Scala Language 01

***Intro***

*In this video and the next series of videos I’m going to discuss Scala Language features. Some of this material can be found at* [*http://go/scalaWorkshop*](http://go/scalaWorkshop)*, so if you need a refresher you can go to that wiki and practice. Or you can rewatch the videos.*

***Vars and Vals***

*The first thing I want to talk about is how to define variables in Scala. Scala makes a distinction between two types of variables, those which are mutable, and those which are immutable. Mutable variables are called vars and immutable variables are called vals. In Scala the use of vals are strongly encouraged over vars.*

*This is because Scala is a functional language and in the functional paradigm, we are changing the state of the program by evaluating functions instead of mutating variables.*

*There’s other good reasons to prefer vals over vars. For one thing, it’s easier to reason about a program when you know the value of a particular variable won’t change. Another reason to prefer vals is that programs with immutable state are inherently thread-safe.*

*Let’s see vars in actions*

*declare an variable named x of type int with value 3*

***var x:Int = 3***

*In scala, types are declared on the right hand side of the variable name, separate by a colon.*

*Now let’s increment it*

***x = x + 1***

*And we get a value of 4.*

*Now let’s try the same thing with a val instead of a var*

*This time we say*

***val x:Int = 3***

*and when we try*

***x = x + 1***

*Scala throws an exception*

*this expression also works:*

***val x = 3***

*Notice there’s no declared type. You might say, but wait, I thought you said Scala is strongly typed. Well it is. It infers that x is an integer based on the type of expression on the right hand side of the assignment. In this case the expression is an int, so x is of type Int. An indeed, if you look at the result, it infers that x is of type Int.*

***Object Basics***

*In Scala, everything’s an object, even numbers. So*

***1.getClass()***

*tells us that the symbol 1 is an instance of the Class Int*

***1 + 2***

*or we can write the equivalent*

***1.+(2)***

*Here we’re showing a couple features*

* *First, the normal method syntax we know from Java can be written as an infix operator, so 1.+paren 2 paren is the same as 1 + 2, omitting dots and parens*
* *Second, methods can have non-alphanumeric names*

*numbers have other methods like*

***1.to(3)***

*which can also be written*

***1 to 3***

***Strings***

*You can have simple strings*

***“ho"***

*Or interpolate expressions by prefixing a string with an s*

***s”the time is now ${java.util.Date}"***

*Here the expression inside the ${} is interpolated into the string*

*If you prefix a string with an f you can use printf-style formatting*

***f”There’s a \* 20 spaces from here =====>${“\*”}%20s"***

*Notice that you can even have quotes inside the ${}*

*You can write multiline strings using triple quotes*

*“”"*

*hello*

*there*

*bob*

*“”"*

***Functions***

*Let’s talk about functions and start with an example*

***def addThree(x:Int):Int = {***

***x + 3***

***}***

*We’ve see this syntax before, but let’s go over it again.*

*The def keyword means that we’re defining a named function.*

*The name of the function is the next symbol which is addThree*

*The name is followed by a parameter list in parenthesis.*

*The parameters are separated by commas and consist of name-colon-type expressions*

*To the right of the parameter list is the return type of the function, separated by a colon.*

*Then an equals sign separates the function signature with the body of the function*

*Everything to the right of the equals is the body of the function.*

*In Scala, we don’t need to supply a “return” keyword to return a value from the function- the last expression evaluated is returned from the function.*

*Also, you’ve probably already noticed by now that semicolons are optional and should usually be omitted.*

*We call a function just like we would in Java, by entering the name of the function and it’s arguments in parens`:*

***addThree(4)***

*If the function is simple, we can drop the curly braces, and some cases even write the function on one line.*

***def addThree(x:Int):Int = x + 3***

*Notice the signature returned in the REPL: (x:Int)Int*

*The parameter types are in parens and the return type is to the right.*

*Unit*

*If a function doesn’t return a value, it’s return type is Unit, which is equivalent to void in Java.*

***def sayHi():Unit = { println(“Hi!”) }***

*when we call it*

***sayHi()***

*it doesn’t return a value… there’s no “res<whatever>” in the REPL*

*A function which has Unit return type can omit the return type from the signature and the = sign between the signature and the body*

***def sayHi() { println(“Hi!”) }***

*Back to our original function, addThree*

***def addThree(x:Int):Int = x + 3***

*Let’s talk about the type of the expression that’s evaluated in the function body. We know that x is an int, and it’s being added to an int, therefore the type of the result is an int.*

*Scala can use reasoning like this to infer the return type of a function. If the type can be safely inferred, it can be omitted from the function signature:*

***def addThree(x:Int) = x + 3***

*So, that definitely saves some typing. Now should you drop return types? It’s debatable. On one hand, there’s less noise in the code, on the other hand, the code is less self-documenting, since the code reader may have to do some work to figure out the return type. In general, I’d say it’s at the discretion of the programmer.*

*If a function has zero parameters, as in*

***def f() = 7***

*Then you can call it like normal*

***f()***

*or omit the parens*

***f***

*Default, name, and vararg parameters*

*You can supply default parameters for a method like this*

***def hi(first:String=“John”, last:String=“Smith”) = s”hi, $first $last!"***

*You can call the function normally*

***hi(“Jane”, Doe”)***

*Or omit some or all of the arguments*

***hi(“Jane”)***

***hi()***

*Functions can be called by naming arguments, instead of matching parameter positions*

***hi(last=“Doe”)***

*If the function takes a variable number of arguments, add a \* after the parameter type. For example*

***def addThem(nums:Int\*) = nums.sum***

*Then you can call it like*

***addThem(1,2,3,4)***

*You can also “spread out” an ordered collection in an argument call by suffixing the argument with a :\_\**

***addThem(List(2,4,6):\_\*)***

*Multiple parameter Lists*

*A function can have multiple parameter lists, as in*

***def addExWhy(x:Int)(y:Int) = x + y***

*You call it by supplying multiple argument lists*

***addExWhy(3)(2)***

*You can create a new function by binding one parameter list and leaving the other parameter list unbound. The unbound parameter list uses the \_ placeholder. This is called partial application, or currying.*

***def addFive = addExWhy(5)\_***

*We now have a new function with a single parameter that adds five to its input, and you can call it like any other “normal” function*

***addFive(4)***

*Function Literals*

*You can define functions without names like this*

***{(x:Int) => x \* 2}***

*Here the function is in curly braces.*

*The parameter list is on the left, then there’s an arrow (called a rocket operator), then the body.*

*You can apply arguments to the function directly like*

***({(x:Int) => x \* 2})(5)***

*or you can assign it to a val and apply arguments to the val.*

***val timesTwo = {(x:Int) => x \* 2}***

***timesTwo(4)***

*In this case it ends up looking like a named function call*

*In cases where the type of the input parameters can be inferred, a placeholder can be used instead of declare a parameter*

***val timesFour: Int => Int = { \_ \* 4 }***

*Here the Int-rocket-int after the val identifier is the signature of the function, saying that there’s an input parameter of type int and the function returns an int*

*You can have multiple placeholder arguments, in which case, the order of occurrence of the argument matches the order of the parameters.*

***val multiply: (Int, Int) => Int = { \_ \* \_ }***

***multiply(4,2)***

*Functions are objects*

*If you create an object with an apply method, like*

***object timesSix {***

***def apply(x: Int):Int = x \* 6***

***}***

*as you might expect, you can call apply on this object like so*

***timesSix.apply(4)***

*but what’s cooler is that in Scala apply is special and you can drop .apply altogether*

***timesSix(4)***

*which looks a lot like an “ordinary” function call. In this way you can treat objects as functions.*

*Call By Name*

*Suppose we have a function*

***def f(time: Long) {***

***println(s”time1 is $time”)***

***Thread.sleep(100)***

***println(s”time2 is $time”)***

***}***

*When we call it like this*

***f(new java.util.Date().getTime)***

*time1 is the same as time2.*

*Why?*

*Because java.util.Date.getTime is evaluated only once, before the function call. The print statements are using the result of that evaluation.*

*Scala offers a different way of evaluating parameters called call-by-name. With call by name, expressions passed into a function call are substituted into the method body, which defers the evaluation of the expression until it is encountered at execution time. Also the expression is evaluated each time it’s encountered in the code. That means the expression may be evaluated 0 to many times, not just once, and that the time of evaluation will be after the function is called.*

*To use a call-by-name parameter, use the rocket operator between the : and the type. For example*

***def f2(time: => Long) {***

***println(s”time1 is $time”)***

***Thread.sleep(100)***

***println(s”time2 is $time”)***

***}***

***f2(new java.util.Date().getTime)***

*Now, time1 is different than time2. Why? It’s because this function is equivalent to*

***def f3() {***

***println(s”time1 is ${new java.util.Date().getTime}”)***

***Thread.sleep(100)***

***println(s”time2 is ${new java.util.Date().getTime}”)***

***}***

*Nested Functions*

*In Scala functions can be nested inside each other. For example,*

***def cube(x:Int) = {***

***def timesX(y:Int) = x \* y***

***timesX(x\*x)***

***}***

*Here the nested function timesX has access to the scope of parent function, so it can use x in addition to it’s own local parameter. This is very handy for helper functions- functions that are only used by one other parent function.*

***Collections***

*Scala has a rich collection library which allows you to easily build, traverse and manipulate data structures. I’m going to talk about some of the most frequently used data structures.*

*Tuples*

*First is a tuple (two-pull or tupple?). A tuple is ordered collection of elements of fixed size. Unlike a list or an array, each position in the tuple can be associated with a different type.*

*Tuples are easy to write. They consist of parenthesis with comma separate values.*

*For example*

***val t1: (Int, String) = (3, “yes”)***

*t1 is strongly typed- the first position is an int, and the second is a string.*

*Type inference also works*

***val t2 = (4, “perhaps”)***

*To access members of tuple, use the .\_<number> postfix operator.*

*For example*

***t2.\_1***

*access the first element*

*and*

***t2.\_2***

*accesses the second element.*

*You can also use a destructuring bind pattern match to get at the constituents of a tuple*

***val (index, status) = (6, “definitely”)***

*Arrays*

*Scala has arrays that are implement as native Java arrays. You can create an array like this*

***val a = new Array[Int](5)***

*The Square brackets are the type parameter, in contrast to angle brackets in Java. The argument 5 is the size of the array*

*To access the array you use parenthesis, not square brackets, like*

***a(1)***

*This makes accessing an array look as if we’re calling a function called a with an argument of 1. Very functional.*

*We can assign a value to an element of an array using an expression such as*

***a(1) = 2***

***a***

*The array is mutated, as it should be.*

*Actually a(1) is really calling the update method on an array, so*

***a.update(2,3)***

***a***

*works as well.*

*Actually, update is one of the magic methods like apply, that you can use for your own objects. For example*

***object Foo {***

***def update(x:Int, y:String) =******println(s”Updating index $x with value $y”)***

***}***

***Foo(2) = “four"***

*does what you’d hope it would do.*

*Lists*

*You can create a list like*

***val l = List(1,2,3,4)***

*You can get the first element using the head member of list*

***l.head***

*You can get the remaining elements using the tail list member*

***l.tail***

*The empty list is written as*

***List()***

*or equivalently*

***Nil***

*which is a predefined object representing the empty list.*

*To build a list, you use the cons operator to append items onto the front of the list. The const operator is represented as double-colons (::). For example*

***1 :: List(2,3)***

*This might look a little weird, since it seems that the cons operator is a method on number. In fact it’s a member on list. Scala has a rule that if an operator ends with a colon (:), then it is right associative.*

*You can append two lists using ++, like*

***List(3,4) ++ List(4,5)***

*Maps*

*You can create a Map using like*

***val m = Map(“a” -> 1, “b” -> 2)***

*to associate a with 1 and b with 2. You might be wondering about the arrow (->), if that’s special Scala syntax. Actually it’s an operator which generates a tuple. For example*

***“a” -> 1***

*So we can also write the previous expression like*

***Map((“a”,1), (“b”,2))***

*We access the map using functional syntax*

***m(“a”)***

*If we try accessing a map key which doesn’t exist, we get an error*

***m(“x”)***

*We’ll learn how to deal with this later.*

*What happens if we try to update the map?*

***m(“a”) = 42***

*Scala is not happy with that.*

*To update maps, we need to create a mutable map. Note the type parameters in the square brackets.*

***val m = collection.mutable.Map[String,Int]()***

*Now try again*

***m(“a”) = 42***

***m***

*All is right now.*