

Next Steps with Scala

More Scala Basics, and Using Worksheets

Agenda

1. IntelliJ and Scala Worksheets
2. Method Parameters and Return Types
3. Expressions vs Statements
4. Tuples
5. Re-writing Rules
6. Collections
7. Extension Methods (Intro Only)
8. Functional vs Imperative Style

Scala Projects in IntelliJ

- Group Activity, load a project into IntelliJ IDEA
- Unzip the `exercises.zip` file
- Open IntelliJ IDEA
- Import Project
- Find `build.sbt` in the exercises, highlight and click OK
- Accept all of the defaults, wait...

Creating (or Opening) a Worksheet

- Open the Projects tab, then scripts (scripts is just a convention I use, it can be anywhere)
- Double-click on a worksheet (or right click on scripts to create a new one)
- Anything you type will be evaluated, like the REPL (but with full IDE features)
- Examples in this course are included as worksheets under scripts

Method Parameters and Return Types

- Let's take a look at a method definition:

```
def max(x: Int, y: Int): Int = if (x > y) x else y
```

- This is *fully typed*, in that the parameter, and return, types are specified. We can drop the return type, since Scala can infer it

```
def min(x: Int, y: Int) = if (x < y) x else y
```

- As you will see in the worksheet, the method has the same exact type, the *Int return type is inferred by the compiler*
- However *you cannot leave the type annotations off of the parameters*, since Scala has no context to infer those from

Methods with No Return Types

- Java (and some other languages) have a `void` keyword, which denotes "no return type"
- In Scala, **every** method and variable has a type, there is no `void`
- The rough equivalent is `Unit`, of which there is only one instance: `()`

```
def sayHi(name: String): Unit = println(s"hello $name")
```

- Methods resulting in `Unit` must have side effects in order to be useful (IO is one such side effect)
- Scala still has *procedural syntax*, which has the same effect **but is deprecated**

```
def sayHello(name: String) {  
  println(s"hello $name")  
}
```

- You will get a warning, and IntelliJ will try to correct you, **always use `:Unit`** = instead of procedural syntax

Expressions vs Statements

- An expression returns its payload as a return argument with a type, e.g.:

```
val min = if (x < y) x else y
```

- A statement returns Unit and has to have some side effect to be useful:

```
if (x > y) println(s"max is $x") else println(s"max is $y")
```

- Functional programming style prefers expressions over statements
- Remember that if, try...catch, for, and other common constructs in Scala are *expressions*
- while and do...while are the only built in control flow constructs that only return Unit:

```
var doIt: Boolean = true  
val result = while (doIt) {  
  println("Hello")  
  doIt = false  
}
```

Statements and Expressions

- `val` and `var` also produce `Unit` return types, this is surprising at first:

```
var x = 5
val y = x = 10
println(x) // 10
println(y) // ()
```

- A common mistake when first learning Scala is ending a code block with a `val`:

```
def add(a: Int, b: Int) = {
  val result = a + b
}

val sum = add(5, 6) // sum will be (): Unit!
```

- Can be avoided by adding the expected return type `Int` which is considered good practice

Tuples

- So far we have looked at simple types like `Int`, `String`, and `Unit`, also our methods have returned just one of these
- What if we want to return more than one thing from a method? Enter *tuples*

```
def sumAndDifference(a: Int, b: Int): (Int, Int) = {  
  val sum = a + b  
  val difference = a - b  
  (sum, difference)  
}
```

- Getting the result parts:

```
val results = sumAndDifference(10, 5)  
  
results._1    // 15: Int  
results._2    // 5: Int
```

- The types are carried through, `_1` and `_2` can be thought of as item 1 and item 2

Tuples

- There's a nicer way to get the parts:

```
val (sm, df) = sumAndDifference(10, 5)
```

- And the tuple can have more than 2 items, and mixed types:

```
val (a,b,c,d,e) = (0, 'u', 8, 1, "too")  
a    // 0: Int  
b    // 'u': Char  
c    // 8: Int  
d    // 1: Int  
e    // "too": String
```

- Tuples can have arity up to 22, because it had to stop somewhere
- Future versions of Scala may (probably will) create tuple arities on the fly

Re-writing Rules, infix

- Scala has no operators (as such), although it appears to:

```
val x = 1 + 2
```

- So what's + if it's not an operator? A method! The above can be re-written:

```
val y = 1.+(2)
```

- This is known as **infix notation**, it works for all methods on an instance with one parameter, e.g.

```
val s = "hello"
```

```
s.charAt(1)
s charAt 1 // same result as above
```

- It does not work without an instance before the method though:

```
println "hello" // will not compile, needs parens
```

Re-writing Rules, apply

- Let's create an array:

```
val arr = Array("scooby", "dooby", "doo")
```

- Getting items out of an array can be achieved with the `apply` method:

```
println(arr.apply(1)) // prints "dooby"
```

- Scala has a shortcut for `apply`, any item (other than a method) followed by parens calls `apply` with the contents of the parens (if any):

```
println(arr(0)) // prints "scooby", same as arr.apply(0) would
```

- In fact, the Array creation line above also uses this rule:

```
Array("scooby", "dooby", "doo")  
// is re-written to  
Array.apply("scooby", "dooby", "doo")
```

which calls the `apply` method on the *companion* object using *varargs* (we will learn about both of these soon)

Re-writing Rules, update

- What if we update the value in an Array (arrays are mutable so they can be updated):

```
arr(0) = "scrappy"
```

- This is re-written to a call to update with the value in parens as the first argument, and the value after the equals as the second, so:

```
arr(1) = "dappy"  
// is re-written to  
arr.update(1, "dappy")
```

- The result of update is defined as Unit so in order to do anything useful, it must have a side effect

Re-writing Rules General Notes

- We will see other Scala re-writing rules as we go through material
- Re-writing is **only** done if the code won't typecheck without a re-write
- If an item doesn't have an apply or update method, the re-write will be attempted but will fail to compile:

```
val z = 10  
z(2) // "Application does not take parameters" compile error
```

```
val xs = List(1,2,3) // could be written List.apply(1,2,3)  
xs(1) // works, gives back 2: Int  
xs(1) = 10 // compile error, since no update method on immutable List
```

Quick Collections Intro

- So far we have seen `Array` which is mutable, and just now `List` which is immutable
- Both collection types have a **type parameter** specifying what they hold:

```
val array1: Array[Int] = Array(1,2,3)
val list1: List[String] = List("scooby", "dooby", "doo")
```

- The **type parameter** is not optional, but can be inferred from the initialization contents

```
val array2 = Array(1,2,3) // Array[Int] is inferred
val list2 = List("scooby", "dooby", "doo") // List[String] is inferred
```

- When **specifying a collection type in a method parameter** (or return parameter), the **type parameter must be provided!**

```
def squareRootsOf(xs: List[Int]): List[Double] =
  for (x <- xs) yield math.sqrt(x)
```

List Initialization

- As seen you can initialize a list using the `List.apply` method (or `List(contents...)` using re-writing)

```
val lista = List(1,2,3)
```

- For lists only, you can also use the **cons form of initialization**, using `::`

```
val listb = 4 :: 5 :: 6 :: Nil
```

- **`::` is right associative**, that is, it applies the parameter on the *left* side to the item on the right, e.g.

```
val listb = ((Nil::(6))::(5))::(4)
```

- Any **operator ending in `:`** is right associative in Scala
- Another list-only operator is **concatenate, `:::`** which joins two lists (again right associative):

```
val listc = lista ::: listb
```


Sequences

- List and Array are both sequences in Scala, subtypes of Seq
- There are others, notably Vector:

```
val v = Vector(1,2,3,4)
```

- All can be passed in to a method requiring a Seq of the right type:

```
def squareRootOfAll(xs: Seq[Int]): Seq[Double] =  
  xs.map(x => math.sqrt(x))
```

- Now, List[Int], Array[Int] and Vector[Int] can all be passed in:

```
squareRootOfAll(v)  
squareRootOfAll(listc)  
squareRootOfAll(array2)
```

- Don't worry about the `x => math.sqrt(x)` notation just yet, we will deal with function literals soon

Sets

- A **Seq (sequence)** is an ordered collection of homogenous values that may be repeated
- By contrast, a **Set** is an unordered collection of homogenous values that are unique

```
val set1 = Set(1,2,3,1,2,4,5) // Produces a Set(5,1,2,3,4)
```

- A **Set** cannot be passed to a function expecting a **Seq**, it is not a sub-type of **Seq**:

```
squareRootOfAll(set1) // will not compile
```

(Im)mutability of Collections

- Array is mutable, may be grown, values may be updated, etc.
- List and Vector are immutable, once created the only way to change the size or update them is to transform them into another reference (or use a var to reassign the reference)
- Set has both mutable and immutable implementations:

```
import scala.collection._  
  
val s1 = mutable.Set(1,2,3)  
var s2 = immutable.Set(1,2,3)
```

- Now if we use += on both of these:

```
s1 += 4 // works because s1 has a += operator  
s2 += 4 // works because s2 is a var
```

- For s2, Scala uses a *re-writing* rule to the expression to `s2 = s2 + 4`
- It is **not** required (nor recommended) to use a var and a mutable collection together

Maps

- A Map can be thought of as an associative sequence of `tuple2`s, the first item of the tuple can be used to look up the second item
- Like Sets, Maps have both mutable and immutable implementations

```
val m1 = mutable.Map('a' -> 1, 'b' -> 2, 'c' -> 3)
var m2 = immutable.Map('d' -> 4, 'e' -> 5, 'f' -> 6)
```

- Updating the maps

```
m1 += m2 // calls += on the mutable map
m2 += 'g' -> 7 // re-writes to m2 = m2 + 'g' -> 7
```

- What's this `'g' -> 7` syntax about?
- It's not syntax, it's an extension method

The -> extension method

- -> can be called on an instance of any type with one parameter of any other type
- The result is a tuple2[FirstType, SecondType] with the values of both instances
- It's mainly syntactic sugar for creating maps, but it's not a keyword. Here's how it works:

```
1 -> "one"  
// is re-written to  
1.->("one")  
// is expanded to  
ArrowAssoc(1).->("one")
```

- No such -> method exists on Int, but an implicit called ArrowAssoc provides it just in time
- Implicits will be covered in-depth later in the course

Simple Map Iteration

- All that effort for `->` is to make maps easy and pretty to initialize

```
val mapToRiches = Map(  
  1 -> "steal underpants",  
  2 -> "???",  
  3 -> "profit"  
)
```

- They are also easy (and pretty) to iterate over with a `for` expression

```
for ((step, instruction) <- mapToRiches) {  
  println(s"Step $step - $instruction")  
}
```

- The `(step, instruction)` unpacks the `tuple2` from the sequence in the map
- However, remember that the `order may vary in some map implementations`

Mutability vs Functional Style

- Statements, side-effects, vars, and mutability are not functional programming style
- Instead, aim for expressions, vals and immutability whenever possible
- Use vars or mutability when dictated by performance or other factors
- You don't need a var with a mutable collection, instead choose one or the other
- Don't let mutability escape into the API
- Don't optimize for performance prematurely
- Also keep methods short and uncomplicated, separate early and often

Opening and Reading a File

```
import scala.io.Source  
  
for (line <- Source.fromFile("somefile.txt").getLines()) {  
  println(line)  
}
```

- Source is not often used in production code, but it is useful for demos and learning Scala

Module 2 Exercises

- Find Module02 test class in exercises
- Run class in Scalatest
- Follow instructions in class to complete exercises and get all tests passing
- There may be a surprise or two in the exercises, ask questions...