SCALA CHEATSHEET

SCALACHEAT

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 Unit-returning procedure; causes havoc

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.

GOOD
`(1 to 5).map(2 *)`
 	
BAD
`(1 to 5).map(* 2)`
	anonymous function: bound infix method.
	Use `2 * _` for sanity's sake instead.

`  y` `}`	expression.

`(1 to 5) filter {`
`  _ % 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.

packages

`import scala.collection._`	wildcard import.

`import scala.collection.Vector`
 	
`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.

`var xs = List(1, 2, 3)`	list (immutable).
`xs(2)`	paren indexing. (slides)
`1 :: List(2, 3)`	cons.

`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.

`if (Math.random < 0.1)`
`  break`
`}`
`}`
	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

pattern matching

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`

object orientation

`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)

`trait T { ... }`
 	
`class C extends T { ... }`
 	
`class C extends D with T { ... }`
	traits.
	interfaces-with-implementation. no constructor params. mixin-able.

`trait T1; trait T2`
 	
`class C extends T1 with T2`
 	
`class C extends D with T1 with T2`
	multiple traits.

`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

`Option(null) == None` `Option(obj.unsafeMethod)`	Null-safe optional value factory

`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

`case Some(x) => f(x)`
`case None => None`
`}`
	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

same as
`option match {`
`  case Some(_) => true`
`  case None => false`
`}`
	True if not empty

`option.isEmpty`
same as
`option match {`
`  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`
`}`
	Zero if empty, otherwise one

`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

same as
`option match {`
`  case Some(x) => x`
`  case None    => throw new Exception`
`}`
	Return value, throw exception if empty

`option.orNull`
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 _               => false`
`}`
	Apply predicate on optional value or false if empty

`option.forall(f(_))`
same as
`option match {`
	Apply predicate on optional value or true if empty

`option.contains(y)`
same as
`option match {`
`  case Some(x) => x == y`
`  case None    => false`
`}`
	Checks if value equals optional value or false if empty

Contributors to this page:

 ashawley  kotobotov  iphayao  heathermiller

DOCUMENTATION

Getting Started

API

Overviews/Guides

Language Specification

DOWNLOAD

Current Version

All versions

COMMUNITY

Community

Mailing Lists

Chat Rooms & More

Libraries and Tools

The Scala Center

CONTRIBUTE

How to help

Report an Issue

SCALA

Blog

Code of Conduct

License

SOCIAL

GitHub

Twitter