The Seductions of Scala

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

April 26, 2013

SOME RIGHTS RESERVED

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

Monday, April 22, 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/>

Co-author,
Programming Scala

programmingscala.com

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

Available now from oreilly.com, Amazon, etc.

Why do we need a new language?

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

Monday, April 22, 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

...

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

Monday, April 22, 13

First reason, we need the benefits of FP.

... for concurrency.

... for concise code.

... for correctness.

... for data.

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

#2
We need a better Object Model

...

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

... for composability.

... for scalable designs.

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

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...

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

Monday, April 22, 13

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!

#3
We need better Productivity.

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

But we want to keep our investment in Java.

For practical reasons.

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

Monday, April 22, 13

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

Scala is...

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

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

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.

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

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.

Succinct Code

Better Productivity, Part I

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

Monday, April 22, 13

Let's cover a few bits of syntax we'll need soon and also see how they help accelerate productivity. We'll see other contributors when we discuss functional programming.

Infix Operator Notation

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

same as

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


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

Monday, April 22, 13

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]
```


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

Monday, April 22, 13

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]
```


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

Monday, April 22, 13

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]
```

Type parameters shown with […].

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

Monday, April 22, 13

Scala allows you to define operators “<“ and “>”, so it reserves “[…]” for type parameters.

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

no () required.
Semicolons inferred.

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

Monday, April 22, 13

Other things aren't needed; Functions that don't take arguments are often defined without empty () and aren't called with them. However, it's conventional to use them anyway for functions with side effects, e.g., "factoryForFoo()".
Also, semicolons are inferred when unambiguous.

```
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

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

Monday, April 22, 13

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!

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

Monday, April 22, 13

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 (for starters…).

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

Monday, April 22, 13

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.

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

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!

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
}
```


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

Note the method names. There's no argument list for the "firstName" method and the = separates the function signature from the body. In contrast, the setter function "firstName_=" follows a protocol the compiler understands so we can write code like 'obj.firstName = "Bubba"' and it will call the function, even though it looks like we are setting a bare field on "obj"!

```
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...

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

Monday, April 22, 13

Why don't reader/writer methods start with "get/set"? The "Uniform Access Principle"; client writes 'myperson.firstName = "Bubba"' and doesn't care if they are calling a method or accessing a public field. There is a @BeanProperty annotation for when you need JavaBeans get/set methods.

Exercise

Declaring classes.

ex1-classes.scala

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

Monday, April 22, 13

A very quick exercise to practice what we've learned so far.

Even Better: Case Classes

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


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

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

Each arg automatically a val.

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

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

Monday, April 22, 13

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

```
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...

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

Monday, April 22, 13

We'll see what the pattern-matching means shortly.

…

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

And a companion object with factory methods.

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

Monday, April 22, 13

It generates a factory method so we can say “val p = Person(...)”, rather than “val p = new Person(...)”. That alone isn’t a lot, although it reduces boilerplate. The important part is that we now have clients calling a factory method that could be enhanced in the future to return one of several possible subtypes, say depending on the arguments. The built-in Map (key-value hash) type does this. It picks the representation to use based on the number of key-value pairs.

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", ...)
```


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

Monday, April 22, 13

Factory methods in Scala are defined using a special convention, which we'll explain shortly.

Exercise

Case classes.

ex2-case-classes.scala

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

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

Monday, April 22, 13

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

Objects as Functions

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

Monday, April 22, 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)
 }
}
```


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

Monday, April 22, 13

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

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

Monday, April 22, 13

We saw apply used already as a factory method, when it's defined on a case object. Here apply is defined as an instance method. What's that all about??

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

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

val error = new Logger(ERROR)

…
error("Network error.")
```


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

Monday, April 22, 13

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.")
```


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

Monday, April 22, 13

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.

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

Monday, April 22, 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.

Lists, Maps, DSLs, and Implicits.

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

Monday, April 22, 13

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

Lists as DSLs

Domain Specific Languages

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

Monday, April 22, 13

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
```


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

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

Monday, April 22, 13

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)
```


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

Monday, April 22, 13

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. Exclusions include . , ( ) [ ] Characters like _ # can't be used alone.

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

Monday, April 22, 13

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!

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

Monday, April 22, 13

"::" 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 ")".

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

Monday, April 22, 13

Infix operator notation.

Infix Operator Notation

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

is actually just

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


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

Monday, April 22, 13

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 built-ins...

Also in many third-party libraries.

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

Examples include the built-in Map type hierarchy.

Oh, and Maps

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

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

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...

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

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:

What Map.apply() actually wants:

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


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

Monday, April 22, 13

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:

What Map.apply() actually wants:

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

More succinct syntax for Tuples

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

Monday, April 22, 13

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

Implicits to the Rescue

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

Monday, April 22, 13

A tool for doing this conversion that we want in an elegant way.

We need to get from this,

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

to this,

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

There is no String.-> method!

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

Monday, April 22, 13

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)
```


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

Monday, April 22, 13

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).

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.

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

Monday, April 22, 13

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))
```


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

Monday, April 22, 13

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

Exercise

Implicit conversations.

ex3-implicits1.scala

or

ex3-implicits2.scala

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

Two Exercises - Choose One:

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

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

Units: Feet vs. Meters

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


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

Add toXML to Map

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


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

Functional Programming

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

Monday, April 22, 13

Let's do a high-level overview of FP

What is Functional Programming?

In a few minutes...

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

y = sin(x)

Based on Mathematics

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

"Functional Programming" is based on the behavior of mathematical functions and variables, which are more restrictive than the ways we've come to think of them in "imperative" programming.

$$y = \sin(x)$$

Setting x fixes y

$\therefore$ variables are immutable

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

Monday, April 22, 13

"Variables" actually reference immutable values.

20 += 1 ??

We never modify the 20 “object”

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

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

Compose functions of other functions

$\therefore$ functions are first-class

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

Monday, April 22, 13

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

y = sin(x)

No side effects.

Referential Transparency

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

Monday, April 22, 13

Sin(x) does not change any global or object state. So, every call with the same value of x will always return the same value of y. Referential Transparency means we can replace calls to sin(x) for some x with the value it would always return. Similarly, from the SW perspectively, we can call sin(x) anywhere and anytime in our programs and we'll always get the same result. This vastly decreases bugs and the testing challenge!!

Side-effect free functions: other benefits

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

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

Monday, April 22, 13

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.

Side-effect free functions: other benefits

- Easy to invoke anywhere.
- Encourage immutable objects.

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

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.

Side-effect free functions: other benefits

- These functions are also called pure.

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

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.

Concurrency

No mutable state

∴ nothing to synchronize!

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

Monday, April 22, 13

FP is breaking out of the academic world, because it offers a better way to approach concurrency, which is becoming essential for every developer to master, since vertical scaling is becoming less viable (end of Moore's Law), so we must scale horizontally.

When you share mutable state...

Hic sunt dracones
(Here be dragons)

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

Multithreaded (synchronized, shared state) is the assembly language of concurrency. It's too low level for most needs.

“Big Data”

Both FP and data analysis are based on Mathematics.

∴ Data is a killer app for FP!

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

Monday, April 22, 13

I also think the growth of focusing on data-centric applications (at least more-so than before…) is also a “killer app” for FP.

Recursion

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

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

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

Monday, April 22, 13

A familiar example of a recursive mathematical function.
Notice the declarative nature; we're not telling you how to calculate, per se. We're just telling you the relationships between values.
In pure FP, you can't increment a loop counter, because it wouldn't be immutable!. An alternative way to do iteration is to use 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

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

Monday, April 22, 13

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

What about stack overflow?

What about function call overhead?

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

Aren't there efficiency and resource exhaustion concerns with recursion??

Prefer tail recursions.

Scala implements the tail call optimization.

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

Monday, April 22, 13

Scala can apply the tail-recursion optimization, where it converts the recursions into while loops (effectively).

Fibonacci is not tail recursive.

What about factorial?

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

I also think the growth of focusing on data-centric applications (at least more-so than before...) is also a “killer app” for FP.

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

Wait! Is this tail recursive?

No.

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

Monday, April 22, 13

It's not tail recursive because we make the recursive call, THEN do work, namely the multiplication.

```
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)
}
```


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

Monday, April 22, 13

It's not tail recursive because we make the recursive call, THEN do work, namely the multiplication.

```
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)
}
```

Nested function

Recursion in the tail position

Call f

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

Monday, April 22, 13

It's not tail recursive because we make the recursive call, THEN do work, namely the multiplication.

Homework

Look up

Left vs. Right Recursion

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

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

Monday, April 22, 13

It turns out there are important differences, in the general case, whether you traverse left or right. I'll let you look those up...

Typing?

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

Some FP languages really push static typing...

tan(Θ) = sin(Θ)/cos(Θ)

y = exp(pi*x)

Properties of Types

are very important

We'll see more of this later...

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

Monday, April 22, 13

The first equation would not work for integers. It works for reals. in the second equation, x could be an integer, but y must be real.

Contrasting Approaches

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

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

Monday, April 22, 13

Currently, there is a lot of debate about the merits of TDD in functional programming. Old time functional programmers are skeptical...

Exercise

Collection Methods

ex4-collections.scala

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

Classic Operations on Collection Types

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

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

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

Monday, April 22, 13

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.

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

Monday, April 22, 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!

Functions as Objects

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

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

list map {
  s => s.toUpperCase
}

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


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

Monday, April 22, 13

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. use "{" or "("

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

function argument list

function body

"function literal" a.k.a. "anonymous function"

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

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

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

Monday, April 22, 13

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...

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

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

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

Monday, April 22, 13

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

"contravariant", "covariant" typing

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

  def apply(a:A): R

  ...
}
```

No method body: => abstract

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

Monday, April 22, 13

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

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

Monday, April 22, 13

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.

Recap

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

list map {
  s => s.toUpperCase
}

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

{…} ok, instead of (…)

Function "object"

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

Monday, April 22, 13

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.

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

Monday, April 22, 13

But I'm not suggesting you do this; recall our earlier discussion about side-effect free functions! However, one "legitimate" use is caching.

```
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
```


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

Monday, April 22, 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!

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

Monday, April 22, 13

This is really, really important. Small, side-effect (or side-effect scoping) functions are easily composed to build up larger, more complex computations.
We'll see a larger-scala approach using "traits" later.

Exercise

A Tree
"Abstract Data Type"
ex5-trees.scala

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

More Functional Hotness

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

Monday, April 22, 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

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

Monday, April 22, 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?

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

Monday, April 22, 13

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

Use pattern matching to extract the value (or not)

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

Monday, April 22, 13

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

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

Monday, April 22, 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 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].

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!!

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

Monday, April 22, 13

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).

Covariant Type

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

+T means T or any subtype of T

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

Monday, April 22, 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 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.*

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

Case Object

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

A singleton. Only one instance will exist.

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

Monday, April 22, 13

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.

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

Monday, April 22, 13

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].

Remember our memo “function”?

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

```scala
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
```


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

Monday, April 22, 13

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

For Comprehensions

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

Monday, April 22, 13

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

Remember This?

Literal syntax for map creation

```
val m =
  Map("one" -> 1, "two" -> 2)
…
m.get("two") match {
 case Some(i) => println(i)
 case None    => // do nothing
}
```

Let's write it more concisely

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

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

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

This prints nothing if the map doesn't have a "two". It extracts the integer from the Some() and prints the value if the map has a "two".

```
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

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

Monday, April 22, 13

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

```scala
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
```


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

Monday, April 22, 13

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

```scala
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 ";"

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

```
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

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

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

Pattern Matching:

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

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

Pattern match again; only take the Somes.

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

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

Monadic 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!

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

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.

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

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

"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

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

Monday, April 22, 13

Also, or memo function needed to handle both cases.

Exercise

Creating a list of tuples using a for comprehension

ex6-for-comprehensions.scala

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

Ranges

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


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

Monday, April 22, 13

This will be useful for the exercise.

List (i,j,k) Tuples Where:

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

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


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

Look up these operations in the scaladocs for List.

Scala's Object Model: Traits

Composable Units of Behavior

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

Monday, April 22, 13

Fixes limitations of Java's object model.

Mixin Composition

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

We would like to compose objects from mixins.

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

Java: What to Do?

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

"Server is a Logger"?

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

Logger isn't an interface!

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

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.

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

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 ...

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

Monday, April 22, 13

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

Traits

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

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

… 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.

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

Monday, April 22, 13

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

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

Monday, April 22, 13

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.

Modifying Existing Behavior

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

Example

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

A pure abstraction (in this case...)

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

Monday, April 22, 13

A very simple abstraction for a Queue.

Log put Calls

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


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

Monday, April 22, 13

(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 Calls

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

What is super bound to??

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

Monday, April 22, 13

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

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

Concrete (boring) implementation

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

Monday, April 22, 13

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

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

Monday, April 22, 13

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

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

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

Stackable Traits

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

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
}
```


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

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

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

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

Monday, April 22, 13

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

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

Defines "veto"

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

Monday, April 22, 13

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

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

loop from -2 to 2

Example use

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

Monday, April 22, 13

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

```
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

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

Monday, April 22, 13

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

What if we reverse the order of the Trait declarations?

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

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

Order switched

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

```
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!

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

Monday, April 22, 13

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

Loosely speaking, the precedence goes right to left.

"Linearization" algorithm

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

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

Method Lookup Order

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

Simpler cases, only...

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

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

Note: traits can't have constructors.

Must initialize fields other ways.

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

Monday, April 22, 13

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

Exercise

Traits

ex7-traits.scala

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

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.

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

Modular Composition

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

Monday, April 22, 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

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

Monday, April 22, 13

We just saw a small example of composing little behaviors, implemented as traits, into a object. Let's look at a larger-scale approach.

Like a Layer Cake

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

A simple example 3-tier service with a UI, middle tier, and datastore, all of which share a common logging service.

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

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

A few simple-minded classes for an example.

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
}
```


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

See Daniel Speiwak's Keynote at the NE Scala Symposium, 2013, for a more in depth look. Slides: https://dl.dropboxusercontent.com/u/1679797/NE%20Scala/Bakery%20from%20the%20Black%20Lagoon.pdf, video: http://www.youtube.com/watch?v=yLbdw06tKPQ.

```
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

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

Monday, April 22, 13

Unit is not the same as void, it's more like a zero-element tuple, i.e., with only one possible value. Or, think of it as special class with no fields and hence only one instance.

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...

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

Monday, April 22, 13

When we use a self-type annotation, we are specifying a dependency yet to be resolved. It's like we extended a trait; we are allowed to make calls on Service and Logging, but they haven't yet been mixed into the UI trait explicitly. In fact, they are just abstractions; we will still need concrete implementations of these traits.
So, when we construct a real object that mixes in the UI, i.e., we instantiate some sort of class, we must resolve these dependencies by then!

Self Type Annotations

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

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

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

Monday, April 22, 13

This technique is convenient when you have method overrides and you want to explicitly call the version defined for one of the self types, say “foo.toString”.

Service Trait

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

Datastore method

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

See Daniel Speiwak's Keynote at the NE Scala Symposium, 2013, for a more in depth look. Slides: https://dl.dropboxusercontent.com/u/1679797/NE%20Scala/Bakery%20from%20the%20Black%20Lagoon.pdf, video: http://www.youtube.com/watch?v=yLbdw06tKPQ.

Datastore Trait

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

Abstract method

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

See Daniel Speiwak's Keynote at the NE Scala Symposium, 2013, for a more in depth look. Slides: https://dl.dropboxusercontent.com/u/1679797/NE%20Scala/Bakery%20from%20the%20Black%20Lagoon.pdf, video: http://www.youtube.com/watch?v=yLbdw06tKPQ.

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

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

```
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!!")
}
```


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

Monday, April 22, 13

Now wire it all together. We are simply defining concrete method implementations for all the abstract methods. In a real app, you would probably implement these in derived traits and then mix them together as shown.

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!!")
}
```


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

I'm just using App for convenience here; not recommending it in general. All it does is treat the constructor body as the main routine and run it immediately (more or less…).

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!!")
}
```


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

Monday, April 22, 13

Now wire it all together. We are simply defining concrete method implementations for all the abstract methods. In a real app, you would probably implement these in derived traits and then mix them together as shown.

```
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

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

Monday, April 22, 13

Now wire it all together. We are simply defining concrete method implementations for all the abstract methods. In a real app, you would probably implement these in derived traits and then mix them together as shown.

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!!)
```


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

Actual implementations tend to be more sophisticated...

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

We kept it simply to highlight the key points. There are more subtleties that tend to crop up in real implementations. See Daniel Speiwak's Keynote at the NE Scala Symposium, 2013, for a more in depth look. Slides: https://dl.dropboxusercontent.com/u/1679797/NE%20Scala/Bakery%20from%20the%20Black%20Lagoon.pdf, video: http://www.youtube.com/watch?v=yLbdw06tKPQ.

Traits Recap

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

Traits let us compose disjoint behaviors and modify existing behaviors.

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

Not shown, nesting of traits...

Case Classes (Revisited)

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

Monday, April 22, 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

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

Monday, April 22, 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)
{…}
```

"Singleton"

```
object Complex{
```

"Factory" method

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

  …
}
```

"Companions"

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

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

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

Calls Complex.apply(...)

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

Monday, April 22, 13

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

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...

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

Note that pattern of the object and its apply method. It takes the same 2 arguments as the primary constructor and it calls that constructor.

Case Class

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


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

Monday, April 22, 13

Scala generates common boilerplate if you add the case keyword.

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*.

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

If you want var fields, add the var keyword in the c'tor argument list, as needed.

Remember Map?

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

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

0 to N Tuple2s

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

Monday, April 22, 13

This is how the “factory method Map” actually works. Map is an object and the factory method is apply. Note that * in the argument list: 0-N tuples.

Case classes are very convenient for “structural” types.

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

That is, simple types that encapsulate several fields, but don’t necessarily have a lot of complexity nor the need for mutable state. If the class is complex and has mutable state, case classes aren’t such a good fit, but maybe the class needs to be rethought anyway!!

Note:

Inheriting one case class from another is disallowed.

equals, hashCode don't work properly

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

There is also an unapply method...

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

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

Monday, April 22, 13

We'll look at these two topics next...

Pattern Matching

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

Monday, April 22, 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:

"cons"

empty list

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


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

We build up a literal list with the "::" cons operator to prepend elements, starting with an empty list, the Nil "object". "::" binds to the right, so the second form shown is equivalent to the first.

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

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

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

Monday, April 22, 13

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

Note recursive call

Return type required

```
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

Explicit return type required for all methods that are recursive!

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

lprint(list)

// => 1, 2, 3, 4, 5,
```


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

Monday, April 22, 13

We build up a literal list with the "::" cons operator to prepend elements, starting with an empty list, the Nil "object".
Note the more specific matches come before the less specific. We can use "_" for "don't cares".

MOAR
Pattern Matching...

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

```
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)
```


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

Monday, April 22, 13

This time, we match on the elements themselves. Note that the more specific matches appear before the less specific, because matching is eager. . We can use "_" for "don't cares". Note that foreach doesn't "visit" the Nil at the end, so we never go through the "case Nil" clause.

The extractors used for Complex and Tuple3 are unapply methods.

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

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

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


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

Add the “unapply” method to our Complex object. Note that unapply returns a tuple with the field values, wrapped in a Some.

```
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
```


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

Algorithm

1. Is type Complex?

2. Call unapply

3. Returns Some?

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

  case Complex(r,i) => // #2
```


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

Monday, April 22, 13

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

Unapply is generated for case classes.

You rarely write your own!

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

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
} }
```


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

Monday, April 22, 13

We actually wrote more than necessary before.

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!

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

Monday, April 22, 13

If we pass a partial function, then the _ match is automatically handled for us by the partial function object.
A set of case clauses inside braces is the literal syntax for PartialFunctions, just as (a,b,c) is a literal syntax for tuples, etc., etc.

A PartialFunction is partial in that it doesn't apply for the whole range of possible inputs.

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

Monday, April 22, 13

From math, division is partial for the denominator, which can't be 0!

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

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

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*.

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

If you want var fields, add the var keyword in the c'tor argument list, as needed.

The keyword is named case because its facilitates pattern matching.

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

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

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

Monday, April 22, 13

I don't mean that they play the same role, just that they are both intrinsic and fundamental concepts, used constantly.

Exercise

Pattern matching

ex8-pattern-matching.scala

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

Odds and Ends...

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

Monday, April 22, 13

Several miscellaneous features to know that we can cover quickly.

Primary and Secondary Constructors

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

Monday, April 22, 13

Because you sometimes need more than one c'tor.

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

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

primary constructor

secondary constructor

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

Secondary c'tor's must invoke the primary c'tor first, then they can do additional processing. This model helps avoid duplication in c'tors.

Parent and Child Classes

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

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

pass parameters to parent constructor

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

Monday, April 22, 13

Derived class passes arguments to the parent class' primary c'tor.

Currying

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

Monday, April 22, 13

Not just a feature of Asian cuisine...

Applying a function's arguments one at a time.

Named after Haskell Curry

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

Monday, April 22, 13

Haskell Curry was a pioneer in mathematical Combinatory Logic.

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

2 param. lists

```
def mod3 = mod(3) _
```

"_" required

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

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

Currying methods.

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

Monday, April 22, 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.

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

You can only curry a whole argument list, so if you want to curry each argument, put each one in its own list.

```
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

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

Monday, April 22, 13

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

What does this mean?

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

- modc is a function value that takes an Int argument.
- … and returns a function that takes an Int and returns an Int.

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

What does this mean?

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

- It binds left to right.

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


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

User-defined Operators

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

Monday, April 22, 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"

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

Note: only immutable objects (preferred!). You don't modify "(1,2)" when you add "(.3,.4)" to it!

```scala
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)

  ...
```

"operators"

"Operator overloading"

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

Monday, April 22, 13

Almost all "operators" in scala are just methods defined on types. Very few operators are special cases in the language grammar.

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



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

"Operator overloading"

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

Monday, April 22, 13

Must use "override" modifier when overriding a concrete method.

Sugar for "unary minus"

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

Required when overriding concrete method

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

"Operator overloading"

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

Monday, April 22, 13

Must use "override" modifier when overriding a concrete method.

```
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

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

Monday, April 22, 13

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

Packages and Imports

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

Packages and Imports: Java Style

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


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

Monday, April 22, 13

Normal java-style package declarations and imports, but we'll see slight syntax differences in the next slide.

Packages and Imports: Alternative Style

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


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

Monday, April 22, 13

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

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

namespace style

all types

selective

"rename"

suppress

everything else...

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

Monday, April 22, 13

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

Declarations In the Namespace

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

Even in the same file!

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

Monday, April 22, 13

Declarations can also go in the namespace.

File names and Directories?

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

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...

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

Monday, April 22, 13

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

Directory and package names don't have to match.

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

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

Monday, April 22, 13

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

The Type Hierarchy

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

Monday, April 22, 13

Several miscellaneous features to know that we can cover quickly.

Base of the Type Hierarchy

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

Monday, April 22, 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 *other types*. No *instances!*

AnyRef is equivalent to Java's Object

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

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

`List[+A]`	Immutable, pervasive functional type
`Set[+A]`	Immutable and mutable versions
`Map[+A]`	Immutable and mutable versions

Scala "encourages" using immutables

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

Monday, April 22, 13

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

`TupleN[A1, …,AN]`	literal `(x1,x2,x3, …)`
`Option[A]`	`Some(a)` or `None`
`FunctionN[ -A1,…,-AN,+R]`	Function literals

Option helps avoid nulls

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

Monday, April 22, 13

Tuples are very convenient for ad hoc, fixed sized groups of things.
Option we'll explore later. For now, think of it as a better alternative than allowing something to "null". Rather, it's "optional".

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

Type "variance annotations"

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

Monday, April 22, 13

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

What Does [+A] Mean?

```
class List[+A] {…}

new List[String]
```

is a subclass *of*

```
new List[AnyRef]
```

Covariant

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

The + means the subclassing behavior is “covariant”, i.e., the subclassing of the parameterized type “goes in the same direction” as the subclassing of the type parameter(s).

Example

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

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


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

Monday, April 22, 13

We'll describe this literal syntax for lists later.
The function expects a list of reference type objects. It calls "getClass" on each, which is not supported by Any or AnyVal. So, if we pass it a list of Strings, the function works fine and is none the wiser...

By the way...

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

// Try these variants:
l.foreach(println(_.getClass))

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

Fails!

Works!

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

Monday, April 22, 13

Variations. Note the use of the wildcard "_". Unfortunately, the first one doesn't work, but the second one does. The second one is more idiomatic of functional programming, where we use mapping and iteration. Also, in "foreach(println)" we are passing the println "function" as an argument to foreach.

And this...

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

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

2nd one works

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

Monday, April 22, 13

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

What Does [-A] Mean?

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

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

is a subclass *of*

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

Contravariant in the arg. types.
Covariant in the return types.

3, Dean Wampler, All Rights Reserved

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

Recall that:

```
(AnyRef,AnyRef) => String
```

is equivalent to

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

"Function Literal" syntax

erved

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 Function1...

Function literal (value)

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

Monday, April 22, 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.

Example

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

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

Error!

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

Monday, April 22, 13

What if we pass a (String) => String function? Not allowed by the “contract” of mapper (even though in this example, there would be no type cast errors!)

Example

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

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

Works!

Why arguments must be contravariant!

erved

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.isInstanceOf[Int])
```

Works!

Boolean (and String) are subclasses of Any

erved

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]
```


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

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).

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)
```


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

Monday, April 22, 13

What if we pass a (String) => String function? Not allowed by the “contract” of mapper (even though in this example, there would be no type cast errors!)

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!

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

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.

Scala has declaration-site inheritance...

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

Monday, April 22, 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.

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

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

In Java:

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

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

Variance defined at the call site

erved

In Scala, the type designer specifies the right behavior.

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

Monday, April 22, 13

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

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

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

Monday, April 22, 13

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

Exercise

Look at covariance and contravariance through the Function1 type.

ex9-function-types.scala

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

Function1[-A,+R]

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

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

Avoiding Mutable State (Revisited)

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

Monday, April 22, 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

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

Recall the classic operations on functional data types

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

Monday, April 22, 13

List being the most commonly-used data type.

```
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?

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

Monday, April 22, 13

object …{ def main …} just like Java's static main.

```
$ 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.

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

object …{ def main …} just like Java's static main.

```
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.

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

```
object Main {

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

  args.map(_.toInt)
       .toSet
       .toList
       .sortWith(_ < _)
       .foreach(println(_))
  }
}
```

"_" is a placeholder

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

Monday, April 22, 13

Each "_" matches an argument to the anonymous function. Note the "sortWith". It takes two arguments, so it has two "_".

```
object Main {

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

  args.map(_.toInt)
       .toSet
       .toList
       .sortWith(_ < _)
       .foreach(println(_))
   }
 }
```

Also not needed!

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

Monday, April 22, 13

Each “_” matches an argument to the anonymous function. Note the “sortWith”. It takes two arguments, so it has two “_”.

```
object Main {

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

  args.map(_.toInt)
       .toSet
       .toList
       .sortWith(_ < _)
       .foreach(println)
   }
 }
```


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

Monday, April 22, 13

Passing (_) to println is also inferred (for any single-argument function)

Recap

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

Scala is...

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

a better Java,

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

object-oriented and functional,

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

succinct,
elegant,
and
powerful.

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

Some General Lessons:

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

Objects can be functions.
Functions are objects.

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

Observation:

Any object graph decomposes into values and collections of values.

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

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

Collections:

Construct, iterate, and manage them functionally.

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

Values:

Prefer immutable values.

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

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

Values:

Carefully specify the properties of value types.

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

Monday, April 22, 13

What are allowed for numbers, strings, …, a person's age, street address, … Can we define these properties with "mathematical precision" and test them?

What are the properties of

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

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

Monday, April 22, 13

What are allowed for numbers, strings, …, a person's age, street address, … Can we define these properties with "mathematical precision" and test them?

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

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

Monday, April 22, 13

Sometimes mutable objects make sense. Be sure that their lifecycle “state machine” is well defined.

Prefer functions and mixins (traits) for composition.

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

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

Thanks!

dean@concurrentthought.com
@deanwampler

polyglotprogramming.com/
programmingscala.com

SOME RIGHTS RESERVED

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

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

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

Monday, April 22, 13

Topics we probably won't have time to cover.

Domain-Specific Languages

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

Monday, April 22, 13

Why Scala is so good for creating DSLs.

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

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

Internal DSLs

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

Internal DSLs are idiomatic syntax in the host language

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

Features for Building Internal DSLs

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

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

Internal (or embedded) DSLs are idiomatic ways of writing the host language itself. They aren't separate languages with a dedicated parser, etc.

Infix operator notation

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

same as

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


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

Monday, April 22, 13

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

Infix operator notation

```
1 hour fromNow
```

instead of

```
1.hour(fromNow)
```


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

Implicit Type Converters

```
1 hour fromNow
```

Need the hour method on Int

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

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…

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

Monday, April 22, 13

"now" is not shown. You can write a routine with Java's (broken) Date and Calendar classes (or use JodaTime).

Implicit Type Converters

Must import...

```
import Hour._

1 hour fromNow
```

fromNow() *passed to hour.*

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

int2Hour called, returning Hour(1).

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

Monday, April 22, 13

Must import the Hour object's methods.

Make your own controls

```
// Print with line numbers.


loop (new File("…")) {
  (n, line) =>

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


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

Monday, April 22, 13

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?

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

Output on itself:

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


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

```
import java.io._

object Loop {

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


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

Monday, April 22, 13

Here's the code that implements loop...

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

_ like * in Java

"singleton" class == 1 object

```
object Loop {
```

loop "control"

two parameters

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

function taking line # and line

like void

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

Monday, April 22, 13

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

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

Monday, April 22, 13

The oval highlights the comma separating the two parameters in the list. The callout at the top right is a reminder of what we want client code to look like when the DSL is used. 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

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

Monday, April 22, 13

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

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

Monday, April 22, 13

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...

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

Monday, April 22, 13

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...

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

Monday, April 22, 13

Here is the nested method, doLoop.

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

Pure functional "looping" technique

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

It Is Tail Recursive

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

Scala optimizes tail recursion into loops

ed

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

Exercise

The whileTrue loop

ex10-while-true.scala

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

I want to write:

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


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

A poor-man's while loop...

Implement whileTrue

- Here is the declaration:

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

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

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

Monday, April 22, 13

Note: no parentheses, even empty ones, before the => in the arg. types.

By-name vs. by-value

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

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


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

By-value parameters aren't just functions, they are anything else that isn't explicitly a by-value. What's the benefit of by-name? They aren't evaluated until called.

By-name vs. by-value

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

"by-name" parameter

Called w/out parentheses

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

"by-value" parameter

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

By-value parameters aren't just functions, they are anything else that isn't explicitly a by-value. What's the benefit of by-name? They aren't evaluated until called.

Why use by-name parameters?

They are evaluated each time they are referenced.

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

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

Monday, April 22, 13

This is important; we want the block to be evaluated for each iteration, not evaluated once with the result being passed as the second argument to whileTrue and reused over and over again.

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

They return something different each time!

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

Monday, April 22, 13

This is important; we want the block to be evaluated for each iteration, not evaluated once with the result being passed as the second argument to whileTrue and reused over and over again.

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

Evaluated each time

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

External DSLs

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

Features for Building External DSLs

- Parser Combinator Library

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

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

Consider this Grammar

```
repeat 10 times {
  say “hello”
}
```

BNF grammar

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

repetition

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

Monday, April 22, 13

There are several ways to write a BNF grammar.

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
```


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

Monday, April 22, 13

Yellow are all methods. “~>” means don’t save the thing on the left. “<~” means don’t save the thing on the right. “wholeNumber” and “stringLiteral” are parser combinators defined by the JavaTokenParsers trait that we will reuse.

```scala
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
}
```


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

Monday, April 22, 13

The required import statement and object (or class) wrapper.

```
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.

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

Monday, April 22, 13

The ^^{…} are how we tell it what to really do with the parsed tokens.
Without these additions, each method just returns a parse object.

In Action...

```
val input =
"""repeat 10 times {
  say "hello"
}"""

RepeatParser.parseAll(
  RepeatParser.repeat, input)
```


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

Monday, April 22, 13

Actually running the thing on some input.

In Action...

```
“hello”
“hello”
“hello”
“hello”
“hello”
“hello”
“hello”
“hello”
“hello”
“hello”
```

The output

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

The result of our script in our DSL.