Data-Intensive Distributed Computing CS431/451/631/651

Module 3 Interlude - Scala

What is Scala?

Scala is a language built on top of the JVM

Key Features:

- Both Functional and Object Oriented
- Every value is an object, every function is a value
  - Even methods are values!

Hello Scala
Or, “How to print stuff”

```
// Comment
println(“Hello, World”) // print a String
println(2 + 2)          // print an Int
println(1,2,3)          // 3x Int print Combo!

printf(“Welcome to Scala, CS%d”, 451)
```

(Bonus Slide) – Strings

Strings use the same format as C, C++, etc.

```
“This is a string with a newline\n”
```

There are also format strings

```
f“2 + 2 is ${2 + 2}”
```

becomes

```
“2 + 2 is 4”
```

Rosetta Stone

Jimmy Lin has Scala versions of some of the MapReduce examples in Bespin.

Because Scala uses the JVM, you can write MapReduce code in Scala.

Variables

```
var vi : Int = 3 // mutable integer
val ci : Int = 5 // constant integer

vi = 4 // valid
ci = 4 // invalid, cannot reassign to val
```

Interactive

You can compile Scala to java class files but can also run it in an interactive shell.

<For your entertainment, Dan will now demonstrate the shell. It'll look like this>

scala> 2 + 2

res0: Int = 4

Assignment

All your favorite operators are here!

```
var x = 3 // type inference!  x is an Int
x = 4     // assignment!
x += 2    // assignment!
x = “foo” // invalid! x is Int, not String
```

Collections

```
var myArray = Array(1, 2.0, 3)
```

Q: What type is myArray?

A: Array[Double]

Q: Square Brackets?

A: Yes. C++/Java would say Array<Double>, Scala says Array[Double]

Any

```
var myArray = Array[Any](1, 2.0, 3)
```

OR

```
var myArray : Array[Any] = Array(1, 2.0, 3)
```

The any type can hold any value. Now the array is a mix of Int and Double values.

Array Operators

```
scala> Array(1,2) ++ Array(3)
Array[Int] = Array(1,2,3)
scala> Array(1,2) :+ 3
Array[Int] = Array(1,2,3)
```

++ makes a new array by concatenating two arrays

:+ makes a new array by appending a single value

Array Operators

++= and :+= exist

They do NOT modify the Array!

```
> val a = Array(1,2)
> a :+= 3
Error: :+= is not a member of Array[Int].
     Cannot convert to assignment as a is not
     assignable
```

Maps

Two forms of Map

Default is immutable.

```
> Map(“a”->3, “b”->4)
Map[String,Int] = …
> scala.collection.mutable.Map(“a”->3, “b”->4)
scala.collection.mutable.Map[String,Int] = …
```

Maps (bonus slide)

(“a” -> 3) is just alternate syntax for (“a”, 3)

(“a” -> 3, “b” -> 4) is equivalent to ((“a”, 3), (“b”, 4))

It’s usually used for Maps to make key-value pairs more expressive. Also to avoid parentheses.

A Tale of Two Maps

Immutable

```
> var m = Map(“a”->3)
> m += (“b” -> 4)
```

map + pair makes a new map

+= is assigning this new map to m

Mutable

```
> val m =
  mutable.Map(“a” -> 3)
> m += (“b” -> 4)
```

mutable.Map has its own += operator that mutates the existing value.

NOT trying to reassign m!

Isn’t an immutable hash map inefficient to update?

Who said a HashMap means Hash Table?!

An Immutable Hash Map is a Trie that uses the hash code as the “String”.

Immutable trees that don’t involve rotations! Producing a new tree based on the old one is O(h). The tree has a fixed height.

Loops

```
for (i <- 1 to 10) println(i) // stops after 10
for (i <- 0 until 10) println(i) // stops before 10
```

You can add conditionals

```
for (i <- 0 until 10 if i % 2 == 1) println(i)
```

Loop to Vector

```
> for (i <- 1 to 10) yield i + 1
… = Vector(1,2,3,4,5,6,7,8,9,10)
```

That’s right, for loops have a value! Or they can, if you yield values

For loops and collections

```
val m = Map("a"->1, "b"->2)
val a =  Array(1,2,3,4)

for (i <- a) println(i)
for ((k, v) <- m) printf("%s -> %d\n", k, v)
```

<- m is iterating over key-value pairs
By putting a tuple here, k and v get bound to the key and value (respectively)

Functions

```
def f(p1 : t1, p2 : t2, …) : [return type] = expr
```

Return type is optional (**if it can be inferred**).

If you use return statements it cannot infer the type. Don't worry about WHY

Parameters' types are **mandatory**

Return is optional (will return value of final expression)

The expr is usually a {block} like in C++, but doesn't need to be

Function Calls

Exactly like C

```
F(x,y)
```

If a function has no parameters you don't need ()

`F()` is the same as F

(This can make it a bit hard to tell if something is a field or a method with no parameters)

Functions

```
def add(x : Int, y : Int) = x + y
```

Returns x + y. Infers return type is int. No need for {braces}

If this makes you uncomfortable, use braces

Anonymous Functions

```
(x : Int, y : Int, …) => x + y
(x : Int) => x * x
```

Parameters types are **mandatory** (unless passing directly to a higher order function)

```
Array(1,2,3).map((x) => x * x)
```

The map method is expecting a function with an int parameter, so it will infer that x is intended to be int

More Anonymous Functions

If you only use a parameter once, you can use an underscore expression.

- This ONLY works as a parameter to a higher order function

```
List(1,2,3).map(_ + 1)
res0: List[Int] = List[Int](2,3,4)
```

An expression with n underscores is interpreted as an anonymous function with n parameter. 1st parameter = 1st underscore, etc

Unit?

Unit is the Scala name for Void. A special type that indicates the lack of a value.

C++ functions that were “void” are “Unit” functions in Scala.

If you see “Expected X but got Unit” you probably forgot to return something, or your function ended with a valueless expression

match

Match is the coolest. It’s one of my fav things in Racket and it’s in Scala too!

```
def intToTroll(x : Int) : String = x match {
  case 0 => “None”
  case 1 => “One”
  case 2 => “Two”
  case 3 => “Many”
  case _ => “Lots”
}
```

Match

So it's switch? You have a VERY low bar for cool, sir.

Au Contraire, it's so much more! The thing after case can be a pattern!

```
def f(x : Any) : String = x match {
  case List(y) => f(y)
  case y : String => y
  case _ => “?”
}
```

Option, Some

Option[A] is a collection that can hold up to 1 A type value.

It's either None, or a Some[A]. To get the A value from a Some, use get

```
def maybeAdd1(x : Option[Int]): Option[Int] = x match {
  case Some(y) => Some(y + 1)
  case None => None
}
```

Maps

Didn't you mention these already?

Yeah, it's a brief pitstop now we've seen Option.

Map[K,V].get(k) returns Option[V]

Map[K,V](k) returns a V or throws a key not found exception.

Interoperability with Java

Scala can interact with Java objects.

Scala constructs don’t work with Java collections, but you can convert.

Step 1:

```
import scala.collection.JavaConverters._
```

Step 2:

for (i <- javaArrayBuffer.asScala) { … }

The End…or The Beginning?

I didn’t mention how Classes, Traits, Mixins, sealing, and a bunch of other things work. You (mostly) don’t need them for the assignments.

I’ll point out new features as we come across them in Spark.