Scala Exercises

Contents

Note on Scala vs Spark versions and Scala installation

Since Scala runs on top of the JVM machine, spark uses scala jars embedded.

To know which Scala version is using your current Spark installation, when you start the spark-shell look at the print-out to see which version of scala is using:

```
$ spark-shell
-- some output--
```

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.5.0-cdh5.5.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_65)
```

In these exercises we will use scala version 2.10.4 to be compatible with Spark 1.5.0

Check available Scala versions at

http://www.scala-lang.org/download

We will use

http://www.scala-lang.org/download/2.10.4.html

That is downloaded as a *.tgz file. This file has been unpacked and placed under

```
$ cd /usr/local/share/scala
```

To have quick access to the following environment variables are added in the .bashrc

```
export SCALA_HOME=/usr/local/share/scala
export PATH=$PATH:$SCALA_HOME/bin
```

- Consider install SBT for building your projects (sbt is like a "maven for scala" with steroids): http://www.scala-sbt.org/
- Installing instructionts at: http://www.scala-sbt.org/0.13/tutorial/Manual-Installation.html

Note on IntelliJ Installation

- IntelliJ is a modern IDE which supports Scala development using a plugin
- For these exercises the express edition is installed and downloaded from: https://www.jetbrains.com/idea/download/
- Idea is untar in /usr/local/share/idea-IC. To start the IDE type idea.sh from everywhere (needed environment variables have been added to the .bashrc file).
- When creating a new project select Scala (or sbt if you have it installed) and choose scala-sdk-2.10.4

Exercise 1 – Basic operations in RPEL / Worksheet

Open a shell. At the prompt type:

```
$ scala
Welcome to Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_91).
Type in expressions to have them evaluated.
```

```
Type :help for more information.
scala>
```

You can use IntelliJ Worksheet for the exercises. From IntelliJ sbt or scala project click File / New / Scala Worksheet / filename `Exercises`

Type:

```
scala> println("Hello, World")
Hello, World
scala>
```

Let's try performing a simple arithmetic operation. At a new prompt type 5 * 7 and press Return. Your display should look like this:

```
scala> 5 * 7
res0: Int = 35
scala>
```

These time your Scala command did not print output, but instead returned a value, the product of 5 and 7. When your commands return (or are themselves) values, the REPL will assign them to a new, constant variable so that you can refer to the value in later operations. These "res" variables (a shortened version of "result") are sequentially numbered so that there will always be a unique container for your command's result.

Now that res0 contains the output of the multiplication command, let's make use of it. Type **2 * res0** at a fresh prompt and press Return. You should see something like this:

```
scala> 2 * res1
res1: Int = 70
scala>
```

1. Although `println()` is a good way to print a string, can you find a way to print a string without `println`? Also, what kinds of numbers, strings, and other data does the REPL support?
2. In the Scala REPL, convert the temperature value of 22.5 Centigrade to Fahrenheit. The conversion formula is cToF(x) = (x * 9/5) + 32.
3. Take the result from exercise 2, halve it, and convert it back to Centigrade. You can use the generated constant variable (e.g., "res0") instead of copying and pasting the value yourself.
4. The REPL can load and interpret Scala code from an external file with the :load <file> command. Create a new file named Hello.scala and add a command that will print a greeting, then execute it from the REPL.
5. Another way to load external Scala code is to paste it into the REPL in "raw" mode, where the code is compiled as if it were actually in a proper source file. To do this, type `:paste`, hit Return, and then paste the contents of your source file from exercise 4. After exiting "paste" mode you should see the greeting. (Ctrl + D to exit).

Exercise 2 – Literals, Values, Variables & Types

1. Write a new Centigrade-to-Fahrenheit conversion (using the formula (x * 9/5) + 32), saving each step of the conversion into separate values. What do you expect the type of each value will be?

2. Modify the Centigrade-to-Fahrenheit formula to return an integer instead of a floating-point number.
3. Using the input value 2.7255, generate the string “You owe $2.73.” Is this doable with string interpolation?
4. Is there a simpler way to write the following?

```
val flag: Boolean = false
val result: Boolean = (flag == false)
```

6. Convert the number 128 to a Char, a String, a Double, and then back to an Int. Do you expect the original amount to be retained? Do you need any special conversion functions for this?

Exercise 3 – Expressions and conditionals

For the rest of exercises you can use a scala or sbt project in IntelliJ or if you prefer type in a text editor and use scala compiler:

```
$ scala <source file>
```

Or inside of REPL type:

```
scala> :load file.scala
Loading file.scala...
```

1. Given a string name, write a match expression that will return the same string if nonempty, or else the string “n/a” if it is empty.
2. Given a double amount, write an expression to return “greater” if it is more than zero, “same” if it equals zero, and “less” if it is less than zero. Can you write this with if..else blocks? How about with match expressions?
3. Write an expression to convert one of the input values cyan, magenta, yellow to their six-char hexadecimal equivalents in string form. What can you do to handle error conditions?
4. Print the numbers 1 to 100, with each line containing a group of five numbers. For example:
   1, 2, 3, 4, 5,
   6, 7, 8, 9, 10
   ....
5. Write an expression to print the numbers from 1 to 100, except that for multiples of 3, print “type,” and for multiples of 5, print “safe.” For multiples of both 3 and 5, print “typesafe.”
6. Can you rewrite the answer to exercise 5 to fit on one line? It probably wont be easier to read, but reducing code to its shortest form is an art, and a good exercise to learn the language.

Exercise 4 – Functions

1. Write a function that computes the area of a circle given its radius.
2. Provide an alternate form of the function in #1 that takes the radius as a +String+. What happens if your function is invoked with an empty +String+ ?
3. Write a recursive function that prints the values from 5 to 50 by fives, without using +for+ or +while+ loops. Can you make it tail-recursive?
4. Write a function that takes a milliseconds value and returns a string describing the value in days, hours, minutes and seconds. What's the optimal type for the input value?
5. Write a function that calculates the first value raised to the exponent of the second value. Try writing this first using +math.pow+, then with your own calculation. Did you implement it with

variables? Is there a solution available that only uses immutable data? Did you choose a numeric type that is large enough for your uses?

6. Write a function that calculates the difference between a pair of 2d points (x and y) and returns the result as a point. Hint: this would be a good use for tuples (<<tuples_section>>).
7. Write a function that takes a 2-sized tuple and returns it with the +Int+ value (if included) in the first position. Hint: this would be a good use for type parameters and the +isInstanceOf+ type operation.
8. Write a function that takes a 3-sized tuple and returns a 6-sized tuple, with each original parameter followed by its +String+ representation. For example, invoking the function with +(true, 22.25, "yes")+ should return +(true, "true", 22.5, "22.5", "yes", "yes")+. Can you ensure that tuples of all possible types are compatible with your function? When you invoke this function, can you do so with explicit types not only in the function result but in the value that you use to store the result?

Exercise 5 – First-Class Functions

1. Write a function literal that takes two integers and returns the highest number. Then write a higher-order function that takes a 3-sized tuple of integers plus this function literal, and uses it to return the maximum value in the tuple.
2. The library function +util.Random.nextInt+ returns a random integer. Use it to invoke the previous function with 3 random integers plus a function that returns the larger of 3 given integers. Do the same with a function that returns the smaller of 3 given integers, and then a function that returns the first integer every time.
3. Write a higher-order function that takes an integer and returns a function. The returned function should take a single integer argument (say, "x") and return the product of x and the integer passed to the higher-order function.
4. Let's say that you happened to run across this function while reviewing another developer's code:

```
`def fzero[A](x: A)(f: A => Unit): A = { f(x); x }`
```

   What does this function accomplish? Can you give an example of how you might invoke it?

5. There's a function named "square" that you would like to store in a function value. Is this the right way to do it? How else can you store a function in a value?

```
def square(m: Double) = m * m
val sq = square
```

6. Write a function called "conditional" that takes a value x and two functions, p and f, and returns a value of the same type as x. The p function is a predicate, taking the value x and returning a +Boolean+ b. The f function also takes the value x and returns a new value of the same type. Your "conditional" function should only invoke the function f(x) if p(x) is true, and otherwise return x. How many type parameters will the "conditional" function require?
7. There is a popular coding interview question I'll call "typesafe", in which the numbers 1 - 100 must be printed one per line. The catch is that multiples of 3 must replace the number with the

word "type", while multiples of 5 must replace the number with the word "safe". Of course, multiples of 15 must print "typesafe".

Use the "conditional" function from exercise 6 to implement this challenge.

Would your solution be shorter if the return type of "conditional" did not match the type of the parameter x? Experiment with an altered version of the "conditional" function that works better with this challenge.

Exercise 6 – Common Collections

1. Create a list of the first 20 odd +Long+ numbers. Can you create this with a for-loop, with the +filter+ operation, and with the +map+ operation?
2. Write a function titled "factors" that takes a number and returns a list of its factors, other than 1 and the number itself. For example, +factors(15)+ should return +List(3, 5)+. Then write a new function that applies "factors" to a list of numbers. Try using the list of +Long+ numbers you generated in exercise 1. For example, executing this function with the +List(9, 11, 13, 15)+ should return +List(3, 3, 5)+, as the factor of 9 is 3 while the factors of 15 are 3 again and 5. This is a good place to use +map+ to check and then +flatMap+
3. Write a function, +first[A](items: List[A], count: Int): List[A]+, that returns the first x number of items in a given list. For example, +first(List('a','t','o'), 2)+ should return +List('a','t')+. You could make this a one-liner by invoking one of the built-in list operations that already performs this task, or (preferably) implement your own solution. Can you do so with a for-loop? With +foldLeft+? With a recursive function that only accessed +head+ and +tail+?
4. Write a function that takes a list of strings and returns the longest string in the list. Can you avoid using mutable variables here? This is an excellent candidate for the list-folding operations we studied. Can you implement this with both +fold+ and +reduce+ ? Would your function be more useful if it took a function parameter that compared two strings and returned the preferred one? How about if this function was applicable to generic lists, ie lists of any type?
5. Write a function that reverses a list. Can you write this as a recursive function? This may be a good place for a +match+ expression.
6. Write a function that takes a +List[String]+ and returns a +(List[String],List[String])+, a tuple of string lists. The first list should be items in the original list that are palindromes (written the same forwards and backwards, like "racecar"). The second list in the tuple should be all of the remaining items from the original list. You can implement this easily with +partition+, but are there other operations you could use instead?
7. The last exercise in this chapter is a multi-part problem. We'll be reading and processing a forecast from the excellent and free OpenWeatherMap API. To read content from the url we'll use the Scala library operation +io.Source.fromURL(url: String)+, which returns an +io.Source+ instance. Then we'll reduce the source to a collection of individual lines using the +getLines.toList+ operation. Here is an example of using +io.Source+ to read content from a url, separate it into lines and return the result as a list of strings.

```
scala> val l: List[String] =
io.Source.fromURL(url).getLines.toList
```

Here is the url we will use to retrieve the weather forecast, in XML format.

```
scala> val url =
"http://api.openweathermap.org/data/2.5/forecast?mode=xml&lat=55&lon=0"
```

Go ahead and read this url into a list of strings. Once you have it, print out the first line to verify you've captured an xml file. The result should look pretty much like this:

```
scala> println( l(0) )
<?xml version="1.0" encoding="utf-8"?>
```

If you don't see an xml header, make sure that your url is correct and your internet connection is up.

Let's begin working with this +List[String]+ containing the xml document.

a) To make doubly sure we have the right content, print out the top 10 lines of the file. This should be a one-liner.
b) The forecast's city's name is there in the first 10 lines. Grab it from the correct line and print out its xml element. Then extract the city name and country code from their xml elements and print them out together (e.g., "Paris, FR"). This is a good place to use regular expressions to extract the text from xml tags.

   If you don't want to use regular expression capturing groups, you could instead use the +replaceAll()+ operation on strings to remove the text surrounding the city name and country name.

c) How many forecast segments are there? What is the shortest expression you can write to count the segments?
d) The "symbol" xml element in each forecast segment includes a description of the weather forecast. Extract this element in the same way you extracted the city name and country code. Try iterating through the forecasts, printing out the description.

   Then create an informal weather report by printing out the weather descriptions over the next 12 hours (not including the xml elements).

e) Let's find out what descriptions are used in this forecast. Print a sorted listing of all of these descriptions in the forecast, with duplicate entries removed.
f) These descriptions may be useful later. Included in the "symbol" xml element is an attribute containing the symbol number. Create a +Map+ from the symbol number to the description. Verify this is accurate by manually accessing symbol values from the forecast and checking that the description matches the xml document.
g) What are the high and low temperatures over the next 24 hours?
h) What is the average temperature in this weather forecast? You can use the "value" attribute in the temperature element to calculate this value.

Exercise 7 – More Collections

1. In the example for +Array+ collections we used the +java.io.File(<path>).listFiles+ operation to return an array of files in the current directory. Write a function that does the same thing for a directory, and converts each entry into its +String+ representation using the +toString+ method. Filter out any dot-files (files which begin with the character '.') and print the rest of the files separated by a semi-colon (';'). Test this out in a directory on your computer that has a significant number of files.
2. Write a function to return the product of two numbers.. that are each specified as a +String+, not a numeric type. Will you support both integers and floating-point numbers? How will you convey if either or both of the inputs are invalid? Can you handle the converted numbers using a +match+ expression? How about with a for-loop?
3. Write a function to safely wrap calls to the JVM library method +System.getProperty(<String>)+, avoiding raised exceptions or null results. +System.getProperty(<String>)+ returns a JVM environment property value given the property's name. For example, +System.getProperty("java.home")+ will return the path to the currently running Java instance while +System.getProperty("user.timezone")+ returns the time zone property from the operating system. This method can be dangerous to use, however, since it may throw exceptions or return +null+ for invalid inputs. Try invoking +System.getProperty("")+ or +System.getProperty("blah")+ from the Scala REPL to see how it responds.

   Experienced Scala developers build their own libraries of functions that wrap unsafe code with Scala's monadic collections. Your function should simply pass its input to the method and ensure that exceptions and null values are safely handled and filtered. Call your function with the example property names above, including the valid and invalid ones, to verify that it never raises exceptions or returns null results.

Exercise 8 – Classes

1. We're working on a gaming site, and need to track popular consoles like the Xbox Two and Playstation Five (I'm planning for the future here).
   a) Create a console class that can track the make, model, debut date, wifi type, physical media formats supported, and maximum video resolution. Override the default +toString+ method to print a reasonably-sized description of the instance (< 120 chars).
      - The debut date (or launch date) should be an instance of +java.util.Date+
      - Keep the wifi type (b/g, b/g/n, etc) field optional, in case some consoles don't have wifi.
      - The physical media formats should be a list. Is a +String+ the best bet here, or an +Int+ that matches a constant value?
      - The maximum video resolution should be in a format that would make it possible to sort consoles in order of greatest number of pixels.
   b) Test our your new console class by writing a new class that creates 4 instances of this console class. All of the instances should have reasonably accurate values.

c) Now its time for games. Create a game class that includes the name, maker, and a list of consoles it supports, plus an "isSupported" method that returns true if a given console is supported.
d) Test out this game class by generating a list of games, each containing one or more instances of consoles. Can you convert this list to a lookup table of consoles to a list of supported games? How about a function that prints a list of the games, sorted first by maker and then by game name?

2. For a change of pace, let's create a directory listing class. The constructor fields should be the full path to the directory and a predicate function that takes a String (the file name) and returns true if the file should be included. The method "list" should then list the files in the directory.

   To implement this, create an instance of +java.io.File+ and use its +listFiles(filter: FilenameFilter)+ to list files that match the given filter. You'll find javadocs for this method and for the +java.io.FilenameFilter+ class, but you will need to figure out how this would be called from Scala. You should pass in the +FilenameFilter+ argument as an anonymous class.

   - Is there any part of this class that would work well as a lazy value?
   - Would it make sense to store the anonmyous subclass of +java.io.FilenameFilter+ as a lazy val?
   - How about the filtered directory listing?

Exercise 9- Object, Case Classes and Traits

1. Let's cover how to write a unit test in Scala with the ScalaTest framework. This exercise will consist of adding a test to the IDE, executing it, and verifying its successful outcome. If you're already familiar with executing tests in an IDE this should be a fairly simple exercise. To better understand the ScalaTest framework, I recommend that you take a break from this exercise and browse the official documentation at http://www.scalatest.org/[the ScalaTest web site].

   Install plugin sbt in IntelliJ: ctrl + shift + A / plugins / browse repository /sbt / install plugin

   Add next line to build.sbt file

```
libraryDependencies += "org.scalatest" % "scalatest_2.10" % "2.1.3" % "test"
```

   ctrl + shift + A / sbt / and in sbt panel select current project and d click reload button in order to update dependencies (it can take a while)

   We'll start with the "HtmlUtils" object (see <<objects_section>>). Create a new Scala class by right-clicking on the "src/main/scala" directory in the IDE and selecting *New -> Scala Class*. Type the name, "HtmlUtils", and set the type to an object. Replace the skeleton object with the following source:

```
object HtmlUtils {
  def removeMarkup(input: String) = {
    input
      .replaceAll("""</?\w[^>]*>""","")
      .replaceAll("<.*>","")
  }
}
```

The new "HtmlUtils.scala" file should be located in "src/main/scala", the root directory for source code in our project. Now add a new "HtmlUtilsSpec" class under "src/test/scala", creating the directory if necessary. Both SBT and IntelliJ will look for tests in this directory, a counterpart to the main "src/main/scala" directory. Add the following source to the "HtmlUtilsSpec.scala" file.

```
import org.scalatest._
class HtmlUtilsSpec extends FlatSpec with ShouldMatchers {
  "The Html Utils object" should "remove single elements" in {
    HtmlUtils.removeMarkup("<br/>") should equal("")
  }
  it should "remove paired elements" in {
    HtmlUtils.removeMarkup("<b>Hi</b>") should equal("Hi")
  }
  it should "have no effect on empty strings" in {
    val empty = true
    HtmlUtils.removeMarkup("").isEmpty should be(empty)
  }
}
```

We're only using the +FlatSpec+ and +ShouldMatchers+ types from this package, but will import everything so we can easily add additional test utilities in the future (such as "OptionValues", a favorite of mine). The class +FlatSpec+ is one of several different test types you can choose from, modeled after Ruby's *RSpec*. +ShouldMatchers+ adds the +should+ and +be+ operators to your test, creating a domain-specific language that can help make your tests more readable.

The first test starts off a bit differently from the other tests. With the +FlatSpec+, the first test in a file should start with a textual description of what you are testing in this file. Later tests will use the +it+ keyword to refer to this description. This helps to create highly readable test reports.

In the test body, the +equal+ operator ensures that the value preceding +should+ is equal to its argument, here the empty string +""+. If not equal, it will cause the test to fail and exit immediately. Likewise, the +be+ operator fails the test if the value before +should+ isn't the same instance, useful for comparing global instances like +true+, +Nil+, and +None+.

Before running the test, open the IntelliJ *Plugins* preference panel under *Preferences* and ensure that the "jUnit" plugin is installed. The plugin will ensure that your test results will be easily viewable and browseable.

Once you have added the test to your project, go ahead and compile it in the IDE. If it doesn't compile, or it otherwise complains about the lack of a "ScalaTest" package, make sure your build script has the ScalaTest dependency and that you can view it in the "External Libraries" section of the "Project" view in IntelliJ.

Now we'll run it. Right click on the test class's name, "HtmlUtilsSpec", and choose *Run 'HtmlUtilsSpec'*. Executing the test will take no more than a few seconds, and if you entered the test and original application in correctly they will all be successful.

Let's conclude this exercise with an actual exercise for you to implement: add additional tests to our "HtmlUtilsSpec" test class. Are there there any features areas that aren't yet tested? Are all valid HTML markup possibilities supported?

There's also the question of whether Javascript contained within "script" tags should be stripped or appear along with the rest of the text. Consider this a bug in the original version of "HtmlUtils". Add a test to verify that the Javascript text will be tripped out and then run the test. When it fails, fix "HtmlUtils" and re-run the test to verify it has been fixed.

Congratulations, you are now writing tests in Scala! Remember to keep writing tests as you work through the rest of the exercises in this book, using them to assert how your solutions should work and to catch any (unforseeable!) bugs in them.

2. Let's work on a different example from this chapter. Create a new Scala trait titled "SafeStringUtils" and add the following source:

```
trait SafeStringUtils {
  // Returns a trimmed version of the string wrapped in an Option,
  // or None if the trimmed string is empty.
  def trimToNone(s: String): Option[String] = {
    Option(s) map(_.trim) filterNot(_.isEmpty)
  }
}
```

Verify that the trait compiles in the IDE. If it all works, complete the following steps:

a) Create an object version of the trait.
b) Create a test class, "SafeStringUtilsSpec", to test the "SafeStringUtils.trimToNone()" method. Verify that it trims strings and safely handles null and empty strings. You should have 3-5 separate tests in your test class. Run the test class and verify it completes successfully.
c) Add a method that safely converts a string to an integer, without throwing an error if the string is unparseable. Write and execute tests for valid and invalid input. What are the most appropriate monadic collections to use in this function?

d) Add a method that returns a randomly generated string of the given size, limited to only upper- and lower-case letters. Write and execute tests that verify the correct contents are return and that invalid input is handled. Are there any appropriate monadic collections to use in this function?

SOLUTIONS

Exercise 1 – Basic operations in RPEL / Worksheet

1) An alternate method to using +println()+ is the +print()+ command, which prints the given string without appending a newline.

```
scala> print("Please remain on the line")
Please remain on the line
```

When you're working in the REPL (Read-Eval-Print-Loop), you can also just enter your string on its own line. The "Print" functionality in the REPL will just print the given value.

```
scala> "Hello, REPL"
res0: String = Hello, REPL
```

2) The standard math operators and precedence used in other mainstream programming languages also work in Scala. Thus you can be assured that multiplication and division operators take precedence over addition and subtraction operators.

```
scala> 22.5 * 9 / 5 + 32
res0: Double = 72.5
```

3) Using parentheses around the subtraction, which needs to occur before the multiplication and division, will give it highest precedence.

```
scala> 22.5 * 9 / 5 + 32
res0: Double = 72.5
scala> (res0 - 32) * 5 / 9
res1: Double = 22.5
```

4) Here's my "Hello.scala" file:

```
println("Greetings from 'Hello.scala'")
```

To load a file from the same directory, type ":load" and the file name, without quotes.

```
scala> :load Hello.scala
Loading Hello.scala...
Greetings from 'Hello.scala'
```

5) Here is the contents of "Hello.scala", pasted from the clipboard into the REPL's paste mode. As noted, you'll need to press "<control>-d" to exit out of the paste mode and trigger the evaluation of the contents. Make sure to use ":paste" here:

```
scala> :paste
// Entering paste mode (ctrl-D to finish)
println("Greetings from 'Hello.scala'")
// Exiting paste mode, now interpreting.
Greetings from 'Hello.scala'
```

Exercise 2 – Literals, Values, Variables & Types

1) Let's pick a centigrade temperature at random, say 7 degrees. For optimal accuracy we'll use 5.0, a floating-point number, to ensure the remainder of the division is preserved.

```
scala> val celsius = 7
celsius: Int = 7
scala> val fahr1 = celsius * 9
fahr1: Int = 63
scala> val fahr2 = fahr1 / 5.0
fahr2: Double = 12.6
scala> val fahrenheit = fahr2 + 32
fahrenheit: Double = 44.6
```

We now have named values for each step of the calculation, with a final answer of 44.6 degrees fahrenheit.

2) The "5.0" floating-point number value resulted in the final result having the same type. To return an integer, we can switch the one floating-point number to an integer.

```
scala> val fahr2 = fahr1 / 5
fahr2: Int = 12
scala> val fahrenheit = fahr2 + 32
fahrenheit: Int = 44
```

The result, 44, is pretty darn close to the value calculated with a floating-point divisor, 44.6.

3) String interpolation, the act of inserting placeholders for named values, is the right solution here. It will let us use printf formatting to convert the input value down to only two decimal values. The printf format for doing so is "%.2f", which specifies that exactly two of the most significant decimal digits will be printed.

The really hard part is getting the dollar sign that prefixes the amount to print, as it is right up against the dollar sign we'll need to trigger string interpolation. Fortunately a double dollar sign will be recognized as a request to print a literal dollar sign, which we can follow with another dollar sign to begin interpolation.

```
scala> val amount = 2.7255
amount: Double = 2.7255
scala> val s = f"You owe $$$amount%.2f dollars"
s: String = You owe $2.73 dollars
```

4) This was a deliberately easy question, written with the goal to prod you to look for ways to reduce unnecessary code (even in the simple value assignments given here). A correct answer is that both the explicit types and the lengthy comparison can be simplified, requiring one to check both sides of the equation for code that can be removed.

```
scala> val flag = false
flag: Boolean = false
scala> val result = !flag
result: Boolean = true
```

5) You'll need a host of conversion functions for this one! If you're unsure of the exact function name, you can find them in the Scaladoc pages for the source type. You can also try the REPL's tab-based. code completion feature. Go ahead and type "128." followed by a tab to see the available functions for that integer.

```
scala> val orig = 128
orig: Int = 128
scala> val c: Char = orig.toChar
c: Char = 
scala> val s: String = c.toString
s: String = 
scala> val d: Double = s(0).toDouble
d: Double = 128.0
scala> val i: Int = d.toInt
i: Int = 128
```

Exercise 3 – Expressions and conditionals

1) You could use a pattern guard here and check if the size of the string is zero or non-zero. However, it's easier to just match based on an empty string.

```
scala> val name = "Dresden"
name: String = Dresden
scala> name match {
     |   case "" => "n/a"
     |   case n => n
     | }
res0: String = Dresden
```

2) First, here's the if..else block (specifically an if..else-if..else block).

```
scala> val amount = 2.143
amount: Double = 2.143
scala> if (amount > 0) "greater" else if (amount < 0) "lesser" else "same"
res0: String = greater
Second, the match expression that achieves the same purpose.
```

```
scala> amount match {
     |   case x if x > 0 => "greater"
     |   case x if x < 0 => "lesser"
     |   case x => "same"
     | }
res1: String = greater
```

Which solution do you prefer? I find the match expression a bit more readable but the if..else block is certainly more compact.

3) Returning an appropriate hexadecimal version of one of the three possible color names is easy enough. A good error handling solution should check if an unexpected color name was given and log the error, while still returning a useful color.

```
scala> val color = "magenta"
color: String = magenta
scala> color match {
     |   case "cyan" => "00ffff"
     |   case "magenta" => "00ff00"
     |   case "yellow" => "ffff00"
     |   case x => {
     |     println(s"Didn't expect $x !")
     |     "333333"
     |   }
     | }
res0: String = 00ff00
```

4) Two for-loops, one for the rows and one for the columns, can handle this task.

```
scala> for (i <- 1 to 100 by 5) {
     |   for (j <- i to (i + 4)) { print(s"$j, ") }
     |   println("")
     | }
1, 2, 3, 4, 5,
6, 7, 8, 9, 10,
11, 12, 13, 14, 15,
```

5) There's a number of ways to solve this. Here's a solution that isn't very compact but plainly checks for the greatest factor (15) before checking for 5 and 3.

```
scala> for (i <- 1 to 100) {
     |   i match {
     |     case x if x % 15 == 0 => println("typesafe")
     |     case x if x % 5 == 0 => println("safe")
     |     case x if x % 3 == 0 => println("type")
     |     case x => println(x)
     |   }
```

```
     | }
1
2
type
4
safe
```

6) To make this fit on a single line, I switched to a mutable string which could hold "type" and "safe". It can also be checked for emptiness and thus print the current integer value.

```
scala> for (i <- 1 to 100) { var s = ""; if (i%3==0) s="type"; if (i%5==0) s+="safe";
if(s.isEmpty) s += i; println(s) }
1
2
type
4
safe
type
7
8
```

Exercise 4 – Functions

1) A circle's area is equal to the square of its radius times Pi. A quick and dirty solution of multiplying the radius twice with a reasonable hard coded version of Pi will get us close to the solution.

```
scala> def circleArea(r: Double) = r * r * 3.14159
circleArea: (r: Double)Double
scala> circleArea(8)
res0: Double = 201.06176
```

We can make use of +scala.math+ to handle the square of the radius and a more approximate version of Pi, if more accuracy is desired.

```
scala> def circleArea2(r: Double) = math.pow(r,2) * math.Pi
circleArea2: (r: Double)Double
scala> circleArea2(8)
res1: Double = 201.06192982974676
```

2) Providing the radius as a double number encoded as a string makes the function a little more complex. Not only will it be necessary to try to convert the string, but we'll have to figure out an acceptable result when the string is empty.

As this function takes up more than a single line, I'll add an explicit return type for clarity and also wrap the function body in curly braces. I'll also go ahead with the number zero as the return value when the input radius is empty.

A more advanced version of this function would also handle completely invalid radius values in the input string, such as "Hello" or "123abc". However, this function won't include that case, as it requires exception handling which we haven't yet covered.

```
scala> def circleArea3(r: String): Double = {
     |   r.isEmpty match {
     |     case true => 0
     |     case false => math.pow(r.toDouble,2) * math.Pi
     |   }
     | }
circleArea3: (r: String)Double
scala> circleArea3("8")
res2: Double = 201.06192982974676
scala> circleArea3("")
res3: Double = 0.0
```

3) Recursive functions that have no return value are rather easy to write, as the result of invoking the function is ignored. The "@annotation.tailrec" will work easily as there is no return value which may be saved locally; we can simply exit immediately after the invocation.

```
scala> @annotation.tailrec
     | def fives(cur: Int, max: Int): Unit = {
     |   if (cur <= max) {
     |     println(cur)
     |     fives(cur + 5, max)
     |   }
     | }
fives: (cur: Int, max: Int)Unit
scala> fives(0, 20)
0
5
10
15
20
```

4) The common format for using milliseconds to describe a time is using the value zero to indicate 12:00 am GMT on January 1st, 1970. Also known as "epoch time", this format is supported by the JVM library classes +java.util.Date+ and +java.util.Calendar+, among others.

However, we can write this function without making use of the JVM classes. Here's a version that expects the epoch time in milliseconds specified as a long value.

Using a long value (64 bits) ensures that the code will work for millenia.

The first step is to reduce the milliseconds down to a more reasonable seconds value. The days is computed by dividing the seconds by the seconds in a day, while the remaining values are computed by modules of the nearest larger time unit.

```
scala> def descTime(epochMs: Long) = {
     |   val secs = epochMs / 1000
     |   val days = secs / 86400
     |   val hours = (secs % 86400) / 3600
     |   val minutes = (secs % 3600) / 60
     |   val seconds = secs % 60
     |   s"$days days, $hours hours, $minutes minutes, $seconds seconds"
     | }
descTime: (epochMs: Long)String
scala> descTime(123456789000L)
res20: String = 1428 days, 21 hours, 33 minutes, 9 seconds
```

5) Here's the easy way, using the +math.pow+ library function.

```
scala> def pow(x: Double, y: Double): Double = math.pow(x,y)
pow: (x: Double, y: Double)Double
scala> pow(2, 5)
res0: Double = 32.0
```

Using a custom calculation, here's a version that stores the intermediate results in a variable. It's not especially robust, since it doesn't do validation on the exponent (eg, checking if a negative number was given).

```
scala> def pow(x: Double, y: Int): Double = {
     |   var p = 1.0; for (i <- 1 to y) p *= x; p
     | }
pow: (x: Double, y: Int)Double
scala> pow(2, 5)
res1: Double = 32.0
```

Hmm but using immutable data is more of a challenge. However, it looks like we can rewrite this version using tail recursion, where the final line in the function is a value passed to the next recursive call. No actual mutable data will be necessary.

```
scala> @annotation.tailrec
     | def pow(x: Double, y: Int, accum: Double = 1): Double = {
     |   if (y < 1) accum
     |   else pow(x, y - 1, accum * x)
     | }
pow: (x: Double, y: Int, accum: Double)Double
scala> pow(2, 5)
res2: Double = 32.0
```

6) A 2-sized tuple is a good parameter type for specifying a point in 2d space. It's components are unnamed, but we can imagine the first component as corresponding to x (the horizontal value) and its second as corresponding to y (the vertical value).

```
scala> def offset(src: (Int, Int), dest: (Int, Int)): (Int, Int) = {
     |   (dest._1 - src._1, dest._2 - src._2)
     | }
offset: (src: (Int, Int), dest: (Int, Int))(Int, Int)
scala> offset( (4, 9), (122, 27) )
res0: (Int, Int) = (118,18)
```

7) This question should have stirred you to consider how you may change the type signature of a given tuple, and how changing the ordering of types in a tuple affects the return type of a function. Ultimately the resulting function must be non-type-safe, as the caller cannot expect to know in which order the types and values of a given tuple will be returned. Hopefully this solution is useful for learning the language, as it isn't really a recommended solution for actual coding (due to the lack of type safety).

```
scala> def intFirst[A,B](t: (A,B)): (Any,Any) = {
     |   def isInt(x: Any) = x.isInstanceOf[Int]
     |   (isInt(t._1), isInt(t._2)) match {
     |     case (false, true) => (t._2, t._1)
     |     case _ => t
     |   }
     | }
intFirst: [A, B](t: (A, B))(Any, Any)
scala> intFirst( ('a', 2) )
res0: (Any, Any) = (2,a)
scala> intFirst( (1, false) )
res1: (Any, Any) = (1,false)
scala> intFirst( (1, 4) )
res2: (Any, Any) = (1,4)
```

If the second item, but not the first item, is an integer than this function returns the tuple with the values switched. Otherwise it just returns the original tuple. Due to the reordering of the values, there is no way to alert the caller of the function what the types of each value will be. Thus the function is forced to return a tuple of the type +(Any, Any)+.

8) The phrase "tuples of all possible types" indicates that we'll need to have type parameters for the input tuple. We can reuse them for the return tuple's tupe, since the return tuple should keep the same ordering of input types.

```
scala> def stringify[A,B,C](t: (A,B,C)): (A,String,B,String,C,String) = {
     |   (t._1, t._1.toString, t._2, t._2.toString, t._3, t._3.toString)
     | }
stringify: [A, B, C](t: (A, B, C))(A, String, B, String, C, String)
```

Let's invoke it and store the result in a value of an explicit type. The return type is based on interpolating the input tuple's types with +String+ types so this should be easy to figure out.

```
scala> val t: (Int,String,Char,String,Boolean,String) = stringify( (1, 'c', true) )
t: (Int, String, Char, String, Boolean, String) = (1,1,c,c,true,true)
```

Exercise 5 – First-Class Functions

1) The answer to the first part of the question is to write a _lambda expression_, i.e. a function literal. If you can write this answer then you'll understand lambda expressions and function literals.

```
scala> val max = (x: Int, y: Int) => if (x > y) x else y
max: (Int, Int) => Int = <function2>
scala> max(23, 32)
res0: Int = 32
```

The second part of the question should give you pause. After all, how can you use a two-input function literal to compare three numbers? Hopefully you were able to figure out something like this:

```
scala> def pickOne(t: (Int, Int, Int), cmp: (Int, Int) => Int): Int = {
     |   cmp(t._1, cmp(t._2, t._3))
     | }
pickOne: (t: (Int, Int, Int), cmp: (Int, Int) => Int)Int
scala> pickOne( (14, 7, 9), max )
res1: Int = 14
```

2) We can start with the max function or exercise 1 that takes two input numbers. First, here's the "max()" function literal invoked with two random numbers.

```
scala> max(util.Random.nextInt, util.Random.nextInt)
res0: Int = 1436693791
To keep our code brief and avoid retyping "util.Random.nextInt", let's add a wrapper
function.
scala> def nextInt = util.Random.nextInt
nextInt: Int
scala> max(nextInt, nextInt)
res1: Int = 1955118075
Okay, here's the "pickOne()" function invoked with three random numbers and the "max()"
comparison function literal.
scala> val t = (nextInt, nextInt, nextInt)
t: (Int, Int, Int) = (-233084145,1722379081,683222211)
scala> pickOne(t, max)
res2: Int = 1722379081
Let's try some varieties of the comparison functions. Since we're passing in function
literals, this should be straightforward.
```

```
scala> val t = (nextInt, nextInt, nextInt)
t: (Int, Int, Int) = (433839089,-1002398428,2101699998)
scala> pickOne(t, (x, y) => if (x < y) x else y)
res3: Int = -1002398428
scala> val t = (nextInt, nextInt, nextInt)
t: (Int, Int, Int) = (913142586,1268532435,94040930)
scala> pickOne(t, (x, y) => x)
res4: Int = 913142586
```

3) Here's our first higher-order function in the exercises which returns another function. While the exercise's description is necessarily complicated, talking about the higher-order function AND the function it returns, the answer is fortunately brief.

```
scala> def multy(x: Int) = (y: Int) => x * y
multy: (x: Int)Int => Int
Let's try it out!
scala> val tripler = multy(3)
tripler: Int => Int = <function1>
scala> tripler(10)
res4: Int = 30
```

4) The purpose of this exercise is to help you gain familiarity with Scala's higher-order functions in the field. Here's a helper function that may appear mysterious at first (especially its fabulous display of punctionation and single-letter type names), but after careful reading will become understandable.

Would it be easier to read this function if it was bound to a specific type, and the two parameter lists combined? Here's the "fzero()" method simplified and redefined with an explicit +String+ type.

```
scala> def fzero(x: String, f: String => Unit): String = { f(x); x }
fzero: (x: String, f: String => Unit)String
scala> fzero("Hello", s => println(s.reverse))
olleH
res0: String = Hello
```

The "fzero()" function, in specialized and simplified form here, executes the given function parameter on a given value and then returns the original value. The original version in the exercise adds type parameters and currying support.

Here's an example of invoking the original function with a specific type and separate parameter lists.

```
scala> fzero[Boolean](false) { b => println(s"b was $b") }
b was false
res1: Boolean = false
```

5) If you haven't figured it out, try the example code in the REPL.

```
scala> def square(m: Double) = m * m
```

```
square: (m: Double)Double

scala> val sq = square

<console>:8: error: missing arguments for method square;

follow this method with `_' if you want to treat it as a partially applied function

       val sq = square

Indeed, using a wildcard to assign a function would help make "sq" into a function value.
Or, alternatively, give it the explicit type of a function value.

scala> val sq1 = square _

sq1: Double => Double = <function1>

scala> sq1(4.0)

res0: Double = 16.0

scala> val sq: Double => Double = square

sq: Double => Double = <function1>

scala> sq(5.0)

res2: Double = 25.0
```

6) Another example of the question being far longer than the answer. Here's our "conditional()" function with a value parameter and two function parameters. We'll try it out with a predicate that will fail with the given input, verifying that the original input is returned.

```
scala> def conditional[A](x: A, p: A => Boolean, f: A => A): A = {
     |   if (p(x)) f(x) else x
     | }
conditional: [A](x: A, p: A => Boolean, f: A => A)A

scala> val a = conditional[String]("yo", _.size > 4, _.reverse)
a: String = yo
```

7) Here's one example of using the "conditional" function to produce the "typesafe" sequence. In this one, the predicate function checks if the current iteration matches multiples of either 3 or 5, whie the other function prints a message and returns zero.

```
scala> for (i <- 1 to 100) {
     |   val a1 = conditional[Int](i, _ % 3 == 0, x => { print("type"); 0 })
     |   val a2 = conditional[Int](i, _ % 5 == 0, x => { print("safe"); 0 })
     |   if (a1 > 0 && a2 > 0) print(i)
     |   println("")
     | }
1
2
type
4
safe
...
```

This isn't a great solution, however. The function literals are writing to the console, a side-effect that one shouldn't have to see in good functional programming exercise answers.

To fix this, let's rework the "conditional" function so it can take a number and return a string. If the given predicate is false, it should return an empty string. We can then use the output strings to create a single final result string for printing (or for re.turning from a function)

```
scala> def conditional[A](x: A, p: A => Boolean, f: A => String): String = {
     |   if (p(x)) f(x) else ""
     | }
conditional: [A](x: A, p: A => Boolean, f: A => String)String
scala> for (i <- 1 to 100) {
     |   val a1 = conditional[Int](i, _ % 3 == 0, _ => "type")
     |   val a2 = conditional[Int](i, _ % 5 == 0, _ => "safe")
     |   val a3 = conditional[Int](i, _ % 3 > 0 && i % 5 > 0, x => s"$x")
     |   println(a1 + a2 + a3)
     | }
1
2
type
4
safe
```

Now the only side effect is in the single "println" statement at the end of each loop. How about we do one more iteration to clean it up, converting the loop's code into a function that takes a single number and returns the correct response? This way we can separate the loop and printout code from the core logic.

```
scala> def typeSafely(i: Int): String = {
     |   val a1 = conditional[Int](i, _ % 3 == 0, _ => "type")
     |   val a2 = conditional[Int](i, _ % 5 == 0, _ => "safe")
     |   val a3 = conditional[Int](i, _ % 3 > 0 && i % 5 > 0, x => s"$x")
     |   a1 + a2 + a3
     | }
typeSafely: (i: Int)String
scala> val sequence = 1 to 100 map typeSafely
sequence: scala.collection.immutable.IndexedSeq[String] = Vector(1, 2, type, 4, safe, type,
7, 8, ...)
scala> println(sequence.mkString("\n"))
1
2
type
4
safe
type
...
```

Our sequence generation, handled with the updated "conditional" function, is now packaged in a reusable function.

Exercise 6 – Common Collections

1) There's many ways to generate this list with the collections library, too many to list here. Here are some examples of generating it with a for-loop, filtering, and mapping.

```
scala> for (i <- 0L to 9L; j = i * 2 + 1) yield j
res0: scala.collection.immutable.IndexedSeq[Long] = Vector(1, 3, 5, 7, 9,
11, 13, 15, 17, 19)
scala> 0L to 20L filter (_ % 2 == 1)
res1: scala.collection.immutable.IndexedSeq[Long] = Vector(1, 3, 5, 7, 9, 11, 13, 15, 17,
19)
scala> 0L to 9L map (_ * 2 + 1)
res2: scala.collection.immutable.IndexedSeq[Long] = Vector(1, 3, 5, 7, 9, 11, 13, 15, 17,
19)
```

2) Here's the "factors" function.

```
scala> def factors(x: Int) = { 2 to (x-1) filter (x % _ == 0) }
factors: (x: Int)scala.collection.immutable.IndexedSeq[Int]
```

We could use +map+ to map each item in the list to a new factor list and then +flatten+ the list of lists into a single list. Or just use +flatMap+ once.

```
scala> def uniqueFactors(l: Seq[Int]) = l flatMap factors
uniqueFactors: (l: Seq[Int])Seq[Int]
scala> uniqueFactors(List(9, 11, 13, 15))
res13: Seq[Int] = List(3, 3, 5)
```

3) Although using a built-in list operation won't help you build out your Scala collection skills, it does let you solve the problem with a simple one-liner. And this is often the solution you will choose first while writing your own applications.

```
scala> val chars = ('a' to 'f').toList
chars: List[Char] = List(a, b, c, d, e, f)
scala> def first[A](items: List[A], count: Int): List[A] = items take count
first: [A](items: List[A], count: Int)List[A]
scala> first(chars, 3)
res0: List[Char] = List(a, b, c)
```

Now that we have the easy solution working, let's try writing out our own solutions. Here's one that uses a for-loop.

```
scala> def first[A](items: List[A], count: Int): List[A] = {
     |   val l = for (i <- 0 until count) yield items(i)
     |   l.toList
     | }
```

```
first: [A](items: List[A], count: Int)List[A]
scala> first(chars, 3)
res1: List[Char] = List(a, b, c)
```

This works, but it's performance is going to be terrible with long non-indexed collections such as linked lists.

Here's a version using +foldLeft+, which then relies on +reverse+ to return the items in their original order. Starting with an empty list, this function adds each element in the input list to its accumulated list.

```
scala> def first[A](items: List[A], count: Int): List[A] = {
     |   items.foldLeft[List[A]](Nil) { (a: List[A], i: A) =>
     |     if (a.size >= count) a else i :: a
     |   }.reverse
     | }
first: [A](items: List[A], count: Int)List[A]
scala> first(chars, 3)
res2: List[Char] = List(a, b, c)
```

Finally, let's solve this with an old-fashioned recursive function. Limited to accessing the +head+ and +tail+ components, this uses non-tail recursion to accumulate a list starting with the final recursive call.

```
scala> def first[A](items: List[A], count: Int): List[A] = {
     |   if (count > 0 && items.tail != Nil) items.head :: first(items.tail, count - 1)
     |   else Nil
     | }
first: [A](items: List[A], count: Int)List[A]
scala> first(chars, 3)
res3: List[Char] = List(a, b, c)
```

4) Again, let's start with a short and simple implementation that takes advantage of the collections library.

```
scala> def longest(l: List[String]): String = names.sortBy(0 - _.size).head
longest: (l: List[String])String
scala> val names = List("Harry", "Hermione", "Ron", "Snape")
names: List[String] = List(Harry, Hermione, Ron, Snape)
scala> longest(names)
res0: String = Hermione
```

Using +fold+ and +reduce+ is a natural fit for this function, as we're reducing a list down to one of its elements.

```
scala> def longest(l: List[String]): String = {
     |   names.fold("")((a,i) => if (a.size < i.size) i else a)
     | }
```

```
longest: (l: List[String])String
scala> longest(names)
res1: String = Hermione
scala> def longest(l: List[String]): String = {
     |   names.reduce((a,i) => if (a.size < i.size) i else a)
     | }
longest: (l: List[String])String
scala> longest(names)
res2: String = Hermione
```

Now, if we're going to make this function applicable to any kind of list we will need a new way to compare elements. Here's a solution that takes a plain comparison function and uses it to reduce the list. We can invoke it with a function literal that returns the string with the longer size.

```
scala> def greatest[A](l: List[A], max: (A,A) => A): A = {
     |   l reduce ((a,i) => max(a,i))
     | }
greatest: [A](l: List[A], max: (A, A) => A)A
scala> greatest[String](names, (x,y) => if (x.size > y.size) x else y)
res3: String = Hermione
```

5) We have already made use of "reverse" in answers to previous exercise questions, so it's a good idea to try and provide it ourselves. This version uses two list parameters, one of which gets initialized to +Nil+, to reverse the elements one at a time.

```
scala> def reverse[A](src: List[A], dest: List[A] = Nil): List[A] = {
     |   if (src == Nil) dest else reverse(src.tail, src.head :: dest)
     | }
reverse: [A](src: List[A], dest: List[A])List[A]
scala> val names = List("Harry", "Hermione", "Ron", "Snape")
names: List[String] = List(Harry, Hermione, Ron, Snape)
scala> reverse(names)
res0: List[String] = List(Snape, Ron, Hermione, Harry)
def reverse2[A](src: List[A]): List[A] =  src match {
  case Nil => Nil
  case head::tail => reverse2 (tail) ::: List (head)
}
reverse2 (names)
```

6) First, the easy way! Let's invoke +partition+ with a function that returns true if the string is a palindrome.

```
scala> def splitPallies(l: List[String]) = l partition (s => s == s.reverse)
splitPallies: (l: List[String])(List[String], List[String])
scala> val pallies = List("Hi", "otto", "yo", "racecar")
pallies: List[String] = List(Hi, otto, yo, racecar)
```

```
scala> splitPallies(pallies)
res0: (List[String], List[String]) = (List(otto, racecar),List(Hi, yo))
```

If you think about it, +partition+ is a kind of list reduction function. It reduces a list to a single tuple, which happens to contain two lists. Let's use +foldLeft+ to reduce the list down to the tuple.

```
scala> def splitPallies(l: List[String]) = {
     |   l.foldLeft((List[String](),List[String]())) { (a, i) =>
     |     if (i == i.reverse) (i :: a._1, a._2) else (a._1, i :: a._2)
     |   }
     | }
splitPallies: (l: List[String])(List[String], List[String])
scala> splitPallies(pallies)
res0: (List[String], List[String]) = (List(racecar, otto),List(yo, Hi))
```

7)

a) We'll load the url and verify the first 10 lines here.

```
scala> val url = "http://api.openweathermap.org/data/2.5/forecast?mode=xml&lat=55&lon=0"
url: String = http://api.openweathermap.org/data/2.5/forecast?mode=xml&lat=55&lon=0
scala> val l: List[String] = io.Source.fromURL(url).getLines.toList
l: List[String] = List(<?xml version="1.0" encoding="utf-8"?>, <weatherdata>, "
<location>", "    <name>Whitby</name>", "    <type/>", "    <country>GB</country>", "
<timezone/>", "    <location altitude="0" latitude="54.48774" longitude="-0.61498"
geobase="geonames" geobaseid="0"/>", "  </location>", "  <credit/>", "  <meta>", "
<lastupdate/>", "    <calctime>1.067</calctime>", "    <nextupdate/>", "  </meta>", "  <sun
rise="2014-12-26T08:24:05" set="2014-12-26T15:42:23"/>", "  <forecast>", "    <time
from="2014-12-26T21:00:00" to="2014-12-27T00:00:00">", "      <symbol number="500"
name="light rain" var="10n"/>", "      <precipitation value="0.5" unit="3h" type="rain"/>",
"      <windDirection deg="175.502" code="S" name="South"/>", "      <windSpeed mps="4.46"
name="Gentle Bre...
scala> l take 10
res0: List[String] = List(<?xml version="1.0" encoding="utf-8"?>, <weatherdata>, "
<location>", "    <name>Whitby</name>", "    <type/>", "    <country>GB</country>", "
<timezone/>", "    <location altitude="0" latitude="54.48774" longitude="-0.61498"
geobase="geonames" geobaseid="0"/>", "  </location>", "  <credit/>")
```

b) To make parsing easier, trim that white space from each line. Also, a function to retrieve the child text of a simple xml block makes the job easier.

```
scala> val k = l map (_.trim)
k: List[String] = List(<?xml version="1.0" encoding="utf-8"?>, ...
scala> def getChild(tag: String) = k filter (_ contains s"<$tag>") mkString ""
replaceAll(".*>(\\w+)</.*","$1")
getChild: (tag: String)String
scala> val cityName = getChild("name")
cityName: String = Whitby
scala> val countryCode = getChild("country")
countryCode: String = GB
```

c) Measuring the number of lines with the end segment "</time>" seems like a good way to do this.

```
scala> val segments = l.filter(_ contains "</time>").size
segments: Int = 41
```

d) To get the weather description, we'll need to grab the "<symbol>" lines and retrieve the contents of the "name" field. I'm writing this as a reusable function because, yes, we'll be reusing it!

```
scala> def attribute(tag: String, attr: String) = {
     |   k.filter(_ contains s"<$tag")
     |     .filter(_ contains s"$attr=")
     |     .map { s => s.replaceAll(s""".*$attr="([^"]+)".*""", "$1") }
     | }
attribute: (tag: String, attr: String)List[String]
scala> val names = attribute("symbol", "name")
names: List[String] = List(light rain, light rain, light rain, overcast clouds, ...
```

Looking at the full weather feed, you can see that each time segment covers three hours. To do a 12-hour forecast all we need are the first four segments. Plus, it would be nice to include the start time of each period in the forecast.

It's a good thing we have an "attribute" function. First, let's use it to retrieve a tuple of the time and description for the next 12 hours. Then we'll print it out with minor improvements to the time.

```
scala> val forecast = attribute("time", "from") zip attribute("symbol", "name") take 4
forecast: List[(String, String)] = List((2014-12-26T21:00:00,light rain), (2014-12-
27T00:00:00,light rain), (2014-12-27T03:00:00,light rain), (2014-12-27T06:00:00,overcast
clouds))
scala> {
     |   println("12 hour forecast")
     |   forecast foreach { case (time, desc) =>
     |     val when = time.replaceAll("""T(\d+).*""","""" at $100""")
     |     println(s"$when | $desc")
     |   }
     | }
12 hour forecast
2014-12-26 at 2100 | light rain
2014-12-27 at 0000 | light rain
2014-12-27 at 0300 | light rain
2014-12-27 at 0600 | overcast clouds
```

e) Easy enough with our "attribute" function.

```
scala> val terms = attribute("symbol", "name").distinct.sorted
terms: List[String] = List(broken clouds, few clouds, light rain, overcast clouds,
scattered clouds, sky is clear)
```

f) Let's make another list of tuples, and then turn them into a map. Fortunately there's a helper function called "toMap" which will build one from a list of 2-sized tuples.

```
scala> val symbolsToDescriptions = attribute("symbol", "number") zip attribute("symbol",
"name")

symbolsToDescriptions: List[(String, String)] = List((500,light rain), (500,light rain),
(500,light rain), (804,overcast clouds), (500,light rain), ...

scala> val symMap = symbolsToDescriptions.distinct.map(t => t._1.toInt -> t._2).toMap

symMap: scala.collection.immutable.Map[Int,String] = Map(500 -> light rain, 802 ->
scattered clouds, 804 -> overcast clouds, 800 -> sky is clear, 801 -> few clouds, 803 ->
broken clouds)

scala> println("Light rain? Yup, " + symMap(500))

Light rain? Yup, light rain
```

g) The "max" and "min" functions for numeric lists are really helpful here.

```
scala> val maxC = attribute("temperature", "max").map(_.toDouble).max

maxC: Double = 7.743

scala> val minC = attribute("temperature", "min").map(_.toDouble).min

minC: Double = 3.042
```

h) This time I'll use the "sum" function for numeric lists along with the handy "attribute" function.

```
scala> val temps = attribute("temperature", "value").map(_.toDouble)

temps: List[Double] = List(4.04, 3.96, 5.03, 5.61, 5.92, 5.5, 5.11, 4.93, 6.47,
6.8200000000001, 6.88, 6.51, 6.01, 7.22, 7.22, 6.4, 5.4, 5.06, 5.95, 6.37, 6.808, 7.42,
7.743, 7.701, 7.41, 6.236, 5.508, 5.34, 5.476, 6.644, 6.794, 6.043, 5.701, 5.199, 4.482,
3.755, 3.042, 4.897, 4.507, 3.824, 3.902)

scala> val avgC = temps.sum / temps.size

avgC: Double = 5.7278536585365885
```

Exercise 7 – More Collections

1) We'll start with a function that returns a list of the names of files in the given directory (cleansed of the "./" prefix). Then we'll filter out the dot files and print the rest as a semicolon-delimited string.

```
scala> def listFiles(path: String): List[String] = {
     |   val files = new java.io.File(path).listFiles.toList
     |   files.map( _.toString.replaceFirst("./","") )
     | }

listFiles: (path: String)List[String]

scala> val files = listFiles(".")

files: List[String] = List(.DS_Store, .git, .gitignore, .idea, .idea_modules, akka,
atmosphere, auth, commands, common, CONTRIBUTING.markdown, core, CREDITS.md,
crosspaths.sbt, example, fileupload, jetty, json, LICENSE, notes, project, publishing.sbt,
README.markdown, sbt, scalate, scalatest, slf4j, specs2, spring, swagger, swagger-ext,
target, test, version.sbt)

scala> val files = listFiles(".").filterNot(_ startsWith ".")

files: List[String] = List(akka, atmosphere, auth, commands, common, CONTRIBUTING.markdown,
core, CREDITS.md, crosspaths.sbt, example, fileupload, jetty, json, LICENSE, notes,
project, publishing.sbt, README.markdown, sbt, scalate, scalatest, slf4j, specs2, spring,
swagger, swagger-ext, target, test, version.sbt)

scala> println("Found these files: " + files.mkString(";") )
```

```
Found these files:
akka;atmosphere;auth;commands;common;CONTRIBUTING.markdown;core;CREDITS.md;crosspaths.sbt;e
xample;fileupload;jetty;json;LICENSE;notes;project;publishing.sbt;README.markdown;sbt;scala
te;scalatest;slf4j;specs2;spring;swagger;swagger-ext;target;test;version.sbt
```

2) The two number parameters are strings, and so will need to be normalized into numeric values if possible. To start, I'll write a function that will parse the strings into doubles but wrapped with an +Option+ to denote the presence or absence of a value.

```
scala> def toDouble(a: String) = util.Try(a.toDouble).toOption
toDouble: (a: String)Option[Double]
scala> val x = toDouble("a")
x: Option[Double] = None
```

Now that we have a function to handle parsing the input strings, our "product" function can focus on handling the case when both parameters are valid. In this function, the match expression ensures that the product will only be returned when both numbers are present.

```
scala> def product(a: String, b: String): Option[Double] = {
     |   (toDouble(a), toDouble(b)) match {
     |     case (Some(a1), Some(b1)) => Some(a1 * b1)
     |     case _ => None
     |   }
     | }
product: (a: String, b: String)Option[Double]
scala> val x = product("yes", "20")
x: Option[Double] = None
scala> val x = product("99.3", "7")
x: Option[Double] = Some(695.1)
```

Using a for-loop with two options is even more concise than a match expression, as the +None+ type is automatically returned in case the loop exits early.

```
scala> def product(a: String, b: String): Option[Double] = {
     |   for (a1 <- toDouble(a); b1 <- toDouble(b)) yield a1 * b1
     | }
product: (a: String, b: String)Option[Double]
scala> val x = product("11", "1.93")
x: Option[Double] = Some(21.23)
scala> val x = product("true", "")
x: Option[Double] = None
```

3) Let's try out the problem see if we can get back an exception or null value.

```
scala> System.getProperty(null)
java.lang.NullPointerException: key can't be null
  at java.lang.System.checkKey(System.java:829)
  at java.lang.System.getProperty(System.java:705)
```

```
  ... 32 elided
scala> val arch = System.getProperty("os.arch")
arch: String = x86_64
scala> val blarg = System.getProperty("blarg")
blarg: String = null
```

Wrapping the call in a +util.Try+ and then handling the potential null result with +Option+ makes it possible to safely deal with the null results or exceptions.

```
scala> def getProperty(s: String): Option[String] = {
     |   util.Try( System.getProperty(s) ) match {
     |     case util.Success(x) => Option(x)
     |     case util.Failure(ex) => None
     |   }
     | }
getProperty: (s: String)Option[String]
scala> getProperty(null)
res1: Option[String] = None
scala> val arch = getProperty("os.arch")
arch: Option[String] = Some(x86_64)
scala> val blarg = getProperty("blarg")
blarg: Option[String] = None
```

Exercise 8 – Classes

1) Instead of answering this in four separate parts, I'll just give my complete solution for the exercise. It would be difficult to replace the package name and import statement across all the code blocks just to see it included once with the final class.

If you're working from an IDE, copy and paste the contents into a "GameConsole.scala" file and compile. As it only uses a JDK class, it should be compatible with any running IDE project.

```
package gamingsite
import java.util.Date
abstract class MediaFormat
class DvdMediaFormat extends MediaFormat
class BluRayMediaFormat extends MediaFormat
class USBMediaFormat extends MediaFormat
class CartridgeMediaFormat extends MediaFormat
abstract class VideoResolution(pixels: Int)
class HD extends VideoResolution(1280 * 720)
class FHD extends VideoResolution(1920 * 1080)
class QHD extends VideoResolution(2560 * 1440)
class UHD4K extends VideoResolution(3840 * 2160)
/**
```

```
 * A console that can play games built for it with one or more video resolutions.
 */
class GameConsole(val make: String, val model: String, val debut: Date, val wifiType: Option[String],
                  val mediaFormats: List[MediaFormat], val maxVideoResolution: VideoResolution) {
  override def toString = s"GameConsole($make, $model), max video resolution = ${maxVideoResolution.getClass.getName}"
}
class GameConsoleLibrary {
  def strToDate(s: String): Date = java.text.DateFormat.getInstance.parse(s)
  val chanduOne = new GameConsole("Chandu", "One", strToDate("2/12/2007 0:00 AM"), Some("a/b"),
      List(new CartridgeMediaFormat), new HD)
  val aquaTopia = new GameConsole("Aqua", "Topia", strToDate("5/2/2012 0:00 AM"), Some("a/b/g"),
      List(new DvdMediaFormat), new HD)
  val oonaSeven = new GameConsole("Oona", "Seven", strToDate("3/3/2014 0:00 AM"), Some("b/g/n"),
      List(new BluRayMediaFormat, new DvdMediaFormat), new FHD)
  val leoChallenge = new GameConsole("Leonardo", "Challenge", strToDate("12/12/2014 0:00 AM"), Some("g/n"),
      List(new USBMediaFormat), new UHD4K)
}
/**
 * A game developed for one or more game consoles
 */
class Game(val name: String, val maker: String, val consoles: List[GameConsole]) {
  def isSupported(console: GameConsole) = consoles contains console
  override def toString = s"Game($name, by $maker)"
}
class GameShop {
  val consoles = new GameConsoleLibrary()
  val games = List(
    new Game("Elevator Action", "Taito", List(consoles.chanduOne)),
    new Game("Mappy", "Namco", List(consoles.chanduOne, consoles.aquaTopia)),
    new Game("StreetFigher", "Capcom", List(consoles.oonaSeven, consoles.leoChallenge))
  )
  val consoleToGames: Map[GameConsole, List[Game]] = {
    val consoleToGames1: List[(GameConsole, Game)] = games flatMap (g => g.consoles.map(c => c -> g))
    val consoleToGames2: Map[GameConsole, List[(GameConsole, Game)]] = consoleToGames1 groupBy (_._1)
    val consoleToGames3: Map[GameConsole, List[Game]] = consoleToGames2 mapValues(_ map (_._2))
    consoleToGames3
```

```
  }
  def reportGames(): Unit = {
    games sortBy (g => s"${g.maker}_${g.name}") foreach { game =>
      val consoleInfo = game.consoles.map(c => s"${c.make} ${c.model}").mkString(", ")
      println(s"${game.name} by ${game.maker} for $consoleInfo")
    }
  }
}
```

3) Sometimes it's good to have a simpler exercise mixed in with the others. This exercise will get you familiar with working with Java API's, especially the ones that take interface implementations as parameters.

I've marked the file and filter as lazy values, as they are really only required if the "list" function is called. In regular practice I probably would have not added the lazy designation, as the two values are not resource-intensive.

```
import java.io.{FilenameFilter, File}
class DirLister(path: String, f: String => Boolean) {
  lazy val file: File = new File(path)
  lazy val filter = new FilenameFilter {
    override def accept(dir: File, name: String): Boolean = f(name)
  }
  def list: List[String] = file.list(filter).toList
}
```

Exercise 9 - Object, Case Classes and Traits

1) A good use of tests is to test the challenging cases in addition to the "happy path" of simple test cases. I wasn't sure that the html utils function would be able to support html elements that cover more than a single line, but happily they were correctly stripped out.

Here is the "HtmlUtils" object updated to strip out any Javascript code in `<script>` tags. The flag expression "(?s)" enables "DOTALL" mode which allows the regular expression to ignore line boundaries.

```
object HtmlUtils {
  def removeMarkup(input: String) = {
    input
      .replaceAll("(?s)<script.*</script>", "")
      .replaceAll("""</?\w[^>]*>""","")
      .replaceAll("<.*>","")
  }
}
```

Here is its test class with two new tests, one to verify that multi-line elements are stripped and another to verify that Javascript code is filtered out.

```
import org.scalatest._
class HtmlUtilsSpec extends FlatSpec with ShouldMatchers {
  "The Html Utils object" should "remove single elements" in {
    HtmlUtils.removeMarkup("<br/>") should equal("")
  }
  it should "remove paired elements" in {
    HtmlUtils.removeMarkup("<b>Hi</b>") should equal("Hi")
  }
  it should "have no effect on empty strings" in {
    val empty = true
    HtmlUtils.removeMarkup("").isEmpty should be(empty)
  }
  it should "support multiline tags" in {
    val src = """
<html>
<body>
Cheers
<div class="header"></div>
</head></html>
    """
    HtmlUtils.removeMarkup(src).trim should equal("Cheers")
  }
  it should "strip Javascript source" in {
    val src = """
<html>
<head>
<script type="text/javascript">
  console.log("Yo");
</script>
</head></html>
    """
    HtmlUtils.removeMarkup(src) should not include "console.log"
  }
}
```

2)

a) Creating an object version of a trait is a popular way to extend the usefulness of that trait.

```
object SafeStringUtils extends SafeStringUtils
```

b) A good test should indicate a specific feature, whether functional or non-functional. Here are additional tests that clearly indicate the desired behavior from the object.

```
import org.scalatest._
class SafeStringUtilsSpec extends FlatSpec with ShouldMatchers {
  "The Safe String Utils object" should "trim empty strings to None" in {
    SafeStringUtils.trimToNone("") should be(None)
    SafeStringUtils.trimToNone(" ") should be(None)
    SafeStringUtils.trimToNone("           ") should be(None) // tabs and spaces
  }
  it should "handle null values safely" in {
    SafeStringUtils.trimToNone(null) should be(None)
  }
  it should "trim non-empty strings" in {
    SafeStringUtils.trimToNone(" hi there ") should equal(Some("hi there"))
  }
  it should "leave untrimmable non-empty strings alone" in {
    val testString = "Goin' down that road feeling bad ."
    SafeStringUtils.trimToNone(testString) should equal(Some(testString))
  }
}
```

c) The new "parseToInt" function first trims the input string, and then passes the value (if present) to a +toInt+ funciton that is wrapped in +Try+ and converted to +Option+. The +flatMap+ operation is used here as +toOption+ returns its own option, and we don't need two levels of options.

Also, this is a good time to convert the "trimToNone" comment into a full scaladoc header, describing the input parameter and return value.

```
import scala.util.Try
trait SafeStringUtils {
  /**
   * Returns a trimmed version of the string wrapped in an Option,
   * or None if the trimmed string is empty.
   *
   * @param s the string to trim
   * @return Some with the trimmed string, or None if empty
   */
  def trimToNone(s: String): Option[String] = {
    Option(s) map(_.trim) filterNot(_.isEmpty)
  }
  /**
   * Returns the string as an integer or None if it could not be converted.
   *
```

```
   * @param s the string to be converted to an integer
   * @return Some with the integer value or else None if not parseable
   */
  def parseToInt(s: String): Option[Int] = {
    trimToNone(s) flatMap { x => Try(x.toInt).toOption }
  }
}
object SafeStringUtils extends SafeStringUtils
```

Here's the full test class including three new tests for the "parseToInt" function.

```
import org.scalatest._
class SafeStringUtilsSpec extends FlatSpec with ShouldMatchers {
  "The Safe String Utils object" should "trim empty strings to None" in {
    SafeStringUtils.trimToNone("") should be(None)
    SafeStringUtils.trimToNone(" ") should be(None)
    SafeStringUtils.trimToNone("           ") should be(None) // tabs and spaces
  }
  it should "handle null values safely" in {
    SafeStringUtils.trimToNone(null) should be(None)
  }
  it should "trim non-empty strings" in {
    SafeStringUtils.trimToNone(" hi there ") should equal(Some("hi there"))
  }
  it should "leave untrimmable non-empty strings alone" in {
    val testString = "Goin' down that road feeling bad ."
    SafeStringUtils.trimToNone(testString) should equal(Some(testString))
  }
  it should "parse valid integers from strings" in {
    SafeStringUtils.parseToInt("5") should be(Some(5))
    SafeStringUtils.parseToInt("0") should be(Some(0))
    SafeStringUtils.parseToInt("99467") should be(Some(99467))
  }
  it should "trim unnecessary white space before parsing" in {
    SafeStringUtils.parseToInt("  5") should be(Some(5))
    SafeStringUtils.parseToInt("0  ") should be(Some(0))
    SafeStringUtils.parseToInt("  99467  ") should be(Some(99467))
  }
  it should "safely handle invalid integers" in {
    SafeStringUtils.parseToInt("5 5") should be(None)
    SafeStringUtils.parseToInt("") should be(None)
    SafeStringUtils.parseToInt("abc") should be(None)
    SafeStringUtils.parseToInt("1!") should be(None)
```

```
  }
}
```

d) Here's the final version of SafeStringUtils with the new random string function.

```
import scala.util.{Random, Try}
trait SafeStringUtils {
  /**
   * Returns a trimmed version of the string wrapped in an Option,
   * or None if the trimmed string is empty.
   *
   * @param s the string to trim
   * @return Some with the trimmed string, or None if empty
   */
  def trimToNone(s: String): Option[String] = {
    Option(s) map(_.trim) filterNot(_.isEmpty)
  }
  /**
   * Returns the string as an integer or None if it could not be converted.
   *
   * @param s the string to be converted to an integer
   * @return Some with the integer value or else None if not parseable
   */
  def parseToInt(s: String): Option[Int] = {
    trimToNone(s) flatMap { x => Try(x.toInt).toOption }
  }
  /**
   * Returns a string composed of random lower- and upper-case letters
   *
   * @param size the size of the composed string
   * @return the composed string
   */
  def randomLetters(size: Int): String = {
    val validChars: Seq[Char] = ('A' to 'Z') ++ ('a' to 'z')
    1 to size map { _ => Random nextInt validChars.size } map validChars mkString ""
  }
}
object SafeStringUtils extends SafeStringUtils
```

Following is the final version of the test class with three new tests.

```
import org.scalatest._
class SafeStringUtilsSpec extends FlatSpec with ShouldMatchers {
  "The Safe String Utils object" should "trim empty strings to None" in {
    SafeStringUtils.trimToNone("") should be(None)
```

```
    SafeStringUtils.trimToNone(" ") should be(None)
    SafeStringUtils.trimToNone("           ") should be(None) // tabs and spaces
  }
  it should "handle null values safely" in {
    SafeStringUtils.trimToNone(null) should be(None)
  }
  it should "trim non-empty strings" in {
    SafeStringUtils.trimToNone(" hi there ") should equal(Some("hi there"))
  }
  it should "leave untrimmable non-empty strings alone" in {
    val testString = "Goin' down that road feeling bad ."
    SafeStringUtils.trimToNone(testString) should equal(Some(testString))
  }
  it should "parse valid integers from strings" in {
    SafeStringUtils.parseToInt("5") should be(Some(5))
    SafeStringUtils.parseToInt("0") should be(Some(0))
    SafeStringUtils.parseToInt("99467") should be(Some(99467))
  }
  it should "trim unnecessary white space before parsing" in {
    SafeStringUtils.parseToInt("  5") should be(Some(5))
    SafeStringUtils.parseToInt("0  ") should be(Some(0))
    SafeStringUtils.parseToInt("  99467  ") should be(Some(99467))
  }
  it should "safely handle invalid integers" in {
    SafeStringUtils.parseToInt("5 5") should be(None)
    SafeStringUtils.parseToInt("") should be(None)
    SafeStringUtils.parseToInt("abc") should be(None)
    SafeStringUtils.parseToInt("1!") should be(None)
  }
  it should "generate random strings with only lower- and upper-case letters" in {
    SafeStringUtils.randomLetters(200).replaceAll("[a-zA-Z]","") should equal("")
  }
  it should "be sufficiently random" in {
    val src = SafeStringUtils.randomLetters(100).toList.sorted
    val dest = SafeStringUtils.randomLetters(100).toList.sorted
    src should not equal dest
  }
  it should "handle invalid input" in {
    SafeStringUtils.randomLetters(-1) should equal("")
  }
}
```