The Seductions of Scala

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

April 3, 2011

SOME RIGHTS RESERVED

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

<shameless-plug/>

Co-author,
Programming Scala

programmingscala.com

Available now from oreilly.com, Amazon, etc.

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

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

#1

We need *Functional Programming*

• • •

First reason, we need the benefits of FP.

... for *concurrency*.

... for *concise* code.

... for *correctness*.

#2
We need a better
Object Model

• • •

... for *composability*.

... for *scalable designs*.

Java's object model (and to a lesser extent, C#'s) has significant limitations.

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

Despite surface contradictions...

We think of objects as mutable and methods as state-modifying, while FP emphasizes immutability, which reduces bugs and often simplifies code. Objects don't have to be mutable!

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

We rarely have the luxury of starting from scratch...

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.

Odersky is the creator of Scala. He's a prof. at EPFL in Switzerland. Many others have contributed to it, mostly his grad. students.
GJ had generics, but they were disabled in javac until v1.5.

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

Objects as *Functions*

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

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


A simple wrapper around your favorite logging library (e.g., Log4J).

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

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

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

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

val error = new Logger(ERROR)

…
error(“Network error.”)
```


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

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

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

apply is called

"function object"

```
…
error("Network error.")
```


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

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

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!

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

Int, Double, etc. are true *objects*.

But they are compiled to primitives.

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!

Functions as *Objects*

First, About Lists

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

The same as this "list literal" syntax:

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

We build up a literal list with the "::" cons operator to prepend elements, starting with an empty list, the Nil "object".

Baked into the Grammar?

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

No, just method calls!

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


But this isn't something backed into the grammar; we're just making method calls on the List type!

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

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

Method names can contain almost any character.

There are some restrictions, like square brackets [and], which are reserved for other uses.

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

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

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

"::" binds to the right, so the second form shown is equivalent to the first.

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

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

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

Infix operator notation.

Infix Operator Notation

```
“hello” + “world”
```

is actually just

```
“hello”.+(“world”)
```


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

Similar mini-DSLs have been defined for other types.

Also in many third-party libraries.

Examples include the built-in Map type hierarchy.

Oh, and Maps

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

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...

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

Oh, and Maps

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

What we like to write:

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

What Map() actually wants:

We won’t discuss implicit conversions here, due to time....

Oh, and Maps

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

What we like to write:

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

What Map() actually wants:

More succinct syntax for Tuples

We won't discuss implicit conversions here, due to time....

We need to get from this,

```
"name" -> "Dean"
```

to this,

```
Tuple2("name", "Dean")
```

There is no String.-> method!

X

We won't discuss implicit conversions here, due to time....

Implicit Conversions

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

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


String doesn't have ->, but ArrowAssoc does! Also, it's -> returns a Tuple2. So we need to somehow convert our strings used as keys, i.e., on the left-hand side of the ->, to ArrowAssoc object, then call -> with the value on the right-hand side of the -> in the Map literals, and then we'll get the Tuple2 objects we need for the Map factory method.

The trick is to declare a method (or in some contexts, a value) with the keyword "implicit". The compiler will look for those in scope and then call them to convert the object without the desired method (a string and -> in our case) to an object with the method (ArrowAssoc).

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))
```


We won’t discuss implicit conversions here, due to time....

Classic Operations on *"Container" Types*

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

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

```
val list = “a” :: “b” :: Nil

list map {
  s => s.toUpperCase
}

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


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

map called on list (dropping the ".")

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

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

"function literal"

function argument list

function body

Note that the function literal is just the "s => s.toUpperCase". The {…} are used like parentheses around the argument to map, so we get a block-like syntax.

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

inferred type

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

Explicit type

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

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

How the Sausage Is Made

Parameterized type

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

Declaration of map

The function argument

map's return type

Here's the declaration of List's map method (lots of details omitted…). Scala uses [...] for parameterized types, so you can use "<" and ">" for method names!
Note that explicitly show the return type from map (List[B]). In our previous examples, we inferred the return type. However, Scala requires types to be specified on all method arguments!

How the Sausage Is Made

like an "abstract" class

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

No method body: => abstract

We look at the actual implementation of Function1 (or any FunctionN). Note that the scaladocs have links to the actual source listings.
(We're omitting some details…) The trait declares an abstract method "apply" (i.e., it doesn't also *define* the method.)
Traits are a special kind of abstract class/interface definition, that promote "mixin composition". (We won't have time to discuss…)

What the Compiler Does

```
(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

You use the function literal syntax and the compiler instantiates an anonymous class using the corresponding FunctionN trait, with a concrete definition of apply provided by your function literal.

```
val list = “a” :: “b” :: Nil

list map {
  s => s.toUpperCase
}

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

Can use {…} instead of (…)

Function “object”

Back to where we started. Note again that we can use “{…}” instead of “(…)” for the argument list (i.e., the single function) to map. Why, to get a nice block-like syntax.

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

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

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.

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 Abstract Data Type

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].

```
// 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?

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

In Use:

Literal syntax for map creation

```
val m =
  Map("one" -> 1, "two" -> 2)
…
val n = m.get("four") match {
 case Some(i) => i
 case None    => 0 // default
}
```

Use pattern matching to extract the value (or not)

Here's idiomatic scala for how to use Options. Our map if of type Map[String,Int]. We match on the Option[V] returned by map.get. If Some(i), we use the integer value I. If there is no value for the key, we use 0 as the default. Note: Option has a short-hand method for this idiom: m.getOrElse("four", 0).

Option Details: sealed

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

All children must be defined in the same file

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

Case Classes

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

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

Object

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

A "singleton". Only one instance will exist.

The scala runtime controls (lazy) instantiation of the single instance. Since the user can't instantiate the instance, objects can't have constructor argument lists, but they are allowed to define fields inside the class body (i.e., primary - and only - constructor body).

Nothing

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

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

None is used when there are 0 items, can be a singleton object, because it has no state! Note that type parameter for the parent Option. In the type system, Nothing is a subclass of all other types, so it substitutes for instances of all other types. This combined with a proper called covariant subtyping means that you could write “val x: Option[String = None” it would type correctly, as None (and Option[Nothing]) is a subtype of Option[String].

Succinct Code

A few things we've seen so far.

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

Infix Operator Notation

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

same as

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


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

Type *Inference*

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

vs.

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


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

Type *Inference*

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

vs.

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


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

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

no () needed.
Semicolons inferred.

Other things aren't needed...

User-defined *Factory* Methods

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

no new needed.
Can return a subtype!

Factory methods on the cheap...

```java
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

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

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

Typical Scala!

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

Class body is the "primary" constructor

Parameter list for c'tor

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

Makes the arg a field with accessors

No class body {…}. nothing else needed!

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

Actually, not *exactly* the same:

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

Doesn't follow the JavaBean convention.

Note that Scala does not define an argument list for "firstName", so you can call this method as if it were a bare field access. The client doesn't need to know the difference!

However, these *are* function calls:

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

  def firstName = _firstName

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

Uniform Access Principle

x

Note that Scala does not define an argument list for “firstName”, so you can call this method as if it were a bare field access. The client doesn’t need to know, nor care, if it is making a bar field access or going through a method.

Even Better...

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

Constructor args are automatically vals, plus other goodies.

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

Scala's Object Model: *Traits*

Composable Units of Behavior

Fixes limitations of Java's object model.

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

Java: What to Do?

```
class Server
 extends Logger { … }
```

"Server is a Logger"?

```
class Server
 implements Logger { … }
```

Logger isn't an interface!

Made-up example Java type. The "is a" relationship makes no sense, but the Logger we implemented earlier isn't an interface either.

Java's object model

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

Chances are, the “logging” and “filtering” behaviors are reusable, yet Java provides no built-in way to “mix-in” reusable implementations. Ad hoc mechanisms must be used.

Traits

Like interfaces with implementations,

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

Traits

… or like

abstract classes +

multiple inheritance

(if you prefer).

… and another way.

Logger as a Mixin:

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

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

I changed some details compared to our original Logger example. Traits don't have constructor argument lists (for various technical reasons), but we can define fields for them, as shown. Here, I make the field abstract, which means that any class that mixes in the trait will have to define "level".

Logger as a Mixin:

```
trait Logger {
 val level: Level // abstract
 …
}

val server =
  new Server(…) with Logger {
    val level = ERROR
}
server.log("Internet down!!")
```

mixed in Logging

abstract member defined

Note that could have declared a type, say "class ServerWithLogger(…) extends Server(...) with Logger {…}, but if you only need one instance, we can just do it "on the fly!" Note that the level is defined as a body for this object, much the same way you define an anonymous inner class and define its abstract members.

Building Our Own Controls

DSLs Using First-Class Functions

Recall *Infix* Operator Notation:

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

also the same as

```
1 + {2}
```

Why is this useful??

x

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

Make your own *controls*

```
// Print with line numbers.

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

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


If I put the “(n, line) =>” on the same line as the “{“, it would look like a Ruby block.

Make your own *controls*

```
// Print with line numbers.

loop (new File("…")) {
  (n, line) =>

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

control?

File to loop through

Arguments passed to…

what do for each line

How do we do this?

Output on itself:

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

```
import java.io._

object Loop {

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


Here’s the code that implements loop...

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

_ like * in Java

"singleton" class == 1 object

```
object Loop {
```

loop "control"

two parameters

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

like "void"

function taking line # and line

Singleton "objects" replace Java statics (or Ruby class methods and attributes). As written, "loop" takes two parameters, the file to "numberate" and a the function that takes the line number and the corresponding line, does something, and returns Unit. User's specify what to do through "f".

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

```
object Loop {

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

two parameters

The oval highlights the comma separating the two parameters in the list. Watch what we do on the next slide...

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

```
object Loop {

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

two parameters lists

We convert the single, two parameter list to two, single parameter lists, which is valid syntax.

Why 2 Param. Lists?

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

loop (new File("…")) {
  (n, line) =>

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

import

1st param.: a file

2nd parameter: a "function literal"

Having two, single-item parameter lists, rather than one, two-item list, is necessary to allow the syntax shown here. The first parameter list is (file), while the second is {function literal}. Note that we have to import the loop method (like a static import in Java). Otherwise, we could write Loop.loop.

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

nested method

Finishing Numberator...

Finishing the implementation, loop creates a buffered reader, then calls a recursive, nested method "doLoop".

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

recursive

"f" and "reader" visible from outer scope

Finishing Numberator...

Here is the nested method, doLoop.

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

Classic Functional Programming technique

It is *Tail* Recursive

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

Scala optimizes tail recursion into loops

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

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

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

When you share mutable state...

Hic sunt dracones
(Here be dragons)

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

Actor Model

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

Actor Model

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

The actor model is not new!!

2 Actors:

Our example. An actor for drawing geometric shapes and another actor that drives it.

```
package shapes

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

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

abstract "draw" method

Hierarchy of geometric shapes

"Case" classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape. The "case" keyword makes the arguments "vals" by default, adds factory, equals, etc. methods. Great for "structural" objects.
(Case classes automatically get generated equals, hashCode, toString, so-called "apply" factory methods - so you don't need "new" - and so-called "unapply" methods used for pattern matching.)

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

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

concrete "draw" methods

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


Case classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape.
For our example, the draw methods will just do "println("drawing: "+this.toString)".

```
package shapes
import akka.actor._

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

Use the "Akka" Actor library

Actor

receive and handle each message

Actor for drawing shapes

An actor that waits for messages containing shapes to draw. Imagine this is the window manager on your computer. It loops indefinitely, blocking until a new message is received...

Note: This example uses the Akka Frameworks Actor library (see http://akka.io), which I prefer over Scala's native actors.

Receive method

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

Actor for drawing shapes

“Receive” blocks until a message is received. Then it does a pattern match on the message. In this case, looking for a Shape object, the “exit” message, or an unexpected object, handled with the last case, the default.

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

pattern matching

Actor for drawing shapes

Each pattern is tested and the first match “wins”. The messages we expect are a Shape object, the “exit” string or anything else. Hence, the last “case” is a “default” that catches anything, we we treat as an unexpected error.

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

draw shape & send reply

done

unrecognized message

After handling each message, a reply is sent to the sender, using “self” to get the handle to our actor “nature”.

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

Altogether

Even compressed on a presentation slide, there isn't a lot of code!

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

a "singleton" type to hold main

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

! is the message send "operator"

driver to try it out

Here's the driver actor. It is declared as an "object" not a class, making it a singleton.
When we start, we send the "go!" message to the Driver actor that is defined on the next slide. This starts the asynchronous message passing.
The "!" is the message send method (stolen from Erlang).

Its "companion" object Driver was on the previous slide.

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

driver to try it out

Here's the driver actor "companion class" for the object on the previous slide that held main.
Normally, you would not do such synchronous call and response coding, if avoidable, as it defeats the purpose of using actors for concurrency.

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

driver to try it out

Here's the driver actor, a scala script (precompilation not required) to drive the drawing actor.
Normally, you would not do such synchronous call and response coding, if avoidable, as it defeats the purpose of using actors for concurrency.

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

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

"<-" and "->" messages may be interleaved!!

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

Note that the -> messages will always be in the same order and the <- will always be in the same order, but the two groups may be interleaved!!

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

Functional-style pattern matching

Object-oriented-style polymorphism

A powerful combination!

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

Recap

Scala is...

a *better*

Java and C#,

object-oriented

and

functional,

succinct,

elegant,

and

powerful.

Thanks!

dean@deanwampler.com
@deanwampler

polyglotprogramming.com/talks
programmingscala.com

thinkbiganalytics.com

SOME RIGHTS RESERVED

Extra Slides

Modifying Existing Behavior with Traits

Example

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

A pure abstraction (in this case...)

A very simple abstraction for a Queue.

Log put

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


(We're ignoring "get"…) "Super" is not yet bound, because the "super.put(t)" so far could only call the abstract method in Logging, which is not allowed. Therefore, "super" will be bound "later", as we'll so. So, this method is STILL abstract and it's going to override a concrete "put" "real soon now".

Log put

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

What is "super" bound to??

(We're ignoring "get"…) "Super" is not yet bound, because the "super.put(t)" so far could only call the abstract method in Logging, which is not allowed. Therefore, "super" will be bound "later", as we'll so. So, this method is STILL abstract and it's going to override a concrete "put" "real soon now".

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

Concrete (boring) implementation

Our concrete class. We import scala.collection.mutable.ArrayBuffer wherever we want, in this case, right were it's used. This is boring; it's just a vehicle for the cool traits stuff...

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

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

Example use

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

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*.

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

Traits are a powerful *composition* mechanism!

Not shown, nesting of traits...

Functional Programming

What is *Functional Programming*?

Don't we already write "functions"?

$$y = \sin(x)$$

Based on *Mathematics*

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

$$y = \sin(x)$$

Setting *x* fixes *y*

∴ *variables* are *immutable*

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

20 += 1 ??

We never *modify* the 20 “object”

Concurrency

No *mutable state*

∴ *nothing* to *synchronize*

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

When you share mutable state...

Hic sunt dracones
(Here be dragons)

$y = \sin(x)$

Functions don't

change state

$\therefore$ *side-effect* free

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

Side-effect free functions

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

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

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

Compose functions of other functions

$\therefore$ *first-class citizens*

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

Even More Functional *Hotness*

For "Comprehensions"

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

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

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

No "if" statement

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