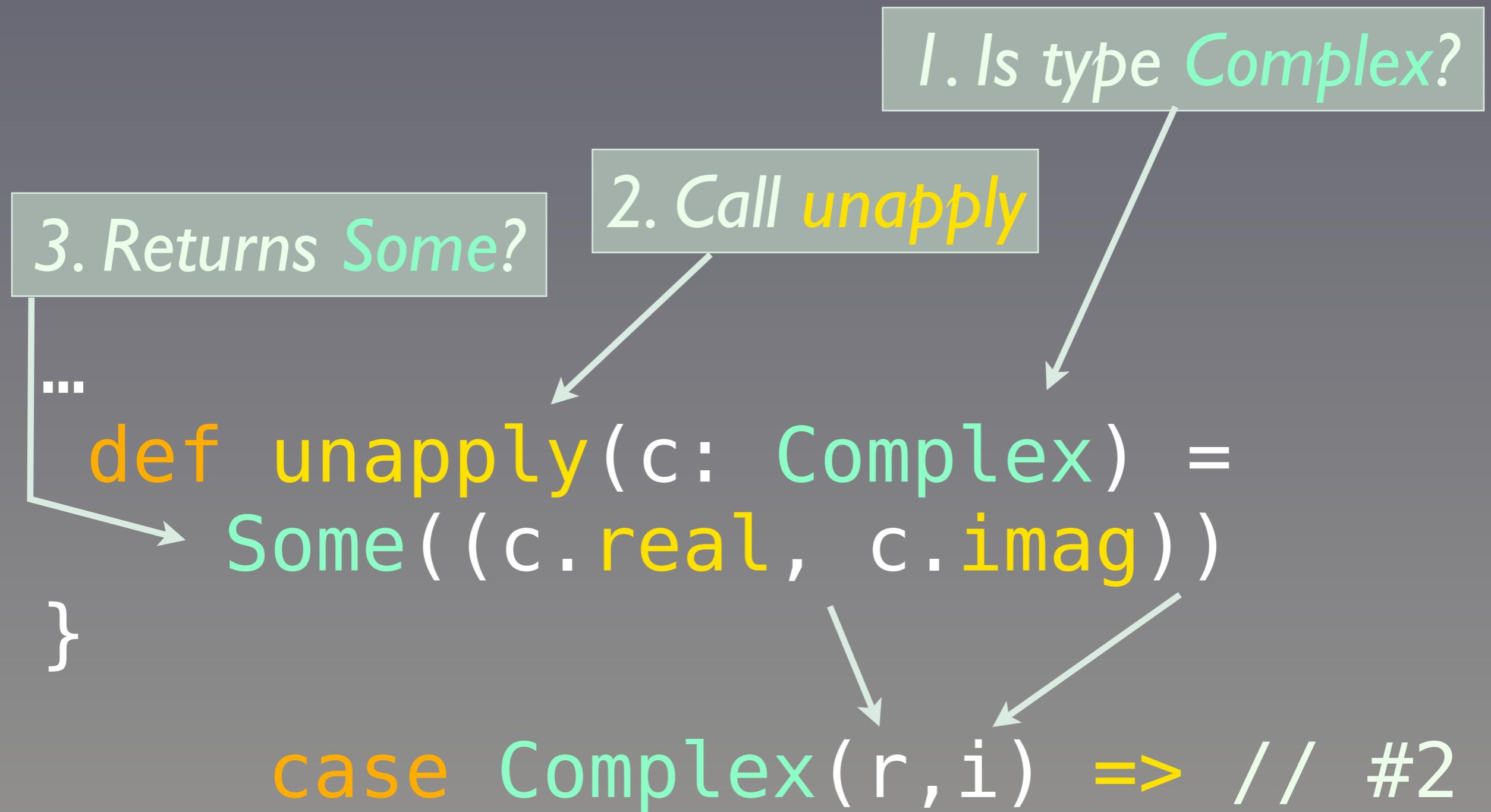
```
class Complex(val real: Double,  
             val imag: Double)  
{...}
```

```
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



Unapply is
generated for
case classes.

You rarely write your own!

Aside:

NOT needed

```
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  
    } }  
}
```

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**;
this is the literal syntax for one!*

A `PartialFunction` is
partial in that it
doesn't apply for the
whole range of
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 immutable fields.
 - The `val` keyword is not required.
- `equals`, `hashCode`, `toString`:
 - Based on the field values.
 - Companion object w/ *apply*, *unapply*.

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

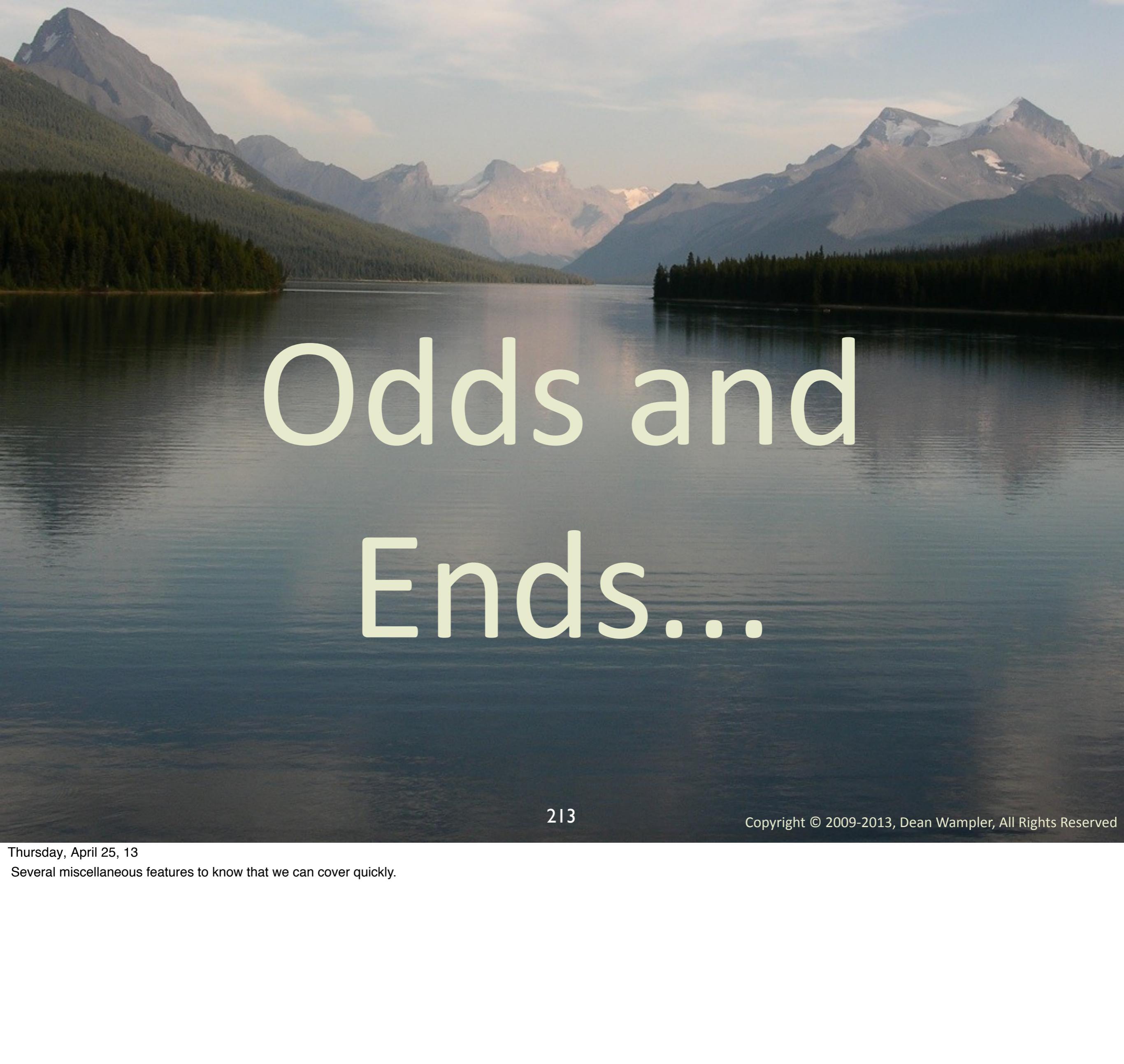
Pattern matching
plays a fundamental role
in FP analogous to
polymorphism in OOP.

Exercise

Pattern matching

ex8-pattern-matching.scala





Odds and Ends...

213

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Several miscellaneous features to know that we can cover quickly.

Primary and Secondary Constructors

214

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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

216

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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

*pass parameters to
parent constructor*

Currying

218

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Not just a feature of Asian cuisine...

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.

220

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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
// => modc = (m:Int)(n:Int)...
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

What does this mean?

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

- modc is a function value that takes an `Int` argument.
- ... 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

225

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Create your own data types with arithmetic operators.

```
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”

```
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”

```
...
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 = 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

Packages and Imports

231

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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 => JWriter,  
        _} ← selective  
    ...  
}
```

suppress → ,

“rename” → Reader => JReader, Writer => JWriter,

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?

236

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

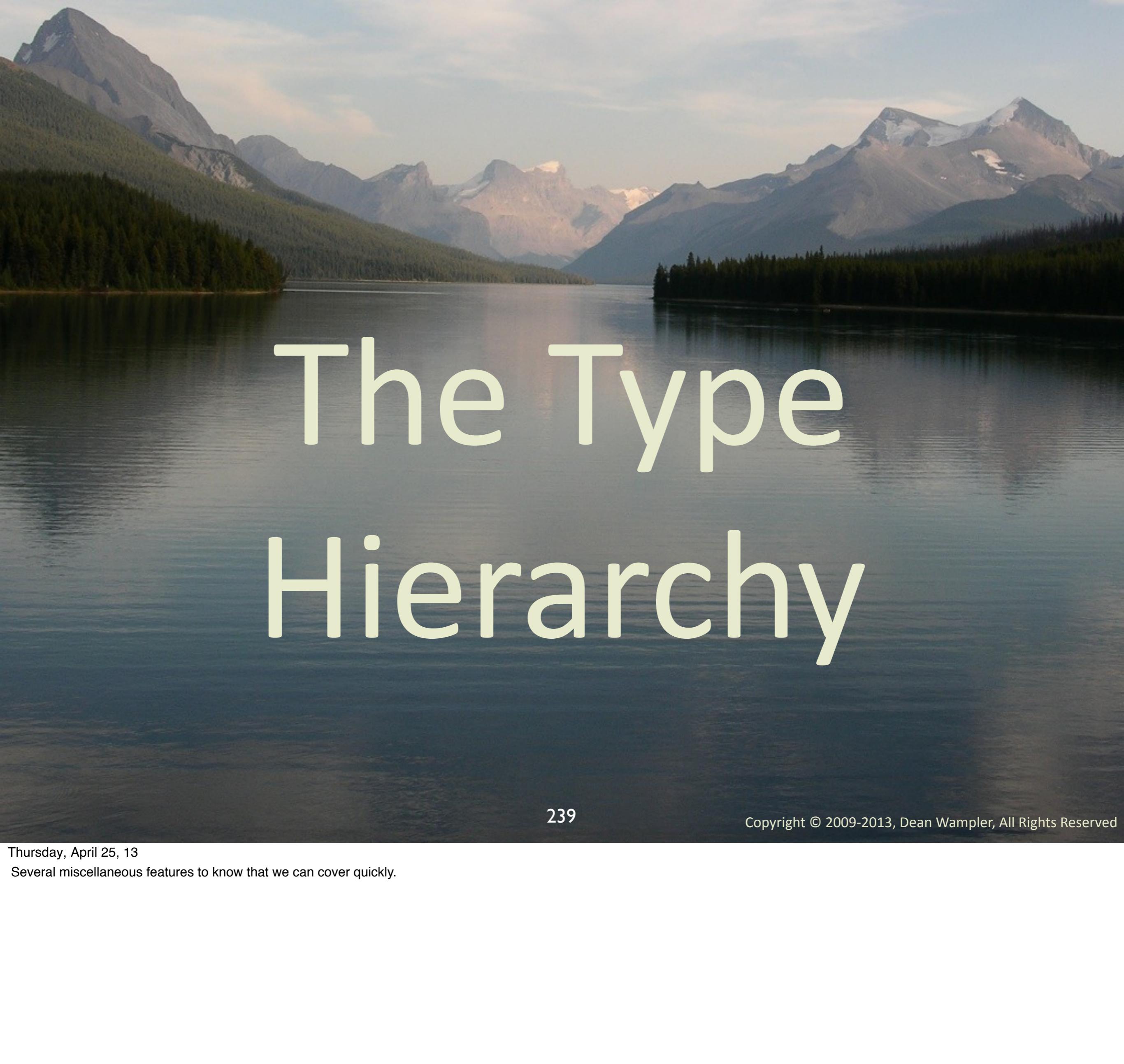
If you can use the namespace construct, what does that mean for a required directory layout, if any? And what about file names matching type names?

File and type names don't have to match.

But it's okay to follow Java conventions...

Directory and package
names
don't have to match.

But it's okay to follow Java conventions...

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 coniferous trees borders the water. The middle ground is dominated by a range of mountains with rugged peaks and some snow patches. The sky is a clear, pale blue with a few wispy clouds.

The Type Hierarchy

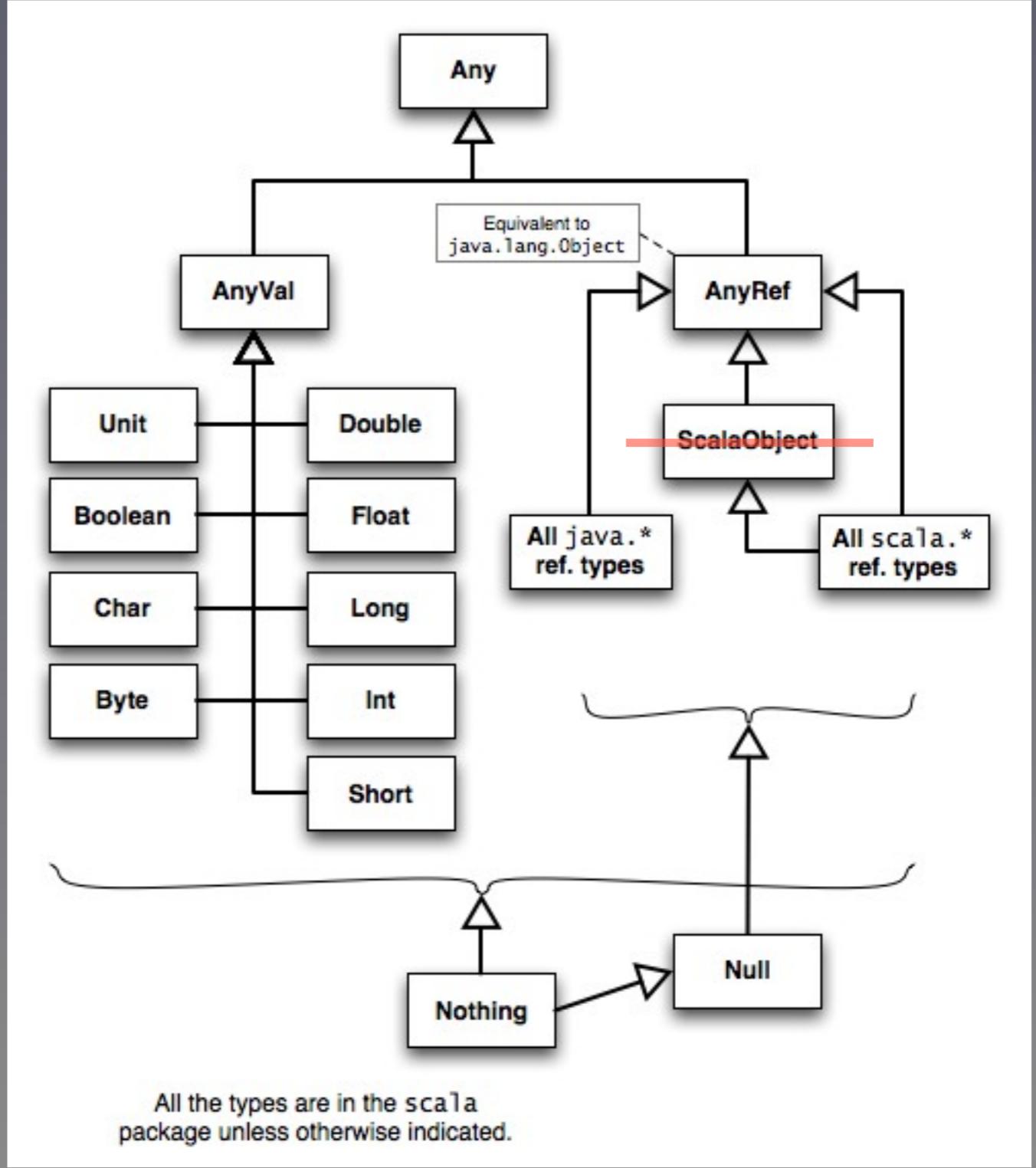
239

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Several miscellaneous features to know that we can cover quickly.

Base of the Type Hierarchy



240

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

There are no primitives in Scala. The VM primitive types are encapsulated in the `AnyVal` types. All other “reference” types are subtypes of `AnyRef`. `ScalaObject` is now obsolete, as of v2.10.

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

AnyRef is equivalent to Java's Object

List [+A]	Immutable, pervasive functional 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!

And this...

```
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 in the arg. types.
Covariant in the return types.

, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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]

250

“Function Literal”
syntax

erved

Thursday, April 25, 13

Syntactic sugar. The first form is more familiar to functional programmers. The second form is more familiar to OO programmers.

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 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!

253

*Why arguments must
be contravariant!*

erved

Thursday, April 25, 13

The 2nd argument is a function of type “(AnyRef) => Unit” (equivalent to “Function1[AnyRef,Unit]”). Next, we call forEachItem, passing a function literal as the 2nd argument.

Covariant Returns

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

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

Works!

254

*Boolean (and String) are
subclasses of Any*

erved

Thursday, April 25, 13

The 2nd argument is a function of type “(AnyRef) => Unit” (equivalent to “Function1[AnyRef,Unit]”). Next, we call forEachItem, passing a function literal as the 2nd argument.

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...

258

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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.

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.

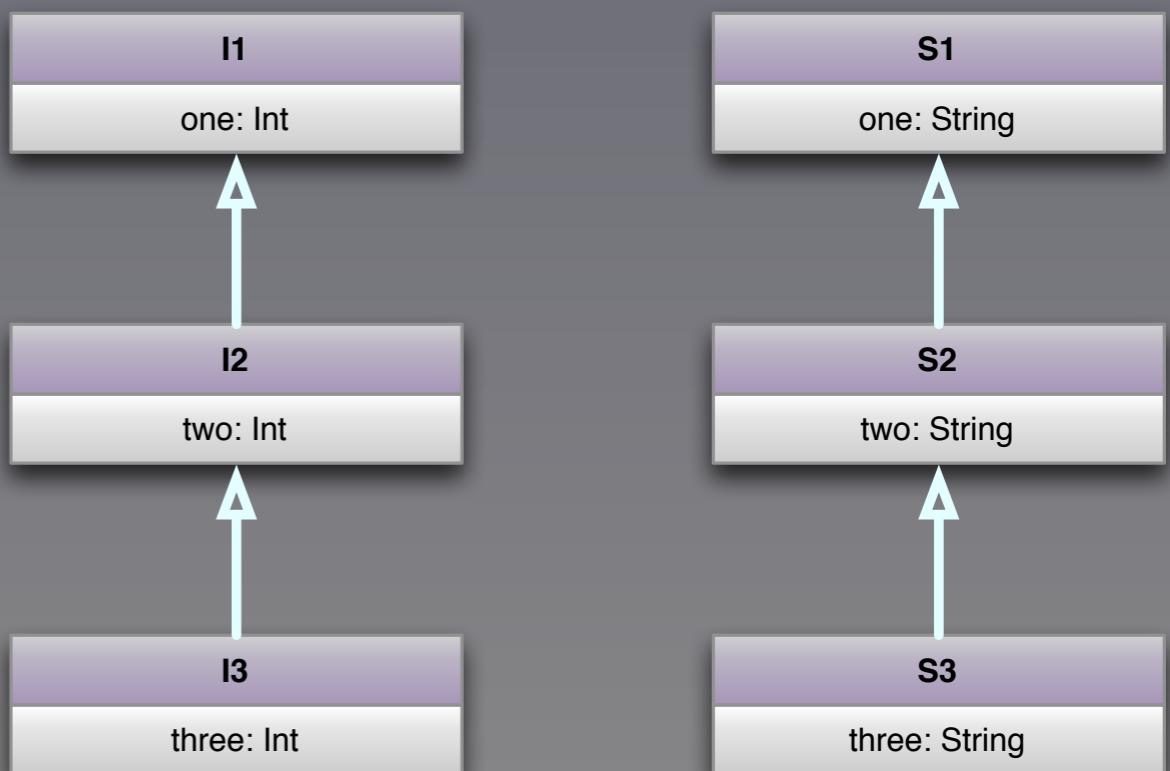
Exercise

Look at covariance and contravariance through the **Function1** type.
`ex9-function-types.scala`



Function1[-A,+R]

- Play with examples of Function1 in a simple type hierarchy to illustrate covariant and contravariant typing.



Avoiding Mutable State (Revisited)

265

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

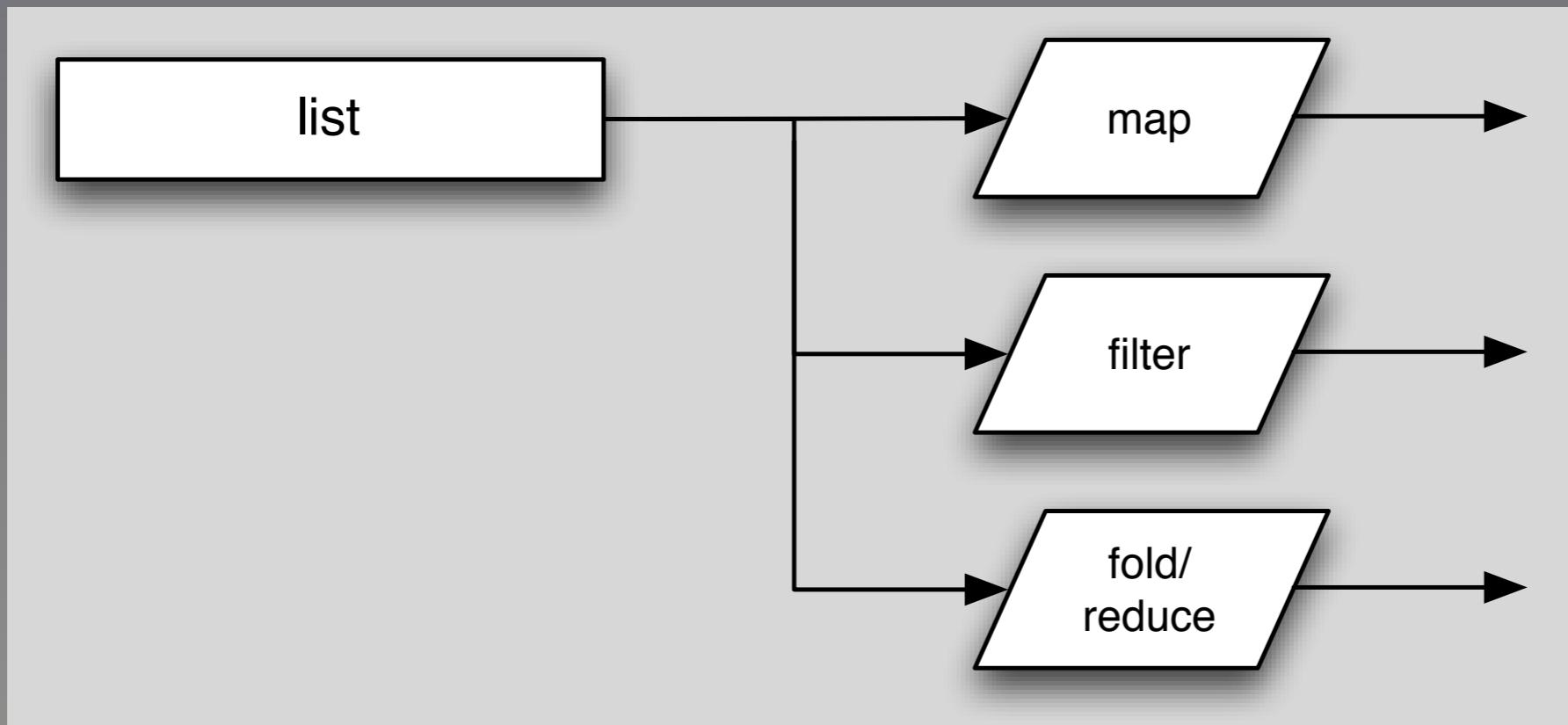
Thursday, April 25, 13

And also a powerful example of a low-level data-oriented DSL in action, that's built into the collections library!

Chaining collection
operations that
handle iteration
for us.

Avoiding loop counters

Recall the classic operations on functional data types



```
object Main {  
    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
```

*Sort integer arguments,
print out unique values.*

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

These variables are unnecessary.

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

“_” is a placeholder

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

Also not needed!

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



Recap

274

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Scala is...

275

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

a better Java,

276

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

object-oriented
and
functional,

succinct,
elegant,
and
powerful.

Some General Lessons:

279

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Objects can be functions.
Functions are objects.

Observation:
Any object graph
decomposes
into values and
collections of values.

Collections:

Construct, iterate,
and manage them
functionally.

Values:
Prefer
immutable
values.

283

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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

Values:

Carefully specify
the properties
of value types.

What are the properties of

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

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

Prefer functions and mixins (traits) for composition.

Thanks!

dean@concurrentthought.com

@deanwampler

polyglotprogramming.com/
programmingscala.com

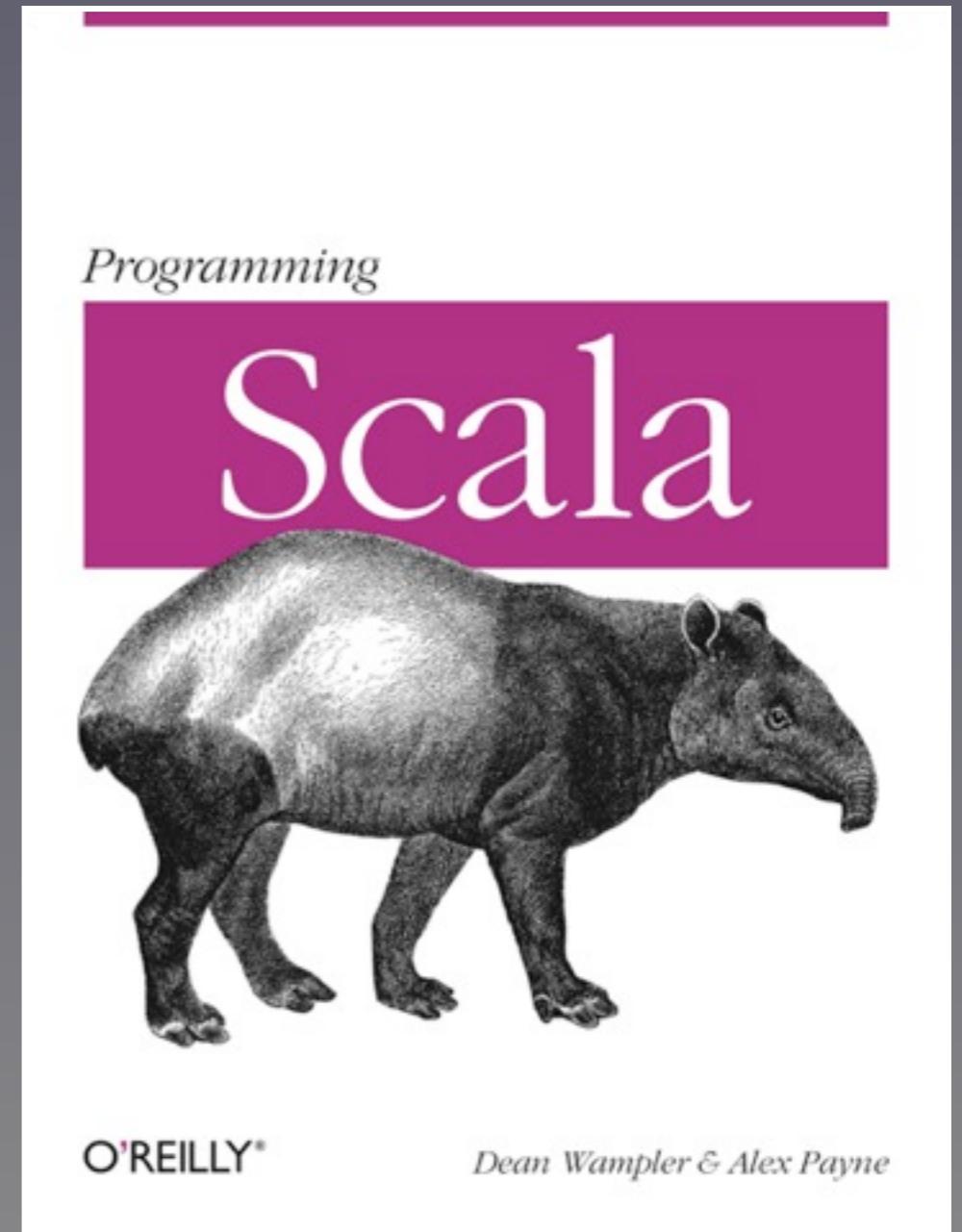
288

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

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.

<http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>



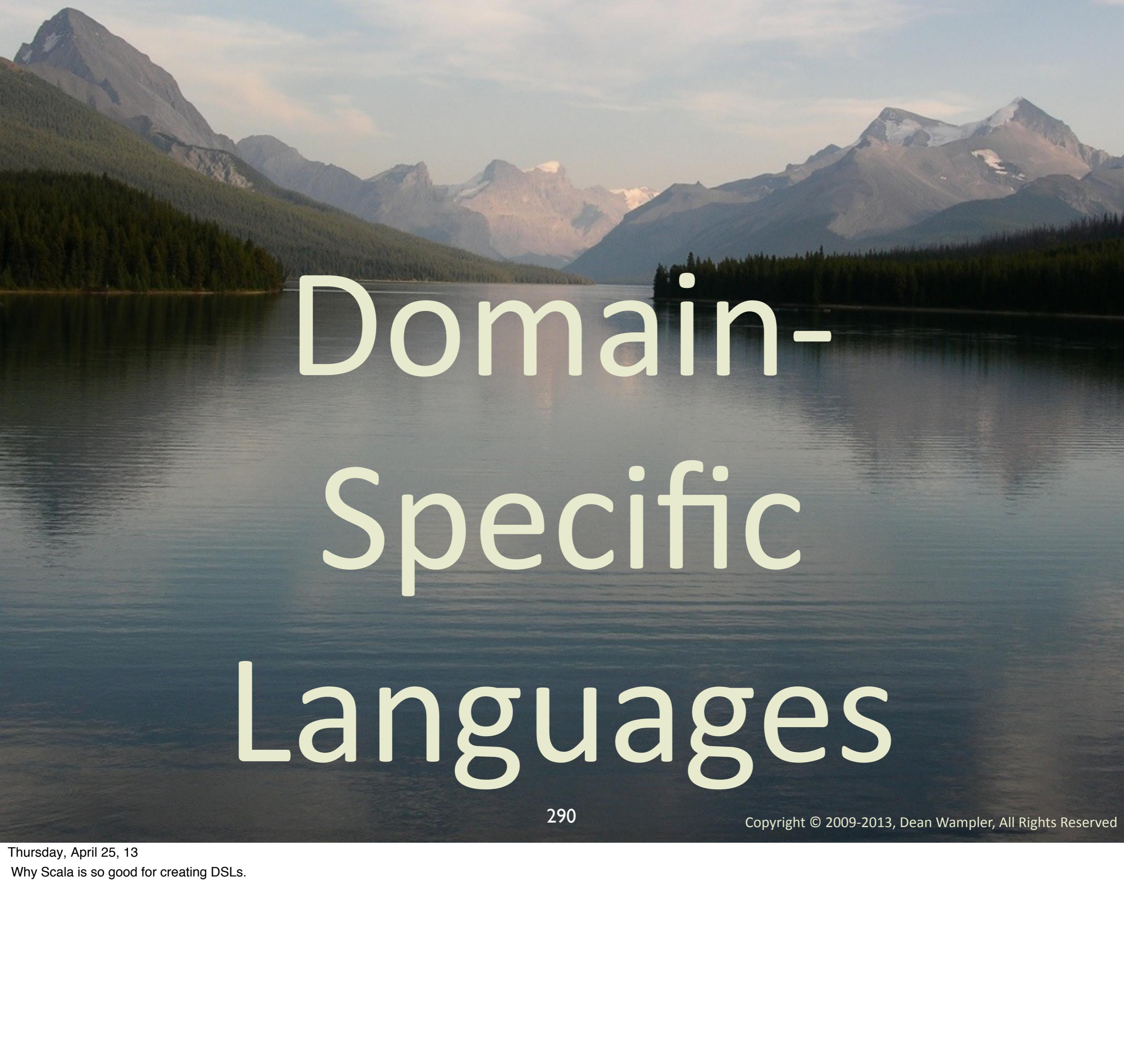
Extra Material

289

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Topics we probably won't have time to cover.

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, their peaks partially obscured by a hazy sky. The lighting suggests either sunrise or sunset, casting a warm glow over the scene.

Domain- specific Languages

290

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Why Scala is so good for creating DSLs.

We've seen several
features that promote
creation of Domain-
Specific Languages
(DSLs)...

Internal DSLs

292

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Internal DSLs are
idiomatic syntax in the
host language

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
```

*int2Hour called,
returning Hour(1).*

*Hour.hour(...)
called.*

*fromNow()
passed to hour.*

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) =  
{ ... } ← like void  
}
```

function taking line # and line

```
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

import

loop (new File("...")) {

(n, line) =>

1st param.: a file

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

}

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...

```
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...

Note: doLoop
is recursive.
There is no mutable
loop counter!

Pure functional “looping” technique

It Is Tail Recursive

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

311

*Scala optimizes tail
recursion into loops*

Exercise

The **whileTrue** loop
ex10-while-true.scala



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)
```

...

...

```
}
```

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

...

```
}
```

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`.

Note that by-name
parameters are not
side-effect free!

They return something different each time!

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

*Evaluated
each time*

External DSLs

320

Copyright © 2009-2013, Dean Wampler, All Rights Reserved

Thursday, April 25, 13

Features for Building External DSLs

- Parser Combinator Library

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