Exercises — Scala (Week Three)

Preparation for the coursework

Spring term 2017

Overiew

This set of exercises is an introductory lesson on Scala programming in the IntelliJ IDE. You are provided with a starter project where you will complete several Scala functions that exercise your understanding of basic Scala programming. These functions are based on concepts introduced in the first four chapters of *Scala for the Impatient* and equivalent texts. You will also learn how to use the sbt build system and how to run tests.

One of the objectives of this exercise sheet is to prepare you for the coursework assignments and the way in which you should approach their solution.

Learning Objectives

- To learn and exercise Scala variables and values.
- To learn and exercise Scalas basic types.
- To learn and exercise Scala control flow.
- To learn and exercise Scala for comprehensions.
- To learn and exercise basic Scala function declarations.
- To learn basic usage of the IntelliJ IDE.
- To learn how to compile using sbt.
- To learn how to run tests using sbt.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then communicate this to a member of the teaching team immediately.

Test files

In the `src/test/scala` directory, we provide several `ScalaTest` test suites that will help you keep on track while completing these exercises. We recommend you run the tests often and use them to help create a checklist of things to do next.

We recommend that you think about possible cases and add new test cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods handle edge cases such as integer arguments that may be positive, negative, or zero? Many methods only accept arguments that are in a particular range.
- Does your code handle unusual cases, such as empty or maximally sized data structures?

To build good test cases, think about ways to exercise functions and methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case.

Import Project into IntelliJ

Begin by downloading the starter project from the module code repository (`exercises/week03`). Then import it into IntelliJ. To import your project into IntelliJ you must run IntelliJ. If the IDE opens a previously created project, simply close the IDE window (do not close the IDE). It will then bring up a prompt with the following options:

- Create New Project
- Import Project
- Open
- Check out from version control

(Note: depending on the version of IntelliJ that you have these options may differ slightly.)

You should select `Import Project`. You will then need to find the downloaded project in the file menu and select `OK`. You should then select

`Import project from external model`

and then highlight `SBT`.

On the next screen make sure `Use auto import` is selected and then click `Finish`. IntelliJ will then initialise your project (give it a minute) and then show you your project structure on the left. (Again, depending upon the version of IntelliJ you are using the behaviour may vary slightly.)

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we provide unit tests for source files (e.g., classes, methods, functions) that do not yet exist in your code. You can still run the other unit tests though.

The project should contain the following root items:

src/main/scala This is the source folder where all code you are writing must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

src/test/scala The test folder where all of the public unit tests are available.

build.sbt This is the build definition file. It provides build information to construct the project. **Do not remove**.

.gitignore This is a special file used by git source control to ignore certain files. You should not touch this file and please **do not remove** it. This file may not be present if the exercise does not call for git usage.

Testing

As mentioned previously, you are provided with a set of unit tests that will test various aspects of your implementation. Do feel free to add additional tests if you wish to. You should get in the habit of running the tests frequently to see how you are doing and to understand where you might be going wrong. The `ScalaTest` testing framework is configured in the `sbt` tool and you can easily run the tests by issuing the following from the command line:

```
> sbt test
```

This will compile your code and run the public `ScalaTest` unit tests. After you compile and run the tests you will notice that a `target` directory has been created. The `target` directory contains the generated class files from your source code as well as information and results of the tests.

Another very useful approach to test and play with the code you write is the console. You can run the console with this command:

```
> sbt console
```

This will load up the Scala REPL (read eval print loop). You can type code directly into the console and have it executed. If you want to cut and paste a larger segment of code (e.g., function declaration) you simply type `:paste` in the console, then paste in your code, then type control-D.

Implementing Functions

After you download and import your project into IntelliJ you will be able to navigate the project structure. This exercise comes with a single source file for you to modify:

```
src/main/scala/basics/ScalaBasics.scala
```

This file contains a single Scala object declaration with stubs for functions that you will need to implement.

Begin by opening this file in the IDE. Take a look at each of the function stubs defined. You will notice that after each functions there is an empty body designated by `???`. This allows the Scala compiler to compile your project, but indicates that the function is not implemented. Indeed, if you were to execute these functions it will result in thrown exceptions telling you that the function was not implemented. Your job will be to replace each `???` by your implementation of the function.

You will also notice that `Scaladoc` documentation precedes each function declaration. We will cover `Scaladoc` at a later time, however, you should be able to read the documentation easily and each gives you constraints and hints on how to implement the function. The following is a summary of the functions you must implement.

1. `ScalaBasics.add`: You are to implement a simple add function that returns the sum of the two integer parameters.

2. `ScalaBasics.inRange`: Here you must implement a function that returns a `Range` type starting at `start` and ending at `end`. The documentation provides a hint on where you should look to find the answer to this.

3. `ScalaBasics.oddRange`: This function returns a `Range` type that returns `n` odd integers starting at `1`.

4. `ScalaBasics.minWhile`: You must implement a function that returns the minimum integer in the given non empty `Array` parameter. Your implementation must conform to the following constraints:

   - You must use a `while` loop.
   - You may use both immutable (`val`) and mutable (`var`) variables.
   - You must use an `if` expression.

5. `ScalaBasics.minFor`: You must implement a function that returns the minimum integer in the given non empty `Array` parameter. Your implementation must conform to the following constraints:

   - You must use a `for` loop (not a *for comprehension*).
   - You may use both immutable (`val`) and mutable (`var`) variables.
   - You may not use an `if` expression. (hint: you should find an appropriate function in the Scala library to determine the *min*).

6. `ScalaBasics.minRecursive`: You must implement a function that returns the minimum integer in the given non empty `Array` parameter. Your implementation must conform to the following constraints:

   - You may not use any loops.
   - You may not use any immutable (`val`) or mutable (`var`) variables.
   - You are allowed to use an `if` expression.
   - Your implementation must be *recursive*.
   - You may not use any library calls.

7. `ScalaBasics.base36`: In the real world it is often useful to be able to generate unique folder and file names. An easy way to do this is to use a `BigInt` and convert the `BigInt` into base 36 (which uses valid characters for folder and file names). You must implement a function that returns a base 36 string representation of the given `BigInt` value.

8. `ScalaBasics.splitInHalf`: You must implement a function that takes a string as input and returns a 2-tuple (pair) where the first element in the pair is the first half of the given string and the second element is the second half of the string. If the string has an odd number of characters it the first element will be the shorter half. Your implementation must conform to the following constraints:

   - You may not use any loops.
   - You may not use recursion.
   - You may not use any immutable (`val`) or mutable (`var`) variables.

9. `ScalaBasics.isPalindrome`: You must implement a function that returns `true` if the given string parameter is a palindrome[1] and `false` otherwise. You should normalise the string before you determine if it is a palindrome. That is, you must remove the characters '. (period), '? (question mark), ', (comma), '; (semi colon), '- (dash), ' (space), and ' (single quote) to eliminate punctuation. Your implementation must conform to the following constraints:

   - You must use a *for comprehension*.
   - You may not use any other loops.
   - You may not use any mutable (`var`) variables.

10. `ScalaBasics.wordCounter`: You must implement a function that counts the number of space delimited words in the provided array of strings. This function takes an array of strings that represent lines in a text file. It returns a `Map` from `String` to `Int` where the `String` is a word found across all lines and the `Int` is the number of times that word was seen (see function documentation for an example). You may assume that words are delimited by spaces and you need not worry about punctuation. You can implement this however you wish. We do not provide a test for this function, however, you are welcome to implement your own by mimicking the structure of the provided public tests. We will be covering `ScalaTest` in more detail at a later time.

As you implement these functions you should run the tests frequently as described above. You should also take advantage of the REPL (read eval print) loop and test out your functions by cut and pasting them from your source code into the console to test. If you prefer not to cut and paste you can also import the functions using the following import from the console:

```
scala> import basics.ScalaBasics._
```

and then proceed by calling the individual functions:

```
scala> add(4, 5)
```

[1]`https://en.wikipedia.org/wiki/Palindrome`