Scala Installation:

You can write Scala code in any text editor, compile it with scalac, and run it with scala. Alternatively, you can use the browser-based IDE provided by Typesafe. You can also use the Eclipsed-based Scala IDE, IntelliJ IDEA, or NetBeans IDE. You can download the Scala binaries and Typesafe Activator from www.scala-lang.org/download. The same site also provides links to download the Eclipsed-based Scala IDE, IntelliJ IDEA, or NetBeans IDE.

You launch it by typing scala in a terminal.

$ cd /path/to/scala-binaries
$ bin/scala

Exercises:

The easiest way to get started with Scala is by using the Scala interpreter, which provides an interactive shell for writing Scala code. It is a REPL (read, evaluate, print, loop) tool

Variables

// VALUES are immutable constants. You can't change them once defined.

```
val hello: String = "Inceptez!"          //> hello  : String = Inceptez!
println(hello)                           //> Hola!

 // Notice how Scala defines things backwards from other languages - you declare the
 // name, then the type.
 // VARIABLES are mutable

var helloThere: String =" hello"         //> helloThere  : String = Hola!
helloThere = "hello" + " There!"
println(helloThere)                      //> Hola! There!
```

// One key objective of functional programming is to use immutable objects as often as possible.
// Try to use operations that transform immutable objects into a new immutable object.
// For example, we could have done the same thing like this:

```
val immutableHelloThere = hello + "There!"
                                    //> immutableHelloThere : String = Hola!There!
println(immutableHelloThere)        //> Hola!There!

// Some other types:

val numberOne : Int = 1             //> numberOne : Int = 1
val truth : Boolean = true          //> truth : Boolean = true
val letterA : Char = 'a'            //> letterA : Char = a
val pi : Double = 3.14159265        //> pi : Double = 3.14159265
val piSinglePrecision : Float = 3.14159265f
                                    //> piSinglePrecision : Float = 3.1415927

val bigNumber : Long = 1234567890l  //> bigNumber : Long = 1234567890
val smallNumber : Byte = 127        //> smallNumber : Byte = 127
```

Control Structures:

```
object LearningScala2 {
// Flow control
// If / else syntax
if (1 > 3)
    println("Impossible!")
else
   println("The world makes sense.")
                                    //> The world makes sense.

if (1 > 3) {
       println("Impossible!")
         }
else {
       println("The world makes sense.")
}                                   //> The world makes sense.
```

```
// Matching - like switch in other languages:

val number = 3                          //> number  : Int = 3
number match {
        case 1 => println("One")
        case 2 => println("Two")
        case 3 => println("Three")
        case _ => println("Something else")
        }

        // For loops

        for (x <- 1 to 4) {
                val squared = x * x
                println(squared)
        }                                                   //|> 1
                                                            //| 4
                                                            //| 9
                                                            //| 16
```

While loops:

```
var x = 10                                                  //> x  : Int = 10
while (x >= 0) {
        println(x)
        x -= 1
}                               //> 10
                                //| 9
                                //| 8
                                //| 7
                                //| 6
                                //| 5
                                //| 4
                                //| 3
                                //| 2
                                //| 1
                                //| 0

x = 0
do { println(x); x+=1 } while (x <= 10)         //> 0
                                                //| 1
                                                //| 2
                                                //| 3
                                                //| 4
                                                //| 5
                                                //| 6
                                                //| 7
                                                //| 8
                                                //| 9
```

```
//| 10
```

Expressions :

```
// "Returns" the final value in a block automatically

        {
        val x = 10; x + 20
         }                                  //> res0: Int = 30

      println ({val x = 10; x + 20})        //> 30
```

EXERCISE :

```
// Write some code that prints out the first 10 values of the Fibonacci sequence.
// This is the sequence where every number is the sum of the two numbers before it.
        // So, the result should be 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

Functions:

```
object LearningScala3 {

 // Functions
 // Format is def <function name>(parameter name: type...) : return type = { expression }
 // Don't forget the = before the expression!

 def squareIt(x: Int) : Int = {
       x * x
 }                                          //> squareIt: (x: Int)Int

 def cubeIt(x: Int): Int = {x * x * x}      //> cubeIt: (x: Int)Int

 println(squareIt(2))                       //> 4

 println(cubeIt(2))                         //> 8


 // Functions can take other functions as parameters

 def transformInt(x: Int, f: Int => Int) : Int = {
       f(x)
 }                                          //> transformInt: (x: Int, f: Int => Int)Int

 val result = transformInt(2, cubeIt)       //> result  : Int = 8
 println (result)                           //> 8
```

```
// "Lambda functions", "anonymous functions", "function literals"
// You can declare functions inline without even giving them a name
// This happens a lot in Spark.

transformInt(3, x => x * x * x)                  //> res0: Int = 27

transformInt(10, x => x / 2)                     //> res1: Int = 5

transformInt(2, x => {val y = x * 2; y * y})     //> res2: Int = 16

// This is really important!
```

EXERCISE:

// Strings have a built-in .toUpperCase method. For example, "foo".toUpperCase gives you back FOO.

// Write a function that converts a string to upper-case, and use that function of a few test strings.

// Then, do the same thing using a function literal instead of a separate, named function.

Data Structures:

```
object LearningScala4 {

// Data structures
// Tuples (Also really common with Spark!!)
// Immutable lists
// Often thought of as database fields, or columns.
// Useful for passing around entire rows of data.

val captainStuff = ("Picard", "Enterprise-D", "NCC-1701-D")

//> captainStuff : (String, String, String) = (Picard,Enterprise-D,NCC-1701-D)

println(captainStuff)                  //> (Picard,Enterprise-D,NCC-1701-D)

// You refer to individual fields with their ONE-BASED index:

println(captainStuff._1)               //> Picard
println(captainStuff._2)               //> Enterprise-D
println(captainStuff._3)               //> NCC-1701-D

// You can create a key/value pair with ->

val picardsShip = "Picard" -> "Enterprise-D"
                          //> picardsShip : (String, String) = (Picard,Enterprise-D)
println(picardsShip._2)                //> Enterprise-D
```

// You can mix different types in a tuple

val aBunchOfStuff = ("Kirk", 1964, true)
//> aBunchOfStuff : (String, Int, Boolean) = (Kirk,1964,true)

Lists :

// Like a tuple, but it's an actual Collection object that has more functionality.
// Also, it cannot hold items of different types.
// It's a singly-linked list under the hood.

val shipList = List("Enterprise", "Defiant", "Voyager", "Deep Space Nine")
println(shipList(1))

//> shipList : List[String] = List(Enterprise, Defiant, Voyager, Deep Space Nine
// Access individual members using () with ZERO-BASED index (confused yet?)

// head and tail give you the first item, and the remaining ones.

println(shipList.head) //> Enterprise
println(shipList.tail) //> List(Defiant, Voyager, Deep Space Nine)

Iterating though a list:

for (ship <- shipList) {println(ship)} //> Enterprise
//| Defiant
//| Voyager
//| Deep Space Nine

// Let's apply a function literal to a list! map() can be used to apply any function to every item in a collection.

val backwardShips = shipList.map((ship: String) => {ship.reverse})
//> backwardShips : List[String] = List(esirpretnE,
tnaifeD, regayoV, eniN eca
//| pS peeD)
for (ship <- backwardShips) {println(ship)} //> esirpretnE
//| tnaifeD
//| regayoV
//| eniN ecapS peeD

// reduce() can be used to combine together all the items in a collection using some function.

```
val numberList = List(1, 2, 3, 4, 5) //> numberList  : List[Int] = List(1, 2, 3, 4, 5)
val sum = numberList.reduce( (x: Int, y: Int) => x + y)
                                                  //> sum  : Int = 15
println(sum)                                      //> 15
```

// filter() can remove stuff you don't want. Here we'll introduce wildcard syntax while we're at it.

```
val iHateFives = numberList.filter( (x: Int) => x != 5)
                                  //> iHateFives  : List[Int] = List(1, 2, 3, 4)
val iHateThrees = numberList.filter(_ != 3)
                                  //> iHateThrees  : List[Int] = List(1, 2, 4, 5)
```

// Note that Spark has its own map, reduce, and filter functions that can distribute these operations. But they work the same way!
// Also, you understand MapReduce now :)

Concatenating lists:

```
val moreNumbers = List(6, 7, 8)
                                          //> moreNumbers  : List[Int] = List(6, 7, 8)
val lotsOfNumbers = numberList ++ moreNumbers
                                  //> lotsOfNumbers  : List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)
```

// More list fun

```
val reversed = numberList.reverse
 //> reversed  : List[Int] = List(5, 4, 3, 2, 1)

val sorted = reversed.sorted
//> sorted  : List[Int] = List(1, 2, 3, 4, 5)

val lotsOfDuplicates = numberList ++ numberList
 //> lotsOfDuplicates  : List[Int] = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)

val distinctValues = lotsOfDuplicates.distinct
//> distinctValues  : List[Int] = List(1, 2, 3, 4, 5)

val maxValue = numberList.max
//> maxValue  : Int = 5

val total = numberList.sum
 //> total  : Int = 15
```

val hasThree = iHateThrees.contains(3)
//> hasThree : Boolean = false

//Maps
// Useful for key/value lookups on distinct keys
// Like dictionaries in other languages

val shipMap = Map("Kirk" -> "Enterprise", "Picard" -> "Enterprise-D", "Sisko" -> "Deep Space Nine", "Janeway" -> "Voyager")

// shipMap : scala.collection.immutable.Map[String,String] = Map(Kirk -> Ente rprise, Picard -> Enterprise-D, Sisko -> Deep Space Nine, Janeway -> Voyage//

println(shipMap("Janeway")) //> Voyager

// Dealing with missing keys

println(shipMap.contains("Archer")) //> false

val archersShip = util.Try(shipMap("Archer")) getOrElse "Unknown"
//> archersShip : String = Unknown
println(archersShip) //> Unknown

EXERCISE:
// Create a list of the numbers 1-20; your job is to print out numbers that are evenly divisible by three. (Scala's
// modula operator, like other languages, is %, which gives you the remainder after division. For example, 9 % 3 = 0
// because 9 is evenly divisible by 3.) Do this first by iterating through all the items in the list and testing each
// one as you go. Then, do it again by using a filter function on the list instead.

// That's enough for now!
// There is MUCH more to learn about Scala. We didn't cover many other collection types, including mutable collections.
// And we didn't even touch on object-oriented Scala. The book "Learning Scala" from O'Reilly is great if you want to
// go into more depth - but you've got enough to get through this course for now.

}

A Standalone Scala Application

So far, you have seen just snippets of Scala code. In this section, you will write a simple yet complete standalone Scala application that you can compile and run.

A standalone Scala application needs to have a singleton object with a method called main. This main method takes an input of type Array[String] and does not return any value. It is the entry point of a Scala application. The singleton object containing the main method can be named anything.

A Scala application that prints “Hello World!” is shown next.

```
object HelloWorld {
   def main(args: Array[String]): Unit = {
     println("Hello World!")
   }
}
```

You can put the preceding code in a file, and compile and run it. Scala source code files have the extension .scala. It is not required, but recommended to name a Scala source file after the class or object defined in that file. For example, you would put the preceding code in a file named HelloWorld.scala.