Starting Scala

Using SBT and the REPL, Language Basics

Agenda

1. The Scala REPL
2. SBT and the REPL - `sbt console`
3. Variables and Values
4. Types
5. Function Definitions
6. If expressions
7. Try..Catch..Finally expressions
8. Simple Loops

The Scala REPL

- If you downloaded Scala already, you can start it by just typing `scala` at the command line

```
$ scala
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.
```

- So let's try `:help`

```
scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to show)
:h? <string>             search the history
...
```

- The REPL tries to evaluate anything you type in

```
scala> 1 + 2
res0: Int = 3
```

SBT

- SBT, (build tool for Scala) also lets you start the REPL
- In addition, it lets you switch Scala versions very easily (no other install needed)
- Using SBT to run Scala 2.12:
  1. Create a new folder on your computer: `mkdir MyDemoProject`
  2. Create a `build.sbt` file in the new folder with the contents `scalaVersion := "2.12.4"` (or some other scala version)
  3. Run `sbt console` at the command line
  4. Possibly wait a little bit as things download

```
$ sbt console
[info] Loading global plugins from /home/dwall/.sbt/0.13/plugins
[info] Set current project to testproject (in build file: ~/TestProject/)
[info] Starting scala interpreter...
[info]
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161).
Type in expressions for evaluation. Or try :help.

scala>
```

First Time in the REPL

- To quit, either use `Ctrl-D` or `:quit`
- REPL: Read, Evaluate, Print, Loop
- Attempts to evaluate each line as you type it

```
scala> val x = 1 + 2
x: Int = 3
```

- If it can't evaluate the current line, it gives you a continuation

```
scala> val x =
     | 1 + 2
x: Int = 3
```

- If you get stuck in continuation lines, hit `enter` twice in a row

```
scala> val x =
     |
     |
You typed two blank lines.  Starting a new command.
```

vals and vars

- A `val` is a final variable definition, it cannot be re-assigned with a different value

```
scala> val x = 10
x: Int = 10

scala> x = 11
<console>:12: error: reassignment to val
       x = 11
```

- A `var` is a mutable variable definition, it can be reassigned with another value of the **same type**

```
scala> var y = 10
y: Int = 10

scala> y = 11
y: Int = 11

scala> y = "eleven"
<console>:12: error: type mismatch;
 found   : String("eleven")
 required: Int
       y = "eleven"
```

Hiding a val with another val

- What about?

```
scala> val x = 10
x: Int = 10

scala> println(x)
10

scala> val x = 11
x: Int = 11

scala> println(x)
11
```

- Didn't this just re-assign a val?

Scopes in the REPL

- In fact the second `val x` hides the first, it's like this:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val x = 10
println(x)

{  // start a new scope
  val x = 11  // hides x in the outer scope
  println(x)
}

// Exiting paste mode, now interpreting.

10
11
```

- Every new REPL prompt is like a new scope
- Also note `:paste` mode, which allows you to decide when the REPL will evaluate your code

Scala and types

- Remember:

```
scala> var x = 10
x: Int = 10

scala> x = "ten"
<console>:12: error: type mismatch;
 found   : String("ten")
 required: Int
       x = "ten"
```

- Scala won't let us re-assign an integer var to a String
- This is because x has a **type** of `Int` when we create it
- We could have initialized it like this:

```
var x: Int = 10
```

- The type goes after the name of the variable, separated by a `:`
- Scala will infer the best type it can if you don't specify it

Method/Function Definitions

- In addition to `val` and `var`, Scala has `def`
- `def`s can take parameters and act on them

```
scala> def add(x: Int, y: Int): Int = x + y
add: (x: Int, y: Int)Int
```

- Note the = before the function body
- The return type can be inferred, but the parameter types cannot:

```
scala> def add(x: Int, y: Int) = x + y
add: (x: Int, y: Int)Int

scala> def add(x, y) = x + y
<console>:1: error: ':' expected but ',' found.
def add(x, y) = x + y
```

- Scala has nothing to infer the parameter types from

If expressions

- In Scala, `if..else` expressions can return values (unlike many more imperative languages):

```
scala> val a = 10

scala> val b = 12

scala> val m = if (a > b) a else b
m: Int = 12
```

- You can ignore the returned value, then it looks just like other languages:

```
scala> var m = 0

scala> if (a > b) {
     |   m = a
     | } else {
     |   m = b
     | }

scala> m
res6: Int = 12
```

Functional Style

- Ignoring the return value means you will need side effects to do anything useful, e.g. in this example we mutate a variable as our side effect
- Purely functional code avoids side-effects (and does not need vars)
- Hence Scala places a greater emphasis on expressions (that return values) rather than statements (that do not)
- In a Scala expression consisting of more than one nested expression, the final inner expression becomes the value of the overall expression, e.g.

```
scala> val maxSquaredDoubled = if (a > b) {
     |   val aSquared = a * a
     |   aSquared * 2
     | } else {
     |   val bSquared = b * b
     |   bSquared * 2
     | }
maxSquaredDoubled: Int = 288
```

- Scala does have a `return` keyword, but there is rarely any need to use it

Try..Catch..Finally expressions

- Scala also opts for expressions over statements in exception handling

```
scala> val divided = try {
     |   10 / 0
     | } catch {
     |   case ae: ArithmeticException => 0
     | } finally {
     |   println("This always runs, but does not affect the result")
     |   42
     | }
This always runs, but does not affect the result
divided: Int = 0
```

- The `finally` block will always run, but will not return a value (hence its only use is side-effecting)
- If an exception is caught, a value may be returned to *recover*
- Uncaught exceptions are automatically re-thrown
- You may also choose to re-throw a caught exception (or throw another):

```
case ae: ArithmeticException => throw new RuntimeException("Can't divide by 0")
```

Simple Loops

- Scala has only one true looping construct: `while` (and the associated `do..while`)
- `while` is a statement, and has no useful return type of its own
- `while` is non-functional and is often replaced by `foreach` or `map` functions over collections, or by `for` and `for..yield` blocks (we will cover these soon)
- `while` is still used for various reasons, including performance

```
var x = 0

while (x < 10) {
  println(s"the square of $x is ${x * x}")
  x += 1
}
```

- `while` must have a side-effect to do anything useful
- In Scala, everything has a return type, there is no `void`
- `Unit` is provided as a return type for statements, it has one instance: `()`

do..while

- While checks the *predicate* first before running the body of the loop
- Possibly the body will never be executed
- Do while executes the body at least once, and checks the predicate to see if it should repeat:

```
var x = 0

do {
  println(s"the square of $x is ${x * x}")
  x += 1
} while (x < 10)
```

- Note the use of string interpolation: `s"the square of $x is ${x * x}"`

Running and Loading Scala Scripts

- You can also run Scala scripts (or load them into the repl)
- Create a file with some Scala code in it, e.g. `squares.sc` with:

```
var x = 1

while (x <= 10) {
  println(s"The square of $x is ${x * x}")
  x += 1
}
```

- To run it as a script, use `scala squares.sc`
- To load it, start `scala` or `sbt console`, then `:load squares.sc`. *note* we use `.sc` since otherwise `sbt` will try and compile it (which won't work for reasons we will see later)
- `Ctrl-C` will break out of a running script if things go wrong

Module 1 Exercises

1. Create a Scala script file called `timestable.sc` in your working directory (make a directory if you need to)

2. Write two while loops from 1 to 5, one inside the other. Call one loop variable `x` and the other `y`

3. `println` a message in the inner loop that says `s"$x times $y is ${x * y}`

4. Remember to increment both `x` and `y` in their respective loops

5. Either run your script using `scala timestable.sc` or `sbt console` then `:load timestable.sc` to check that it works. You should get 25 lines of output.

Module 1 Exercises - Extra Credit

- `toString` is a method that can be called on anything in Scala, for example:

```
scala> val x = 123
scala> x.toString  // results in a string of "123"
```

- Furthermore, Strings have a `.contains` method that checks to see if a Char is contained anywhere in the String:

```
scala> val s = "123"
s: String = 123

scala> s.contains('3')
res2: Boolean = true

scala> s.contains('4')
res3: Boolean = false
```

- Alter your previous `timestable.sc` to only print out lines if the result of the multiplication contains either a `'4'` or a `'6'` digit in the number produced. `||` is the logical or operator in Scala