A Short Introduction to Scala

Otfried Cheong

February 22, 2018

1 Running Scala

Like Python, Scala can be executed interactively, or using a Scala script. To execute Scala interactively, just call scala from the command line. You can then type commands interactively, and they will be executed immediately:

```
Welcome to Scala version 2.8.1.final.
scala> println("Hello World")
Hello World
scala> println("This is fun")
This is fun
```

This is a good way to get to know a new programming language and to experiment with different types of data.

In most cases, however, you will want to write a script, consisting of Scala functions and statements to be executed. For example, we can create a file *for1.scala* with the following contents:

```
def square(n : Int) {
  println(n + "\t: " + n * n)
}

for (i <- 1 to 10)
  square(i)
```

We can run this script with the command `scala for1.scala`.

We will see later that there is a third way to execute Scala programs, namely to create one or more files that are compiled using the Scala compiler. The advantage is that such a compiled programs starts much faster.

2 Syntax

Comments in Scala look like in Java and C: Everything after a double-slash `//` up to the end of the line is a comment. A long comment that continues over several lines can be written by enclosing it in `/*` and `*/`.

Programming languages allow you to group together statements into *blocks*. For instance, the body of a function, or the body of a `while`-loop is a block. In Python, this so-called *block structure* is indicated using indentation. Python is very much an exception: in nearly all programming languages, indentation, and in fact any white space, does not matter at all. In Scala, like in Java and C, block structure is indicated using braces: `{` and `}`. For example, a typical `while`-loop may look like this:

```
var i = 0
while (i <= 10) {
  printf("%3d: %d\n", i, i * i)
  i += 1
}
```

In Java and C, all statements have to be terminated by a semicolon `;`. In Scala, this semicolon can nearly always be omitted when the statement ends at the end of the line. If you want to put several statements on the same line, you have to separate them using semicolons:

```
val s = "hello"; println(s)
```

The precise rule for line endings is the following: A line ending is considered to terminate a statement (like a semicolon) *unless*

- the line ends with a word or operator that is not legal as the end of a statement (like an infix operator or a period), or
- the next line starts with a word that cannot start a new statement, or
- the line ends while inside parenthesis `(...)` or brackets `[...]`.

3 Dynamic and static typing

Every piece of data handled by a Scala program is an *object*. Every object has a *type*. The type of an object determines what we can do with the object. Examples of objects are numbers, strings, files, and digital images.

A *variable* is a name assigned to an object during the execution of a program. The object currently assigned to a name is called the *value* of the name or variable.

In dynamically typed programming languages like Python, the same name can be assigned to all kinds of different objects:

```
# This is Python!
m = 17          # int
m = "seventeen" # str
m = 17.0        # float
```

This is flexible, and makes it easy to quickly write some code. It also makes it easy to make mistakes, though. If you assign an object of the incorrect type, you only find out during the execution of the program that something doesn't work—you get a runtime error.

Scala is a *statically typed* language. This means that a variable can only be assigned to objects of one fixed type, the *type of the variable*:

```
var m : Int = 17
m = 18          // ok
m = "seventeen" // error!
m = 18.0        // error!
```

The advantage of a statically typed language is that the compiler can catch type errors during the compilation, before the program is even run. A good compiler can also generate more efficient code for a statically type language, as the type of objects is already known during the compilation. Consider the following Python function:

```
# This is Python!
def test(a, b):
  print(a + b)
```

The `+` operator here could mean integer addition, float addition, string concatenation, or list concatenation, depending on the type of the parameters `a` and `b`. The code created for this function must look at the types of the arguments and determine which method to call. Now consider the following Scala function:

```
def test(a : Int, b : Int) {
  println(a + b)
}
```

Here it is clear to the compiler that the `+` operator means integer addition, and so the code for the function can immediately add the two numbers.

Other statically typed languages are Java, C, and C++. One disadvantage of statically typed languages is that one has to write type names everywhere, leading to code that is much more verbose than code in, say, Python. Indeed, in languages like Java or C, the programmer has to write down the type for every variable she uses:

```
/* This is C */
int m = 17;
float f = 17.0;
```

Scala makes our life easier and our code shorter by using *type inference*. If Scala can detect what the type of a variable name should be, then we do not need to indicate the type:

```
scala> var m : Int = 17  // ok
m: Int = 17
scala> var n = 18        // also ok!
n: Int = 18
scala> var f = 19.5
f: Double = 19.5
scala> var h = "Hello World"
h: java.lang.String = Hello World
```

Note how the interactive Scala interpreter tells you what the type of the name is even though we have not indicated it at all. When you are not sure if it is okay to omit the type of a name, you can always write it.

4 `val` variables and `var` variables

Scala has two different kinds of variables: `val` variables and `var` variables. `val` stands for *value*, and a `val` variable can never change its value. Once you have defined it, its value will always remain the same:

```
val m = 17
m = 18      // error!
```

A `var` variable, on the other hand, can change its value as often as you want:

```
var n = 17
n = 18      // ok
```

So why are `val` variables useful? Because they make it easier to understand and to discuss a program. When you see a `val` variable defined somewhere in a large function, you know that this variable will always have exactly the same value. If the programmer had used a `var` variable instead, you would have to carefully go through the code to find out if the value changes somewhere.

It is considered good style in Scala to use `val` variables as much as possible.

5 Some basic data types

The most important basic types you will need first are:

`Int` An integer in the range -2^{31} to $2^{31}-1$.

`Boolean` Either `true` or `false` (be careful: different from Python, in Scala these are written with small letters).

`Double` A floating-point representation of a real number.

`Char` A single character. To create a `Char` object, write a character enclosed in single quotation marks, like `'A'`. Scala supports Unicode characters, so a `Char` object can also be a Hangul syllable.

`String` A sequence of `Char`. To create a `String` object, write a string enclosed in double quotation marks. Scala also supports the triple-quote syntax from Python to create long strings possibly containing new-line characters.

You will sometimes see that Scala indicates the type of a string as `java.lang.String`. Since Scala runs using the Java virtual machine and can use Java libraries, Scala uses Java strings for compatibility. We can just write `String` to indicate the string type.

Special characters can be placed in `Char` and `String` literals using a backslash. For instance, `\n` is a new-line character, `\t` a tabulator, `\'` and `\"` are single and double quotes, and `\\` is a backslash character. (Inside triple-quote strings, the backslash is not interpreted like this!)

A list of important `String`-methods is in Appendix A.

Sometimes we want to use integers larger than the maximal number allowed by the `Int` type. In that case, we can use the type `Long`, which covers the range 2^{-63} to $2^{63}-1$. To create a `Long` object, write a number followed by `L`, like `123L`. (If you need even larger integers, you can use the type `BigInt`, which represents arbitrarily large integers. But `BigInt` is much slower.)

For completeness, the other basic types in Scala are `Byte`, `Short`, and `Float`. Don't use them unless you know what you are doing.

You can convert objects between the basic types by using their conversion methods:

```
scala> 98.toChar
res1: Char = b
scala> 'c'.toDouble
res2: Double = 99.0
scala> 'c'.toString
res3: java.lang.String = c
scala> 98.55.toInt
res4: Int = 98
```

6 Operators

Scala's numerical operators are `+`, `-`, `/`, and `%`. Like in Python 2, Java, and C, the division operators `/` and `%` perform an *integer division* if both operands are integer objects. Note that there is no power operator in Scala (you can use the `math.pow` library function to do exponentiation).

You can use the shortcuts `+=`, `-=`, etc.

You can compare numbers or strings using `==`, `!=`, `<`, `>`, `<=`, and `>=`.

The boolean operators are `!` for *not*, `&&` for *and*, and `||` for *or*, as in C and Java. (These somewhat strange operators were invented for C in the early 1970s, and have somehow survived into modern programming languages.) Like in Python, Java, and C, these operators only evaluate their right hand operand when the result is not already known from the value of the left hand operand.

Sooner or later, you may meet the *bitwise operators*: `&`, `|`, `^`, `~`, `<<`, `>>`, and `>>>`. They work on the bit-representation of integers. (A common mistake is to try to use `^` for exponentiation—it does something completely different, namely a logical exclusive-or.) Don't use these operators unless you know what you are doing.

The equality and inequality operators `==` and `!=` can be used for objects of any type, even to compare objects of different type.

One of the great features of Scala is that you can define your own operators in a very flexible way. It is easy to abuse this feature, though, and we will not use it until later in the course.

In Scala, the mathematical constants and functions are methods of the package `math`, the equivalent of the Python `math` module. For instance:

```
scala> val pi = math.Pi
pi: Double = 3.141592653589793
scala> val t = math.sin(pi/6)
t: Double = 0.49999999999999994
scala> math.sin(pi/4)
res1: Double = 0.7071067811865475
scala> math.sqrt(2.0)/2
res2: Double = 0.7071067811865476
```

If you don't like the long function names, you can *import* some of the functions:

```
scala> import math.sin
import math.sin
scala> sin(math.Pi/4)
res1: Double = 0.7071067811865475
```

Here, we imported only the `sin`-function. We could also import several functions at once:

```
scala> import math.{Pi,sin,cos,sqrt}
import math.{Pi, sin, cos, sqrt}
scala> sin(Pi/4)
res1: Double = 0.7071067811865475
scala> sqrt(2.0)/2
res2: Double = 0.7071067811865476
```

You can even import *everything* in the `math`-package using this syntax:

```
scala> import math._
```

But note that unlike in Python, it is *not* necessary to write an `import`-statement to use the `math`-package!

7 Writing functions

Scala function definitions look very much like mathematical function definitions. For instance, in mathematics we would define a function f as follows

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \ f(a, b) = a + b$$

In Scala, this function would be written like this

```
scala> def f(a:Int, b:Int) : Int = a + b
f: (a: Int,b: Int)Int
scala> f(3, 9)
res1: Int = 12
```

Note that we have indicated the type of each parameter of the function as well as the type of the result.

The *body* of the function is a *block* of statements. A block can either be a single expression as in function `f` above, or it consists of several statements surrounded by curly braces:

```
def g(a: Int, b : Int) : Int = {
  val m = a*a - b*b
  val s = a - b
  m/s
}
```

As mentioned above, indentation has no meaning to the Scala compiler at all. Nevertheless, you should indent your programs to be readable!

One big difference between Scala and Python, C, or Java is that *no return statement is needed* in functions. The function automatically returns the value of the *last expression* in the function body. You can still use `return` statements when you want to return from the function earlier.

You *have* to write the name of each parameter type for a Scala function. The result type is actually optional—you can try to omit it, and see if the Scala compiler can figure it out. I recommend always writing the result type until you have more experience in Scala programming. Including the result type also makes your program easier to read.

Like in Python, every Scala function returns a value. A function that does not need to return anything will return the special value `()`, the only object of the special type `Unit`, similar to the special value `None` in Python.

So a function that returns nothing useful could be written like this:

```
def greet(name : String) : Unit = {
  println("Hello " + name)
}
```

Since this case is common, Scala provides a special shorthand notation for it:

```
def greet(name : String) {
  println("Hello " + name)
}
```

So we omit both the result type *and the equals sign.*

The parameters of a function are `val` variables, and so it is not allowed to change their value inside the function. This is different from Python, Java, and C. So the following is illegal:

```
def test(a: Int) {
  a = a + 7  // illegal!
  println(a)
}
```

One interesting feature of Scala is that it supports *parameterless* functions. Consider the different syntax of functions `f1` and `f2` here:

```
scala> def f1() { println("Hi f1") }
f1: ()Unit
scala> def f2 { println("Hi f2") }
f2: Unit
```

`f1` is a normal function with an empty parameter list, so you would normally call it as `f2()`. However, `f2` is a parameterless function—you call it simply as `f2`:

```
scala> f1()
Hi f1
scala> f2
Hi f2
```

Note that Scala is somewhat generous and allows you to call `f1` without the empty parentheses. However, calling `f2` *with parentheses* is illegal:

```
scala> f1
Hi f1
scala> f2()
<console>:7: error: f2 of type Unit
           does not take parameters
```

We will see later why this is actually a very useful feature.

8 Conditional expressions

The syntax of the conditional expression is

```
if (pts > 80) {
  s = "well done!"
} else {
  s = "practice harder"
}
```

As in Python, the `else` part can be omitted:

```
if (pts > 80) {
  println("good student")
}
```

The then-part and the else-part are *blocks*, that is, either a single statement or a sequence of statements surrounded by braces. So the example above can be abbreviated like this:

```
if (pts > 80)
  s = "well done!"
else
  s = "practice harder"
```

You should only do this if it makes the code easier to read, not harder!

So far, Scala's `if` statement looks exactly like in Java or C. There is a difference though: Scala's `if` expression *has a value* itself, and so you can write code like this:

```
  val msg = if (pts > 80)
      "well done"
    else
      "practice harder!"
```

You can therefore use `if` expressions inside other expressions.

9 While loops

Scala's `while` loop looks exactly like Java's or C's (and much like Python's):

```
var i = 1
while (i < 10) {
  println(i + "\t: " + i*i)
  i += 1
}
```

Scala does not have `break` and `continue` statements. (You may understand why later in the course.)

The simple `while` loop above could have been written more easily using a `for` loop as follows:

```
for (i <- 1 until 10)
  println(i + "\t: " + i*i)
```

Here `i until j` includes the integers $\{i, i+1, \ldots, j-1\}$. Note that curly braces are not needed since there is only one statement inside the loop.

Alternatively, you could have written

```
for (i <- 1 to 10)
  println(i + "\t: " + i*i)
```

This time, the loop goes from 1 to 10, as `i to j` includes the integers $\{i, i+1, \ldots, j\}$.

10 Arrays

To store many objects of the same type, we can use an array:

```
scala> val L = Array("CS109", "is",
                     "the", "best")
L: Array[java.lang.String] =
   Array(CS109, is, the, best)
scala> L.length
res1: Int = 4
scala> L(0)
res2: java.lang.String = CS109
scala> L(3)
res3: java.lang.String = best
```

Note the type of `L`. It is not just `Array`, but `Array[java.lang.String]`. `Array` is a *parameterized type*, where the type parameter indicates the type of the objects stored inside the array.

The elements of the array can be accessed as `L(0)` to `L(L.length-1)`. Scala uses normal round parentheses to access the elements of an array, not square brackets like Java, C, or Python.

An array is a consecutive block of memory that can store a fixed number of objects. This is the same as arrays in Java or C, but completely different from lists in Python: You cannot append elements to an array, or change its length in any way. (We will later meet other data structures where you can do this.)

Above we saw how to create an array when you already know the objects that go in the array. In many cases you want to create an array that is initially empty:

```
scala> val A = new Array[Int](100)
A: Array[Int] = Array(0, ... 0)
```

Here, `Array[Int]` indicates the type of object you want to create, and `100` is the number of elements. When you create an array of numbers, all elements are initially zero. It's different for other arrays:

```
scala> val B = new Array[String](5)
B: Array[String] =
   Array(null, null, null, null, null)
```

Here, the elements of the new array have the special value `null`, similar to `None` in Python. This value typically means that a variable has not been initialized yet. Except for the basic numeric types, a variable of any type can have the value `null`.

You can use a `for`-loop to look at the elements of an array one-by-one:

```
scala> for (i <- L)
     |   println(i)
CS109
is
the
best
```

This is called "iterating over an array" and is quite similar to the `for`-loop in Python.

A list of the most important array methods can be found in Appendix B.

11 Command line arguments

When you call a Scala script from the command line, you can add *command line arguments*:

```
scala script.scala I love CS109!
```

The command line arguments are available inside the script in the array `args`. If *script.scala* looks like this:

```
for (s <- args)
  println(s)
```

then the output of the command line call above would be this:

```
> scala script.scala I love CS109!
I
love
CS109!
```

Here is another example script, *triangle.scala*:

```
val n = args(0).toInt

for (i <- 1 to n) {
  for (j <- 1 to i)
    print("*")
  println()
}
```

```
> scala triangle.scala 3
*
**
***
> scala triangle.scala 6
*
**
***
****
*****
******
```

12 Tuples

Like Python, Scala supports tuples:

```
scala> var a = (3, 5.0)
a: (Int, Double) = (3,5.0)
```

Unlike in Python, the parentheses are stricly necessary. Note the type: the first element of tuple `a` is an `Int`, the second one a `Double`. You can assign only tuples of the correct type to the variable `a`:

```
scala> a = (2, 7)
a: (Int, Double) = (2,7.0)
scala> a = (2.0, 7)
<console>:5: error: type mismatch;
 found   : Double(2.0)
 required: Int
```

You can access the elements of a tuple as fields `_1`, `_2`, and so on:

```
scala> a._1
res1: Int = 3
```

You can also "unpack" a tuple like in Python:

```
scala> val (x,y) = a
x: Int = 3
y: Double = 5.0
```

The parentheses around `x` and `y` are required—if you omit them, something completely different happens (try it)!

13 Reading from the terminal

You already noticed the functions `print`, `println`, and `printf` for printing to the terminal.

To read input from the terminal, there are a number of `read`-methods:

- `readLine()` reads a line and returns it as a string (without the new-line character at the end);
- `readLine(prompt)` prints a prompt and then reads a line as above, as in this example:

  ```
  val s = readLine("What do you say? ")
  println("You said: '" + s + "'")
  ```

- `readInt()` reads a line and returns it as an integer;
- `readDouble()` reads a line and returns a floating point number;
- `readBoolean()` reads a line and returns a `Boolean` (“yes”, “y”, “true”, and “t” for `true`, and anything else for `false`); and
- `readChar()` reads a line and returns the first character.

For instance:

```
print("How many beer? ")
val n = readInt()
printf("You ordered %d beer\n", n)
print("Are you sure? ")
if (readBoolean())
  printf("Serving %d beer\n", n)
```

All the `print` and `read` methods mentioned above are actually methods of the `Console` object, but Scala makes them available as if they were global functions because they are so common.

14 Selecting alternatives

The `match` expression is similar to the switch-statement in Java and C, but really much more powerful, as we will see later. For the moment, you can use it to conditionally execute different expressions depending on the value of some object:

```
val food = args(0)

val banchan = food match {
  case "rice" => "kimchi"
  case "noodles" => "gakdugi"
  case "pizza" => "pickles"
  case "bread" => "cheese"
  case _ => "huh?"
}
printf("With %s, we recommend %s\n",
       food, banchan)
```

Note that there is no `break` statement: A `case` ends when the next case starts. The `match` expression really is an expression and returns a value: in this case, we assign it to a variable.

The default case is indicated by an underscore `_`.

A String methods

Since `String` is such an important class and the on-line documentation is in Java syntax, here is a list of the most important string methods. Here, `S` and `T` are strings.

- `S.length` is the length of the string in characters;
- `S.substring(i)` returns the part of the string starting at index `i`.
- `S.substring(i,j)` returns the part of the string starting at index `i` and going up to index `j-1`. You can write `S.slice(i,j)` instead.
- `S.contains(T)` returns `true` if `T` is a substring of `S`;
- `S.indexOf(T)` returns the index of the first occurrence of the substring `T` in `S` (or -1);
- `S.toLowerCase` and `S.toUpperCase` return a copy of the string with all characters converted to lower or upper case;
- `S.capitalize` returns a new string with the first letter only converted to upper case;
- `S.reverse` returns the string backwards;
- `S.isEmpty` is the same as `S.length == 0`;
- `S.nonEmpty` is the same as `S.length != 0`;
- `S.startsWith(T)` returns `true` if `S` starts with `T`;
- `S.endsWith(T)` returns `true` if `S` ends with `T`;
- `S.replace(c1, c2)` returns a new string with all characters `c1` replaced by `c2`;
- `S.replace(T1, T2)` returns a new string with all occurrences of the substring `T1` replaced by `T2`;
- `S.trim` returns a copy of the string with white space at both ends removed;
- `S.format(arguments)` returns a string where the percent-placeholders in `S` have been replaced by the arguments (see example below);
- `S.split(T)` splits the string into pieces and returns an array with the pieces. `T` is a regular expression (not explained here). To split around white space, use `S.split("\\s+")`.

For example:

```
scala> val S = "CS109 is nice"
S: java.lang.String = CS109 is nice
scala> S.contains("ice")
res0: Boolean = true
scala> S.indexOf("ice")
res1: Int = 10
scala> S.indexOf("rain")
res2: Int = -1
scala> S.replace('i', '#')
res4: java.lang.String = CS109 #s n#ce
scala> S.split("\\s+")
res5: Array[java.lang.String] = Array(CS109, is, nice)
scala> S.toLowerCase
res6: java.lang.String = cs109 is nice
scala> S.toUpperCase
res7: java.lang.String = CS109 IS NICE
scala> S.substring(5)
res8: java.lang.String = " is nice"
scala> S.substring(5,8)
res9: java.lang.String = " is"
scala> S.reverse
res10: String = ecin si 901SC
scala> val F = "%5s %3d %-3d %g"
F: java.lang.String = %5s %3d %-3d %g
scala> F.format("cs206", 12, 3, math.Pi)
res11: String = cs206  12 3   3.14159
```

B Array methods

A short list of the most useful methods of arrays:

- `A.head` the first element of the array;
- `A.last` the last element of the array;
- `A.contains(x)` tests if array contains an element equal to `x`;
- `A.take(n)` returns a new array that has the first `n` elements of `A`;
- `A.drop(n)` returns a new array that has the elements of `A` except for the first `n` elements;
- `A.max`, `A.min`, `A.sum` return the largest, smallest, and sum of elements in the array;
- `A.reverse` returns a new array with the elements of `A` in reverse order;
- `A.sorted` returns a new Array with the elements of `A` in sorted order;
- `A.mkString` returns a string with all elements of `A` concatenated together;
- `A.mkString(sep)` returns a string with all elements of `A` concatenated, using `sep` as a separator;
- `A.mkString(start, sep, end)` returns a string with all elements of `A` concatenated, using `sep` as a separator, prefixed by `start` and ended by `end`;
- `A ++ B` returns a new array that contains the elements of arrays `A` and `B` concatenated;
- `A :+ el` returns a new array that contains the element of `A` with the element `el` appended at the back;
- `el +: A` returns a new array that contains the element of `A` with the element `el` appended at the front.

For example:

```
scala> val L = Array("CS109", "is", "the", "best")
L: Array[java.lang.String] = Array(CS109, is, the, best)
scala> L.head
res0: java.lang.String = CS109
scala> L.last
res1: java.lang.String = best
scala> L.mkString
res2: String = CS109isthebest
scala> L.mkString(" ")
res3: String = CS109 is the best
scala> L.mkString("--")
res4: String = CS109--is--the--best
scala> L.mkString("<", "_", ">")
res5: String = <CS109_is_the_best>
scala> (L.min, L.max)
res6: (java.lang.String, java.lang.String) = (CS109,the)
scala> L.sorted
res7: Array[java.lang.String] = Array(CS109, best, is, the)
scala> L.reverse
res8: Array[java.lang.String] = Array(best, the, is, CS109)
scala> L ++ Array("course", "at", "KAIST")
res9: Array[java.lang.String] = Array(CS109, is, the, best, course, at, KAIST)
scala> L :+ "course"
res10: Array[java.lang.String] = Array(CS109, is, the, best, course)
scala> "The" +: L
res11: Array[java.lang.String] = Array(The, CS109, is, the, best)
```