Custom Control Structures

Abstracting Control Structures With First Class Functions

Agenda

1. Simplifying Code Using Higher Order Functions
2. Loans and Resource Management
3. Currying and Multiple Parameter Lists
4. Higher Order Functions
5. Function Arity
6. By-name Functions
7. Custom Looping

Using the Contents of a File

```
import java.io.File
import scala.io.Source

def fileContainsQuestion(file: File): Boolean = {
  val source = Source.fromFile(file)

  try {
    source.getLines().toSeq.headOption.map { line =>
      line.trim.endsWith("?")
    }.getOrElse(false)
  } finally source.close()
}
```

```
def emphasizeFileContents(file: File): String = {
  val source = Source.fromFile(file)

  try {
    source.getLines().toSeq.headOption.map { line =>
      line.trim.toUpperCase
    }.getOrElse("")
  } finally source.close()
}
```

Using Generics and HoFs

```
def withFileContents[A](file: File, fn: String => A, default: A): A = {
  val source = Source.fromFile(file)

  try {
    source.getLines().toSeq.headOption.map { line =>
      fn(line)
    }.getOrElse(default)
  } finally source.close()
}
```

- **A** is a *type parameter* and can often be inferred
- We supply a function literal from `String => A` to the function
- We also supply a fallback default value of type `A`

Calling the Generic Method

```
val hamlet = new File(fileLoc, "hamlet.shkspr")

withFileContents(hamlet, _.trim.endsWith("?"), false)
// false

withFileContents(hamlet, _.trim.toUpperCase, "")
// THE LADY DOTH PROTEST TOO MUCH, METHINKS.

// something more complex?
// find most common letter
withFileContents(hamlet, { line =>
  val letters = line.toLowerCase.filterNot(_ == ' ').toSeq
  val grouped = letters.groupBy(identity)
  grouped.maxBy { case (char, seq) => seq.length }._1
}, 'e')
// 't'
```

- It works, but that syntax is awkward to both write and read

Currying Revisited

- From previous example

```
val add3: (Int, Int, Int) => Int = (a, b, c) => a + b + c
// (Int, Int, Int) => Int

val add3curried = add3.curried
// Int => (Int => (Int => Int))
```

- We could write the function this way ourselves:

```
val add3c: Int => Int => Int => Int = a => b => c => a + b + c
// Int => (Int => (Int => Int))
```

- The parens are not required, but can clarify what's happening
- To call these curried functions:

```
add3(1,2,3)           // 6
add3curried(1)(2)(3)  // 6
add3c(1)(2)(3)        // 6

add3c.apply(1).apply(2).apply(3)  // 6
```

Currying in Methods

- Scala methods can likewise be curried:

```
def add3method(a: Int)(b: Int)(c: Int) = a + b + c
add3method(1)(2)(3)   // 6
```

- When a parameter list has one parameter, can swap **{}**s for **()**s

```
add3method { 1 } { 2 } { 3 }  // 6
```

- In parens, you can have commas, in curlies, you get semi-colon inference
- This syntax trick is useful for cleaning up our generic implementation

Curried Generic Loan

```
def withFileContents[A](file: File, default: A)(fn: String => A): A = {
  val source = Source.fromFile(file)

  try {
    source.getLines().toSeq.headOption.map { line =>
      fn(line)
    }.getOrElse(default)
  } finally source.close()
}

withFileContents(hamlet, false)(_.trim.endsWith("?"))  // curried with parens
withFileContents(hamlet, "")(_.trim.toUpperCase)

// find most common letter
withFileContents(hamlet, 'e') { line =>   // curried with curlies
  val letters = line.toLowerCase.filterNot(_ == ' ').toSeq
  val grouped = letters.groupBy(identity)
  grouped.maxBy { case (char, seq) => seq.length }._1
}
```

- Often, function parameters are curried in a separate parameter list at the end of the method definition

Function Arity

- Functions have an Arity, which means the number of input parameters

```
val sq: Int => Int = x => x * x  // Function1[Int, Int]
val add: (Int, Int) => Int = (a, b) => a + b  // Function2[Int, Int, Int]
val mult3: (Int, Int, Int) => Int = _ * _ * _ // Function3[Int, Int, Int, Int]
```

- There is also a `Function0`:

```
import scala.util.Random
val makeARandom: () => Double = () => Random.nextDouble()

makeARandom()  // some double value
makeARandom()  // some different double value
```

- The function takes no parameters, but is not evaluated until () is applied, and is evaluated each time an apply happens

Writing Our Own Loop

```
import scala.annotation.tailrec

@tailrec
def fruitLoop(pred: () => Boolean)(body: () => Unit): Unit = {
  if (pred()) {
    body()
    fruitLoop(pred)(body)
  }
}

var x = 0

fruitLoop(() => x < 5) { () =>
  println(x * x)
  x += 1
}
```

- This looks kind of like a while loop, except for those `() =>` bits when we call it

By-name Functions

- To provide nicer syntax at the call site, Scala has *by-name* functions as an alternative to Function0:

```
@tailrec
def fruityLoop(pred: => Boolean)(body: => Unit): Unit = {
  if (pred) {
    body
    fruityLoop(pred)(body)
  }
}

var y = 0
fruityLoop(y < 5) {
  println(y * y)
  y += 1
}
```

- We have now constructed a loop syntactically identical to `while`
- The by-name function is evaluated without ()s, "by-name" only (except if you call another method expecting a by-name)
- By-names are easy to get wrong, beware! Convert to `Function0` ASAP