The Seductions of Scala

Dean Wampler

dean@deanwampler.com
@deanwampler
polyglotprogramming.com/talks
programmingscala.com
Co-author, Programming Scala

programmingscala.com

Available now from oreilly.com, Amazon, etc.
Why do we need a new language?

I picked Scala in late 2007 to learn because I wanted to learn a functional language. Scala appealed because it runs on the JVM and interoperates with Java...
First reason, we need the benefits of FP.
.... for concurrency.
.... for concise code.
.... for correctness.
We need a better Object Model
... for composability.
... for scalable designs.
Scala’s *Thesis*: Functional Programming Complements Object-Oriented Programming

Despite surface contradictions...

We think of objects as mutable and methods as state-modifying, which is often appropriate. But using functional idioms reduces bugs and often simplifies code. You can make your objects immutable, too!
But we want to keep our investment in Java/C#.
Scala is...

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

- Helped design java generics.
- Co-wrote GJ that became javac (v1.3+).
- Understands Computer Science and Industry.
Everything can be a Function
Objects as Functions
class Logger(val level: Level) {

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

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

Note how variables are declared, “name: Type”.

makes “level” an immutable field

class body is the “primary” constructor

method
class Logger(val level: Level) {

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

Note how variables are declared, "name: Type".
class Logger(val level: Level) {
  def apply(message: String) = {
    // pass to logger system
    log(level, message)
  }
}

val error = new Logger(ERROR)
...
error("Network error.")
class Logger(val level: Level) {

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

val error = new Logger(ERROR)

apply is called

"function object"

error("Network error.")

Tuesday, July 20, 2010
Adding a parameterized arg. list after an object causes the compiler to invoke the object's "apply" method.
Put an arg list after any object, apply is called.
Everything is an Object
Int, Double, etc. are true objects.

But they are compiled to primitives.
First, About Lists

The same as this “list literal” syntax:

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

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. Note the "operator notation"; x.m(y) ==> x m y
val list = Nil.

1 :: 2 :: 3 :: 4 :: 5 :: Nil

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

val list = Nil.::(5).::(4).::(3).::(2).::(1)
“Operator” Notation

“hello” + “world”

is actually just

“hello”.+(“world”)
Similarly

“hello” compareTo “world”

is actually just

“hello”.compareTo(“world”)
val map = Map(
  "name" => "Dean",
  "age"  => 39)

Maps also have a literal syntax. Is this a special case in the language grammar?
Oh, and Maps

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

"baked" into the language grammar?

No, just method calls...

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

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

An “implicit conversion” converts a string to a type that has the -> method.
Collections like List and have a set of common operations that can be used on them.
Back to functions as objects...
Let's map a list of strings with lower-case letters to a corresponding list of uppercase strings.
map called on list

list map {
  s => s.toUpperCase
}

map argument list

“function literal”

function argument list

function body

No () and ; needed!

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.
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’ve used type inference a lot...
How the Sausage Is Made

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

Parameterized type (like <A> in Java)

Declaration of `map`

The function argument

Here's the declaration of List's map method (lots of details omitted...).
How the Sausage Is Made

like an “abstract” class

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

Java’s “Object”

No method body: => abstract

Tuesday, July 20, 2010
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 important details...) The trait defines an abstract method “apply”. 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

becomes:

new Function1[String, String] {
def apply(s: String) = {
  s.toUpperCase
}
}

No “return” needed

Compiler generates an anonymous class

You use the function literal syntax and the compiler instantiates an anonymous class using the corresponding FunctionN trait, with a concrete definition of apply provided by your function literal.
```scala
list map {s => s.toUpperCase}
```

*using a function value*

```scala
val f = new Function1[...,...] {
  def apply(s: String) = {
    s.toUpperCase
  }
}
```

```scala
list map {s => f(s)}
```

*Alternative*

Tuesday, July 20, 2010

Back to where we started. Note again that we can use "(...)" instead of "(...)" for the argument list (i.e., the single function) to map.
Since *functions* are *objects*, they could have *state*...
class Counter[A](val inc:Int = 1) extends Function1[A,A] {
  var count = 0
  def apply(a:A) = {
    count += inc
    a // return input
  }
}

val f = new Counter[String](2)
val l1 = "a" :: "b" :: Nil
val l2 = l1 map {s => f(s)}
println(f.count) // 4
println(l2)      // List("a","b")
We've seen a lot of syntax. Let's recap a few of the ways Scala keeps your code succinct.
Infix Operator Notation

"hello" + "world"

same as

"hello".+("world")

Great for DSLs!
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 uses [] rather than <>, so you can use “<” and “>” as method names!
// Scala
val persons = new HashMap[String, Person]

no () needed.
Semicolons inferred.
User-defined *Factory Methods*

```scala
val persons = Map(
    "dean" -> deanPerson,
    "alex" -> alexPerson)
```

*no new needed.*

*Returns an appropriate subtype.*
class Person {
  private String firstName;
  private String lastName;
  private int age;

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

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

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

  public int getAge() { return this.age; }
  public void setAge(int age) {
    this.age = age;
  }
}
class Person(
    var firstName: String,
    var lastName: String,
    var age: Int)

Typical Scala!

Scala is much more succinct. It eliminates a lot of boilerplate.
class Person(
  var firstName: String,
  var lastName: String,
  var age: Int)

Class body is the “primary” constructor
Parameter list for c’tor

Makes the arg a field with accessors
No class body {...}. nothing else needed!

Scala is much more succinct. It eliminates a lot of boilerplate.
Actually, not exactly the same:

```scala
val person = new Person("dean",...)
val fn = person.firstName
person.firstName = "Bubba"

// Not:
// val fn = person.getFirstName
// person.setFirstName("Bubba")
```

Doesn’t follow the *JavaBean convention.*
However, these are function calls:

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

    def firstName = _firstName

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

Uniform Access Principle

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!
Scala's Object Model: Traits

Composable Units of Behavior

Fixes limitations of Java's object model.
Java

class Queue extends Collection implements Logging, Filtering
{
    ...
}
Java’s object model

• **Good**
  • Promotes abstractions.

• **Bad**
  • No *composition* through reusable *mixins*.

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

... or like

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

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

A pure abstraction (in this case...)

Tuesday, July 20, 2010
Log **put** and **get**

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

("get" is similar.) “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` and `get`

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

What is “super” bound to??

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

Tuesday, July 20, 2010
class StandardQueue[T] extends Queue[T] {
  import ... ArrayBuffer
  private val ab =
    new ArrayBuffer[T]
  def put(t: T) = ab += t
  def get() = ab.remove(0)
  ...
}

Concrete (boring) implementation
val sq = new StandardQueue[Int]
  with QueueLogging[Int]

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

Example use

We instantiate StandardQueue AND mixin the trait. We could also declare a class that mixes in the trait.
The “put(10)” output comes from QueueLogging.put. So “super” is StandardQueue.
val sq = new StandardQueue[Int] with QueueLogging[Int]

sq.put(10) // #1
sq.get()  // #2

// => put(10) (on #1)
// => get(10) (on #2)
Like Aspect-Oriented Programming?

Traits give us advice, but not a join point "query" language.

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

trait QueueFiltering[T] extends Queue[T] {
  abstract override def put(t: T) = {
    if (veto(t))
      println(t + " rejected!")
    else
      super.put(t)
  }
  def veto(t: T): Boolean
}

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

trait QueueFiltering[T] extends Queue[T] {
  abstract override def put(t: T) = {
    if (veto(t))
      println(t + " rejected!")
    else
      super.put(t)
  }

  def veto(t: T): Boolean
}

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”

Anonymous Class

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)
// => 0, 1, 2

Filtering occurred before logging

Example use

The filter trait’s “put” is invoked before the logging trait’s put.
What if we reverse the order of the Traits?
val sq = new StandardQueue[Int]
  with QueueFiltering[Int]
  with QueueLogging[Int]
  {
    def veto(t: Int) = t < 0
  }
for (i <- -2 to 2) {
    sq.put(i)
}
println(sq)

// => put(-2)
// => -2 rejected!
// => put(-1)
// => -1 rejected!
// => put(0)
// => put(1)
// => put(2)
// => 0, 1, 2

logging comes before filtering!

Now, the logger trait’s “put” is invoked before the filtering trait’s put.
Loosely speaking, the precedence goes right to left.

"Linearization" algorithm
Method Lookup Order

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

*Simpler cases, only...*
Traits are also powerful for mixin composition.
Logger, revisited:

```scala
trait Logger {
  def log(level: Level, message: String) = {
    Log.log(level, message)
  }
}

val dean = new Person(...) extends Logger

dean.log(ERROR, "Bozo alert!!")
```

I changed some details compared to our original Logger example, e.g., no "level" field. Mix in Logger.
DSLs

Yet more features for DSL creation...

Fixes limitations of Java’s object model.
Building Our Own Controls

Exploiting First-Class Functions
Recall *infix* operator notation:

\[
\begin{align*}
1 + 2 & \quad // \quad => \quad 3 \\
1. + (2) & \quad // \quad => \quad 3
\end{align*}
\]

*also the same as*

\[
1 + \{2\}
\]

*Why is this useful??*
Make your own controls

// Print with line numbers.

loop (new File("...")) {
    (n, line) =>
        printf("%3d: %s\n", n, line)
}
Make your own controls

// Print with line numbers.

control? File to loop through

loop (new File("...")) {
(n, line) => Arguments passed to...

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

what do for each line

How do we do this?
Output on itself:

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

object Loop {

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

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

"singleton" class == 1 object

loop "control"
two parameters

function taking line # and line

_ like * in Java

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".
The oval highlights the comma separating the two parameters in the list. Watch what we do on the next slide...
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) =>
        printf("%3d: %s\n", n, line)
}

import new method

1st param.: a file

2nd parameter: a “function literal”

Tuesday, July 20, 2010

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(n:Int) = {...}
        doLoop(1)
    }
}
object Loop {
  ... 
  def doLoop(n: Int): Unit = {
    val l = reader.readLine()
    if (l != null) {
      f(n, l)
      doLoop(n+1)
    }
  }
}
doLoop is Recursive. There is no mutable loop counter!

Classic Functional Programming technique
It is *Tail* Recursive

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

*Scala optimizes tail recursion into loops*

A tail recursion - the recursive call is the last thing done in the function (or branch).
Recap: Make a DSL

// Print with line numbers.
import Loop.loop

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

Tuesday, July 20, 2010
We’ve used some syntactic sugar (infix operator notation, substituting (...) for (...)) and higher-order functions to build a tiny DSL. Other features supporting DSLs include implicits
FP is going mainstream because it is the best way to write robust concurrent software. Here's an example...
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](t: T)

Provides factory, pattern matching, equals, toString, and other goodies.

Tuesday, July 20, 2010

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].
Which is the better API?

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

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

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

No “if” statement

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

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.
FP is going mainstream because it is the best way to write robust concurrent software. Here’s an example...
When you share *mutable* state...

*Hic sunt dracones*
*(Here be dragons)*

**Hard!**
Actor Model

• *Message* passing between autonomous *actors*.
• No *shared* (mutable) *state*. 
The actor model is not new!!
2 Actors:

“self”

???

“exit”

Display

draw
draw
error!

exit
package shapes

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

abstract class Shape {
  def draw()
}

Hierarchy of geometric shapes

Tuesday, July 20, 2010

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

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

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

Hierarchy of geometric shapes

Tuesday, July 20, 2010

“Case” classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape. The “case” keyword makes the arguments “vals” by default, adds factory, equals, etc. methods. Great for “structural” objects.
(Case classes automatically get generated equals, hashCode, toString, so-called “apply” factory methods - so you don’t need “new” - and so-called “unapply” methods used for pattern matching.)
Case classes for 2-dim. points and a hierarchy of shapes. Note the abstract draw method in Shape. For our example, the draw methods will just do "println(this.toString)".
An actor that waits for messages containing shapes to draw. Imagine this is the window manager on your computer. It loops indefinitely, blocking until a new message is received...
An actor that waits for messages containing shapes to draw. Imagine this is the window manager on your computer. It loops indefinitely, blocking until a new message is received...
```
receive {
  case s: Shape => s.draw()
  sender ! "drawn"
  case "exit" => println("exiting...")
  sender ! "bye!"
  // exit
  case x => println("Error: " + x)
  sender ! ("Unknown: " + x)
}
```

“Receive” blocks until a message is received. Then it does a pattern match on the message. In this case, looking for a Shape object, the “exit” message, or an unexpected object, handled with the last case, the default.
Each pattern is tested and the first match “wins”. The messages we expect are a Shape object, the “exit” string or anything else. Hence, the last “case” is a “default” that catches anything.
After handling each message, a response is sent to the sender, which we get by calling the “sender” method. The “!” is the message send method (from Erlang).
package shapes
import ...
object ShapeDrawingActor extends Actor {
  def act() {
    loop {
      receive {
        case s: Shape =>
          s.draw()
          sender ! "drawn"
        case "exit" =>
          println("exiting...")
          sender ! "bye!" //; exit
        case x =>
          println("Error: " + x)
          sender ! ("Unknown: " + x)
      }
    }
  }
}
import shapes._
import scala.actors.Actor._

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

    self.receive {
        case reply => println(reply)
    }
}
The `!` method sends a message. Then we wait for a reply. In our receive method, we match on any “reply” and just print it.

This script uses the method “self” to get the actor corresponding to “me”.

---

**Tuesday, July 20, 2010**

The “!” method sends a message. Then we wait for a reply. In our receive method, we match on any “reply” and just print it. This script uses the method “self” to get the actor corresponding to “me”.

... ShapeDrawingActor.start()
sendAndReceive(
    Circle(Point(0.0,0.0), 1.0))
sendAndReceive(
    Rectangle(Point(0.0,0.0), 2, 5))
sendAndReceive(3.14159)
sendAndReceive("exit")

// => Circle(Point(0.0,0.0),1.0)
// => drawn.
// => Rectangle(Point(0.0,0.0),2.0,5.0)
// => drawn.
// => Error: 3.14159
// => Unknown message: 3.14159
// => exiting...
// => bye!
... ShapeDrawingActor.start()
sendAndReceive(
    Circle(Point(0.0,0.0), 1.0))
sendAndReceive(
    Rectangle(Point(0.0,0.0), 2, 5))
sendAndReceive(3.14159)
sendAndReceive("exit")

// => Circle(Point(0.0,0.0),1.0)
// => drawn.
// => Rectangle(Point(0.0,0.0),2.0,5.0)
// => drawn.
// => Error: 3.14159
// => Unknown message: 3.14159
// => exiting...
// => bye!
...receive {
  case s:Shape =>
  s.draw()
  sender ! "drawn"
  case ...
  case ...
}
Recap
Scala is...
a better Java and C#,
object-oriented and functional,
succinct, elegant, and powerful.
Thanks!

dean@deanwampler.com
@deanwampler

polyglotprogramming.com/talks
programmingscala.com
Functional Programming
What is Functional Programming?

Don’t we already write “functions”?
y = sin(x)

Based on Mathematics

“Functional Programming” is based on the behavior of mathematical functions and variables.
\[ y = \sin(x) \]

Setting \( x \) fixes \( y \)

\[ \therefore \text{variables are immutable} \]
20 += 1

We never modify the 20 “object”
Concurrency

No mutable state

∴ nothing to synchronize

FP is breaking out of the academic world, because it offers a better way to approach concurrency, which is becoming ubiquitous.
When you share *mutable* state...

*Hic sunt dracones*
(Here be dragons)
A math function doesn’t change any “object” or global state. All the work it does is returned by the function. This property is called “referential transparency”.

\[ y = \sin(x) \]

Functions don’t change state.

\[ \therefore \text{side-effect free} \]
Side-effect free functions

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

*Side-effect free* functions are far less likely to introduce subtle integration bugs, especially in concurrent systems. By encouraging immutable objects (e.g., when methods return new objects, rather than modify existing ones), that improves concurrency robustness.
$\tan(\Theta) = \frac{\sin(\Theta)}{\cos(\Theta)}$

Compose functions of other functions

:. *first-class citizens*
More on DSLs

Yet more features for DSL creation...

Tuesday, July 20, 2010
Fixes limitations of Java's object model.
“Pimp My Library”

Typesafe “monkey patching”

A funny name for a powerful feature, simulating open types (i.e., adding behavior to types) in a type-safe way.
Recall our “Loop”

// Print with line numbers.
import Loop.loop

loop (new File("...")) {
  (n, line) =>
    printf("%3d: %s\n", n, line)
}
Suppose we want a loop method on File instead?
val file = new File("...")

file.loop {
  (n, line) =>
  printf("%3d: %s\n", n, line)
}
In languages with open types, like Ruby, we can easily add new methods to types or individual objects. Can’t do that in Scala, right?
Implicits

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

object WithLoop {
  implicit def file2WithLoop(file: File) = 
    new WithLoop (file)
}

You can't do that in Scala, but “implicits” let you define a conversion from the type you have to a new type that has the method you want. The syntax you use looks as if you have the method on the original type. The compiler finds the best-matching implicit conversion method in scope and applies it for you.
The implicit function will be used by the compiler to convert a File to a WithLoop, then we can call the loop method, which now has only one argument, the function to apply to each line in the file.
Technical note. Because of closures, we don't actually need to make the File a field (val) of the class. Also, it's okay for the object and class to have the same names; they are called “companions”.
The compiler sees you try to call loop() on a File object, but that method doesn’t exist. So, it looks for another type that has the loop method and an implicit conversion method in scope to convert from File to the other type. It then calls loop on the new object.
Implicits can be used to mimic Haskell-like type classes.

http://debasishg.blogspot.com/2010/06/scala-implicits-type-classes-here-i.html