Control Structures in Scala

Built in control structures, statements vs expressions

Agenda

1. Expressions vs Statements
2. Unit
3. Scala's if expression
4. val and var
5. try..catch..finally
6. While loops
7. For expressions
8. The Four Gs of For
9. For, not just for loops
10. Match expressions
11. Guards
12. String interpolation

Expressions vs Statements

- An expression has a return value, a statement does not (at least not a useful one)
- In Functional Programming, a *pure* expression has no effects other than those seen in the return value
- Scala has no `void` keyword for expressions, void is the absense of a return type, but in Scala **everything** has a return type
- But for statements, that return type is `Unit`, conversely, a return type of `Unit` denotes an expression

```
// an expression
val x = 1 + 2  // x: Int = 3

// a statement
println(x)  // prints 3

// since everything has a return type
val un = println(x)  // un: Unit = ()

un == ()  // () is the only instance of Unit
// Warning: comparing values of types Unit and Unit using `==`
// will always yield true
```

Unit

- A `Unit` return type implies that a method must have a side effect to do something useful
- E.g. I/O, set or update a variable
- A non-`Unit` return type does not imply that there are no side effects, however
- Many built in constructs in Scala are expressions rather than statements, few return only `Unit`
- Even if you have side effects, there may be something more useful than `Unit` that you can return, you can always ignore it if you don't want to use it
- There is only one instance of `Unit`, it is `()` (sometimes referred to as empty tuple)
- `Unit` is descended from `AnyVal` like the "primitive" types

Returning Something Other Than Unit

- E.g. a simple file writer:

```
import java.io._

class WriterOutput(writer: PrintWriter) {
  def write(s: String): Unit = writer.println(s)
}

val ex1 = new PrintWriter(new File("ex1.txt"))

val out1 = new WriterOutput(ex1)

out1.write("Hello")
out1.write("to")
out1.write("you")

ex1.close()
```

- `write` method returns `Unit`, to write out multiple things, invoke it on the same unit multiple times
- `close()` method also returns `Unit` (and has a side effect). In Scala, the convention is to always put parens on zero parameter methods that have side effects

Returning `this` Instead Of `Unit`

- If you return `this` instead in the above example, you get a fluent API cheaply

```
class WriterOutput2(writer: PrintWriter) {
  def write(s: String): WriterOutput2 = {
    writer.println(s)
    this
  }
}

val ex2 = new PrintWriter(new File("ex2.txt"))

val out2 = new WriterOutput2(ex2)

out2.write("Hello").write("to").write("you")

ex2.close()
```

Scala's `if` Expression

- In many languages, e.g. Java, `if` is a statement (has no return value)

```
// this is Java
String fileName = "default.txt";  // defines a variable

if (args.length > 0) {
  fileName = args[0];  // side effect
}
```

- Java also has a ternary operator which is an expression:

```
String fileName = (args.length) > 0 ? args[0] : "default.txt";
```

- Scala combines these two things into one (`if` is an expression)

```
val fileName = if (args.length > 0) args(0) else "default.txt"  // can now be val
```

- The return type is the combination of the types on both sides of the else, e.g.

```
val res = if (x > 0) x else false  // type of res will be AnyVal
```

`val` and `var`

- As we saw in module 1, `var` can change but `val` cannot
- Prefer `val` when you can use it, it makes code easier to reason about, and easier to refactor
- IntelliJ will give code hints when something can be a `val`
- You can also make it highlight `vars`, for example color them red
- When you have more expressions, you can have more `vals`

try … catch … finally

- Like `if`, Scala's `try … catch … finally` is an expression
- The result (and type) is decided by the `try` and `catch` blocks
- If present, the `finally` block always runs, but doesn't affect the result or type

```
val args = Array.empty[String]

val fileName2 =
  try {
    args.head // throws exception on empty array
  }
  catch {
    case _: NoSuchElementException => "default.txt"
  }
  finally {
    println("Wheeeee")
    "the finally block"
  }

// fileName2: String = default.txt
```

While Loop

- A statement (returns `Unit`)

```
def greet(n: Int): Unit = {
  var i = 0
  while (i < n) {
    i += 1
    println("hello")
  }
}

greet(5)
```

You can avoid the `while` (and the `var`):

```
@tailrec
def greet(n: Int, curr: Int = 0): Unit = {
  if (curr < n) {
    println("hello")
    greet(n, curr + 1)
  }
}
```

There is also a `do..while` loop where the body is always called at least once

For

```
for (i <- 1 to 10) println i * i
```

```
(1 to 10).foreach(i => println(i * i))
```

```
for (i <- 1 to 3; j <- 1 to 3) println(i * j)
```

```
(1 to 3).foreach(i => (1 to 3).foreach(j => println(i * j)))
```

```
for {
  i <- 1 to 3  // {}s turn on semi-colon inference
  j <- 1 to 3
} {            // multi-line block (looks weird without yield)
  println(i * j)
}
```

Without `yield` block, `foreach` is used and `Unit` is the result type

For ... Yield

More idiomatic in Scala (does not need to have side effect)

```
for (i <- 1 to 10) yield i * i

(1 to 10).map(i => i * i)

for (i <- 1 to 3; j <- 1 to 3) yield i * j

(1 to 3).flatMap(i => (1 to 3).map(j => i * j))

for {
  i <- 1 to 3
  j <- 1 to 3
  k <- 1 to 3
} yield {
  i * j * k
}

(1 to 3).flatMap(i => (1 to 3).flatMap(j => (1 to 3).map(k => i * j * k)))
```

- For yield blocks, all generators are `flatMap` except the last which is `map`

The Four Gs of For

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

- **G**enerator: `file <- filesHere`, denoted by the `<-`, all generators in the same for block must be of the same type (e.g. a collection, or a Try).
- **G**uard: `if file.getName.endsWith(".sc")`, guards short-circuit the for expression, causing filtering behavior.
- inline assi**G**nment: `trimmed = line.trim`, a simple val expression that may be used in both the remainder of the for block, and in the yield block.
- **G**ive (or the yield), after the `yield` keyword comes the payoff of the for block setup.

```
filesHere.filter(_.getName.endsWith(".sc")).flatMap { file =>
  fileLines(file).filter(_.trim.matches(".*for.*")).map { line =>
    line.trim.length
  }
}
```

For is More Than Just Loops

```
import scala.concurrent._
import duration._
import ExecutionContext.Implicits.global

val f1 = Future(1.0)
val f2 = Future(2.0)
val f3 = Future(3.0)

val f4 = for {
  v1 <- f1
  v2 <- f2
  v3 <- f3
} yield v1 + v2 + v3

Await.result(f4, 10.seconds)
```

- Easy asynchronous programming
- Also `Try`, `Option`, `Either`, `*your type here*`
- All you need is a type with `foreach`, `map`, `flatMap`, `withFilter` with the correct type signatures

Match Expressions

Like Java's `switch` but more powerful:

```
val x = 1

x match {
  case 1 => println("it's one")
  case 2 => println("it's two")
  case _ => println("it's something else")
}
```

It's an expression:

```
val res = x match {
  case 1 => "one"
  case 2 => "two"
  case _ => "something else"
}

// res: String = "one"
```

Match Expression Guards

```
val n = -1

n match {
  case 0 => "It's zero"
  case v if v > 0 => s"It's positive $v"
  case v => s"It's negative ${v.abs}"
}

// res0: String = It's negative 1
```

The progression of tests only continues until the first match of everything to the left of the =>

Match and More Match

Can match Strings, Lists, more...

```
def matchIt(x: Any): String = x match {
  case "Hello" => "Well, hello back"
  case 1 :: rest => s"A list beginning with 1, rest is $rest"
  case Nil => "The empty list"
  case 5 => "The number 5"
  case _: List[_] => "Some kind of list, not empty and not starting with 1"
}

matchIt(5)           // The number 5
matchIt(List(1,2,3)) // A list beginning with 1, rest is List(2, 3)
matchIt(List(1))     // A list beginning with 1, rest is List()
matchIt(List(2,3))   // Some kind of list, not empty and not starting with 1
matchIt(Nil)         // The empty list
matchIt(2.0)         // Exception!: MatchError
```

More on match later in the course

String Interpolation

```
val x = 10

val y = 2.12

val name = "Fred"

s"$name $x $y"              // Fred 10 2.12

s"$name is ${x * y}"        // Fred is 21.200000000000003

f"$name is ${x * y}%08.4f"  // Fred is 021.2000

s"$names"                   // won't compile!

s"${name}s"                 // Freds

"\t\n"
raw"\t\n"                   // \t\n

"""\t\n"""                  // \t\n
```

- `f` interpolation follows the printf notation
- `raw` does not escape literals in the string

Exercises for Module 4

- Find the Module04 class and run it with ScalaTest
- Follow the instructions and fix the tests until everything is green