CS 2340 Objects and Design - Scala

Control Structures

Christopher Simpkins
chris.simpkins@gatech.edu

Control Structures

Scala has only four built-in control structures:

- `if`
- `while`
- `for`
- `try`

Other control structures are defined in libraries using function values – more on that next time. Another example of how Scala *scales* elegantly.

`if, for, try` return values, which means they are *expressions*

`if` Expressions

Scala's `if` works just like `if` in other languages, except that it returns a value.
Consider a typical imperative `if`:

```
var port = 9999
if (!args.isEmpty) {
  port = args(0).toInt
}
```

In Scala, the extra variable initialization is unnecessary:

```
val port = if (!args.isEmpty) args(0).toInt else 9999
```

Also, with initialization and selection combined, we can use a `val` instead of a `var` ...

The value of `val`. A Rumination on Functional Style

`val`s are *referentially transparent*, a property of functional programs which means

- you can reason about programs algebraically (as opposed to keeping a state machine in your head while you read an imperative program)
- you can safely substitute equivalent expressions

So you could use the `port` variable like this if you need to use it later:

```
val port = if (!args.isEmpty) args(0).toInt else 9999
val sock = new Socket(``127.0.0.1'', port)
```

... or like this if you don't need the port after your socket is initialized:

```
val sock =
  new Socket(``127.0.0.1'', if (!args.isEmpty) args(0).toInt else 9999)
```

`Socket` happens to have a `getPort` method. However, better to have extra `val port = ...` line for readability.

`while` Loops

- Only built-in control structure that's not an expression
- Still has a result type: `Unit`, whiich is a type with a single value, the *unit value*, written `()`

Works just like in imperative languages

```
while (!world.isTerminal(s)) {
  a = chooseAction(Q, s)
  s1 = world.act(s, a)
  r = rewards(s1)
  Q((s,a)) = Q((s,a)) + alpha * (r + (gamma*max(Q,s1)) - Q((s,a)));
  s = s1
}
```

There's also a `do-while`, which performs its test after the loop body, meaning the loop body will execute at least one time

```
do {
  something()
} while (you == can)
```

`for` Expressions

Extremely versatile. Here's the syntax from the Scala spec:

```
Expr1         ::= 'for' ('(' Enumerators ')' | '{' Enumerators '}')
                    {nl} ['yield'] Expr
Enumerators   ::= Generator {semi Enumerator}
Enumerator    ::= Generator
                | Guard
                | 'val' Pattern1 '=' Expr
Generator     ::= Pattern1 '<-' Expr [Guard]
Guard         ::= 'if' PostfixExpr
```

Here's a simpler way to think of it using EBNF:

```
forExpr      ::= for '('enumerator+')' { body } [yield] resultExpr
enumerator   ::= generator ([guard] | [varDef])*
generator    ::= scalaVarId <- scalaSeq
guard        ::= if booleanExpr
varDef       ::= scalaVarId = scalaExpr
scalaVarId   ::= <any valid Scala variable identifier>
scalaExpr    ::= <any valid Scala expression>
```

Basic `for` Expression

Iterating through a collection:

```
scala> val xs = List(1, 2, 3, 4, 5, 6)
xs: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> for (x <- xs) print(math.pow(x, 2)+" ")
1.0 4.0 9.0 16.0 25.0 36.0
```

- `x <- xs` is the generator (no `guard` or `varDef` in this example)
- A new iteration variable, `x`, is generated for each element of `xs`
- Body is executes `xs.length` times, each time with `x` bound to a new element of `xs`
- `x` not visible outside `for` expression

```
scala> x
<console>:8: error: not found: value x
       x
       ^
```

Imperative-style Iteration

Could use an imperative-style index variable if you wanted:

```
scala> for (i <- 0 to 5) print(math.pow(xs(i), 2)+" ")
1.0 4.0 9.0 16.0 25.0 36.0
```

... but that's not funcitonal style, and it's clearer to simply generate the sequence elements directly.
Why bother thinking about off-by-one errors?

```
scala> for (i <- 1 to xs.length) print(math.pow(xs(i), 2)+" ")
4.0 9.0 16.0 25.0 36.0 java.lang.IndexOutOfBoundsException: 6
at scala.collection.LinearSeqOptimized ...
```

(Did you notice the other off-by-one error?)

Filtering `for` Expression

If we only want even numbers, we can “filter them in” with a guard expression

```
scala> xs
res10: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> for (x <- xs if x % 2 == 0) print (x+" ")
2 4 6
```

`if x % 2 == 0` is a *guard* expression

Nested Iteration

Multiple generators separated by ; produce nested iterations:

```
scala> val xs = List('a, 'b, 'c)
xs: List[Symbol] = List('a, 'b, 'c)

scala> val ys = List(1, 2, 3)
ys: List[Int] = List(1, 2, 3)

scala> for (x <- xs; y <- ys) print((x,y)+" ")
('a,1) ('a,2) ('a,3) ('b,1) ('b,2) ('b,3) ('c,1) ('c,2) ('c,3)
```

Which is shorthand for:

```
scala> for (x <- xs)
     |   for (y <- ys)
     |     print((x,y)+" ")
```

If you put the enumerator expressions in { } instead of ( ), you can put each clause on a separate line without using a semicolon. Parentheses turn off semicolon inference.

Mid-stream Variable Binding

We can bind variables to save computation, for example:

```
scala> val phoenetics = Map('a -> "alpha", 'b -> "bravo", 'c ->
    "charlie")
...
scala> for (x <- xs; xspell = phoenetics(x); y <- ys)
    print((xspell,y)+" ")
(alpha,1) (alpha,2) (alpha,3) (bravo,1) (bravo,2) (bravo,3)
    (charlie,1) (charlie,2) (charlie,3)
```

which is equivalent to:

```
scala> for (x <- xs) {
     |   val xspell = phoenetics(x)
     |     for (y <- ys)
     |       print((xspell, y)+" ")
     | }
(alpha,1) (alpha,2) (alpha,3) (bravo,1) (bravo,2) (bravo,3)
    (charlie,1) (charlie,2) (charlie,3)
```

Notice we had to use `{ }` to define the loop body of the outer `for` expression

Yielding a New Collection

If we want a new collection we can `yield` it.

```
scala> val xs = List(1, 2, 3, 4, 5, 6)
xs: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> for (x <- xs) yield math.pow(x, 2)
res26: List[Double] = List(1.0, 4.0, 9.0, 16.0, 25.0, 36.0)
```

- `yield`ed collections match the collection type of the generator they came from
- `yield` must come after generator definitions and *before* loop body
- intermediate variables are in scope in `yield` expression, loop body variables are not (see previous point)

```
scala> val ys = Array(1, 2, 3, 4, 5, 6)
ys: Array[Int] = Array(1, 2, 3, 4, 5, 6)

scala> for (y <- ys; dub = y*2) yield math.pow(dub, 2)
res29: Array[Double] = Array(4.0, 16.0, 36.0, 64.0, 100.0, 144.0)
```

Throwing Exceptions

You throw exceptions in Scala just like in Java

```
val half =
  if (n % 2 == 0)
    n/2
  else
    throw new RuntimeException("n must be even")
```

- In Scala, `throw` has a return type: `Nothing`
- `Nothing` is a bottom type, i.e., a subtype of everything
- So the type of `half` above is `Int`

Catching Exceptions with `try-catch` Expressions

```
try {
  val f = new FileReader("input.txt")
  // Use and close file
} catch {
  case ex: FileNotFoundException => handleMissingFile(ex.getFileName)
  case ex: IOException => handleIOError()
}
```

- Each case is searched until a match is found
- Only one case is executed, i.e., no fall-through as in C-style switch statement
- `case ex: IOException` means "if ex is of type IOException, but not FileNotFoundException (since the previous case didn't match), execute the code to the right of =>"
- If nothing matches, exception propagates
- Can match everything ("catch-all") with `case _ => ...` ("_" is Scala's "wildcard" character).

Ensuring Clean-up with `finally` Clauses

Use `finally` blocks to release resources, such as file handles, database connections, and network sockets

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

- Just like Java, Scala is lexically (a.k.a. statically) scoped, so `file` had to be defined outside of `try-finally` to be visible in both `try` and `finally` blocks
- `finally` clause is always executed, whether exception is caught or propagated
- Next week we'll see a better way to ensure resource release with the *loan* pattern

Value of `try-catch-finally` Expressions

If `try` block does not throw an exception, the last value in `try` block is value of `try-catch-finally` expression

```
scala> def f() = try {
     |   1
     | } catch {
     |   case ex: RuntimeException => 2
     | } finally {
     |   3
     | }
f: ()Int

scala> f
res32: Int = 1
```

Values of `try-catch-finally` Expressions

If `try` block throws an exception that matches a case in `catch` block, value of the case expression is value of `try-catch-finally` expression

```
scala> def f() = try {
     |   throw new RuntimeException
     |   1
     | } catch {
     |   case ex: RuntimeException => 2
     | } finally {
     |   3
     | }
f: ()Int

scala> f
res32: Int = 2
```

Value of `try-catch-finally` Expressions

`finally` clause is intended for side-effects; last value in finally clause is never the value of the `try-catch-finally` expression

```
scala> def g(): Int = try { 1 } finally { 2 }

scala> g
res32: Int = 1
```

Beware return statements, which override normal value determination:

```
scala> def f(): Int = try { return 1 } finally { return 2 }
f: ()Int

scala> f
res33: Int = 2
```

Here we're saying "regardless of the value of the `try-catch-finally` expression, this method will return 2." Scala style is to avoid explicit returns.

Yielding Values from `try-catch-finally` Expressions

Summary:

- If `try` block does not throw an exception, the last value in `try` block is value of `try-catch-finally` expression
- If `try` block throws an exception that matches a case in `catch` block, value of the case expression is value of `try-catch-finally` expression
- `finally` clause is intended for side-effects; last value in finally clause is never the value of the `try-catch-finally` expression

Avoid explicit returns!

Match Expression Basics

Work like the catch clauses of `try-catch` expressions, except that the match variable is not implicitly defined

```
scala> def attitude(weather: String) = weather match {
     |   case "sun" => "hello"
     |   case "rain" => "good bye"
     |   case _ => "whatever" // if no cases above match
     | }
attitude: (weather: String)java.lang.String
scala> attitude("sun")
res35: java.lang.String = hello
scala> attitude("rain")
res36: java.lang.String = good bye
scala> attitude("pizza")
res38: java.lang.String = whatever
```

- `match` expressions return values
- No fall-through - use `case _ => ...` to define default
- Very powerful and cool. We'll see later how case classes and pattern matching support algebraic data types

Scala Has No Built-in Break and Continue

Consider this Java code that finds the first argument that ends in ".scala" but doesn't start with "-"

```
int i = 0; // This is Java
boolean foundIt = false;
while (i < args.length) {
  if (args[i].startsWith("-")) {
    i = i + 1;
    continue;
  }
  if (args[i].endsWith(".scala")) {
    foundIt = true; break;
  }
  i = i + 1;
}
```

How do we eliminate the breaks and continues?

Scala Has No Built-in Break and Continue

Replace continues with ifs and breaks with boolean variables.

```
var i = 0
var foundIt = false
while (i < args.length && !foundIt) {
  if (!args(i).startsWith("-")) {
    if (args(i).endsWith(".scala"))
      foundIt = true
  }
  i=i+1
}
```

Better yet, write a recursive function that captures the meaning of the logic:

```
def searchFrom(i: Int): Int =
  if (i >= args.length) -1
  else if (args(i).startsWith("-")) searchFrom(i + 1)
  else if (args(i).endsWith(".scala")) i
  else searchFrom(i + 1)
val i = searchFrom(0)
```

Closing Points

- Scala is lexically scoped (we've seen examples of this)
- Read the chapter's section on refactoring imperative code
- We'll start to see the "Zen" of Scala and funcitonal programming next week when we define our own control structres