SCALA CHEATSHEET

SCALA CHEATSHEET

Thanks to Brendan O'Connor, this cheatsheet aims to be a quick reference of Scala syntactic constructions. Licensed by Brendan O'Connor under a CC-BY-SA 3.0 license.

variables

`var x = 5`	Variable.
GOOD `x = 6`	
`val x = 5`	Constant.
BAD `x = 6`	

functions

GOOD `def f(x: Int) = { x * x }` **BAD** `def f(x: Int)   { x * x }`	Define function. Hidden error: without `=` it's a procedure returning `Unit` ; causes havoc. Deprecated in Scala 2.13.
GOOD `def f(x: Any) = println(x)` **BAD** `def f(x) = println(x)`	Define function. Syntax error: need types for every arg.
`type R = Double`	Type alias.
`def f(x: R)` vs. `def f(x: => R)`	Call-by-value. Call-by-name (lazy parameters).
`(x: R) => x * x`	Anonymous function.
`(1 to 5).map(_ * 2)` vs. `(1 to 5).reduceLeft(_ + _)`	Anonymous function: underscore is positionally matched arg.
`(1 to 5).map(x => x * x)`	Anonymous function: to use an arg twice, have to name it.
`(1 to 5).map { x =>` `  val y = x * 2` `  println(y)` `  y`	Anonymous function: block style returns last expression.

`  _ % 2 == 0` `} map {` `  _ * 2` `}`	Anonymous functions: pipeline style (or parens too).
`def compose(g: R => R, h: R => R) =` `  (x: R) => g(h(x))` `val f = compose(_ * 2, _ - 1)`	Anonymous functions: to pass in multiple blocks, need outer parens.
`val zscore =` `  (mean: R, sd: R) =>` `  (x: R) =>` `  (x - mean) / sd`	Currying, obvious syntax.
`def zscore(mean: R, sd: R) =` `  (x: R) =>` `  (x - mean) / sd`	Currying, obvious syntax.
`def zscore(mean: R, sd: R)(x: R) =` `  (x - mean) / sd`	Currying, sugar syntax. But then:
`val normer =` `  zscore(7, 0.4) _`	Need trailing underscore to get the partial, only for the sugar version.
`def mapmake[T](g: T => T)(seq: List[T]) =` `  seq.map(g)`	Generic type.
`5.+(3); 5 + 3` `(1 to 5) map (_ * 2)`	Infix sugar.
`def sum(args: Int*) =` `  args.reduceLeft(_+_)`	Varargs.

packages

`import scala.collection.{Vector, Sequence}`	Selective import.
`import scala.collection.{Vector => Vec28}`	Renaming import.
`import java.util.{Date => _, _}`	Import all from `java.util` except `Date`.
At start of file: `package pkg` *Packaging by scope:* `package pkg {` `  ...` `}` *Package singleton:* `package object pkg {` `  ...` `}`	Declare a package.

data structures

`(1, 2, 3)`	Tuple literal (`Tuple3`).
`var (x, y, z) = (1, 2, 3)`	Destructuring bind: tuple unpacking via pattern matching.
BAD `var x, y, z = (1, 2, 3)`	Hidden error: each assigned to the entire tuple.
`var xs = List(1, 2, 3)`	List (immutable).

`1 to 5` *same as* `1 until 6` `1 to 10 by 2`	Range sugar.
`()`	Empty parens is singleton value of the Unit type. Equivalent to `void` in C and Java.

control constructs

`if (check) happy else sad`	Conditional.
`if (check) happy` *same as* `if (check) happy else ()`	Conditional sugar.
`while (x < 5) {` `  println(x)` `  x += 1` `}`	While loop.
`do {` `  println(x)` `  x += 1` `} while (x < 5)`	Do-while loop.
`import scala.util.control.Breaks._` `breakable {` `  for (x <- xs) {` `  if (Math.random < 0.1)`	Break (slides).

`` for (x <- xs if x % 2 == 0) yield x * 10 same as xs.filter(_ % 2 == 0).map(_ * 10) ```	For-comprehension: filter/map.
`` for ((x, y) <- xs zip ys) yield x * y same as (xs zip ys) map { case (x, y) => x * y } ```	For-comprehension: destructuring bind.
`` for (x <- xs; y <- ys) yield x * y same as xs flatMap { x => ys map { y => x * y } } ```	For-comprehension: cross product.
`` for (x <- xs; y <- ys) { val div = x / y.toFloat println("%d/%d = %.1f".format(x, y, div)) } ```	For-comprehension: imperative-ish. `sprintf` style.
`` for (i <- 1 to 5) { println(i) } ```	For-comprehension: iterate including the upper bound.
`` for (i <- 1 until 5) { println(i) } ```	For-comprehension: iterate omitting the upper bound.

GOOD

```
(xs zip ys) map {
  case (x, y) => x * y
}
```

BAD

```
(xs zip ys) map {
  (x, y) => x * y
}
```

Use case in function args for pattern matching.

BAD

```
val v42 = 42
3 match {
  case v42 => println("42")
  case _   => println("Not 42")
}
```

`v42` is interpreted as a name matching any Int value, and "42" is printed.

GOOD

```
val v42 = 42
3 match {
  case `v42` => println("42")
  case _     => println("Not 42")
}
```

`` `v42` `` with backticks is interpreted as the existing val `v42`, and "Not 42" is printed.

GOOD

```
val UppercaseVal = 42
3 match {
  case UppercaseVal => println("42")
  case _            => println("Not 42")
}
```

`UppercaseVal` is treated as an existing val, rather than a new pattern variable, because it starts with an uppercase letter. Thus, the value contained within `UppercaseVal` is checked against `3`, and "Not 42" is printed.

`class C(x: R)`	Constructor params - `x` is only available in class body.
`class C(val x: R)` `var c = new C(4)` `c.x`	Constructor params - automatic public member defined.
`class C(var x: R) {` `  assert(x > 0, "positive please")` `  var y = x` `  val readonly = 5` `  private var secret = 1` `  def this = this(42)` `}`	Constructor is class body. Declare a public member. Declare a gettable but not settable member. Declare a private member. Alternative constructor.
`new {` `  ...` `}`	Anonymous class.
`abstract class D { ... }`	Define an abstract class (non-createable).
`class C extends D { ... }`	Define an inherited class.
`class D(var x: R)` `class C(x: R) extends D(x)`	Inheritance and constructor params (wishlist: automatically pass-up params by default).
`object O extends D { ... }`	Define a singleton (module-like).

	constructor params.
`class C extends D with T { ... }`	mixin-able.
`trait T1; trait T2`	
`class C extends T1 with T2`	Multiple traits.
`class C extends D with T1 with T2`	
`class C extends D { override def f = ...}`	Must declare method overrides.
`new java.io.File("f")`	Create object.
BAD `new List[Int]` **GOOD** `List(1, 2, 3)`	Type error: abstract type. Instead, convention: callable factory shadowing the type.
`classOf[String]`	Class literal.
`x.isInstanceOf[String]`	Type check (runtime).
`x.asInstanceOf[String]`	Type cast (runtime).
`x: String`	Ascription (compile time).

options

`Some(42)`	Construct a non empty optional value.
`None`	The singleton empty optional value.

`Some(null) != None`	
`val optStr: Option[String] = None` ***same as*** `val optStr = Option.empty[String]`	Explicit type for empty optional value. Factory for empty optional value.
`val name: Option[String] =` `  request.getParameter("name")` `val upper = name.map {` `  _.trim` `} filter {` `  _.length != 0` `} map {` `  _.toUpperCase` `}` `println(upper.getOrElse(""))`	Pipeline style.
`val upper = for {` `  name <- request.getParameter("name")` `  trimmed <- Some(name.trim)` `  if trimmed.length != 0` `  upper <- Some(trimmed.toUpperCase)` `} yield upper` `println(upper.getOrElse(""))`	For-comprehension syntax.
`option.map(f(_))` ***same as*** `option match {` `  case Some(x) => Some(f(x))` `  case None    => None` `}`	Apply a function on the optional value.
`option.flatMap(f(_))` ***same as*** `option match {` `  case Some(x) => f(x)`	Same as map but function must return an optional value.

`optionOfOption.flatten` ***same as*** `optionOfOption match {` `  case Some(Some(x)) => Some(x)` `  case _ => None` `}`	Extract nested option.
`option.foreach(f(_))` ***same as*** `option match {` `  case Some(x) => f(x)` `  case None => ()` `}`	Apply a procedure on optional value.
`option.fold(y)(f(_))` ***same as*** `option match {` `  case Some(x) => f(x)` `  case None => y` `}`	Apply function on optional value, return default if empty.
`option.collect {` `  case x => ...` `}` ***same as*** `option match {` `  case Some(x) if f.isDefinedAt(x) => ...` `  case Some(_) => None` `  case None => None` `}`	Apply partial pattern match on optional value.
`option.isDefined` ***same as*** `option match {` `  case Some(_) => true` `  case None => false` `}`	`true` if not empty.

`case Some(_) => false` `case None => true` `}`	`true` if empty.
`option.nonEmpty` ***same as*** `option match {` `case Some(_) => true` `case None => false` `}`	`true` if not empty.
`option.size` ***same as*** `option match {` `case Some(_) => 1` `case None => 0` `}`	`0` if empty, otherwise `1`.
`option.orElse(Some(y))` ***same as*** `option match {` `case Some(x) => Some(x)` `case None => Some(y)` `}`	Evaluate and return alternate optional value if empty.
`option.getOrElse(y)` ***same as*** `option match {` `case Some(x) => x` `case None => y` `}`	Evaluate and return default value if empty.
`option.get` ***same as*** `option match {` `case Some(x) => x` `case None => throw new Exception`	Return value, throw exception if empty.

same as `option match {` `  case Some(x) => x` `  case None    => null` `}`	Return value, `null` if empty.
`option.filter(f)` *same as* `option match {` `  case Some(x) if f(x) => Some(x)` `  case _              => None` `}`	Optional value satisfies predicate.
`option.filterNot(f(_))` *same as* `option match {` `  case Some(x) if !f(x) => Some(x)` `  case _               => None` `}`	Optional value doesn't satisfy predicate.
`option.exists(f(_))` *same as* `option match {` `  case Some(x) if f(x) => true` `  case Some(_)         => false` `  case None            => false` `}`	Apply predicate on optional value or `false` if empty.
`option.forall(f(_))` *same as* `option match {` `  case Some(x) if f(x) => true` `  case Some(_)         => false` `  case None            => true` `}`	Apply predicate on optional value or `true` if empty.

```
  case Some(x) => x == y
  case None    => false
}
```

optional value or `false` if empty.