Scala Tutorial - Quick Reference

This tutorial is a quick reference material for those who are starting or need to refresh their Scala skills.

We strongly recommend you use IntelliJ IDEA.

Create a new source file `src/main/scala/HelloWorld.scala` with the following content:

```scala
@main def helloWorld(): Unit = println("Hello World!")
```

Compile and run the project in IntelliJ by pressing the green arrow next to the @main annotation

Now that you know how to execute your code, we will refresh a bit of the Scala language and standard API.

- Scala is object-oriented, functional, and statically typed.
- Scala runs on the JVM and interoperates with Java.
- Scala offers easy parallelism through immutable data structures.
- In Scala, all types are objects, including functions and primitive types.

Values and variables

Values (`val`) are immutable, whereas variables (`var`) are mutable.

Syntax:

```scala
val x : Int = 3
var y : Int = 4
```

Scala supports type inference, so that types can be omitted:

```scala
val institution = "EPFL" // immutable, institution is inferred to be a String
institution = "ETH" // error
var i = 0 // mutable, inferred to be Int
i = 1 // ok
val (name, age, height) = ("John Doe", 25, 1.75)
```

Functions

Syntax:

```scala
def functionName(param1: Type, param2: Type, ...): ReturnType = expression
```

For instance:

```scala
def square(a: Int): Int = a * a
def fact(x: Int): Int = if (x < 2) 1 else x * fact(x - 1)
```

Scala supports type inference for the returned value. You can omit the return type when the function is not recursive:

```
def square(a: Int) = a * a   // OK
def fact(x: Int) = if (x < 2) 1 else x * fact(x - 1)
// error: recursive method fact needs result type
```

Larger blocks of code are generally enclosed by curly brackets

```
def quadRoots(a: Float, b: Float, c: Float) = { // with brackets
  val deltaRoot = math.sqrt(b * b - 4 * a * c)
  ((-b + deltaRoot) / (2 * a), (-b - deltaRoot) / (2 * a))
}
```

If there is no returned value, you can use the `Unit` type (equivalent to `void` in Java or C):

```
def nicePrint(str: String): Unit = println(s">>> ${str} <<<")
```

Notice the 's' in front of the string literal, which enables string interpolation. This is similar to Python's f-strings.

Nested functions: you can declare functions inside functions.

```
def factorial(i: Int): Int = {
  def fact(i: Int, accumulator: Int): Int = {
    if (i < 2)
      accumulator
    else
      fact(i - 1, i * accumulator)
  }
  fact(i, 1)
}
```

Unlike the previous `fact` implementation, this one is tail recursive, i.e., it executes in constant stack space because the same amount of stack memory is used across the recursive calls.

Higher order functions

Higher order functions are functions that take other functions as arguments or that return a function as a result.

For instance, `sum` computes the sum of all values between `from` and `to` after applying an arbitrary function `f` to each value.

```
def sum(from: Int, to: Int, f: Int => Int): Int = {
  if (from > to) 0 else f(from) + sum(from + 1, to, f)
}
def sumOfSquares(from: Int, to: Int) = sum(from, to, square)
def sumOfFactorials(from: Int, to: Int) = sum(from, to, factorial)
```

Anonymous functions can be passed as arguments as well:

```
sum(1, 5, x => x * x * x)        // Int = 225
```

Classes and objects

- A `Class` can be defined in one line. Its attributes may have default values, and it can be instantiated with `new`.

```
class Point(val x: Int = 0, val y: Int = 0)
val origin = new Point   // Point = Point@129a3c4b
```

By default, `var` or `val` attributes are `public`, but they can be explicitly set to `private`. The default `toString` method prints a hash of the object. If one wants a nicer print, it must be overridden.

```
class Point(val x: Int = 0, val y: Int = 0) {
  override def toString: String = s"($x, $y)"
}
val origin = new Point   // Point = (0, 0)
```

- Singleton objects have only one instance, and are defined by the keyword `object`.

```
object Logger {
  def info(message: String) = println(s"INFO: $message")
}
Logger.info("running...")  // INFO: running...
```

- Companion objects are singleton objects with the same name as a corresponding class. They both can access all members of the companion.

```
class Circle(val radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}
object Circle {
  private def calculateArea(r: Double): Double = 3.1415 * r * r
}
val circle = new Circle(5.0)
circle.area  // Double = 78.53750000000001
```

Case classes are like normal classes. The difference is that all attributes are `public val` by default, and you do not need to use the `new` operator to instantiate them. They are convenient for immutable structures and pattern matching. They also override toString by default. They cannot be extended, as they are implicitly final.

```
case class Person(name: String, age: Int)
val someone = Person("John Doe", 25)
```

```scala
val personString = someone.toString
println(personString) // prints Person("John Doe", 25)
```

Pattern matching

Pattern matching is somewhat similar to `switch/case` constructions in Java/C, but more powerful. It checks whether a value `matches` (keyword `match`) some pattern and executes the corresponding `case` code.

```scala
import scala.util.Random
val x: Int = Random.nextInt(10)

x match {
  case e if e % 2 == 0 => "even"
  case _ => "odd"
}
```

Enums are especially useful for pattern matching:

```scala
enum Notification {
  // equivalent to an abstract class (Notification) extended by two case classes.
  // the scala compiler also knows that the two subclasses are *exhaustive*,
  // there is no other subclass of Notification.
  case Email(sender: String, title: String, body: String)
  case SMS(caller: String, message: String)

  // you can define methods in enums, just like Java abstract classes
  def extractSource: String = {
    this match {
      case Email(sender, _, _) => sender
      case SMS(caller, _) => caller
    }
  }
}

def notifyAbout(notification: Notification): String =
  notification match {
    case Email(sender, title, _) => s"You got an email from $sender with title: $title"
    case SMS(number, _) => s"You got an SMS from $number!"
  }

val someSms = SMS("12345", "Are you there?")
val someEmail = Email("alice@wonderland.com", "meeting", "Let's meet?")
println(notifyAbout(someSms))
println(notifyAbout(someEmail))
println(someEmail.extractSource)
```

Collections

- We will look into a few Scala collections:
  - Array, Vector, List, Set, Map, Tuples

At the same time, we will use some common higher-order functions to manipulate them:

- Transformations: **map**, **flatMap**
- Partitioning: **groupBy**
- Reduction: **reduce**
- Combination: **zip**
- Filtering: **filter**

Array and Vector

Arrays (API) are mutable, indexed collections of values, whereas Vectors (API) are immutable. They provide random access and updates in constant time.

```
val numbers = Array(1, 2, 3, 4)
val first = numbers(0) // read the first element
numbers(3) = 100       // replace the 4th array element with 100.
```

map is a higher-order function that *maps* each element of a collection into another, possibly of a different type. Here is its signature in the Array class:

```
def map[B](f: (A) => B): Array[B]
```

Now, suppose that we want a collection with the cubes of our Array. We could obtain it like this:

```
numbers.map(x => x * x * x) // Array[Int] = Array(1, 8, 27, 1000000)
```

---

filter selects all elements that satisfy a predicate provided as an argument. Here is its signature in the Vector class:

```
def filter(p: A => Boolean): Vector[A]
```

We could filter the prime numbers from a collection like this:

```
def isPrime(n: Int): Boolean =
  !((2 until n-1) exists (n % _ == 0))
(2 to 50).filter(isPrime) // Vector(2, 3, ..., 43, 47)
```

---

reduce is a higher-order function that applies a pair-wise associative operation to all elements of a collection and returns a single value. Here is its signature:

```
def reduce[A](op: (A, A) => A): A
```

Let's say now that we want to obtain the multiplication among all the elements in the **numbers** collection:

```
numbers.reduce((a, b) => a * b) // Int = 600
```

Here is another way of defining **sumOfSquares** that was shown before:

```
def sumOfSquares(from: Int, to: Int) =
  (from to to).map(x => x * x).reduce(_ + _)
```

List

Lists represent ordered collections (API). They have O(1) time complexity for prepend and head/tail access, whereas the other operations are usually O(n) in the number of elements in the list.

`Nil` is the empty List. `::` is a right-associative operator that prepends an element to the List.

```
val mainList = List(3, 2, 1)
val another = 3 :: 2 :: 1 :: Nil
mainList eq another            // Boolean = false. They are different objects
mainList == another            // Boolean = true. They contain the same things
val with4 = 4 :: mainList      // re-uses immutable mainList.
val shorter = mainList.tail    // does not alloc anything
mainList.reverse               // O(n) time complexity
```

flatMap, like **map**, iterates through all the elements of a collection and *maps* them into something else. The difference is that **flatMap** may generate collections for each element, and the result will be flattened, i.e., instead of a List of lists, the result of **flatMap** will be a flat List containing all the elements of the mapped collections. Here is its signature in the List class:

```
final def flatMap[B](f: A => GenTraversableOnce[B]): List[B]
```

Suppose that we have three lines of a file within a List. **flatMap** could be used to obtain a single List with all the words in the file.

```
val line1 = "Let it snow, let it snow, let it snow."
val line2 = "Do or do not. There is no try."
val line3 = "A horse is a horse, of course, of course."
val file = List(line1, line2, line3)
val wordsInFile = file.flatMap(line => line.toLowerCase.split("\W+"))
```

Set

A **Set** is a collection that contains no duplicate elements (API).

```scala
val myset = Set(1, 2, 2, 3, 3, 3) // scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

If we want to find out how many unique words we have in our file example, we could do:

```scala
wordsInFile.toSet.size // Int = 14
```

Tuples

Tuples (API) are immutable values with a fixed number of elements (up to 22), each with their own type. Each individual element is named _1, _2, _3, and so on. Case classes are an alternative to tuples if you prefer more meaningful names.

```scala
val triplet = ("John Doe", 25, 1.75)
println(s"${triplet._1} is ${triplet._2} years old and ${triplet._3}m tall")
```

zip merges a collection with another by combining their elements in pairs (i.e., tuples of two elements). Its signature in the List class is:

```scala
def zip[B](that: Iterable[B]): List[(A, B)]
```

For example,

```scala
val names = List("Alice", "Bob", "Charlie", "Dave", "Eve")
val grades = List(6, 4.5, 5.25, 5.5, 4)
val results = names zip grades
// List((Alice,6.0), (Bob,4.5),
// (Charlie,5.25), (Dave,5.5), (Eve,4.0))
```

If one of the lists is longer than the other, the extra elements will be ignored.

Map

A **Map** (API) maps keys into values.

```scala
val resultsMap = results.toMap
resultsMap("Bob") // Double = 4.5
```

groupBy partitions a collection into a map of collections according to a discriminator function. Here is its signature in the List class:

```scala
def groupBy[K](f: (A) => K): Map[K, List[A]]
```

Using the identity function as a discriminator gives a map where the keys are unique elements of the original collection and values are collections containing all the occurrences for each given key.

```scala
val groupedWords = wordsInFile.groupBy(x => x)
```

Loops

Like Java, you can use **while**, **do while**, and **for**.

```
val letters = ('a' to 'e').toVector
var i = 0
while (i < letters.size) {
  println(letters(i))
  i = i + 1
}

do {
  i = i - 1
  println(letters(i))
} while(i > 0)

for (l <- letters) println(l)
```

For comprehensions offer a concise notation to what would be *nested loops* in imperative languages. They have the form `for (enumerator1; enumerator2; ... [filter]) yield expression` and result in a collection of elements.

Additional Resources

- Scala API
- Scala docs
- Sbt docs
- Coursera series of courses about Functional Programming in Scala. It is possible to audit the courses for free if you do not need/want a certificate.