Scala Language 02

# For Comprehensions

*Scala does away with the for loops used in imperative languages such as Java and C++. In its place is the for-comprehension, which transforms collections. Here’s an example of a simple for-comprehension:*

for (x <- 1 to 3) println(x)

*Recall that 1 to 3 is a range object, and we can use the object in place of the expression which created it*

val r = 2 to 4

for (x <- r) println(x)

*You can iterate over a list like*

for (x <- List(2,3,4)) println(x)

*The yield operator on a for comprehension applies an expression against the variables defined in the head of the for comprehension*

for (x <- 1 to 3) yield 2 \* x

*You can have a nested loop using two collections in the same head*

for (x <- 1 to 3;

y <- 1 to x)

yield (x, y)

*You can have an assignment in the head to compute values*

for (x <- 1 to 3;

y <- 1 to x;

z = x \* y)

yield (x, y, z)

*You can use conditionals to filter collections*

for (x <- 1 to 3;

y <- 1 to x if y != 2;

z = x \* y)

yield (x, y, z)

# Options

*Scala discourages use of null. Instead, it uses options. Options may either be Some(x) or None. Example:*

Some(1)

*Get the “wrapped” value from Some(x)*

Some(1).get

*The “empty” value is called None*

None

*You can use getOrElse to return a default value*

Some(1).getOrElse(2)

None.getOrElse(2)

*Maps return options when using the get method. If a key doesn’t exist in the map, None is returned. You could use getOrElse on the results of map.get*

val m = Map(1->2)

m.get(1)

m.get(3)

m.get(1).getOrElse(42)

m.get(3).getOrElse(42)

*Map has a more convenient getOrElse method which makes this simpler*

m.getOrElse(1,42)

m.getOrElse(3,42)

# Higher Order Functions

*A higher order function is a function that can take functions as arguments. For example,*

def runFunAndAddTwo(f: Int => Int, x: Int) = { f(x) + 2 }

**def timesTwo(x:Int) = 2 \* x**

**runFunAndAddTwo(timesTwo, 8)**

*Here’s another example*

def f(x:Int)(g: (Int => Int)) = g(x) \* 2

*Here we have a function with two parameter lists. The first parameter list takes an int, and the second parameter list takes a function with an int input parameter and int return type.*

*We can call f using the following syntax*

f(2) { \_ + 4 }

*Note that we don’t need parenthesis around the function. If a function argument is the sole argument in a parameter list, the parenthesis can be omitted.*

*Here’s an example of a higher-order function of 0-arity, meaning no input parameters*

def printTime(time: () => Long) = s"The time is now ${time()}"

printTime({ () => new java.util.Date().getTime })

printTime{ () => new java.util.Date().getTime }

printTime( () => new java.util.Date().getTime )

*Note the similarity of a zero-arity function to a call-by-name parameter. A call-by-name parameter just drops the argument list and the parens when invoking the function*

def printTime2(time: => Long) = s"The time is now ${time}"

printTime{ new java.util.Date().getTime }

*Collections use higher-order functions. map is common. It transforms each element by applying a function.*

List(1,2,3).map(timesTwo)

List(1,2,3) map timesTwo

List(1,2,3) map { \_ \* 2 }

*An Option can be considered a collection and also implements Map*

Some(3) map { \_ \* 2 }

val IntNone:Option[Int] = None

IntNone map { \_ \* 2 }

*The return type of the mapping function does not have to be the same as the input type. Here we start with a list of Ints and end up with a List of List of Int*

def f(x: Int):List[Int] = if (x % 2 == 1) List(x) else Nil

List(1,2,3) map f

*foreach is like map but expects a function of return type Unit*

List(1,2,3) map { x => println(x) }

List(1,2,3) foreach { x => println(x) }

*Filter collects all elements that satisfy the predicate*

List(1,2,3) filter { \_ % 2 == 1 }

*Flatmap maps all the elements and then appends them (rather than consing them together plus nil)*

def f(x: Int):List[Int] = if (x % 2 == 1) List(x) else Nil

List(1,2,3) flatMap f

List(1) ++ List() ++ List(3)

Some(3) flatMap { (x:Int) => Some(x + 2) }

IntNone flatMap { (x:Int) => Some(x + 2) }

*Many higher order functions take predicates, that is, boolean-valued functions. Count is an example.*

List(1,2,3) count { \_ % 2 == 1 }

*Find searches for an element that satisfies the predicate and returns an option (None if not found)*

List(1,2,3) find { \_ % 2 == 0 }

List(1,2,3) find { \_ % 2 == 2 }

# Folding

*When you want to iterate over a list and accumulate some value, the fold operator is a functional way of doing that.*

*The fold operator has two parameters, an initial value and an accumulator function. The function has two parameters, one representing the current item and another representing an accumulated value. The result of the function is an accumulated value. For example, to multiply a list of numbers, we could use*

List(1,2,3,4).foldLeft(1) { (x:Int, acc:Int) => x \* acc }

*Here we’re applying the fold operator on the list. The first argument to fold is the initial value of 1. The second argument is a function which takes two parameters, x, which is the current item in the list, and acc which is the accumulated value. It then multiplies the item with the accumulated value and returns the result as the new accumulator.*

*To be more explicit, the fold is doing the following*

def f(x:Int, acc:Int) = x \* acc

var acc = 1

acc = f(1,acc)

acc = f(2,acc)

acc = f(3, acc)

acc = f(4, acc)

*A more compact way of writing foldLeft is using the slash-colon operator*

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

(1 /: l){ \_ \* \_ }

*If the (1 :/ l) is tripping you up, write it more explictly*

l./:(1)({\_ \* \_ })

*or*

l./:(1){\_ \* \_ }

*The parens around the (1 /: l) in the original expression are the way you use infix notation with multiple parameter lists. To show a simpler example*

object foo {

def op1(x:Int) = x

def op2(x: Int)(y:Int) = x + y

}

*Here’s a simple infix example*

foo op1 3

*Here’s an infix example with multiple parameter lists*

(foo op1 3)(4)

*There’s also foldRight, which accumulates values right-to-left*

List(1,2,3,4).foldRight(1) { (x:Int, acc:Int) => x \* acc }

*And it’s cryptic counterpart, the colon-slash operator*

(l :\ 1){ \_ \* \_ }

# Classes

*Classes are defined almost like Java, without the syntatic noise - constructor args are in the header*

abstract class Person(val name: String)

*We can extend classes, ensuring their super constructors are called*

class Employee(override val name: String, val salary:Int) extends Person(name)

*Case classes provide automatic toString, immutable members, and pattern matching (later)*

case class Engineer(override val name: String, override val salary:Int, languages:List[String]) extends Employee(name, salary)

*Create an Engineer*

val mike = Engineer("Mike", 33000, List("Scala"))

*Notice how an instance is constructed. There’s no new operator. That’s similar to the syntax we saw when creating datastructures like Lists.*

*We get Nice printing for free*

mike

*We can access fields using do notation*

mike.name

mike.languages.head

# Traits

*Traits are like interfaces in Java. The main difference is that traits can have implementations for their methods. In contrast to classes, traits may not have constructors. Traits can be used to mixin functionality to classes, adding “flavor” to them. You can specify multiple traits by using the “with” keyword in the class header. Example:*

trait Male { def favoriteColor():String = "blue" }

case class MaleEngineer(override val name: String, override val salary:Int, languages:List[String]) extends Employee(name, salary) with Male

val john = MaleEngineer("john", 100000, Nil)

john.favoriteColor

# Objects

*Instead of static members, Scala has "singleton" objects*

object Widget {

var count = 0

def printCount() { println(f"count is $count") }

}

*Access to singleton members is similar access to Java class members*

Widget.count

Widget.printCount

Widget.count = 2

Widget.printCount

*Singeltons can be paired with classes, hence the term "companion object". They must be defined in the same unit, hence the :paste and ctrl-d in the REPL.*

:paste

object Widget {

private var count = 0

def printCount() { println(f"count is $count") }

}

class Widget {

Widget.count = Widget.count+1

}

<ctrl-d>

*The class is "friends" with the companion object*

Widget.printCount()

val w1 = new Widget()

val w2 = new Widget()

Widget.printCount()