Contents

Scala Basics

Declaring Variables

Declaring values is done using either var or val. val declarations are immutable, whereas vars are mutable. Immutability is a good thing.

val x = 10

x is now 10

x = 20

error: reassignment to val

var y = 10

y = 20

y is now 20

Scala is a statically typed language, yet note that in the above declarations, we did not specify a type. This is due to a language feature called type inference. In most cases, Scala compiler can guess what the type of a variable is, so you don't have to type it every time. We can explicitly declare the type of a variable like so:

```
val z: Int = 10
val a: Double = 1.0
```

Notice automatic conversion from Int to Double, result is 10.0, not 10

```
val b: Double = 10
```

Boolean values

```
true
false
```

Comments and Print

```
// Single line comment
/*
  Multi-line comments, as you can already see from above, look like this.
*/
```

Printing, and forcing a new line on the next print

```
println("Hello world!")
println(10)
```

Printing, without forcing a new line on next print

```
print("Hello world")
```

```
print(10)
```

Basic Operations

Math is as per usual

1 + 1 *2*

2 - 1 *1*

5 * 3 *15*

6 / 2 *3*

6 / 4 *1*

6.0 / 4 *1.5*

Evaluating an expression in the REPL gives you the type and value of the result

```
1 + 7
```

The above line results in:

scala> 1 + 7

res29: Int = 8

This means the result of evaluating 1 + 7 is an object of type Int with a value of 8

Note that "res29" is a sequentially generated variable name to store the results of the expressions you typed, your output may differ.

Strings

"Scala strings are surrounded by double quotes"

'a' A Scala Char

'Single quote strings don't exist' <= This causes an error

Strings have the usual Java methods defined on them

```
"hello world".length
```

```
"hello world".substring(2, 6)

"hello world".replace("C", "3")
```

Expressions inside interpolated strings are also possible. Prepending s to any string literal allows the usage of variables directly in the string.

```
val a = Array(11, 9, 6)

s"My second daughter is ${a(0) - a(2)} years old."
```

"My second daughter is 5 years old."

```
s"We have double the amount of ${n / 2.0} in apples."
```

"We have double the amount of 22.5 in apples."

```
s"Power of 2: ${math.pow(2, 2)}"
```

"Power of 2: 4"

Prepending f to any string literal allows the creation of simple formatted strings, similar to printf in other languages.

```
val height = 1.9d

val name = "James"

println(f"$name%s is $height%2.2f meters tall")
```

James is 1.90 meters tall

Triple double-quotes let strings span multiple rows and contain quotes

```
val html = """<form id="daform">
              <p>Press belo', Joe</p>
              <input type="submit">
             </form>"""
```

Functions

Functions are defined as:

def functionName(args...): ReturnType = { body... }

If you come from more traditional languages, notice the omission of the return keyword. In Scala, the last expression in the function block is the return value.

```
def sumOfSquares(x: Int, y: Int): Int = {
  val x2 = x * x
  val y2 = y * y
  x2 + y2
}
```

The { } can be omitted if the function body is a single expression:

```
def sumOfSquaresShort(x: Int, y: Int): Int = x * x + y * y
```

Syntax for calling functions is familiar:

```
sumOfSquares(3, 4)
```

Ans: 25

You can use parameters names to specify them in different order

```
def subtract(x: Int, y: Int): Int = x - y
subtract(10, 3)      => 7
subtract(y=10, x=3)  => -7
```

In most cases (with recursive functions the most notable exception), function return type can be omitted, and the same type inference we saw with variables will work with function return values:

```
def sq(x: Int) = x * x   Compiler can guess return type is Int
```

Functions can have default parameters:

```
def addWithDefault(x: Int, y: Int = 5) = x + y
addWithDefault(1, 2)  => 3
addWithDefault(1)     => 6
```

Anonymous functions look like this:

```
(x: Int) => x * x
```

Unlike defs, even the input type of anonymous functions can be omitted if the context makes it clear. Notice the type "Int => Int" which means a function that takes Int and returns Int.

```
val sq: Int => Int = x => x * x
```

Anonymous functions can be called as usual:

```
sq(10)    => 100
```

If each argument in your anonymous function is used only once, Scala gives you an even shorter way to define them. These anonymous functions turn out to be extremely common, as will be obvious in the data structure section.

```
val addOne: Int => Int = _ + 1
val weirdSum: (Int, Int) => Int = (_ * 2 + _ * 3)

addOne(5)       => 6
weirdSum(2, 4)  => 16
```

Flow Control

```
1 to 5
val r = 1 to 5
r.foreach(println)
r foreach println
```

Scala is quite lenient when it comes to dots and brackets - study the rules separately. This helps write DSLs and APIs that read like English

```
(5 to 1 by -1).foreach (println)
```

A while loop

```
var i = 0
while (i < 10) { println("i " + i); i += 1 }
```

A do-while loop

```
i = 0
do {
  println("i is still less than 10")
  i += 1
} while (i < 10)
```

Conditionals

```
val x = 10

if (x == 1) println("yeah")
if (x == 10) println("yeah")
if (x == 11) println("yeah")
```

```
if (x == 11) println("yeah") else println("nay")

println(if (x == 10) "yeah" else "nope")
val text = if (x == 10) "yeah" else "nope"
```

Collections

Arrays, Maps & Sets

```
val a = Array(1, 2, 3, 5, 8, 13)

a(0)     // Int = 1
a(3)     // Int = 5
a(21)    // Throws an exception

val m = Map("fork" -> "tenedor", "spoon" -> "cuchara", "knife" -> "cuchillo")

m("fork")        // java.lang.String = tenedor
m("spoon")       // java.lang.String = cuchara
m("bottle")      // Throws an exception

val safeM = m.withDefaultValue("no lo se")
safeM("bottle")   // java.lang.String = no lo se

val s = Set(1, 3, 7)
s(0)      // Boolean = false
s(1)      // Boolean = true
```

Look up the documentation of map here -

http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Map

Lists

Scala Lists are quite similar to arrays which means, all the elements of a list have the same type but there are two important differences. First, lists are immutable, which means elements of a list cannot be changed by assignment. Second, lists represent a linked list whereas arrays are flat.

List of Strings

```
val fruit: List[String] = List("apples", "oranges", "pears")
```

List of Integers

```
val nums: List[Int] = List(1, 2, 3, 4)
```

Empty List.

```
val empty: List[Nothing] = List()
```

Access first or last elements of a list

```
fruit.head
fruit.tail
```

Check if list is empty

```
fruit.isEmpty
```

Access element in the list by index

```
fruit(2)
```

if index is out of bound, it will throw an exception. Safer is to use lift so you can extract the value if it exists and fail gracefully if it does not.

```
fruit.lift(2)
```

Will return None if index if out of bounds.

Merge two lists

```
val fruit1 = "apples" :: ("oranges" :: ("pears" :: Nil))
val fruit2 = "mangoes" :: ("banana" :: Nil)

val allfruits = fruit1 ::: fruit2
List.concat(fruit1, fruit2)
```

Iterating through values of list

```
fruits.foreach { println }
```

Filter Elements in a list

```
val x = List(1,2,3,4,5,6,7,8,9,10)
val evens = x.filter(a => a % 2 == 0)

Ans: evens: List[Int] = List(2, 4, 6, 8, 10)
```

Map through a function

```
val x = List( 1, 2, 3 )
val y = x.map(a => a * 2)
y: List[Int] = List(2, 4, 6)
```

Reverse a list

```
fruit.reverse
```

Tuples

Scala tuple combines a fixed number of items together so that they can be passed around as a whole. Unlike an array or list, a tuple can hold objects with different types but they are also immutable.

(1, 2)

(4, 3, 2)

(1, 2, "three")

(a, 2, "three")

Let's define a function to return a tuple

```
val divideInts = (x: Int, y: Int) => (x / y, x % y)
```

The function divideInts gives you the result and the remainder

```
divideInts(10, 3)
```

Ans: (Int, Int) = (3,1)

To access the elements of a tuple, use _._n where n is the 1-based index of the element

```
val d = divideInts(10, 3)    // (Int, Int) = (3,1)

d._1    // Int = 3
d._2    // Int = 1
```

Alternatively, you can do multiple-variable assignment to tuple, which is more convenient and readable in many cases

```
val (div, mod) = divideInts(10, 3)

div     // Int = 3
mod     // Int = 1
```

Classes and Case Classes

Aside: Everything we've done so far in this tutorial has been simple expressions (values, functions, etc). These expressions are fine to type into the command-line interpreter for quick tests, but they cannot exist by themselves in a Scala file. For example, you cannot have just "val x = 5" in a Scala file. Instead, the only top-level constructs allowed in Scala are:

- objects

- classes

- case classes

Classes are similar to classes in other languages. Constructor arguments are declared after the class name, and initialization is done in the class body.

```
class Dog(br: String) {
  var breed: String = br

  Define a method called bark, returning a String
  def bark = "Woof, woof!"

  Values and methods are assumed public. "protected" and "private" keywords are also available.
  private def sleep(hours: Int) =
    println(s"I'm sleeping for $hours hours")

  Abstract methods are simply methods with no body.
abstract class Dog(...) { ... }
   def chaseAfter(what: String): String
}

val mydog = new Dog("greyhound")
```

```
println(mydog.breed)  => "greyhound"
println(mydog.bark)   => "Woof, woof!"
```

The "object" keyword creates a type AND a singleton instance of it. It is common for Scala classes to have a "companion object", where the per-instance behavior is captured in the classes themselves, but behavior related to all instance of that class go in objects. The difference is similar to class methods vs static methods in other languages. Note that objects and classes can have the same name.

```
object Dog {
  def allKnownBreeds = List("pitbull", "shepherd", "retriever")
  def createDog(breed: String) = new Dog(breed)
}
```

Case classes are classes that have extra functionality built in. A common question for Scala beginners is when to use classes and when to use case classes. The line is quite fuzzy, but in general, classes tend to focus on encapsulation, polymorphism, and behaviour. The values in these classes tend to be private, and only methods are exposed. The primary purpose of case classes is to hold immutable data. They often have few methods, and the methods rarely have side-effects.

```
case class Person(name: String, phoneNumber: String)
```

Create a new instance. Note cases classes don't need "new"

```
val george = Person("George", "1234")
val kate = Person("Kate", "4567")
```

With case classes, you get a few perks for free, like getters:

```
george.phoneNumber   => "1234"
```

Per field equality (no need to override .equals)

```
Person("George", "1234") == Person("Kate", "1236")   => false
```

Easy way to copy

```
otherGeorge == Person("george", "9876")
```

```
val otherGeorge = george.copy(phoneNumber = "9876")
```

Functional Programming

Scala allows methods and functions to return, or take as parameters, other functions or methods.

A function taking an Int and returning an Int

```
val add10: Int => Int = _ + 10
```

List(11, 12, 13) - add10 is applied to each element

```
List(1, 2, 3) map add10
```

Anonymous functions can be used instead of named functions:

```
List(1, 2, 3).map(x => x + 10)
```

And the underscore symbol, can be used if there is just one argument to the anonymous function. It gets bound as the variable

```
List(1, 2, 3).map (_ + 10)
```

If the anonymous block AND the function you are applying both take one argument, you can even omit the underscore

```
List("Dom", "Bob", "Natalia").foreach( println )
```

Combinators

```
val sSquared = List( 1, 2, 3, 4, 5).map( x => x * x )
```

```
sSquared.filter(_ < 10).reduce (_+_)
```

Filters

The filter function takes a predicate (a function from A -> Boolean) and selects all elements which satisfy the predicate

```
List(1, 2, 3) filter (_ > 2)
```

Results will be List(3)

```
case class Person(name: String, age: Int)
List(
  Person(name = "Dom", age = 23),
  Person(name = "Bob", age = 30)
).filter(_.age > 25)
```

Results will be List(Person("Bob", 30))

Imports

Importing things

```
import scala.collection.immutable.List
```

Import all "sub packages"

```
import scala.collection.immutable._
```

Import multiple classes in one statement

```
import scala.collection.immutable.{List, Map}
```

Rename an import using '=>'

```
import scala.collection.immutable.{List => ImmutableList}
```

Java classes can also be imported. Scala syntax can be used

```
import java.swing.{JFrame, JWindow}
```

Main Method

Your programs entry point is defined in an scala file using an object, with a single method, main:

```
object Application {
  def main(args: Array[String]): Unit = {
    stuff goes here.
  }
}
```

Files can contain multiple classes and objects. Compile with scalac

Input and output

To read a file line by line

```
import scala.io.Source
for(line <- Source.fromFile("myfile.txt").getLines())
  println(line)
```

To write a file use Java's PrintWriter

```
val writer = new PrintWriter("myfile.txt")
writer.write("Writing line for line" + util.Properties.lineSeparator)
writer.write("Another line here" + util.Properties.lineSeparator)
```

```
writer.close()
```