

CSE 130: Spring 2012

Crash Course in Scala

Ranjit Jhala, UC San Diego

What is Scala ?

- “Java + ML + Extra Magic”
- Created by [Martin Odersky @ EPFL](#)
- Widely used (Twitter, FourSquare, LinkedIn, ...)

Why Scala?

Why Scala: Multi Paradigm

- Object-Functional Programming
- Post-Functional Programming
- Interesting blend, many exciting new features

Why Scala: Bleeding Edge

- Modern type system (subclassing, generics, implicits, ...)
- Modern concurrency (actors, collections, ...)
- Seamless integration with Java
- Because I wanted to learn how it works!

Plan: Next 6 lectures

Whirlwind Overview + Advanced Features

1. “Crash course”: Expressions, Values, Types
2. Classes, Subtyping & Generic Polymorphism
3. Inheritance, Traits and Mixins
4. Implicits and Type-Classes
5. Laziness
6. Concurrency: Actors, Map-Reduce

Many Resources

Online

- Martin Odersky’s [First Steps in Scala](#)
- Twitter’s [Scala School](#)
- Matt Might’s [Scala In Small Bites](#)
- API Search Engine scalex.org

Books

- Programming in Scala by Odersky, Spoon & Venners
- Scala for the Impatient by Horstmann

Lets Start the “Crash Course”!

Scala 101: Expressions & Types

Like ML, Scala is *statically* typed

- Almost *everything* is an expression, which has a type
- Type system is rather different than ML’s...

Scala 101: Read-Eval-Print Loop

The easiest way to get started is with Scala's REPL

```
$ scala
Welcome to Scala version 2.9.0.1 OpenJDK...
Type in expressions to have them evaluated.
Type :help for more information.
```

```
scala>
```

Scala 101: Read-Eval-Print Loop

Enter expressions and have them evaluated

```
scala> 2
res0: Int = 2
```

```
scala> 2 + 3
res1: Int = 5
```

Everything is an “Object”

So when you type:

```
2 + 3
```

the compiler sees the method call:

```
2.+(3)
```

So + is just a method call! (as are *many* other things...)

Everything is an “Object”: Overloading

Furthermore, unlike ML (and like Java) Scala supports “overloading”

```
scala> 2 + "cat"
res3: String = 2cat
```

```
scala> 2 + 3.0
res4: Double = 5.0
```

Each of these calls the appropriate method on the receiver `Int`.

Everything is an “Object”: Overloading

But don't get carried away...

```
scala> 2 + true
```

```
<console>:8: error: overloaded method value + with alternatives:
  (x: Double)Double <and>
  (x: Float)Float <and>
  (x: Long)Long <and>
  (x: Int)Int <and>
  (x: Char)Int <and>
  (x: Short)Int <and>
  (x: Byte)Int <and>
  (x: String)String
cannot be applied to (Boolean)
     2 + true
       ^
```

Fair enough.

Basic Data

You have your basic built-in types...

```
scala> 4.2 * 6.7
res8: Double = 28.14
```

```
scala> true || false
res9: Boolean = true
```

```
scala> 'c'
res10: Char = c
```

Basic Data

Strings are borrowed from Java...

```
scala> "cat"
res25: java.lang.String = cat
```

```
scala> "cat" + "dog"
res26: java.lang.String = catdog
```

```
scala> "cat" * 3
res26: java.lang.String = catcatcat
```

Basic Data

... and you can take *whatever you like* from Java

```
scala> import java.util.Date
import java.util.Date

scala> new Date()
res27: java.util.Date = Wed May 09 11:17:29 PDT 2012
```

So *every* Java library in the known universe trivially usable in Scala.

Variables

Two ways to introduce variables

- **Immutable** (a.k.a. The “right” way)
- **Mutable** (a.k.a. The “if-you-must” way)

Immutable Variables

- With the `val` keyword

```
scala> val x = 2
x: Int = 2
```

- `val x = e` is like Ocaml’s `let x = e`
- But don’t try to change (re-assign) such a variable!

```
scala> x = x + 1
<console>:8: error: reassignment to val
      x = x + 1
      ^
```

- *Snap.*

... and if you absolutely must: Immutable Variables

```
scala> var z = 1
z: Int = 1

scala> z += 1

scala> z
res1: Int = 2

scala> z = 3
z: Int = 3
```

We will revisit `val` and `var` when we talk about objects and fields.

Compound Data: Tuples

Scala comes equipped with modern amenities like *tuples*

```
scala> val t = ("cat", 12)
t: (java.lang.String, Int) = (cat,12)

scala> val t2 = ("mickey", 12, "mouse")
t2: (java.lang.String, Int, java.lang.String)
    = (mickey,12,mouse)

scala> val t3 = ("mickey", (12, "mouse"))
t3: (java.lang.String, (Int, java.lang.String))
    = (mickey,(12,mouse))
```

the type is written `(T1, T2)` or `(T1, T2, T3)`

Compound Data: Tuples

To access the tuple elements, you can use pattern matching

```
scala> t
res7: (java.lang.String, Int) = (cat,12)

scala> val (p, q) = t
p: java.lang.String = cat
q: Int = 12
```

or the tuple *field* accessors `._1` or `._2` etc.

```
scala> t._1
res5: java.lang.String = cat

scala> t._2
res6: Int = 12
```

Compound Expressions

You can **sequence** expressions to get bigger expressions

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

{ println("hello world")
  2 + 8 }

// Exiting paste mode, now interpreting.

hello world
res0: Int = 10
```

Paste-mode lets you enter multi-line expressions in REPL

Functions

Many ways to define functions

Functions

Many ways to define functions

```
def inc(x: Int): Int = {
  x + 1
}
```

Call a function in the usual way

```
scala> inc(4)
res3: Int = 5
```

Functions

No need for explicit `return`

```
def adder(x: Int, y: Int): Int = {  
  println("adder(x=%d, y=%d)".format(x, y))  
  x + y  
}
```

Block evaluates to last expression

```
scala> adder(2, 3)  
adder(x=2, y=3)  
res1: Int = 5
```

Curried Functions

```
def curriedAdder(x: Int)(y: Int): Int = {  
  println("You called curried adder: %d %d".format(x, y))  
  x + y  
}
```

which is called thus

```
scala> curriedAdder(2)(3)  
You called curried adder: 2 3  
res0: Int = 5
```

Functions with no name

A.k.a *anonymous* functions

Like Ocaml's `fun x -> e`

```
scala> (x: Int) => x + 1  
res3: (Int) => Int = <function1>  
  
scala> (x: Int, y: Int) => x + y  
res2: (Int, Int) => Int = <function2>
```

Note the types of the functions

Functions with no name

Call by placing argument in front

```
scala> ((x: Int) => x + 1)(8)
res3: Int = 9
```

Makes sense to bind to a name first

```
scala> val inc = (x: Int) => x + 1
inc: (Int) => Int = <function1>
```

```
scala> inc(5)
res5: Int = 6
```

Functions are Objects Too!

Functions are Objects Too!

Functions are just objects with `apply` method

```
scala> object inc { def apply(x: Int) = x + 1 }
```

Call them like so

```
scala> inc(5)
res: Int = 6
```

Functions are Objects Too!

Functions are just objects with `apply` method

When you **write** `f(args)` Scala **reads** `f.apply(args)`

- Enables many fun features
- “Uniform Access Principle” ...

Functions are Objects Too!

Anything with `apply` can be *called*

```
scala> val str = "megalomaniac"
str: String = megalomaniac
```

```
scala> str(2)
res: Char = g
```

So far: Basic Data, Variables, Functions

Next: Control Expressions

Control Expressions

The usual suspects ...

- if-else
- try-catch
- while-loops
- for-loops

Control Expressions: If-Else

Put the `then` and `else` expressions inside `{...}`

```
def fac(n: Int) = {
  if (n > 0) {
    n * fac(n-1)
  } else {
    1
  }
}
```

Control Expressions: Pattern Matching (FTW!)

Scala has pattern matching too ...

```
def fac(n: Int): Int =  
  n match {  
    case n if (n > 0) => n * fac(n-1)  
    case _            => 1  
  }
```

... more on this later.

Digression: About Those Types...

(Unlike ML), you have to write down *some* types in Scala.

Function arguments and returns

```
scala> def fac(n: Int) =  
      |   n match {  
      |     case n if (n > 0) => n * fac(n-1)  
      |     case _            => 1  
      |   }  
<console>:10: error: recursive method fac needs result type  
      case n if (n > 0) => n * fac(n-1)
```

Scala's **Local Type Inference** figures out the rest!

Control Expressions: Try-Catch

Put the body expression inside {...}

```
import java.io.File  
  
def fileContents(file: String): List[String] = {  
  val f = new java.io.File(file)  
  try {  
    scala.io.Source.fromFile(f).getLines().toList  
  } catch {  
  
    case e: java.io.FileNotFoundException => {  
      println("WTF! No such file exists")  
      List()  
    }  
  }  
}
```

```
    }  
  }  
}
```

Control Expressions: Try-Catch (contd.)

You can run it thus:

```
scala> val lines = fileContents("lec1-scala-intro.markdown")  
lines: List[String] = List(% CSE 130: Spring 2012, ...)
```

and if you call it with a junk file:

```
scala> val lines = fileContents("foobar")  
WTF! No such file exists  
lines: List[String] = List()
```

Control Expressions: While-loop

Old-school while loops ...

```
def fac(n: Int) = {  
  var res = 1  
  var count = n  
  while (count > 0) {  
    res *= count  
    count -= 1  
  }  
  res  
}
```

Control Expressions: For-loop

... and of course, for-loops too...

```
scala> for (i <- 1 to 5) println(i)  
1  
2  
3  
4  
5
```

Control Expressions: For-loop

Or if you want a step...

```
scala> for (i <- 1 to 5 by 2) println(i)
1
3
5
```

Control Expressions: For-loop

Or count in the other direction ...

```
scala> for (i <- 5 to 1 by -1) println(i)
5
4
3
2
1
```

Control Expressions: For-loop

Actually, for-loops are *quite* special ...

```
scala> for (i <- "special") println(i)
s
p
e
c
i
a
l
```

Hmm. Whats going on?

Scala Batteries Included: Collections

Batteries Included: Collections

Scala bundled with with a rich set of Collections

- Arrays
- Lists
- (Hash) Sets
- (Hash) Maps
- Streams ...

These are *immutable* OR *mutable*

Batteries Included: Arrays

(Mutable) Arrays

```
scala> val a = Array("cow", "dog", "mouse")
res14: Array[java.lang.String] = Array(cow, dog, mouse)
```

Note the type: `Array[String]`.

Arrays have a fixed length

```
scala> a.length
res15: Int = 3
```

Batteries Included: Arrays

(Mutable) Arrays can be randomly **accessed**

```
scala> a(2)
res16: java.lang.String = mouse
```

- “Uniform access principle”
- `a(2)` *read as* lookup method call `a.apply(2)`

Batteries Included: Arrays

(Mutable) Arrays can be randomly **updated**

```
scala> a(0) = "capuchin"

scala> a
res19: Array[java.lang.String] = Array(capuchin, dog, mouse)
```

- “Uniform access principle”
- `a(0) = "capuchin"` *read as* update method call `a.update(0, "capuchin")`

Batteries Included: For-Loops

(Mutable) Arrays can also be looped over...

```
scala> for (animal <- a) println(animal)
capuchin
dog
mouse
```

Loathsome “off-by-one” errors BEGONE!

But *immutable* better if you can help it...

Batteries Included: Lists

Batteries Included: Lists

(Immutable) Lists

```
scala> val xs = List(1,2,3,4)
xs: List[Int] = List(1, 2, 3, 4)

scala> val ys = List("cat", "dog", "moose", "gorilla")
ys: List[java.lang.String] = List(cat, dog, moose, gorilla)
```

Cannot change a list!!

```
scala> ys(0) = "kite"
<console>:9: error: value update is not a member of List[java.lang.String]
      ys(0) = "kite"
      ^
```

Batteries Included

Quite a bit like ML lists...

```
scala> val zs = "chicken" :: ys
zs: List[java.lang.String] = List(chicken, cat, dog, moose, gorilla)
```

can be accessed via patterns

```
def listConcat(xs: List[String]): Int =
  xs match {
    case Nil => ""
    case h::t => h + listConcat(t)
  }
```

Which you can call like so:

```
scala> listConcat(zs)
res10: chickencatdogmoosegorilla
```

Batteries Included

(Immutable) Lists

You can also append them

```
scala> List(1,2,3) ++ List(4,5,6)
res11: List[Int] = List(1,2,3,4,5,6)
```

and, loop over them too ...

```
scala> for (animal <- zs) println(animal)
chicken
cat
dog
moose
gorilla
```

Batteries Included: Hash-Maps

Batteries Included: Hash-Maps

Key-Value Maps, that can be *immutable* (by default)

```
scala> val numNames = Map("one" -> 1, "two" -> 2, "three" -> 3)
numNames: scala.collection.immutable.Map[java.lang.String,Int]
       = Map(one -> 1, two -> 2, three -> 3)
```

Batteries Included: Hash-Map Lookup

Key-Value Maps, that can be *immutable* (by default)

You can *lookup* the value of a key much like arrays

```
scala> numNames("three")
res12: Int = 3
```

Batteries Included: Hash-Maps Lookup

Key-Value Maps, that can be *immutable* (by default)

If the value doesn't exist though... **Exception**

```
scala> numNames("nine")
java.util.NoSuchElementException: key not found: nine
.
.
.
```

Batteries Included: Hash-Maps Membership

Key-Value Maps, that can be *immutable* (by default)

Moral, *look before you leap*

```
scala> numNames.contains("nine")
res13: Boolean = false
```

or using the cleaner notation

```
scala> numNames contains "nine"
res14: Boolean = false
```

Batteries Included: Hash-Maps Adding Keys

Key-Value Maps, that can be *immutable* (by default)

Would be nice to extend a map with new key-value bindings.

How do you think its done?

Batteries Included: Hash-Maps Adding Keys

Key-Value Maps, that can be *immutable* (by default)

Would be nice to extend a map with new key-value bindings...

```
scala> numNames + ("nine" -> 9)
res15: scala.collection.immutable.Map[java.lang.String,Int]
      = Map(one -> 1, two -> 2, three -> 3, nine -> 9)
```

Note the above is a **brand new map** ...

```
scala> numNames contains "nine"
res16: Boolean = false
```

Batteries Included: Hash-Maps Adding Keys

Key-Value Maps, that can be *immutable* (by default)

Would be nice to extend a map with new key-value bindings ...

```
scala> val newMap = numNames + ("nine" -> 9)
newMap: scala.collection.immutable.Map[java.lang.String,Int]
      = Map(one -> 1, two -> 2, three -> 3, nine -> 9)
```

... Bind the result to a new map.

```
scala> newMap("nine")
res17: Int = 9
```

Batteries Included: Mutable Hash-Maps (!)

Batteries Included: Mutable Hash-Maps

(psst.) There are *mutable* Key-Value Maps too...

```
scala> import scala.collection.mutable.HashMap

scala> val mmap : HashMap[String, Int] = HashMap()
scala> mmap += "mon" -> 1
scala> mmap += "tue" -> 2
scala> mmap += "wed" -> 3

scala> mmap("tue")
res18: Int = 2
```

Note: type parameters for the key (String) and value (Int)

Recap: Expressions, Variables, Functions, Collections

QUIZ: What is the value of res?

```
import scala.collection.mutable.HashMap
val mutMap: HashMap[String, Int] = HashMap()
mutMap += "mon" -> 1
mutMap += "mon" -> 2
val res = mutMap("mon")
```

- A. No value, **Type Error** (cannot update val)
- B. No value, **Runtime Exception** (key not found)
- C. 1: Int
- D. 2: Int
- E. None: Option[Int]

QUIZ: What is the value of res?

```
import scala.collection.mutable.HashMap
var immutMap : Map[String, Int] = Map()
immutMap += "mon" -> 1
immutMap += "mon" -> 2
val res = immutMap("mon")
```

- A. No value, **Type Error**
- B. No value, **Runtime Exception** (NotFound)

- C. 1: Int
- D. 2: Int
- E. None: Option[Int]

QUIZ: What is the value of res?

```
import scala.collection.immutable.Map
var immutMap : Map[String, Int] = Map()
var other = immutMap
immutMap += "mon" -> 1
immutMap += "mon" -> 2
val res = other("mon")
```

- A. No value, **Type Error**
- B. No value, **Runtime Exception (NotFound)**
- C. 1: Int
- D. 2: Int
- E. None: Option[Int]

QUIZ: What is the value of res?

```
~~~{scala} import scala.collection.mutable.HashMap var immutMap :
Map[String, Int] = Map() var other = immutMap immutMap += "mon" -> 1
immutMap += "mon" -> 2 val res = other("mon")
```

- A. No value, **Type Error**
- B. No value, **Runtime Exception (NotFound)**
- C. 1: Int
- D. 2: Int
- E. None: Option[Int]

How to operate on collections?

How to operate on collections? HOFs!

Collections and Higher-Order Functions

All collection objects equipped with HOFs

- filter
- map
- foreach
- foldLeft, foldRight
- and many others!

HOFS + Collections: filter

All collection objects equipped with HOFs!

```
scala> "MaSsIvEaTtAcK".filter(c => c.isUpper)
res19: String = MSIETAK
```

Or, with equivalent, simpler syntax for HOFs

```
scala> "MaSsIvEaTtAcK".filter(_.isUpper)
res20: String = MSIETAK
```

HOFS + Collections: filter

All collection objects equipped with HOFs!

```
scala> List(1,2,3,4,5,6,7,8).filter(_ % 2 == 0)
res21: List[Int] = List(2,4,6,8)
```

```
scala> Array(1,2,3,4,5,6,7,8).filter(_ % 2 == 0)
res21: Array[Int] = Array(2,4,6,8)
```

HOFS + Collections: filter

With Map, the filter is over a **tuple**

```
scala> numNames.filter(_._1.length == 3)
res24: scala.collection.immutable.Map[java.lang.String,Int]
      = Map(one -> 1, two -> 2)
```

HOFS + Collections: filter

With Map, the filter is over a **tuple**

You can use anonymous functions over tuples

```
scala> numNames.filter({case (k, v) => k.length == 3})
res23: scala.collection.immutable.Map[java.lang.String,Int]
      = Map(one -> 1, two -> 2)
```

HOFS + Collections: map

All collection objects equipped with HOFs!

```
scala> "MaSsIvEaTtAcK".map(_.toUpperCase)
res24: String = MASSIVEATTACK
```

```
scala> List(1,2,3).map(_ * 100)
res25: List[Int] = List(100, 200, 300)
```

```
scala> Array("cat", "dog", "wabbit").map(_.map(_.toUpperCase))
res26: Array[String] = Array(CAT, DOG, WABBIT)
```

HOFS + Collections: foreach

foreach is like map but does an action, returns ()

```
scala> Array("cat", "dog", "wabbit").foreach(println(_))
cat
dog
wabbit
```

... so whats a for-loop?

HOFS + Collections: for-loops Revisited

for-loop is just a HOF!

```
for (a <- thing) { body }
```

is just same as

```
thing.foreach(a => body)
```

HOFS + Collections: for-loops Revisited

So you can loop over *all* collections

```
scala> numNames.foreach({case (k, v) => println(k + " ~~~~> " + v)})
one ~~~~> 1
two ~~~~> 2
three ~~~~> 3
```

or if you prefer

```
scala> for ((k, v) <- numNames) { println(k + " ~~~~> " + v) }
one ~~~~> 1
two ~~~~> 2
three ~~~~> 3
```

HOFS + Collections: for-loops Revisited

Another example:

```
scala> val dir = new java.io.File(".")
dir: java.io.File = .

scala> for (f <- dir.listFiles) println(f)
./slides.markdown
./scratch.markdown
./Monoid.scala
./Freq.scala
.
.
.
```

HOFS + Collections: for-loops Revisited

Sometimes, want to loop over *some* elements and skip others.

For example, to print names of all `.scala` files:

```
scala> for (f <- dir.listFiles
           if f.getName.endsWith(".scala")) { println(f) }
./Monoid.scala
./Freq.scala
```

QUIZ: HOFS + Collections: for-loops Revisited

Sometimes, want to loop over *some* elements and skip others.

```
for (x <- coll if cond) { body }
```

Is really a nice way of writing

A. `coll.filter(x => body)`

B. `coll.filter(x => cond).foreach(x => body)`

C. `coll.foreach(x => body)`

D. `coll.foreach(x => body).filter(x => body)`

HOFS + Collections: for-loops Revisited

More often, want to **compute** a collection as a value ...

Suppose we have an Array of Strings ...

```
val words = Array("you", "are", "doing", "it", "wrong")
```

... to turn it into a **rant**

```
scala> val rant = for (w <- words) yield w.toUpperCase  
rant: Array[java.lang.String] = List(YOU, ARE, DOING, IT, WRONG)
```

... Note the output is **also** an Array.

HOFS + Collections: for-loops Revisited

More often, want to **compute** a collection as a value ...

Works for any collection...

```
scala> for (w <- fileContents("lec-scalacrash.markdown")  
      yield w.toUpperCase
```

```
res: List[String] = List(% CSE 130: SPRING 2012, ...)
```

QUIZ: HOFS + Collections: for-loops Revisited

More often, want to **compute** a collection as a value ...

```
for (x <- coll) yield expr
```

Is really a nice way of writing

A. `coll.foreach(x => expr)`

B. `coll.filter(x => expr)`

C. `coll.map(x => body)`

HOFS + Collections: for-loops Revisited

More often, want to **compute** a collection as a value ...

... after some processing.

E.g., to **find** all `.scala` files in a directory

```
scala> val scalaFiles = for (f <- dir.listFiles
                          if f.getName.endsWith(".scala"))
                          yield f
```

```
scalaFiles: Array[java.io.File] = Array(/Monoid.scala, /Freq.scala)
```

HOFS + Collections: for-loops Revisited

Sometimes, want to *return* a collection...

... after some processing.

```
scala> import scala.io.Source._
scala> val fileSize = for (f <- dir.listFiles
                          if f.getName.endsWith(".scala"))
                          yield fromFile(f).length
```

```
fileSize: Array[Int] = Array(349, 406)
```

HOFS + Collections: for-loops Revisited

Sometimes, want to *return* a collection...

... after some processing.

```
for (x <- coll if cond) yield expr
```

Is really another way of writing

```
coll.filter(x => cond).map(x => expr)
```

HOFS + Collections: Nested for-loops

You can nest for-loops too:

```
scala> for (i <- 1 to 3; j <- 1 to 3) {  
    println ("%d and %d" format(i, j))  
  }  
1 and 1  
1 and 2  
1 and 3  
2 and 1  
2 and 2  
2 and 3  
3 and 1  
3 and 2  
3 and 3
```

HOFS + Collections: Nested for-loops

You can nest for-loops too,

and then return a collection as the result:

```
scala> for (i <- 1 to 3; j <- 1 to 3) yield (i, j)  
res: scala.collection.immutable.IndexedSeq[(Int, Int)]  
    = Vector((1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3))
```

HOFS + Collections: Nested for-loops

You can nest `for`-loops too, what does this return

and then return a collection as the result

after filtering

```
scala> val res = for (i <- 1 to 3; j <- 1 to 3 if i < j)
                    yield (i, j)
res26: scala.collection.immutable.IndexedSeq[(Int, Int)]
      = Vector((1,2), (1,3), (2,3))
```

HOFS + Collections: Nested for-loops

You can nest `for`-loops too, what does this return

and then return a collection as the result

after filtering

Again: this holds for **any** collection:

```
scala> val res = for (i <- List(1,2,3);
                    j <- List(4,5,6)
                    if i + j == 7)
                    yield (i, j)
res: List[(Int, Int)] = List((1,6), (2,5), (3,4))
```

HOFS + Collections: Nested for-loops

You can nest `for`-loops too, what does this return

and then return a collection as the result

```
val res = for (i <- List(1,2,3);
              j <- List(4,5,6)
              if i + j == 7)
              yield (i, j)
```

So far, inner loop independent of outer but...

QUIZ: HOFS + Collections: Nested for-loops

Of course, you can nest `for`-loops too:

```
scala> val ll = List(List(1,2), List(3,4))
scala> val res = for (xs <- ll;
                     x  <- xs)
                     yield x
```

What is the value of `res`?

- A. `List(1, 3)`
- B. `List(List(1,2), List(3,4))`
- C. `List((1,3), (1,4), (2,3), (2,4))`
- D. `List(1,2,3,4)`
- E. `List()`

HOFS + Collections: Nested for-loops

Of course, you can nest `for`-loops too:

another example (wink wink) ...

```
scala> for (w <- List("cat", "dog", "mouse"); c <- w) yield c
res: List[Char] = List(c, a, t, d, o, g, m, o, u, s, e)
```

... Note the output type is also `List`

Like the top-level sequence!

HOFS + Collections: Nested for-loops

Of course, you can nest `for`-loops too:

yet another example (wink wink) ...

```
scala> for (w <- Array("cat", "dog", "mouse"); c <- w) yield c
res: Array[Char] = Array(c, a, t, d, o, g, m, o, u, s, e)
```

... Note the output type is also `Array`

QUIZ: HOFS + Collections: Nested for-loops

Of course, you can nest for-loops too:

```
val res = for (xs <- Iterator("cat", "dog", "mouse")
               x  <- xs)
  yield x
```

What is the **type** of `res`?

- A. String
- B. List(String)
- C. Iterator(String)
- D. List(Char)
- E. Iterator(Char)

HOFS + Collections: for-loops Revisited

Wait a minute! Remember this?

```
scala> for (i <- 1 to 5) println(i)
1
2
3
4
5
```

How does it work ?

HOFS + Collections: for-loops Revisited

Wait a minute! Remember this?

`1 to 5` is a **method call** `1.to(5)`

```
scala> 1 to 5
res: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
```

HOFS + Collections: for-loops Revisited

If you want a **step**

```
scala> 1 to 10 by 2 // 1.to(10).by(2)
res3: scala.collection.immutable.Range = Range(1, 3, 5, 7, 9)

scala> 10 to 1 by -1 // 10.to(1).by(-1)
res4: scala.collection.immutable.Range = Range(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)

scala> 10 to 1 by -2 // 10.to(1).by(-1)
res5: scala.collection.immutable.Range = Range(10, 8, 6, 4, 2)
```

You can do a LOT with collections!

So what does this do?

```
List(1,2,3,4,5).foldLeft(0)((x1, x2) => x1 + x2)
```

You can do a LOT with collections!

So what does this do?

```
List(1,2,3,4,5).foldLeft(0)(_ + _)
```

Here, `(_ + _)` is a short hand for

```
(x1, x2) => arg1 + arg2
```

Btw, you can do a LOT with collections!

And, what does this do?

```
def foo(n: Int, k: Int) =
  (1 to k).map(_ => n).foldLeft(1)(_ * _)
```

Put things together: Frequencies

Lets write a Frequency Finder

How often does a Char appear in a String

```
def freq(str: String): HashMap[Char, Int] = {  
  val freqMap = new HashMap[Char, Int]  
  for (c <- str) {  
    freqMap(c) = 1 + freqMap.getOrElse(c, 0)  
    // if(freqMap contains c){freqMap(c)}  
    // else { 0 }  
  }  
  freqMap  
}
```

Lets write a Polymorphic Frequency Finder

Lets generalize a bit. (Make it polymorphic)

```
def freq[A](xs: Iterable[A]): HashMap[A, Int] = {  
  val freqMap = new HashMap[A, Int]  
  for (x <- xs) {  
    freqMap(x) = 1 + freqMap.getOrElse(x, 0)  
  }  
  freqMap  
}
```

Iterable[A] describes objects that can be iterated over...

Lets write a Polymorphic Frequency Finder

Can run it on Strings

```
scala> freq("caterpillar")  
res: scala.collection.mutable.HashMap[Char,Int]  
    = Map(c -> 1, a -> 2, e -> 1, i -> 1, r -> 2, t -> 1, l -> 2, p -> 1)
```

or List

```
scala> freq(List(1,2,1,13,1,2,1,3,31,12,1))  
res: scala.collection.mutable.HashMap[Int,Int]  
    = Map(12 -> 1, 3 -> 1, 13 -> 1, 1 -> 5, 2 -> 2, 31 -> 1)
```

or ...

Compiling and Running

Compiling and Running

To make an executable, put the functions in a file e.g. `foo.scala`

```
object Freq { ... }
```

compile with

```
$ scalac foo.scala
```

and run with

```
$ scala FreqShow "lec1-scala-intro.markdown"
...
```

Compiling and Running

Or now run in the REPL

```
scala> val m = Freq("caterpillar")
scala> Freq.show(m, 1)
c : #
a : ##
e : #
i : #
r : ##
t : #
l : ##
p : #
```

Next Time: Types, Classes, Objects, Generics.