BOF3024 - Learning Scala: A Practical Approach

Bob Treacy (@bobtreacy)
http://www.iq.harvard.edu/people/robert-treacy

Michael Bar-Sinai (@michbarsinai)
http://www.iq.harvard.edu/people/michael-bar-sinai
http://mbarsinai.com

IQSS
The Institute for Quantiative Social Science
HARVARD UNIVERSITY

Agenda

1. Why Scala?
2. Colliding Concepts
3. New Concepts
4. Tools
5. Tips or, how to learn Scala *and* keep your job
6. Case study

Slides and code:

https://github.com/IQSS/javaone2014-bof3024F

Why Scala

- Implement a project based on a template that's relatively close to the final project
  - Such as Akka distributed worker template
- Create a web application using Play!
  - has a Java flavor as well, but template engine uses Scala only
- Try some functional programming
  - Scala may not be your best choice
- Try some *postfunctional* programming

> *If you look at the features Scala provides, it is substantially a functional language, but... it does not force you to adopt the functional style. I think* ***postfunctional*** *is a good term for that blend.*
>
> *Martin Odersky, http://www.scala-lang.org/old/node/4960*

- Everyone's favorite reasons:
  - Learn a new and interesting language
  - Bend your mind in new ways.
  - Just because

Colliding Concepts

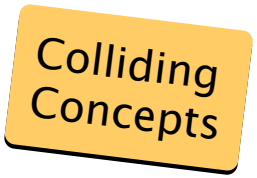
While sharing the JVM, and Java's extensive libraries, Scala and Java are very different languages. Some words have different meanings.

Intuitions created by years of working with Java can be misleading. You may end up doing as much *unlearning* as you do *learning*.

It's a good thing, if you're ready for it.

So We've gathered a few examples.

Syntax



Semicolon inference

Semi-colons are optional at the end of a line containing an expression

- Two expressions, two lines

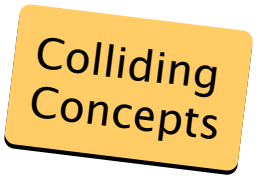
```
print("hello, ")
println("world!")
```

- Two expressions, one line

```
print("hello, "); println("world!")
```

- One expression, multiple lines: a line ending is treated as a semicolon unless:
  1. End in a word that would not be legal at the end of a expression
  2. Next line begins with a word that cannot start a expression
  3. Line ends while inside parentheses or bracket

Syntax

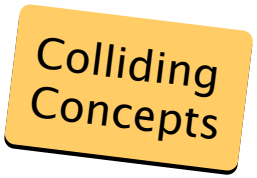


[]s, ()s and {}s

`[]` are for type parameters, `<>` are for XML literals and operators, `()` and `{}` are interchangeable.

→ Array access is done using `array(index)`, not `array[index]`!

Syntax



Type name goes after the variable name

Java:

```
String myString;
```

Scala:

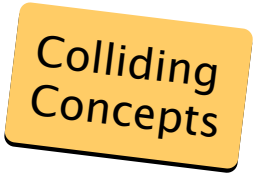
```
  myString: String
```

Personal goal: stop getting confused as early as Q3 of 2041.

However, declaring a value's type is not always necessary – **in many cases Scala can infer types on its own:**

```
val myString = "Hello!" //myString is a String
```

Constructors, Reconstructed



Unlike Java, **each class has a single primary constructor**, whose definition dovetails the class' definition:

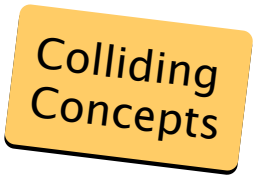
```
class BOF( var title:String,
           val num:Int,
           topic: String ) {
  def desc = title + ", a BOF about " + topic
}
```

- `var v:T` becomes a mutable field of the class
- `val v:T` becomes a final field of the class
- `v:T` is a value that can be used inside the class
- The body of the primary constructor is all the code directly within the class.

Alternative description (Odersky):

> *Classes have parameters, just like methods do.*

Auxiliary Constructors



It is also possible to define **auxiliary constructors**.

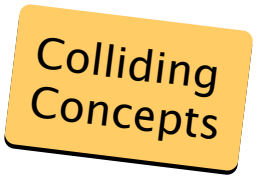
Auxiliary Constructors:

- Are called `this(...)`
- Have to begin by calling the primary constructor, or another auxiliary constructor.
  - Either way, that constructor is referred to as `this`.

```
class Bof( var title:String,
           val num:Int,
           topic: String ) {
  def desc = title + ", a BOF about " + topic

  def this( title:String ) = {
    this( title, title.size, "the topic of '%s'".format(title))
  }
}
```

Method invocation



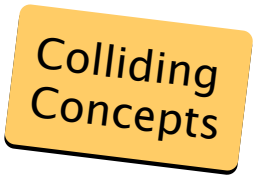
- Methods that take no parameters may be invoked by just calling them – no need for `()` after the name.

```
scala> def f()={ println("hello") }
f: ()Unit
scala> f // prints "hello"
scala> f() // prints "hello"
scala> f _
res9: () => Unit = <function0>
```

- Single parameter methods can be invoked using an alternative to the dot-notation:
  `a + b` is the same as `a.+(b)`
- Right associative methods: When last character of method name is `:`
  `a +: b` is the same as `b.+:(a)`
- The `apply()` method – creates a "default" method for objects – no need to type the `apply` part.
  `instance()` is really `instance.apply` which is really `instance.apply()`

```
scala> (f _)() // prints "hello"
```

The Point of No `return`



All methods return values. **Thus, the `return` keyword is often superfluous**.

In the absence of an explicit return keyword, a method will return the last value computed.

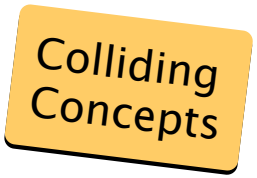
`return` is useful as a type of `break` that returns a value.

These samples are functionally the same:

```
class SumWithInt {
  var sum = 0
  def add(b: Int):Int={
    sum += b
    return sum
  }
  def add1( i:Int )={
    sum += i
    sum
  }
}
```

Code: Returns.scala at presentation repo

`Unit` filling the `void`



All methods return values. **Thus, declaring a method to be of a `void` type is impossible.**

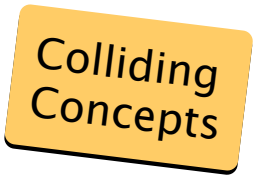
Method that has nothing to return, can return `Unit`, a type with a single value, marked as `()`. This allows everything to be an **expression** – Scala has no statements.

These samples are functionally the same:

```
class SumWithUnit {
  var sum = 0
  def add(b: Int): Unit ={
     sum += b
     return ()
  }
  def add1( i:Int ) {
    sum += i
    ()
  }
  def add2( i:Int ) = {
    sum += i
  }
  def add3( i:Int ) = sum += i
}
```

Code: Returns.scala at presentation repo

`class` vs. `object`



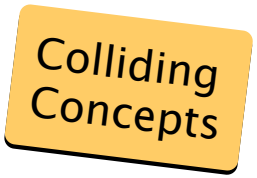
Using the `object` keyword, programs can declare and construct an instance at design time. This is different from the `class` keyword, where instance construction is done at runtime, using the `new` keyword.

```
object Counter {
  var count = new java.util.concurrent.atomic.AtomicInteger(0)
  def up() = count.addAndGet(1)
  def down() = count.addAndGet(-1)
  def value = count.get
}
```

```
scala> Counter.up
res5: Int = 1
```

- The created object is a singleton, initialized the first time its called.
- Objects can extend classes, and traits, as usual.
- The object's type is implicitly defined, but can be accessed (e.g. when creating DSLs).

Companion Objects replace `static`



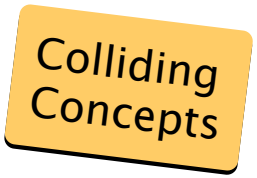
There is no `static` class members in Scala. Functionality that belongs to a class (rather than to its instances) goes in that class' **Companion Object**.

```
class Burrito private( val filling:String ) { ... }

object Burrito {
  var count = new java.util.concurrent.atomic.AtomicInteger(0)
  def makeWith( filling:String ) = {
    count.incrementAndGet()
    new Burrito(filling)
  }
  def brag() = "%,d burritos served".format(count.get)
}
```

```
scala> Burrito makeWith "beans"
res22: Burrito = Burrito with beans
...
scala> Burrito.brag
res14: String = 6 burritos served
```

`apply()` is the new `new`

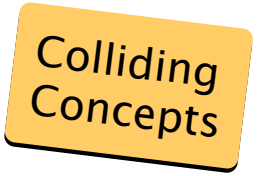


Define an `apply` method on a companion object, and you can skip `new`:

```
object Burrito {
  var count = new java.util.concurrent.atomic.AtomicInteger(0)
  def makeWith( filling:String ) = {
    count.incrementAndGet()
    new Burrito(filling)
  }
 def apply( filling:String ) = makeWith(filling)
  ///...
}
```

```
scala> Burrito("Everything")
res28: Burrito = Burrito with Everything
```

Starting an Application



No `static` → No `public static void main`.

Implementing the `App` trait turns `objects` into executable applications. It uses the body of the object as a main method.

```
object MyApplication extends App {
  if (args.contains("-h")) printHelpAndQuit();
  // rest of code goes here
}
```

- To access the argument list, use `args`.
- There is a `main` method – normally, it does not need to be explicitly overridden.
- The object's fields will not be initialized when the object's body is executed.
- Trait `Application` is deprecated as of version 2.9.0

Collection Library

Scala's collection library is very different from Java's. Scala approach to collections is different.

- Most collections have a mutable and immutable version. Same name, different package.
  - If the package is not imported or the full name is not used, default is immutable version
- Java collections have minimal API, whereas Scala's collections has large API
  - `java.util.List` doesn't have a `last()` method, as it can be implemented by `size()` and `get()`

This may be one of the areas where it's simpler to read the official intro rather than learn as you go.

http://docs.scala-lang.org/overviews/collections/introduction.html

What Do Mean By `List`?

Java's `List`

Ordered collection of elements.

Scala's `List`

The classic linked list from CS 101 / intro to functional programming (with some extras thrown in, and without the annoying TA).

When you want a Java type of `List`, use Scala's `Seq` or `Buffer`

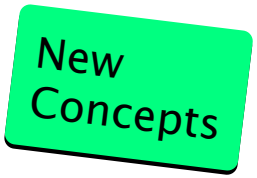
New Concepts

Scala introduces many concepts that are not present in Java. Often, when you look at Scala code, it's unclear where values come from, how can objects extends so many classes, and so on.

This confused us too.

So We've gathered a few examples.

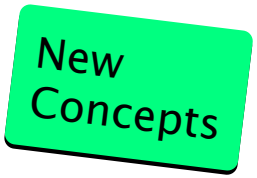
`val` vs. `var`



- `val` – immutable value
- `var` – variable (mutable value)
- `def` – defines methods, but can also be used to define immutable values.
- Immutability bias – when possible, use immutable values and data structures.

```
val workerId = UUID.randomUUID().toString
var currentWorkId: Option[String] = None
```

Traits, not Interfaces



Traits define signatures of supported methods, like Java interfaces, but can also provide method implementations and fields.

```
trait Foo {
  def bar( b:Baz )
}
```

They can also be "mixed in" to create quasi-multiple inheritance

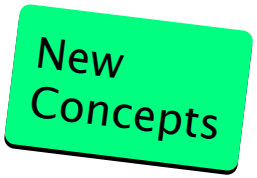
```
trait Fudge {
  def drip(s:Shirt) = {...}
}

trait Sprinkles
trait Peanuts

class Food
class IceCream extends Food

class TraitTest {
  var desert = new IceCream with Fudge with Sprinkles with Peanuts
}
```

Case Classes



A class that's specialized for storing compound immutable values. By deciding to use a case class rather than a normal class, you get:

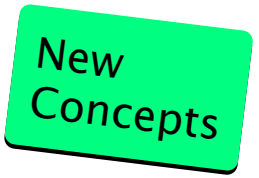
- Automatic, proper `toString`, `equals`, `hashCode`, getters.
- `copy` method to create new objects based on existing ones.
- Companion object, complete with `apply` and `unapply` methods

```
abstract class Session

case class BOF(num:Int, title:String) extends Session
case class Tutorial(num:Int, technology:String) extends Session
case class Keynote( title:String, speaker:String ) extends Session
case class JUG( groupName: String ) extends Session
```

```
scala> val jos = BOF(3024,"Learning Scala the Practical Way")
jos: BOF = BOF(3024,Learning Scala the Practical Way)
scala> val jos2 = jos.copy( num = 1024 )
jos2: BOF = BOF(1024,Learning Scala the Practical Way)
```

Pattern Matching

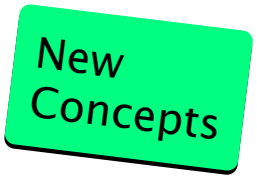



```
object SessionPrinter {
  def print(s:Session)={
    s match {
      case BOF(n,t) => "%d: BOF %s".format(n,t)
      case Tutorial(n,t) => "Learn about %s! (%d)".format(t,n)
      case JUG(g) => "Meet with friends from %s".format(g)
      case _ => s.toString
}}}
```

```
scala> val mySched = Seq( BOF(1,"Learn Scala"), Tutorial(40,"NetBeans"), JUG
scala> mySched.map( SessionPrinter.print(_) ).foreach( println(_) )
1: BOF Learn Scala
Learn about NetBeans! (40)
Meet with friends from jArgentina JUG
```

Pattern matching is a very versatile control structure, whose full options are beyond the scope of this talk. In short, it the mother of all `switch` statements. And then some.

Implicits

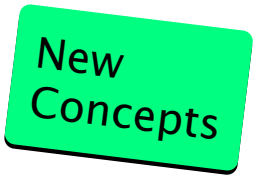


implicit keyword

implicit conversion
if compiler comes across a type error it will look for an appropriate conversion method
when you wish you could add a method to a class you didn't write

implicit parameters
arguments to a method call can be left out
compiler will look for default values to supply with the function call

Not Covered



Scala has many other new concept we can't cover here, but are worth looking into:

- `Option` type
- `Either` type
- Scoping of `private` field modifiers
- Various `import` options
- Delimited Continuations
- Macros
- Rich type system
- Currying
- Working with functions
- Advanced pattern matching
- Tuples
- …

Tools

- scala REPL - play around with code
- IDEs
    - NetBeans - Scala Plugin
    - Intellij IDEA
    - Eclipse
    - Or, text editors such as SublimeText, Atom
- ScalaDoc

Tools

- SBT

```
import com.github.retronym.SbtOneJar._
name := "galileowrapper"
version := "1.0"
scalaVersion := "2.10.3"
libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-contrib" % "2.2.1",
  ...
  "edu.harvard.iq" % "consilience-core" % "1.0-SNAPSHOT",
  "junit" % "junit" % "4.4")
resolvers ++= Seq(
  "Typesafe Repository" at
  "http://repo.typesafe.com/typesafe/releases/",
...
  "Local Maven Repository" at
  "file://"+Path.userHome.absolutePath+"/.m2/repository")

oneJarSettings
mainClass in oneJar := Some("worker.Main")
```

Tips

Spotting a good starter project

- Look at activator templates
- One-offs
- Scripts

Resources

- Books
  - Scala for the impatient by Cay S. Horstmann 1 chapter per lunch, and you're done in 22 business days.
- Meetups
- Typesafe
  - Webcasts, Newsletters
  - DZone Refcard (Scala + Akka)
- MOOCs

Don't Believe Everything You Hear About FP

Tips

Functional programming is great for some things, and not-so-great for others. It is somewhat hyped at the moment.

Scala, a **postfunctional** language, allows you to choose the approach you deem best, or just feel like trying. Don't be confused by the internets.

- Mutability is OK, as long as there's no concurrent updates
  - `java.util.concurrent` is still there for you
  - Object creation and clean-up may be cheap, but it's not free!
  - Side-effects are hard to reason about formally. This is why we still have jobs.
- Tail recursion is not the best thing since sliced bread
  - It's a way to coerce an recursive algorithm into a shape that will allow the compiler to further coerce it into a loop.
  - Makes total sense in Lisp, ML etc. where there are no loops.
  - You're writing in Scala. You can actually use a loop.
  - When you do use a tail recursion, don't forget `@tailrec`.

Having said that...

Ease Into Functional Programming

Tips

At your leisure, consider re-writing existing, "Scala-with-Java-accent" code using functional programming idioms.

Case study: Re-writing in Scala

Problem: Create a poor-man's Markdown to HTML conversion, supporting `<p>`s and single level `<ul>`s.

Input: Sequence of `String`s in markdown syntax

> *Lines starting with `*` should become `<li>`s, grouped and nested in a `<ul>` element.*

Output: Sequence of `<p>` and `<ul>` elements, according to Markdown rules.

Initial code:


```
def html1( rawStrings : Seq[String] ):String = {
  val cleaned = mutable.Buffer[String]()
  for ( l <- rawStrings ) {
    cleaned += l.trim
  }
  val curUl = mutable.Buffer[String]()
  val html = mutable.Buffer[String]()
  for ( l <- cleaned ) {
    if ( l.startsWith("*") ) {
      curUl += "<li>" + l.substring(1).trim + "</li>"
    } else {
      if ( ! curUl.isEmpty ) {
        html += "<ul>"+curUl.mkString + "</ul>"
        curUl.clear
      }
      html += "<p>" + l + "</p>"
    }
  }

  if ( ! curUl.isEmpty ) {
    html += "<ul>"+curUl.mkString + "</ul>"
  }
  return html.mkString
}
```

Code: Rewrite.scala in presentation repo

Initial code:


```
  def html1( rawStrings : Seq[String] ):String = {
    val cleaned = mutable.Buffer[String]()
    for ( l <- rawStrings ) {
      cleaned += l.trim
    }
    val curUl = mutable.Buffer[String]()
    val html = mutable.Buffer[String]()
    for ( l <- cleaned ) {
      if ( l.startsWith("*") ) {
        curUl += "<li>" + l.substring(1).trim + "</li>"
      } else {
        if ( ! curUl.isEmpty ) {
          html += "<ul>"+curUl.mkString + "</ul>"
          curUl.clear
        }
        html += "<p>" + l + "</p>"
      }
    }

    if ( ! curUl.isEmpty ) {
      html += "<ul>"+curUl.mkString + "</ul>"
    }
    return html.mkString
  }
```

Somewhat more functional:


```
def html2( rawStrings : Seq[String] ) = {
  val cleaned = rawStrings.map( _.trim )

  val curUl = mutable.Buffer[String]()
  val html = mutable.Buffer[String]()
  for ( l <- cleaned ) {
    if ( l.startsWith("*") ) {
      curUl += "<li>" + l.substring(1).trim + "</li>"
    } else {
      if ( ! curUl.isEmpty ) {
        html += "<ul>"+curUl.mkString + "</ul>"
        curUl.clear
      }
      html += "<p>" + l + "</p>"
    }
  }

  if ( ! curUl.isEmpty ) {
    html += "<ul>"+curUl.mkString + "</ul>"
  }
  html.mkString
}
```

Somewhat more functional:


```
def html2( rawStrings : Seq[String] ) = {
  val cleaned = rawStrings.map( _.trim )

  val curUl = mutable.Buffer[String]()
  val html = mutable.Buffer[String]()
  for ( l <- cleaned ) {
    if ( l.startsWith("*") ) {
      curUl += "<li>" + l.substring(1).trim + "</li>"
    } else {
      if ( ! curUl.isEmpty ) {
        html += "<ul>"+curUl.mkString + "</ul>"
        curUl.clear
      }
      html += "<p>" + l + "</p>"
    }
  }

  if ( ! curUl.isEmpty ) {
    html += "<ul>"+curUl.mkString + "</ul>"
  }
  html.mkString
}
```

Somewhat more functional:

Code Rewrite

```
def html2( rawStrings : Seq[String] ) = {
  val cleaned = rawStrings.map( _.trim )

  val curUl = mutable.Buffer[String]()
  val html = mutable.Buffer[String]()
  for ( l <- cleaned ) {
    if ( l.startsWith("*") ) {
      curUl += "<li>" + l.substring(1).trim + "</li>"
    } else {
      if ( ! curUl.isEmpty ) {
        html += "<ul>"+curUl.mkString + "</ul>"
        curUl.clear
      }
      html += "<p>" + l + "</p>"
    }
  }

  if ( ! curUl.isEmpty ) {
    html += "<ul>"+curUl.mkString + "</ul>"
  }
  html.mkString
}
```

```
Seq(1,2,3,4).foldLeft(0)( (p,i)=>p+i )
                  // yields 10
```

Even more functional:


```
def html3( rawStrings : Seq[String] ) = {
  val elements = rawStrings.map( _.trim )
                   .map( s => if (s.startsWith("*"))
                                {"<li>"+s.substring(1).trim+"</li>"}
                              else
                                {"<p>" + s + "</p>"} )

  val grouped = elements.tail.foldLeft( List(List(elements.head)) )(
                   (l,s) => {
                      if ( l.last.head(1) != s(1) )
                          l :+ List(s)
                      else
                          l.dropRight(1) :+ (l.last :+ s)})

  grouped.flatMap( l => if (l.head.startsWith("<li>"))
                          List("<ul>"+l.mkString+"</ul>")
                        else
                           l ).mkString
}
```

Using `foldLeft`, we create a list of lists of `Strings`, and operate on them. Then, use `flatMap` to convert the result to a list of strings, and then use `mkString` to create the final string.

Actual code included in presentation repo at GitHub.

Other Sessions

- Scala
  - **CON1740 – Scala Macros: What Are They, How Do They Work, and Who Uses Them?** Thursday, Oct 2, 11:30 AM – 12:30 PM – Hilton – Continental Ballroom 7/8/9
  - **CON1849 – Event-Sourced Architectures with Akka** Wednesday, Oct 1, 8:30 AM – 9:30 AM – Hilton – Continental Ballroom 7/8/9

- IQSS
  - **BOF5619 – Lean Beans (Are Made of This): Command Pattern Versus MVC**
  - **BOF5475 When The PrimeFaces Bootstrap Theme Isn't Enough** Tuesday, Sep 30, 9:00 PM – 9:45 PM – Hilton – Plaza A
  - **CON5575 Bean Validation: Practical Examples from a Real World Java EE7 Application** Tuesday, Sep 30, 4:00 PM – 5:00 PM – Parc 55 – Cyril Magnin I

Thanks

Visit the IQSS data science team at http://datascience.iq.harvard.edu