

# Scala Lab 1: Getting started with Scala

## CSP344/CS-205 | *by* Keoni D'Souza

---

### Part 0/ Welcome

#### 0.1/ Introduction

Hello! You're very welcome to my Scala lab session. I hope you enjoyed the lecture – this lab is designed to let you play around with Scala a bit (perhaps for the first time).

We'll start off with a few of the basics, and then I'll introduce to you some slightly more complicated concepts. You should be able to translate what you learnt in Haskell into its Scala equivalent, using what you know of Java's syntax to help you.

I hope you enjoy this lab session – and, if anything, realise why I almost (let's not be too dramatic!) fell in love with Scala. I'll be on hand to help if you have any questions, hopefully...

Keoni  
921231@swansea.ac.uk

---

## 0.II/ Setting up in the Windows Lab

### Task 1:

- Locate and open IntelliJ in the Unified Desktop under:  
Specialist Apps > College of Science > Computer Science.
- Set up a new project under File > New > Project..., selecting the Scala option on the left sidebar.
- Go for an sbt-based project (which should already be selected), click Next and then name it ScalaLabs.

- Unfortunately, you'll need to locate the JDK. Luckily for you, I found it (though, it did take a surprising amount of time)! Copy in the path or locate it here:

C:\Program Files\Java\jdk1.8.0\_45.

- Then Finish, my friend! But, that's not the end. In the project sidebar, inside

ScalaLabs/src/main/scala,

create a document scalaLab1.scala to work in, labelling each following task as necessary. (You can do this by right-clicking the scala folder and selecting New > Scala Class and naming it ScalaLab1.) Select Object as the kind.

Copy this code into the file and run it:

```
object scalaLab1 extends App {  
  println("Keoni welcomes you to Scala.")  
}
```

Does it print out the message? Good.

Extending the object with App means we don't need to bother with a separate main function. But, if you'd like to do that, try the following:

```
object scalaLab1A {  
  def main(args: Array[String]): Unit = {  
    println("Keoni welcomes you to Scala.")  
  }  
}
```

Right, you happy now? Again, good. Now we can move on.

## Part I/ Writing and reading in Scala

### I.I/ Writing a simple function in Scala

#### I.I.a/ The square function

The square of a number  $n$  is the result of multiplying a number by itself:  $n \times n$ , or  $n^2$ . The math library has a built-in power function:

```
pow(x: Double, y: Double): Double
```

where:

$x$  is the base

$y$  is the exponent

$x^y$  is returned

that allows you to calculate the square easily:

```
scala.math.pow(n,2)
```

You can also write one manually, of course. For example:

```
def square(n: Int): Int = n * n
```

The type annotations (the parts after the colons) don't necessarily have to be included because of Scala's built-in type inference – but you probably should, right? Right.

**Task 2:** Copy the below and run it. Check that what you believe to be the right value is returned:

```
object scalaLab1 extends App {  
  def square(n: Int): Int = n * n  
  println(square(2))  
}
```

#### I.I.b/ +3 function

**Task 3:** Write a function add3 that takes in a number and adds 3.

#### I.I.c/ Returning even numbers

Even numbers are integers divisible by 2. To check if a number is even, we can use modulo division, %.

**Task 4:** Write a function isEven that takes in an integer and returns a corresponding Boolean.

#### I.I.d/ Combining functions

**Task 5:** In a new function squareAnd3, write something that takes in an integer, squares it, adds 3, then returns the result.

## I.II/ Reading user input

The `io.StdIn` library stands for standard input. Add the import statement:

```
import scala.io.StdIn.readLine
```

at the start of your file to allow users to provide data from the keyboard that can be interacted with Scala functions.

**Task 6:** Write a function called `greet()` that uses `readLine()` to take in a name and produces a message not unlike the following:

Congratulations, [name] – you have been called to learn Scala!

Of course, [name] substitutes the name passed in – let's not be silly about it, guys...

## Part II/ Lists and higher order functions in Scala

### II.I/ Lists in Scala

In Scala, lists are immutable by default, as is the language's functional nature. For this part, we will be working with mutable lists to help transition you from Java. To that end, add this line to the start of the file:

```
import scala.collection.mutable.ListBuffer
```

`List` is immutable, so we'll have to use `ListBuffer` which *Scaladoc* explains is "a Buffer implementation backed by a list".

#### II.I.a/ Writing a list

Here is a list of door closers, the different kinds available:

- Automatic
- Concealed
- Overhead
- Slide arm
- Transom
- Electromagnetic
- Floor spring
- Emergency

**Task 7:** Create a variable (`var`) called `doorClosers` that creates a new instance of `ListBuffer` that will take in elements of type `String`.

**Task 8:** Using `++=`, add all the types of door closer above to `doorClosers`.

### II.I.b/ Printing from a list

Scala has three built-in operations that can be performed on a list – one of which, `isEmpty`, is rather self-explanatory, returning a `Boolean`.

**Task 9:** Use `head` to print the first element of `doorClosers`.

**Task 10:** Use `tail` to print the rest of the door closers.

### II.I.c/ Updating a list

**Task 11:** In a shocking turn of events, I reason that *automatic* is less a type of door closer and more of a classification. To remedy this, use `-=` to get rid of it.

**Task 12:** I realise similar can be said for *emergency*. This time, with postfix notation, use `remove()` to delete it.

**Task 13:** Finally, the fragment

```
var doorClosersList = doorClosers.toList
```

provides a copy of `doorClosers` in list-form. Try removing another element. Is it possible? Why/why not? How could the definition be improved?

## II.II/ Higher order functions in Scala

“Higher order functions”, *Scaladoc* explains, “take other functions as parameters or return a function as a result.” They are made possible in Scala because they support operations available to others like being passed as an argument, returned from a function, modified and assigned to a variable (i.e. they are first class values).

### II.II.a/ The `map` operation

`map` takes a predicate and applies it to every element contained within the collection. It's part of the `TraversableLike` trait, so will work on all different types of collections.

**Task 14:** In a `val` called `lengths`, use `map` to create a list that corresponds to the lengths of the names in `doorClosersList`.

### II.II.b/ The `filter` operation

`filter` takes a predicate that returns a `Boolean`. If an element evaluates to `true`, it is returned. Falsely evaluated items are filtered out of the result.

**Task 15:** Write a function `lessThanX` that takes in a list of strings and a length `x`. It should return the items that have a length less than the specified integer.

## Part III/ Interacting Scala with Java

As Scala's syntax is principally a superset of Java's, it doesn't tend to be too hard to invoke code written in Java with Scala. Well, that's what they say anyway...

Here is an example of a program simulating a bank account that is written in Java:

```
1 // Modified bank account example
2 // Original author: Robert Keller
3
4 class BankAccountJava {
5     int balance;
6     private int number;
7     String owner;
8     static int last_acct = 0; // account number counter
9
10    public BankAccountJava(String owner,
11                            int initial_balance) {
12        this.owner = owner;
13        balance = initial_balance;
14        number = ++last_acct;
15    }
16
17    public void deposit(int amount) {
18        balance += amount;
19    }
20
21    public void statement() {
22        System.out.println("Balance in account number " +
23                            number + " is " + balance);
24    }
25
26    public static void main(String[] args) {
27        BankAccountJava a, b, c;
28
29        a = new BankAccountJava("Jens", 15000);
30        b = new BankAccountJava("Yens", 200000);
31        c = new BankAccountJava("Yenz", 100000);
32
33        b.deposit(5000);
34        c.deposit(50000);
35
36        a.statement();
37        b.statement();
38        c.statement();
39    }
40 }
```

Some key points:

- Lines 10–15 define the constructor.
- Lines 17–19 allows you to deposit into an account.
- Lines 21–24 produces a bank statement.
- Line 26 onwards tests out the class.

**Task 16:** In a new file, `BankAccountJava.java`, under a new directory `java` in the main folder, copy and run the code. It should return the following:

```
Balance in account number 1 is 15000
Balance in account number 2 is 205000
Balance in account number 3 is 150000
```

**Task 17:** In a new file, `BankAccount.scala`, define an object called `BankAccount` and write a Scala function `withdrawFunds` that takes in an account and an amount and returns a message that indicates whether the withdrawal has been successful. Indeed, if so, the money should be deducted from the account.

**Task 18:** Write a Scala function `applyInterest` to apply interest to an account's balance. For the time being, use an integer modifier. You can add a message indicating that interest has been applied, if you so wish.

---

That's the end (just about). Thank you for joining me.

References:

Put the references here.