Scala - the scalable language

```
/*
  Set yourself up:

  1) Download Scala - http://www.scala-lang.org/downloads
  2) Unzip/untar to your favourite location and put the bin subdir in your `PATH` environment variable
  3) Start a Scala REPL by running `scala`. You should see the prompt:

  scala>

  This is the so called REPL (Read-Eval-Print Loop). You may type any Scala
  expression, and the result will be printed. We will explain what Scala files
  look like further into this tutorial, but for now, let's start with some
  basics.

*/


////////////////////////////////////////////////
// 1. Basics
////////////////////////////////////////////////

// Single-line comments start with two forward slashes

/*
  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)
// Hello world!
// 10

// Printing, without forcing a new line on next print
print("Hello world")
print(10)
// Hello world!10

// 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

// Boolean operations
!true         // false
!false        // true
true == false // false
10 > 5        // true

// 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.
*/

"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")

// They also have some extra Scala methods. See also: scala.collection.immutable.StringOps
"hello world".take(5)
```

```
"hello world".drop(5)

// String interpolation: notice the prefix "s"
val n = 45
s"We have $n apples" // => "We have 45 apples"

// Expressions inside interpolated strings are also possible
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 apple
s"Power of 2: ${math.pow(2, 2)}"                      // => "Power of 2: 4"

// Formatting with interpolated strings with the prefix "f"
f"Power of 5: ${math.pow(5, 2)}%1.0f"         // "Power of 5: 25"
f"Square root of 122: ${math.sqrt(122)}%1.4f" // "Square root of 122: 11.0454"

// Raw strings, ignoring special characters.
raw"New line feed: \n. Carriage return: \r." // => "New line feed: \n. Carriage return: \r."

// Some characters need to be "escaped", e.g. a double quote inside a string:
"They stood outside the \"Rose and Crown\"" // => "They stood outside the "Rose and Crown""

// 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>"""


/////////////////////////////////////////////////
// 2. Functions
/////////////////////////////////////////////////

// Functions are defined like so:
//
//   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)  // => 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


// The return keyword exists in Scala, but it only returns from the inner-most
// def that surrounds it.
// WARNING: Using return in Scala is error-prone and should be avoided.
// It has no effect on anonymous functions. For example:
def foo(x: Int): Int = {
  val anonFunc: Int => Int = { z =>
    if (z > 5)
      return z // This line makes z the return value of foo!
    else
      z + 2    // This line is the return value of anonFunc
  }
  anonFunc(x)  // This line is the return value of foo
}


//////////////////////////////////////////////////
// 3. Flow Control
```

```
////////////////////////////////////////////////

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

r foreach println
// NB: 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 loops
var i = 0
while (i < 10) { println("i " + i); i += 1 }

while (i < 10) { println("i " + i); i += 1 }   // Yes, again. What happened? Why?

i    // Show the value of i. Note that while is a loop in the classical sense -
     // it executes sequentially while changing the loop variable. while is very
     // fast, faster that Java loops, but using the combinators and
     // comprehensions above is easier to understand and parallelize

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

// Tail recursion is an idiomatic way of doing recurring things in Scala.
// Recursive functions need an explicit return type, the compiler can't infer it.
// Here it's Unit.
def showNumbersInRange(a: Int, b: Int): Unit = {
  print(a)
  if (a < b)
    showNumbersInRange(a + 1, b)
}
showNumbersInRange(1, 14)


// 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"
```

```
/////////////////////////////////////////////
// 4. Data Structures
/////////////////////////////////////////////

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
 * and make sure you can read it
 */


// Tuples

(1, 2)

(4, 3, 2)

(1, 2, "three")

(a, 2, "three")

// Why have this?
val divideInts = (x: Int, y: Int) => (x / y, x % y)

// The function divideInts gives you the result and the remainder
divideInts(10, 3)    // (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


////////////////////////////////////////////////
// 5. Object Oriented Programming
////////////////////////////////////////////////

/*
  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
  - traits

  And now we will explain what these are.
*/

// 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) {
  // Constructor code here
  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. If we uncomment the next
  // line, class Dog would need to be declared abstract
  //   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 behavior. 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")

// And many others. Case classes also get pattern matching for free, see below.


// Traits coming soon!


///////////////////////////////////////////////
// 6. Pattern Matching
///////////////////////////////////////////////

// Pattern matching is a powerful and commonly used feature in Scala. Here's how
// you pattern match a case class. NB: Unlike other languages, Scala cases do
// not need breaks, fall-through does not happen.

def matchPerson(person: Person): String = person match {
  // Then you specify the patterns:
  case Person("George", number) => "We found George! His number is " + number
  case Person("Kate", number)   => "We found Kate! Her number is " + number
  case Person(name, number)     => "We matched someone : " + name + ", phone : " + number
}

val email = "(.*)@(.*)".r  // Define a regex for the next example.

// Pattern matching might look familiar to the switch statements in the C family
// of languages, but this is much more powerful. In Scala, you can match much
```

```scala
// more:
def matchEverything(obj: Any): String = obj match {
  // You can match values:
  case "Hello world" => "Got the string Hello world"

  // You can match by type:
  case x: Double => "Got a Double: " + x

  // You can specify conditions:
  case x: Int if x > 10000 => "Got a pretty big number!"

  // You can match case classes as before:
  case Person(name, number) => s"Got contact info for $name!"

  // You can match regular expressions:
  case email(name, domain) => s"Got email address $name@$domain"

  // You can match tuples:
  case (a: Int, b: Double, c: String) => s"Got a tuple: $a, $b, $c"

  // You can match data structures:
  case List(1, b, c) => s"Got a list with three elements and starts with 1: 1, $b, $c"

  // You can nest patterns:
  case List(List((1, 2, "YAY"))) => "Got a list of list of tuple"

  // Match any case (default) if all previous haven't matched
  case _ => "Got unknown object"
}

// In fact, you can pattern match any object with an "unapply" method. This
// feature is so powerful that Scala lets you define whole functions as
// patterns:
val patternFunc: Person => String = {
  case Person("George", number) => s"George's number: $number"
  case Person(name, number) => s"Random person's number: $number"
}


////////////////////////////////////////////////
// 7. Functional Programming
////////////////////////////////////////////////

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

val add10: Int => Int = _ + 10 // A function taking an Int and returning an Int
List(1, 2, 3) map add10 // List(11, 12, 13) - add10 is applied to each element

// 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

s.map(sq)

val sSquared = s. map(sq)

sSquared.filter(_ < 10)

sSquared.reduce (_+_)

// 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) // List(3)
case class Person(name: String, age: Int)
List(
  Person(name = "Dom", age = 23),
  Person(name = "Bob", age = 30)
).filter(_.age > 25) // List(Person("Bob", 30))


// Scala a foreach method defined on certain collections that takes a type
// returning Unit (a void method)
val aListOfNumbers = List(1, 2, 3, 4, 10, 20, 100)
aListOfNumbers foreach (x => println(x))
aListOfNumbers foreach println

// For comprehensions

for { n <- s } yield sq(n)

val nSquared2 = for { n <- s } yield sq(n)

for { n <- nSquared2 if n < 10 } yield n

for { n <- s; nSquared = n * n if nSquared < 10} yield nSquared

/* NB Those were not for loops. The semantics of a for loop is 'repeat', whereas
   a for-comprehension defines a relationship between two sets of data. */


//////////////////////////////////////////////
// 8. Implicits
//////////////////////////////////////////////

/* WARNING WARNING: Implicits are a set of powerful features of Scala, and
 * therefore it is easy to abuse them. Beginners to Scala should resist the
 * temptation to use them until they understand not only how they work, but also
```

```
 * best practices around them. We only include this section in the tutorial
 * because they are so commonplace in Scala libraries that it is impossible to
 * do anything meaningful without using a library that has implicits. This is
 * meant for you to understand and work with implicts, not declare your own.
 */

// Any value (vals, functions, objects, etc) can be declared to be implicit by
// using the, you guessed it, "implicit" keyword. Note we are using the Dog
// class from section 5 in these examples.
implicit val myImplicitInt = 100
implicit def myImplicitFunction(breed: String) = new Dog("Golden " + breed)

// By itself, implicit keyword doesn't change the behavior of the value, so
// above values can be used as usual.
myImplicitInt + 2                   // => 102
myImplicitFunction("Pitbull").breed // => "Golden Pitbull"

// The difference is that these values are now eligible to be used when another
// piece of code "needs" an implicit value. One such situation is implicit
// function arguments:
def sendGreetings(toWhom: String)(implicit howMany: Int) =
  s"Hello $toWhom, $howMany blessings to you and yours!"

// If we supply a value for "howMany", the function behaves as usual
sendGreetings("John")(1000)  // => "Hello John, 1000 blessings to you and yours!"

// But if we omit the implicit parameter, an implicit value of the same type is
// used, in this case, "myImplicitInt":
sendGreetings("Jane")  // => "Hello Jane, 100 blessings to you and yours!"

// Implicit function parameters enable us to simulate type classes in other
// functional languages. It is so often used that it gets its own shorthand. The
// following two lines mean the same thing:
// def foo[T](implicit c: C[T]) = ...
// def foo[T : C] = ...


// Another situation in which the compiler looks for an implicit is if you have
//   obj.method(...)
// but "obj" doesn't have "method" as a method. In this case, if there is an
// implicit conversion of type A => B, where A is the type of obj, and B has a
// method called "method", that conversion is applied. So having
// myImplicitFunction above in scope, we can say:
"Retriever".breed // => "Golden Retriever"
"Sheperd".bark    // => "Woof, woof!"

// Here the String is first converted to Dog using our function above, and then
// the appropriate method is called. This is an extremely powerful feature, but
// again, it is not to be used lightly. In fact, when you defined the implicit
// function above, your compiler should have given you a warning, that you
// shouldn't do this unless you really know what you're doing.


/////////////////////////////////////////////////
```

```scala
// 9. Misc
//////////////////////////////////////////////

// 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}

// Import all classes, except some. The following excludes Map and Set:
import scala.collection.immutable.{Map => _, Set => _, _}

// Java classes can also be imported. Scala syntax can be used
import java.swing.{JFrame, JWindow}

// 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()
```

Further resources

- Scala for the impatient
- Twitter Scala school
- The scala documentation
- Try Scala in your browser

- Join the Scala user group