Scala 2.9.x

Cheat sheet
Stefan Maetschke
V 1.32, 08.01.2013

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 compiler`	`stop fast`

predef

Predefined types and methods in `Predef.scala` that do not need to be imported. See also next section

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

readf(format:String):List[Any]
readf1(format:String):Any
readf2(format:String):(Any,Any)
readf3(format:String):(Any,Any,Any)

val x = readInt
val y = readFloat
...
val str = readLine
```

`currentThread`	guess what
`exit`	exits application

annotations

Compiler annotations, eg. unchecked, unused, deprecated, inline, native, serializable, volatile, transient, remote, clonable, SerialVersionUID

`@SuppressWarnings(Array("unchecked"))`	Type erasure

```
@SuppressWarnings(Array("unchecked","unused"))

@SuppressWarnings(Array("unchecked"))
override def equals(other:Any) = other match {
  case that:Content[A] => this.content == that.content
  case _ => false
}
```

errors/assertions/preconditions

Automatically imported from `Predef.scala`

`error("message")`	RuntimeException

`assert(x>0)`
`assert(pred, message)`
	throws AssertionError

`require(x>0)`
`require(pred, message)`
	throws IllegalArgumentExeption

`assume(x>0)`
`assume(pred, message)`
	throws AssertionError

variables/constants

`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

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 <- 0 until 10) println(i)         for loop, exclusive
for(i <- 1 to 10) println(i)            for loop, inclusive
for(i <- 0 until 10 by 2) println(i)    for loop,  stride of 2
for(i <- 1 to 10; j <- 1 to 10)         nested for loop
  println((i,j))
for(i <- 0 until 10 if i%2==0)          with guard
  println(i)
for(i <- 0 to 10; sqr = i*i if sqr%2==0) var. binding
  println(sqr)
```

foreach

```
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 <- 1 until 10) if i % 2 == 0) yield I
for (i <- List.range(1, 10) if i % 2 == 0) yield i
```

collections

Three main packages with main traits automatically imported
`scala.collection.immutable`
`scala.collection.mutable`

```
   Traversable
       |
    Iterable
   /   |   \
Map   Set  Sequence
```

collection conversion scala ↔ java

Automatic bidirectional conversion between java and scala collections by implicit conversions.
Import of `collection.JavaConversions._` is needed.

```
import collection.JavaConversions._
val names:Iterable[String] = new ArrayList[String]()
```

traversable

Topmost base trait for collections. It provides all methods needed in for-comprehensions but `foreach()` method is abstract.

object methods

```
Traversable.empty[Int]       => List[Int]()
Traversable(1,2,3)           => List(1,2,3)
Traversable(List(1,2,3):_*)  => List(1,2,3)

Traversable.range(1,3)       => List(1,2)
Traversable.range(1,6,2)     => List(1,3,5)

Traversable.fill(3)(1)       => List(1,1,1)
Traversable.fill(2,2)('a')   => List(List(a,a),List(a,a))
… up to five dimensions

Traversable.tabulate(3)(_+1) => List(1,2,3)
Traversable.tabulate(2,2)((x,y) => (x,y))
=> List(List(0,1), List(2,2))
… up to five dimensions

Traversable.iterate(1,5)(_+1)   => List(1,2,3,4,5)
Traversable.iterate(1,5)(_*2)   => List(1,2,4,8,16)

Traversable.concat(List(1,2), List(3,4), List(5))
=> List(1,2,3,4,5)
```

instance methods

`trav.isEmpty`	true for empty travs
`trav.nonEmpty`	!trav.isEmpty
`trav.hasDefiniteSize`	true for strict travs
`trav.size`[1]	number of elems
`trav ++ trav1`	concat, new trav
`trav ++ itr`	concat, iterable
`trav.addString(builder)`	add to str builder
`trav.addString(builder, sep)`	with separator
`trav.addString(builder,start,sep,end)`	with start,end str
`trav.mkString`	makes string
`trav.mkString(sep)`	with separator
`trav.mkString(start,sep,end)`	with start,end str
`trav.copyToArray(arr,start,len)`	fill array
`trav.copyToArray(arr,start)`	fill array
`trav.copyToBuffer(buffer)`	fill buffer

`trav.reduceLeft(_*_)`	reduce Left
`trav.reduceLeftOption(_*_)`	returns Option
`trav.reduceRight(_*_)`	reduce Right
`trav.reduceRightOption(_*_)`	returns Option
`trav.foldLeft(1)(_*_)`	fold Left
`trav.foldRight(1)(_*_)`	fold Right
`(1/:trav)(_*_)`	fold Left
`(trav:\1)(_*_)`	fold Right
`trav.count(_ > 1)`	count
`trav.slice(from, until)`	exclusive `until`
`trav.splitAt(idx)`	split at index
`trav.drop(n)`	drop first n elems
`trav.dropWhile(_ < 5)`	drop while < 5
`trav.take(n)`	take first n elems
`trav.takeWhile(_ < 5)`	take while < 5
`trav.filter(_ > 5)`	filter for > 5
`trav.filterNot(_ > 5)`	filter for <= 5
`trav.span(_ < 5):(Trav,Trav)`	`takeWhile` & rest
`trav.find(_ > 5):Option[Int]`	return first hit
`trav.forall(_ > 1)`	true if all true
`trav.exists(_ > 1)`	true if some true
`trav.map(f)`	apply f return trav
`trav.collect(pf)`	filter&map together

e.g. `List(1,10,”a”) collect{case i:Int if i>5 => i}`

`trav.flatMap(f)`[2]	apply f and concat
`trav.flatten`[2]	flatten trav
`trav.foreach(f)`	apply f return Unit
`trav.groupBy(f:(A)=>K)`	return Map[K,Trav]
`trav.partion(_ > 5)`	returns two travs
`trav.unzip`	unzip trav of tuples
`trav.transpose`	trans. travs of travs
`trav.head`	first elem
`trav.headOption:Option`	first elem as Option
`trav.last`	last elem
`trav.lastOption:Option`	last elem as Option
`trav.init`	all but last
`trav.tail`	all but first
`trav.min`[3]	minimum
`trav.max`[3]	maximum
`trav.sum`	sum
`trav.product`	product
`trav.toList`	to List
`trav.toSeq`	to Sequence
`trav.toSet`	to Set
`trav.toStream`	to Stream
`trav.view`	creates view
`trav.view(from,until)`	creates view

[1] O(n) for lists!

[2] Especially useful for travs of Options:

```
 List(Some(1),Some(2), None).flatten   => List(1,2)
```

[3] User defined ordering possible, e.g. for list with tuples

```
 trav.min(new Ordering[(Int,Int)] {
   def compare(x:(Int,Int), y:(Int,Int)) = x._2 - y._2 })
```

However, easier is this:

```
 trav.reduceLeft((x,y) => (if(x._2 < y._2) x else y))
```

iterable

object methods

… nothing beyond what is offered by `Traversable`

instance methods

`iter.size`[1]	length of iterable

`iter.iterator`	returns iterator
`iter.sameElements(iter2)`	`iter == iter2`
`iter.takeRight(n)`	take n right elems
`iter.dropRight(n)`	drop n right elems

```
List(1,2,3) zip “test”  => List((1,t),(2,e),(3,s))
List(1,2,3) zipAll (“test”,0,'x')
   => List((1,t),(2,e),(3,s),(0,t))
 “test”.zipWithIndex
   => IndexedSeq((t,0),(e,1),(s,2),(t,3))
```

[1] O(n)

seq

object methods

`seq.unapplySeq(x:Seq)`	for pattern match { case Seq(...) => }
`seq.singleton`	singleton sequence

instance methods

`seq.length`[1]	length of seq
`seq :+ elem`	append elem
`elem +: seq`	prepends elem
`seq1 ++ seq2`	concat, new seq
`seq(idx)`	get idx-th elem
`seq.apply(idx)`	get idx-th elem
`seq.isDefinedAt(idx)`	seq(idx) defined?
`seq.contains(elem)`	tests for elem
`seq.sameElements(seq2)`	test same elems
`seq.containsSlice(seq2)`	tests for seq2
`seq.startsWith(seq2)`	starts with seq2
`seq.startsWith(seq2,offset)`	starts with seq2
`seq.endsWith(seq2)`	ends with seq2

```
seq.corresponds(seq2)(p:(x,y)=>Boolean)
e.g.: Seq.range(1,10).corresponds(Seq.range(0,9))
        ((x,y) => x-1==y)
```

`seq.prefixLength(p:(x)=>Boolean)`	length for p is true
`seq.segementLength(p:(x)=>Boolean,from)`	length for p is true
`seq.indexOf(elem)`	-1 if not found
`seq.indexWhere(p:(x)=>Boolean)`	-1 if not found
`seq.indexWhere(p:(x)=>Boolean,from)`	-1 if not found
`seq.LastIndexWhere(p:(x)=>Boolean)`	-1 if not found
`seq.lastIndexWhere(p:(x)=>Boolean,from)`	-1 if not found
`seq.indexOfSlice(seq2)`	-1 if not found
`seq.indexOfSlice(seq2,from)`	-1 if not found
`seq.lastIndexOfSlice(seq2)`	-1 if not found
`seq.lastIndexOfSlice(seq2,from)`	-1 if not found
`seq.lastIndexOf(elem)`	-1 if not found
`seq.lastIndexOf(elem, from)`	-1 if not found
`seq.findIndexOf(p:(x)=>Boolean)`	-1 if not found
`seq.findLastIndexOf(p:(x)=>Boolean)`	-1 if not found

`seq.sortWith(lt:(x,y)=>Boolean)`	sorting
e.g. `List(1,2,3).sortWith(_>_)`	
`seq.sortBy(f:(x)=>y)`	sort by f(x)
e.g. `List(('c',1),('b',2),('a',3)).sortBy(_._1)`	

`seq.reverse`	reverse
`seq.reverseIterator`	reverse iterator
`seq.reverseMap(f:(x)=>y)`	reversed&map
`seq.removeDuplicates`	remove duplicates
`seq.indices`	list of indices
`seq.padTo(len,elem)`	pad with elem
`seq.grouped(n):Iterator[Seq]`	groups of size n
`seq.sliding(size)`	sliding window
`seq.sliding(size, step)`	sliding window

`seq.span(p:(x)=>Boolean)`	prefix,suffix by p

```
e.g.: Seq(1,2,3).span(_>2)  => (List(), List(1,2,3))
      Seq(1,2,3).span(_<2)  => (List(1), List(2,3))
```

`seq.intersect(seq2)`	intersection
`seq.union(seq2)`	union
`seq.diff(seq2)`	difference

[1] O(1), equal to `seq.size`

range

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

rich types

Rich data type are implicit wrappers around Java types such as boolean, byte, float that add functionality. See package `scala.runtime`

conversion

```
toBoolean, toChar, toShort, toInt, toByte, toFloat,
toDouble, toString
97.toChar => a
'a'.toInt => 97
```

Char

```
isControl, isDigit, isLetter, isLetterOrDigit, isLower,
isUpper, isSpaceChar, isWhitespace, isTitleCase
toLower, toUpper, toTitleCase
'a' to 'c'     => IndexedSeqView(a,b,c)
```

Int, Long

```
ToBinarryString, toHexString, toOctalString, abs
1 to 5 by 2         =>  Range(1,3,5)
1 until 5 by 2      =>  Range(1,3)
```

Double, Float

```
isInfinity, isNegInfinity, isPosInfinity, toDegrees,
toRadians, abs, ceil, floor, round
1.0 to 1.6 by 0.2      =>  NumericRange(1.0,1.2,1.4,1.6)
```

string

Strings are sequences, see `Seq`

`“””multiple lines and raw””””`	raw strings

```
"""|Example of a string with
   |a stripped margin.""".stripMargin
```

`“%.1f %d”.format(3.14, 5)`	str formatting
`str.trim`	trims white spaces
`str.stripLineEnd`	strips line end
`str.stripPrefix(prefix)`	strips prefix str
`str.stripSuffix(suffix)`	strips suffix str
`str.replaceAll(“\\s”,””)`	replace use regex
`str.replaceFirst(“\\s”,”_”)`	replace use regex
`str.split(‘ ‘)`	splits use char
`str.split(“\\s+“)`	split use regex
`str.split(pos)`	split at pos
`str.matches(“[A-Z}+”)`	regular expression
`str.endsWith(string)`	string end
`str.startsWith(string)`	string start
`str.substr(i)`	i to end
`str.substr(i,j)`	i to j-1
`str.capitalize`	capitalize
`str.toLower`	to lower

`str.toBoolean, toDouble, toInt, ….`	
`”ab”*3`	ababab
`str.lines`	iter over lines in str without line ends
`str.linesWithSeparator`	iter over lines in str with line ends

enumeration

Lightweight alternative to case classes

```
object WeekDay extends Enumeration {
   type WeekDay = Value
   val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

object Main extends Application {
  import WeekDay._
  def isWorkingDay(d: WeekDay) = ! (d == Sat || d == Sun)
  WeekDay.values filter isWorkingDay foreach println
}
```

option

`Option` is a “type-safe null” to be used instead of null. `Option[T]` has two values: `Some(T)` and `None`.
See: http://blog.tmorris.net/scalaoption-cheat-sheet/

```
val x:Option[Int] = Some(5)
x.get           => 5
x.getOrElse(0)  => 5
x.isDefined     => true
x match {            // to avoid, better ways see below
  case Some(x) => x
  case None    => 0
}

val y:Option[Int] = None
y.get            => NoSuchElementException
y.getOrElse(0)   => 0
y.isDefined      => false

o.foreach(foo)
<=> if(o!=None) foo(o.get)
<=> o match {case None=> ; case Some(x) => foo(x)}

case class Contact(name:String)
val contacts = List(Some(Contact(“Peter”)), None)
contacts.flatten  => List(Contact(“Peter”))
contacts.flatMap(_.map(_.name)) => List(“Peter”)
contacts.map(_.map(_.name))  => List(Some(“Peter”), None)
contacts.map(_.map(_.name).getOrElse(“NoName”))
=> List(“Peter”, “NoName”)
```

function

`def add(x:Int, y:Int):Int = {x+y}`	function definition
`val inc = add(_:Int, 1)`	partially applied
`val add2 = add _`	partially applied
`def incGen(a:Int) = (x:Int) => x+a`	closure
`def add(a:Int)(b:Int) = a+b`	currying
`(x:Int, y:Int) => x+y`	function literal
`def thunk()`	thunk: no args func
`def f(x: => Int)`	lazy/by-name para
`def f(x(): => Int)`	lazy with thunk
`def inc(x:Int, a:Int = 1) = x+a`	default param
`def sum(xs:Int*) = xs.reduceLeft(_+_)`	var. num. of args
`sum(1,2,3)`	call it
`sum(List(1,2,3):_*)`	unpacking args
`def foo(bar:(Int,Int)=>Int) = {bar(…)}`	higher order func.

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

```
def repeat(n:Int)(action: => Unit) {
  for(i ← 0 until n) action
}

def using[A, B <: {def close(): Unit}]
  (closeable: B) (f: B => A): A =
  try { f(closeable) } finally { closeable.close() }
```

tuple

Tuples are immutable and can contain elements of different types. Index is one-based!

`val t = (“pi”, 3.14)`	a tuple
`t._1`	first element
`t._2`	second element

list

Lists are immutable, no append for lists

`val list = List(1,2,3,4,5)`	filled list
`val list = List.range(1,6)`	filled list
`list.length`	length of list
`list.head`	first element
`list.last`	last element
`list.mkString(“,”)`	list to string
`list.mkString(“[”, “:”, ”]”)`	list to string
`list.count(p:(x)=>Boolean)`	count
`list.find(p:(x)=>Boolean)`	find
`list.filter(p:(x)=>Boolean)`	filter
`list.reverse()`	reverse
`list.sortWith(_>_)`	sorting
`list.partition(p:(x)=>Boolean)`	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.view`	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
`val mat = Array.ofDim[Double](3,6)`	matrix
`mat(2)(3)`	element access
`val cube = Array.ofDim[Int](3,6,4)`	cube
`cube(2)(3)(1) = 1`	element access

map

Maps can be mutable or immutable depending on import

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

`ListMap`	order preserving
`TreeMap`	red-black tree
`HashMap`	`== 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(List((1,2),(2,3)):_*)`	filled map
`val map = Map[Int,Int]().withDefaultValue(0)`	

`val map = List((1,2),(2,3)).toMap`	immutable
`val map = Map[Int,Int]() ++ List((1,2),(2,3))`	immutable
`val map = Map[Int,Int]() ++= List((1,2),(2,3))`	mutable

`map += key -> value`	add pair
`map += (key,value)`	add pair
`map.isDefinedAt(key)`	has value for key
`map(key)`	get value
`map.get(key)`	get value
`map.getOrElse(key, default)`	get val or def
`map.getOrElseUpdate(key:A, op: ⇒ B):B`	get or update
`map.iterator`	itr over entries
`map.valuesIterator`	itr over values
`map.keysIterator`	itr over keys
`map.keySet`	key set
`map.mapValues(f: (B) ⇒ C)`	map on values

```
map.map{case (k,v) => k+”:”+v}.mkString(“\n”)

// histogram/counter of elements in list
list.groupBy(e => e).mapValues(_.size)
```

set

Sets are immutable but a mutable version and a sorted set exist

`val set = Set(1,2,3,4)`	filled set
`val set = Set(List(1,2,3,4):_*)`	set from list
`set + 5`	add to set
`set.add(5) // returns false if 5 in set`	add to set
`set - 4`	remove from set
`set ++ (5 to 7)`	add iterable to set
`vset -- (3 to 4)`	remove from set
`set.clear`	clears set
`set.contains(5)`	true if 5 in set
`set(5)`	true if 5 in set
`set.subsetOf(Set(1,2,3))`	true is subset

`set1 & set2`	intersection
`set1 intersect set2`	intersection
`set1 | set2`	union
`set1 union set2`	union
`set1 &~ set2`	difference
`set1 diff set2`	difference
`set1 -- set2`	set1 - set2

stack

import scala.collection.mutable.Stack

```
val stack = new Stack[Int]()
```

`stack.push(elems:A*)`	push
`stack += elem`	push
`stack.pop`	pop
`stack.top`	no remove
`stack(index)`	getter
`stack.length`	stack size
`stack.clear`	clears stack
`stack.elements`	iter over elems
`stack1 ++= stack2`	pushes stack2 on 1

main method

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

object MyApp extends App {
  args foreach println
```

```
}
```

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 {`	abstract class
`  def method:Int`	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
}

class Frac(val num:Int, val den:Int) extends
  Ordered[Frac] {
  require(den > 0)
  val toDouble = num/den.toDouble
  def *(that:Frac):Frac =
    Frac(this.num*that.num, this.den*that.den)
  def *(c:Int):Frac = Frac(num*c, den)
  def *(c:Double):Double = toDouble*c
  def compare(that:Frac):Int =
    (this.num*that.den) - (that.num*this.den)
  override def equals(other:Any) = other match {
    case that:Frac => (this eq that) ||
       (that.num == this.num && this.den == that.den)
    case _ => false
  }
  override def hashCode = 13*(num+13*den)
  override def toString = "%d/%d" format (num,den)
}

object Frac {
  def apply(num:Int, den:Int) = new Frac(num,den)
  implicit def int2Frac(num:Int):Frac = Frac(num,1)
  implicit def frac2double(frac:Frac):Double =
   frac.toDouble
}
import Frac._
println(Frac(1,3) == Frac(1,2))
println(Frac(1,3) < Frac(1,2))
println(Frac(1,3) * Frac(1,2))
println(Frac(1,2) * 2.5)
println(2.5 * Frac(1,2))
println(Frac(1,2) * 2)
println(2 * Frac(1,2))
```

case class

Used for pattern matching. No `new` required for instantiation, comes with companion `object`, constructor parameters automatically become class member variables, sensible default implementations for `toString`, `hashCode` and `equals`.

```
abstract class C
case class A(x:Int) extends C
case class B(a:A) extends C

B(A(1)) == B(A(1))                      true
B(A(1)) == B(A(0))                      false

val b = B(A(2))                         no new
val (B(A(x)) = b
=> x = 2

def extract(c:C) = c match {            matching
  case B(A(1)) => "x=1"
  case B(a)    => "a="+a
  case _       => "no match"
}
```

access modifiers

Every member without modifier is `public`
There is no static, use companion object instead.

`private`	visible only inside of class (but not in inner class)
`protected`	visible in class and subclasses (not package wide)
`private[X]`	private up to class/package X
`protected[X]`	protected up to class/package X
`private[this]`	access only from same instance

trait

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

```
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
```

implicit

Implicit modification of existing classes
`http://www.scalaclass.com/book/export/html/1`

```
// pimping, e.g. 5.sin
implicit def pimpDouble(x:Double) = new {
  def sin = Math.sin(x)
  def cos = Math.cos(x)
}


// factorial, e.g.  5!
implicit def pimp(n:Int) =
 new { def ! = ((1 to n) :\ 1) (_*_) }
```

```
import Numeric.Implicits._
def sum[N:Numeric](lst:List[N]) = lst :\ (_+_)
```

match expression

Similar to switch in Java but also support pattern matching

```
def count(x:int) = x match {
  case 0 => “zero”
  case 1 => “one”
  case n => “many:”+n
}

def size(x:Any) = x match {
  case n: Int       => n
  case s: String    => s.length
  case l :List[_]   => l.size
  case m: Map[_,_] => m.size
  case _            => 0
}

def isPositive(x:Int) = x match {
  case n: Int if n>=0  => true
  case _               => false
}

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
}
```

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

http://langref.org/scala/pattern-matching

```
val number = “””[0-9]+”””.r                regular expression
number findAllIn “123 45”                  => MatchIterator
number findFirstIn “123 45”                => Option[String]
number findFirstMatchIn “123 45”           => Option[Match]
number.replaceAllIn(“123 45”, “x”)         replaces all
spaces split “123 45”                      => Array

val alpha = “””([a-z]+)”””.r               regex with 1 group
List(“5”,”ab”).collect{case alpha(letters) => letters}

val frac=”””(\d+)/(\d+)”””.r               regex with 2 groups
val frac(num,den) = “2/10”                 extractor, unapply
List(“5”,”1/2”).collect{case frac(num,den) => (num,den)}
```

xml

http://www.scala-lang.org/node/131
http://www.ibm.com/developerworks/java/library/x-scalaxml/index.html
http://burak.emir.googlepages.com/scalaxbook.docbk.html

```
val data = <shopping>
<item name="bread" quantity="3" price="2.50"/>
<item name="milk" quantity="2" price="3.50"/>
</shopping>

val data = <shopping>
{List("bread,3,2.50", "milk,2,3.50") map { row =>
row split ","
} map { item =>
<item name={item(0)} quantity={item(1)} price={item(2)}/>
}}

</shopping>
val res = for (
item <- data \ "item" ;
price = (item \ "@price").text.toDouble ;
qty = (item \ "@quantity").text.toInt)
yield (price * qty)

XML.save(“shopping.xml”, data)
```

files

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

fromFile(“test.txt”).getLines().
  map(_.split(“\\s+”)).foreach(println)

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(file:File) = file.listFiles match {
  case null => Array[File]()
  case list => list
}
dir(new File(“.”)).foreach(println(_.getName))

def walker(file:File, action:(File)=>Unit):Unit = {
  action(file)
  file.listFiles match {
    case null => return
    case list => list.foreach(walker(_,action))}
}
walker("c:/Temp", f=> println(f.getPath) )

// creates a buffered file writer
def writer(filepath:String) = new BufferedWriter(
  new FileWriter(new File(filepath)))


// copy a file
def copy(src:File, dest:File) =
  new FileOutputStream(dest).getChannel.
    transferFrom(new FileInputStream(src).
      getChannel, 0, Long.MaxValue)
```

benchmarking

Benchmarking of classes: `import scala.testing`

```
trait Benchmark {
  def run()            // to implement
  var multiplier = 1   // number of times run() is called
  def runBenchmark(noTimes: Int): List[Long]
}

// example
object Sorter extends Benchmark {
  multiplier = 10     // call run 10 times
  def run = List.range(1,10000).sortWith(_ > _)
}
println( Sorter.runBenchmark(5) )
```

process

Calling external programs

```
import scala.sys.process.{Process, ProcessIO}
import scala.io.Source

def show(s:InputStream) {
  Source.fromInputStream(s).getLines.foreach(println)
}
val pb = Process("""ipconfig.exe""")
val pio = new ProcessIO(
  stdin => (),
  stdout => show(stdout),
  stderr => show(stderr))
pb.run(pio)  // don't wait
```

conversion

Conversion between java and scala collections

```
import scala.collection.JavaConversions._

sIterable = collectionAsScalaIterable(jCollection)
sIterable = iterableAsScalaIterable(jIterable)
sIterator = asScalaIterator(jIterator)

jIterator = asJavaIterator(sIterator)
jIterable = asJavaIterable(sIterable)
...
```

useful snippets

```
// unpacking via case classes
val Array(h,m) = “13:57”.split(':')

// max of a list of tuples according to second component
list.reduceLeft( (a,b) => if(a._2 > b._2) a else b )

// format list of doubles
list.map(“%.2f” format _)
list.formatted(“%.2f”)

import Numeric.Implicits._
def sum[N:Numeric](lst:List[N]) = lst :\ (_+_)

// collect: combines type filter and map
List(1, "a") collect{ case i:Int => i }  => List(1)
val num = “””([0-9]+)”””.r       //regex
List(“12”,”a”).collect{case num(chars) => chars.toInt}

// dot product
def dot(xs:List[Double], ys:List[Double]) =
  (xs, ys).zipped map (_*_) reduceLeft(_+_)


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

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

// replace var names in text by their values
val vars = Map(“{X}”->”1”, “{PI}”->”3.14”)
def replace(text:String) =
  vars.foldLeft(text){case (t,(k,v)) => t.replace(k,v)}
```

```
// print out all methods of String class
(new String()).getClass.getMethods.foreach(println)


// shuffle a list
scala.util.Random.shuffle(List(1,2,3,4))


// organize items in a map according to some attribute
val names = List(“Peter”, “John”, “Jacob”, “Paul”)
val map = names.groupBy(e => e(0)) //group by 1st letter


// write to a logfile with a variable argument list
def log(formatstr:String, args:Any*) =
  logfile.write( formatstr.format(args:_*) )


class Counter[T](xs:Iterable[T]) {
  val counts = xs.groupBy(identity).mapValues(_.size)
  override def toString = counts.toSeq.
     sortBy(_._2).mkString("\n")
}


// 4 digit trinary counter
val n = 3                            // trinary
val digits = Array(0,0,0,0)          // 4 digits
def increment(i:Int = 0):Unit = {
  while(digits(i) < n) {
    if(i<digits.length-1) increment(i+1)
    else println(digits.mkString)
    digits(i) += 1
  }
  digits(i) = 0
}

// use view to iterate efficiently
(1 to 10000).view.filter(_%2==0).sum


case class Person(name:String, age:Int)
val persons = List(Person("Joe", 42), Person("Jane", 30),
 Person("Alice", 14), Person("Bob", 12))
persons.exists(_.age > 18)
persons.filter(_.age > 18)
persons.map(_.name)
persons.foldLeft(0)(_ + _.age)


// prime numbers
def primes(s:Stream[Int]=Stream.from(2)):Stream[Int] =
  Stream.cons(s.head,primes(s.tail filter {_%s.head!=0}))
primes().take(5).foreach(println)
```

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 direkt `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://programming-scala.labs.oreilly.com
http://www.scala-lang.org/node/104
http://jim-mcbeath.blogspot.com/2008/09/scala-syntax-primer.html
Programming in Scala, M. Odersky, L. Spoon, B. Venners, Artima 2008
http://langref.org/scala