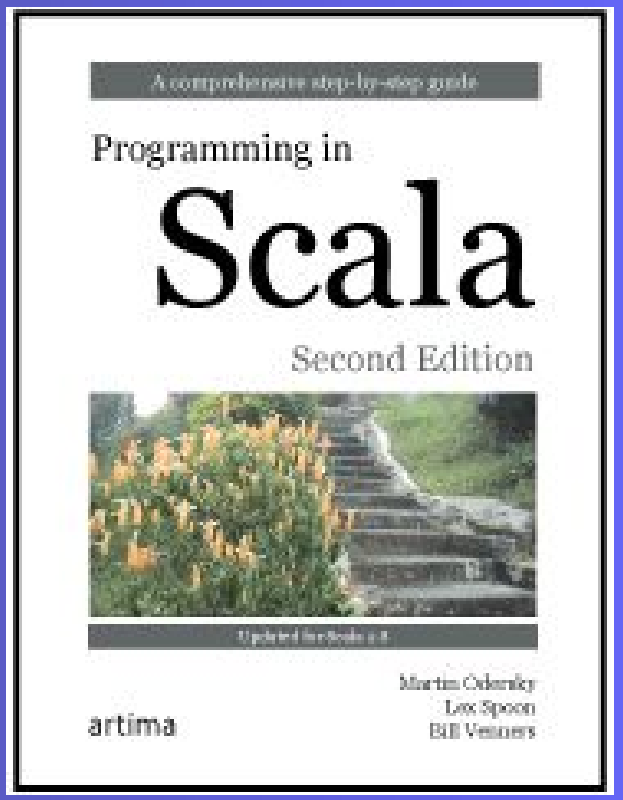
Stairway to Scala - Flight 4

Built-in control structures

Bill Venners
Dick Wall



escalatesoft.com

Flight 4 goal

Look at Scala's built-in control structures,
its Unit value, and
the imperative/functional divide.

If expressions

```
var filename = "default.txt"
if (!args.isEmpty)
  filename = args(0)

val filename =
    if (!args.isEmpty) args(0)
    else "default.txt"
```

Val or var?

Look for opportunities to use vals. They can make your code both easier to read and easier to refactor.

While and do-while loops

```
def gcdLoop(x: Long, y: Long): Long =
{
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  b
}
```

```
var line = ""
do {
  line = readLine()
  println("Read: "+ line)
} while (line != "")
```

Scala's Unit value

```
scala> def greet(): Unit = { println("hi") }
greet: ()Unit

scala> greet() == ()
hi
res0: Boolean = true
```

```
var line = ""
while ((line = readLine()) != "") // This doesn't work!
  println("Read: "+ line)
```

```
var line = ""
while ({line = readLine(); line} != "") // a working alternative
  println("Read: "+ line)
```

A recursive method

```
def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

For “loops”

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere)
    println(file)

scala> for (i <- 1 to 4)
    println("Iteration "+ i)

for (i <- 1 until 4)
    println("Iteration "+ i)

// not common in scala:
for (i <- 0 to filesHere.length - 1)
    println(filesHere(i))
```

For expression filtering

```
val filesHere = (new java.io.File(".")).listFiles
for (file <- filesHere if file.getName.endsWith(".scala"))
    println(file)
```

This is similar in behavior to:

```
for (file <- filesHere)
    if (file.getName.endsWith(".scala"))
        println(file)
```

Multiple filters

```
for (
    file <- filesHere
    if file.isFile
    if file.getName.endsWith(".scala")
) println(file)
```

Nested iteration

```
def fileLines(file: java.io.File) =
    scala.io.Source.fromFile(file).getLines.toList

def grep(pattern: String) =
    for (
        file <- filesHere
        if file.getName.endsWith(".scala");
        line <- fileLines(file)
        if line.trim.matches(pattern)
    ) println(file + ": " + line.trim)

grep(".*gcd.*")
```

Mid-stream assignment

```
def grep(pattern: String) =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(pattern)
  } println(file + ": " + trimmed)

  grep(".*gcd.*")
```

Producing a new collection

```
def scalaFiles() =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
  } yield file
```

```
for (file <- filesHere if file.getName.endsWith(".scala")) {
  yield file  // Syntax error!
}
```

Transforming an Array[File] to an Array[Int]

```
val forLineLengths =
  for {
    file <- filesHere
    if file.getName.endsWith(".scala")
    line <- fileLines(file)
    trimmed = line.trim
    if trimmed.matches(".*for.*")
  } yield trimmed.length
```

Throwing exceptions

```
def half(n: Int) =
    if (n % 2 == 0)
        n / 2
    else
        throw new IllegalArgumentException("n must be even")

def betterHalf(n: Int) = {
    require(n % 2 == 0, "n must be even")
    n / 2
}
```

Try-catch clause

```
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

try {
  val f = new FileReader("input.txt")
  // Use and close file
} catch {
  case ex: FileNotFoundException => // Handle missing file
  case ex: IOException => // Handle other I/O error
}
```

Finally clause

```
import java.io.FileReader

val file = new FileReader("input.txt")
try {
  // Use the file
} finally {
  file.close()  // Be sure to close the file
}
```

Try expression yielding a value

```
import java.net.URL
import java.net.MalformedURLException

def urlFor(path: String): URL =
    try new URL(path)
    catch {
        case e: MalformedURLException =>
            new URL("http://www.scala-lang.org")
    }
```

Match expressions

```
def matchThis(s: String): Unit = {
  s match {
    case "salt" => println("pepper")
    case "fish" => println("chips")
    case "eggs" => println("bacon")
    case _ => println("huh?")
  }
}
```

A match that yields a value

```
val firstArg = if (!args.isEmpty) args(0) else ""

val friend = println(friend)

firstArg match {
    case "salt" => "pepper"
    case "fish" => "chips"
    case "eggs" => "bacon"
    case _ => "huh?"
  }
```

Imperative style multiplication table

```
def printMultiTable() {

  var i = 1
  // only i in scope here

  while (i <= 10) {

    var j = 1
    // both i and j in scope here

    while (j <= 10) {

      val prod = (i * j).toString
      // i, j, and prod in scope here
      var k = prod.length
      // i, j, prod, and k in scope here
```

Imperative style multiplication table

```
    while (k < 4) {
      print(" ")
      k += 1
    }

    print(prod)
    j += 1
  }

  // i and j still in scope; prod and k out of scope
  println()
  i += 1
 }
 // i still in scope; j, prod, and k out of scope
}
```

Functional style alternative

```
// Returns a row as a sequence
def makeRowSeq(row: Int) =
    for (col <- 1 to 10) yield f"${row * col}%4d"

// Returns a row as a string
def makeRow(row: Int) = makeRowSeq(row).mkString

// Returns table as a string with one row per line
def multiTable() = {

    // a sequence of row strings
    val tableSeq = for (row <- 1 to 10) yield makeRow(row)

    tableSeq.mkString("\n")
}
```

Exercises for Flight 4