

Scala 2.7.x

Cheat sheet by Stefan Maetschke
V 0.9, 12.11.2009

predef

Predefined types and methods that do not need to be imported

// these methods throw RuntimeExceptions	
error("message")	error message
assert(x>0), assert(pred, msg)	assertion
require(x>0); require(pred, msg)	precondition
assume(x>0); assume(pred, msg)	precondition

```
// print out
print(x:Any)
println(x:Any); println:Unit
printf(format:String, xs:Any*)
print(x:Any)
```

```
// reading formatted
readf(format:String):List[Any]
readf1(format:String):Any
readf2(format:String):(Any,Any)
readf3(format:String):(Any,Any,Any)
```

```
// reading different types
val x = readInt
val y = readFloat
val str = readLine
```

variables/constants

scala foo.scala	run scala file
var x = 10	variable value
val x = 10	constant value
val x:Int = 10	with type
val x,y,z = 10	multi bind
lazy val list = List(1,2,3)	lazy initialization

dot/operator notation

```
1+2 == 1.+(2)
-2 == (2).unary_-
a.max(b) == a max b
s.indexOf(0, 'c') = s indexOf (0, 'c')
```

import

imports can appear anywhere and can refer to objects as well.
implicitly imported are: java.lang._, scala._ and Predef._

import java.text._	import all
import java.util.{Date,Timer}	import selection
import java.util.{Date=>UDate}	import class as
import java.{util=>U}	import package as
import java.util.{Date=>_, _}	import all but Date

val folder = new File("Maet/data")	create file obj
import folder._	import this obj

if(exists) println(listFiles)	use obj methods
-------------------------------	-----------------

package

package com.get.rich	Java style
package com {	Nested packages
package get {	
package rich {}	
}	
}	

type

type T = Int	type declaration
--------------	------------------

interpreter / compiler

scala foo.scala	run scala file
scala foo	run .class file
scalac foo.scala bar.scala	compile scala files
fsc foo.scala bar.scala	fast compiler
fsc -shutdown	stop fast compiler

control structures

if(cond) {doThis} else {doThat}	if
for(i <- 1 to 10) println(i)	for
while(cond) {doThis}	while
do {doThis} while(cond)	do-while
import Breaks.{break, breakable}	break
breakable {	
for (...) {	
if (...) break	
}	
}	

for

for(i <- 1 to 10) println(i)	for loop
for(i <- 1 to 10; j <- 1 to 10)	nested for loop
println((i,j))	
for(i <- 1 until 10 if i%2==0)	with guard
println(i)	
list.foreach(x => println(x))	for-each loop
list.foreach(println(_))	
list.foreach(println)	

for comprehension

return type is of the same type as the enumerator used

```
for (i <- List.range(1, 10) ) yield i*i           ret List
for (i <- 1 to 10; j <- 1 to 10) ) yield i*j
for (i <- List.range(1, 10) if i % 2 == 0) yield i
```

iterable methods

subclasse: Ranged, Collection {Seq,Map,Set,ArrayStack}	
iter.isEmpty	true if empty
iter1 ++ iter3	append
iter.copyToArray(arr, start)	fill array
iter.copyToBuffer(buffer)	fill buffer
iter.dropWhile(p(A)=>Boolean)	drop while not true
iter.takeWhile(p(A)=>Boolean)	take while true
iter.elements	new iter over elems

iter.exists(p(A)=>Boolean)	true if exists
iter.forall(p(A)=>Boolean)	true if true for all
iter.find(p(A)=>Boolean)	returns Option
iter.filter(p(A)=>Boolean)	keep all true
iter.reduceLeft(_*_)	reduce Left
iter.reduceRight(_*_)	reduce Right
iter.foldLeft(1)(_*_)	fold Left
iter.foldRight(1)(_*_)	fold Right
(1/:iter)(_*_)	fold Left
(iter:\1)(_*_)	fold Right
iter.map(f)	apply f
iter.flatMap(f)	apply f and concat
iter.foreach(f)	apply f return Unit
iter.mkString	string repr.
iter.mkString(sep)	with separator
iter.mkString(start,sep,end)	start,end strings
iter.addString(buf)	to StringBuilder.
iter.addString(buf, sep)	with separator
iter.addString(buf, start, sep,end)	start,end strings
iter.partition(p(A)=>Boolean)	ret. two iter
iter.toList	
iter.toSeq	
iter.toStream	

seq methods

superclass: Seq
subclasses: List, MutableList, SingleLinkedList, Queue, Stack, Buffer,...

seq.isEmpty	true if length==0
seq.length	length
seq.size	length
seq.first	first element
seq.last	last element
seq1 ++ seq2	append
seq.contains(elem):Boolean	
seq1.containsSlice(seq2)	contain slice
seq.take(n)	take first n
seq.takeWhile(p(A)=>Boolean)	take while true
seq.drop(n)	without first n
seq.dropWhile(p(A)=>Boolean)	drop while not true
seq.filter(p(A)=>Boolean)	keep all true
seq.startsWith(seq2)	
seq.endsWith(seq2)	
seq.map(f)	apply f
seq.flatMap(f)	apply f and concat
seq.indexOf(elem)	-1 if not found
seq1.indexOf(seq2)	-1 if not found
seq.lastIndexOf(elem)	-1 if not found
seq.projection	projection
seq.reverse	reverse
seq.slice(from,until)	from until
seq.toArray	append

projection

a projection is a generator derived from a collection

range

1 to 5	inclusive
1 until 5	exclusive
1 to 10 by 2	stride

string

Strings are immutable

```

"""multiple lines and raw"""
"""|Example of a string with
   |a stripped margin.""".stripMargin
"test\n".trim
str.split(' ')
str.split("\\s+")
str.matches("[A-Z]+")
str.endsWith(string)
str.startsWith(string)
str.substr(i)
str.substr(i,j)
"ab"*3

```

raw strings
trims white spaces
splits use char
split use regex
regular expression
string end
string start
i to end
i to j-1
ababab

function

```

def add(x:Int, y:Int):Int = {x+y}
val inc = add(_:Int, 1)
val add2 = add _
def incGen(a:Int) = (x:Int) => x+a
def add(a:Int)(b:Int) = a+b
(x:Int, y:Int) => x+y
def sum(xs:Int*) = xs.reduceLeft(_+_ )
def thunk()
def f(x: => Int)
def f(x(): => Int)

def foo(bar:(Int,Int)=>Int) = {bar(...)}
def everySecond(action:(String)=>Unit, text:String) {
  while(true) { action(text); Thread.sleep(1000) }
}

```

function definition
partially applied
partially applied
closure
currying
function literal
var. num. of args
thunk: no args func
lazy/by-name para
lazy with thunk
higher order func.

tuple

Tuples can contain elements of different types, index is one-based!

```

var t = ("pi", 3.14)
t._1
t._2

```

a tuple
first element
second element

list

Lists are immutable, no append for lists

```

val list = List(1,2,3,4,5)
val list = List.range(1,6)
list.length
list.head
list.last
list.mkString(",")
list.mkString("[", ":", "]")
list.count(_>2)
list.find(_>2)
list.filter(_>2)
list.reverse()
list.sort(_>_)

```

filled list
filled list
length of list
first element
last element
list to string
list to string
count
find
filter
reverse
sorting

list.partition(_>3)	part. into two lists
list1-list2	list difference
list1.diff(list2)	list difference
list1.union(list2)	list union
list1.intersect(list2)	list intersection
list.removeDuplicates	remove dups
list1:::list2	concatenate
elem::list	cons
list.zipWithIndex	enumerate
list.indices	list with indices
list.elements	iterator over elems
list.projection	lazy, generator

array

Arrays are mutable

val a = new Array[Int](10)	empty array
val a = Array(1,2,3,4)	filled array
val a = Array.range(1,5)	filled array
a(11)	element access

map

Maps can be mutable or immutable depending on import

```
import scala.collection.mutable.Map
import scala.collection.immutable.Map
```

val map = Map[String,String]	empty map
val map = Map(1->"I", 2->"II", 3->"III")	filled map
val map = Map((1,2),(2,3),(3,4))	filled map
val map = Map[Int,Int]().withDefaultValue(0)	

map += key -> value	add pair
map += (key,value)	add pair
map(key)	get value
map.get(key)	get value
map.getOrElse(key, default)	get val or def
map.getOrElseUpdate	
map.elements	itr over elems

set

Sets are immutable but mutable version exists

val set = Set(1,2,3,4)	filled set
------------------------	------------

stack

```
import scala.collection.mutable.Stack
val stack = new Stack[Int]()
stack.push(elems:A*)
stack += elem
stack.pop
stack.top
stack(index)
stack.length
stack.clear
stack.elements
stack1 += stack2
```

push
push
pop
no remove
getter
stack size
clears stack
iter over elems
pushes stack2 on 1

application

```
object MyApp {
  def main(args: Array[String]) {
    ...
  }
}

object MyApp extends Application {
  ...
}
```

only single threaded apps

class

Notes: constructor arguments, methods and class variables are transparent, e.g. x in myclass.x can be all of the three,

no "public" keyword but private, protected and override

object O(x:T)	singleton
class C(x:T)	x private
class C(private var x:T)	x private
class C(val x:T)	x public+getter
class C(var x:T)	x public+get+set

class B extends A {...}	extends
class B extends A with T {...}	extends with Trait
class B extends A with T1 with T2 {...}	extends with Traits

abstract class A { def method(): Int }	abstract class abstract method
--	-----------------------------------

class B(x:T) extends A(x)	call super construc.
---------------------------	----------------------

```
class C {
  private var a = "private"
  val b = "public_get"
  var c = "public_get_set"
}
```

```
class Table {
  // table of names
  private var names = new Array[String](10)
  // implements table(index)
  def apply(index:Int) = names(index)
  // implements table(index) = name
  def update(index:Int, name:String) =
    names(index) = name
}
```

```
case class Frac(val num:Int, val den:Int) {
  require(den > 0)
  var toRational = num/den
  def this(t:(Int,Int)) = this(t._1,t._2)
  def *(that:Frac) =
    Frac(this.num*that.num, this.den*that.den)
  def *(c:Int) = Frac(num*c, den)
  def toTuple = (num,den)
  override def toString = "%d/%d".format(num,den)
}
```

trait

Traits are essentially the same as classes with two differences. 1) constructor cannot have parameters, 2) invocation of class methods is stackable.

```
abstract class AbstractNumber extends
  Ordered[AbstractNumber] {
  def ensure(n:Int):Boolean
  def value:Int
  def compare(that:AbstractNumber) =
    this.value - that.value
}
trait Positive extends AbstractNumber {
  override abstract def ensure(n:Int) =
    (n > 0) && super.ensure(n)
}
trait Even extends AbstractNumber {
  override abstract def ensure(n:Int) =
    (n % 2 == 0) && super.ensure(n)
}
class Number(n:Int) extends AbstractNumber {
  assert(ensure(n))
  def ensure(n:Int) = true
  def value = n
}
class EvenPositive(n:Int) extends Number(n)
  with Positive with Even
```

match expression

Similar to switch in Java but more powerful

```
val number = str match {
  case "one" => 1
  case "two" => 2
  case _ => 0
}

List(1,2,3) match {
  case 1::tail => "one"
  case _::tail => "more"
  case Nil => "nothing"
}

def countEven(list:List[Int]):Int = list match {
  case x::tail if x%2==0 => countEven(tail)+1
  case _::tail => countEven(tail)
  case Nil => 0
}

val value: Option[String] = map.get(key)
value match {
  case Some(x) => x
  case None => "No hotness found"
}
```

exceptions

```
try {
  val reader = new FileReader("text.txt")
}
catch {
  case e: FileNotFoundException => println("No file")
  case e: IOException => println("No permission")
}
```

```
finally {
  reader.close()
}
```

regular expressions

```
import java.util.regex
val spaces = """"[;,\s]+"""".r           regular expression

val frac=""""(\d+)/(\d+)"""".r         regex with 2 groups
val frac(num,den) = "2/10"             extractor
```

files

```
import scala.io.Source
for(line <- Source.fromFile("test.txt").getLines)
  for(elem <- line.split("\s+"))
    println(elem)

import java.io.File
def filenames(path:String) = (new File(path)).list
def filenames(path:String, regex:String) = {
  for(fname <- filenames(path) if fname.matches(regex))
    yield fname
}

def dir(path:String) = (new File(path)).listFiles
for(file <- dir(".") if file.isFile)
  println(file.getName)

def walker(path:String, action:(File)=>Boolean):Unit = {
  for(file <- (new File(path)).listFiles if action(file))
    walker(file.getPath, action)
}
walker("c:/Temp", f=>{println(f.getPath); f.isDirectory})
```

useful snippets

```
list.reduceLeft(_ max _)           find maximum
list.reduceLeft(_+_ )              sum
list.reduceLeft(_+","+_ )          make strings
list.map("%.2f" format _)          format list of doubles

val prod = (1/:list)(_*_ )         product

def factorial(n :Int) = (1/(2 to n))(_*_ )

def strsum(text:String) = {
  def convert(text:String) = try{Some(text.toInt)}
    catch{ case _ => None }
  text.split("\s+").flatMap(convert).reduceLeft(_+_ )
}

List((1,2),(3,4)).map{case(x,y)=>x+y}
```

best practice

- reduceLeft is more efficient than reduceRight
- no parentheses for methods, if they have no parameters and have no side effects (getters).
- If a method has side effects it should have parentheses.

notes

there is no direct `break` or `continue` for loops but there is `breakable` (see control structures)

there is no `static`, use the singleton object instead

round brackets `()` can be replaced by curly brackets `{}` if the function has only one parameter: `sqrt(4) == sqrt{4}`

empty brackets can be left out, e.g. `str.length == str.length()`

`protected` works on subclass level (not on package level as in Java)

`override` works for instance variables too

references

<http://jim-mcbeath.blogspot.com/2008/09/scala-syntax-primer.html>