Lecture 20: The Many Hats of Scala: Imperative
10:00 AM, Mar 12, 2018

Contents

1 Imperative Programming in Scala

Objectives

By the end of these notes, you will know:

- How to use basic imperative constructs in Scala

By the end of these notes, you will be able to:

- Write and test basic imperative programs in Scala

1 Imperative Programming in Scala

Although Scala gives us the functional idioms we know and love, it can still be useful to have access to imperative constructs like loops. In this lecture we’ll give you a short crash course in Scala’s imperative features.

First, we’ll start by defining an `object` in which to put our imperative code. Inside the object, we’ll create a pair of lists:

```scala
object ScalaLecture extends App {
    val list1: List[Int] = List(1, 11, 21)
    val list2: List[Int] = List(2, 10, 20)

    // code fragments below will go here
}
```

Recall that, since the `object` extends `App`, we can just put our code inside its body; no need to define a `main` function.

While Loops  

Much like Java, Scala provides `while` loops. The syntax is quite similar. For instance, to print out the numbers from 1 to 10 imperatively, we’d write the following:

```scala
var i = 1
while(i <= 10) {
    println(i)
    i = i + 1
}
```
**For Loops**  Scala also provides Java-style for loops, just with a slightly different syntax. Instead of writing:

```scala
for(int v1 : list1) {
    System.out.println(v1);
}
```

we'd instead write:

```scala
for(v1 <- list1) {
    println(v1)
}
```

Nesting loops is much easier in scala. If we want to obtain all pairs of elements in `list1` and `list2` (regardless of order), we can write:

```scala
for(v1 <- list1; v2 <- list2) {
    println(v1 +"," + v2)
}
```

If you find yourself needing to program imperatively in Scala, you will usually find that the code you write is far more concise than what Java would require.

**Comprehensions**  Scala extends the idea of for loops into a useful construct called *sequence comprehensions*. Essentially, they let us construct lists via a loop, but without the awkward and verbose baggage of an “under construction” helper list object. Here’s an example: let’s look for numbers in our two lists that differ by 1, and build a list of their sums.

```scala
val diff1Sums = for(v1 <- list1; v2 <- list2; if v1 == v2 + 1)
    yield { v1 + v2 }
println(diff1Sums)
```

Running this produces the list we’d expect: `List(21, 41)`. But what is this new `yield` keyword doing? To find out, let’s try running the same code, only without `yield`.

```scala
val diff1Sums = for(v1 <- list1; v2 <- list2; if v1 == v2 + 1)
(v1 + v2)
println(diff1Sums)
```

Instead of the above list, this code produces an odd-looking value: `()`. Scala calls this `unit`\(^1\) and it essentially means that nothing was produced. Note that this is different from raising an error or returning `null`; here there was nothing for the computation to return.

Why is this happening? Because Scala is faithfully following our instructions: it loops through every pair of values in the two lists, and when `v2 == v2 + 1` is true, it evaluates the sum `v1 + v2`...and then discards it, because nothing uses the computed value. In other words, there’s nothing to tell this loop what to do with the numbers it computes.

\(^1\)Some other languages call it “void”.
In contrast, the `yield` keyword tells Scala to *keep the values computed* within the loop, and add them to a list, which the comprehension returns. Combined with functional methods like `map` and `fold`, comprehensions are incredibly useful and concise.

**Arrays (with a note on type annotations)** Arrays in Scala work much like arrays in Java. We can instantiate an array of 5 integers similarly to the way we’d do it in Java:

```scala
val arr1 = new Array[Int](5)
```

Thinking back to Dynamic Programming, we might want to make an array of options. This is much easier to do in Scala than Java:

```scala
val arr2 = new Array[Option[Int]](5)
```

Even better, we can use `Arrays.fill`, which lets us easily initialize the array to `None` (recall that an option either contains `None` or `Some(k)` for some value `k`):

```scala
val arr2: Array[Option[Int]] = Array.fill(5)(None)
```

Java would let us write `for` loops that iterated through a range of indexes. For instance, we could write `for(i=0;i<100;i++){ ... }`. Scala provides the same, albeit in a more readable way. The following code traverses every cell of our array and populates each cell with that cell’s index.

```scala
for(i <- 0 until arr1.length) {
    arr2(i) = Some(i)
}
```

Note that we gave an explicit type to `arr2` when we declared it, rather than leave Scala to infer a type. To see why, try it out for yourself! Type inference isn’t magic—sometimes you need to give Scala an extra hint or two.

**Maps (Mutable, Immutable)** Working with Scala’s maps is also similar to what you’ve seen in Java. Here’s code to create a map that is instantiated with two key-value pairs: "tim" is mapped to 355 and "kathi" is mapped to 309.

```scala
val map1 = Map("tim" -> 355, "kathi" -> 309)
```

By default, maps in Scala are *immutable*. That is, you’re back to a functional world view where, to modify a map, you need to obtain a new object. E.g.,

```scala
val map1a = map1 + ("tim" -> 306) // add to map, return new map
```

If we were to evaluate `map1 == map1a`, we’d get `false`, since they are different objects. One is the original map, and the other has the updated entry for "tim".

But what if you want a Java-style, mutable map? Scala provides immutable and mutable versions of many different data structures, including maps. To get a mutable map, just create one via the `mutable` package:
val map2 = scala.collection.mutable.Map("tim" -> 355, "kathi" -> 309)

To add (or update) a key-value pair via mutation, use `+=` rather than `+`:

map2 += ("tim" -> 316)

And that’s it! If you’re curious about what else you can do with maps in Scala (or arrays, or lists, or...) we encourage you to check out the Scala language reference at https://www.scala-lang.org/api/current/.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs018/feedback