CROSSINGTECH

Scala Quick Reference

Symbolics

->	Returns a two-element tuple containing the key and value	`Map( 1 -> "A", 2 -> "B")` `(1).->("A")`

_	A placeholder, used in imports, function literals, etc.	`import com.xtech._`
		`case _ => value.toString`
		`numbers.filter(_ < 0)`

:	Separator between identifiers and type annotations.	`def add(i: Int): Int = ...`
=	Assignment.	`val one = "1"`
=> ⇒	Used in function literals to separate the argument list from the function body.	`numbers.filter(x => x < 0)`
<- ←	Used in for comprehensions in generator expressions.	`for (arg <- args)`
<:	Upper bounds (a subtype of)Used in parameterized and abstract type declarations to constrain the allowed types.	`def apply[T <: U](x: T)`
<%	View bounds(apply implicit convertion).Used in parameterized and abstract type declarations to convert the type using view.	`def m [A <% B](args): R = ...`
>:	Lower bounds (supertype of)Used in parameterized and abstract type declarations to constrain the allowed types.	`def append[U >: T](x: U) =`
#	Refer to a type declaration nested in another type	`val ic: MyClass#myType = ...`
@	Marks an annotation.	`@deprecated def bad() =`

‘	Symbol	`val s = 'aSymbol`
		`def dolt(r: Symbol)`
		`dolt(s); print(s.name)`

If a method takes 0 or one parameter you can drop the dot and parentheses when calling the function.

Variables

Immutable (Final)

```
val msg = "Hello, world!"
val msg: String = "Hello, world!"
val big = new java.math.BigInteger("12345")
```

Mutable

```
var greets = "Hello, world!"
var greets: String = "Hello, world!"
```

Lazy initialization

(only on Immutable)

```
object Demo {
  lazy val x = { println("initializing x"); "done" }
}
```

Basic Types

Value Type	Range
Byte	8-bit signed two’s complement integer (-2^7 to 2^7 - 1, inclusive)
Short	16-bit signed two’s complement integer (-2^{15} to 2^{15} - 1, inclusive)
Int	32-bit signed two’s complement integer (-2^{31} to 2^{31} - 1, inclusive)
Long	64-bit signed two’s complement integer (-2^{63} to 2^{63} - 1, inclusive)
Char	16-bit unsigned Unicode character (0 to 2^{16} - 1, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

Operators

+, -, *, /, %	Arithmetics
>, <, <=, >=, ==, !=	Relational
&&, ||, !	Logical
&, |, ^, ~	Bitwise (and, or, xor, inv)
<<, >>, >>>	Bitwise shift (left, right, unsigned right)

The operator “==” check the value equality on reference AND primitive type.

Rich Operation

Scala provides “rich wrapper” around basic types via implicit conversions.

Code	Result
`0 max 5`	5
`0 min 5`	0
`-2.7 abs`	2.7
`-2.7 round`	-3L
`1.5 isInfinity`	false
`(1.0 / 0) isInfinity`	true
`4 to 6`	Range(4,5,6)
`"nick" capitalize`	“Nick”
`"nicolas" drop 2`	“colas”

Literals

Integer

`val dec = 31`	Decimal Integer
`val hex = 0XFF`	Hexa Integer
`val long = 31L`	Long (“l” or “L”)
`val little: Short = 367`	Short
`val littler: Byte = 38`	Byte

Floating point

`val double = 1.2345`	Double
`val e = 1.234e4`	Double (“e” or “E”)
`val float = 1.234F`	Float (“f” or “F”)

Character and String

`val aChar = ‘D’`	Char
`val unicode = ’\u0043’`	Unicode Char
`val string = “string”`	String
`val s = “”” it’s “you” “”””`	Raw String (It’s “you”)

Special character

Literal	Meaning
\n	line feed (\u000A)
\b	backspace (\u0008)
\t	tab (\u0009)
\f	form feed (\u000C)
\r	carriage return (\u000D)
\”	double quote (\u0022)
\’	single quote (\u0027)
\\	backslash (\u005C)

Boolean

`val bool = true`	Boolean (true | false)

Check

```
“abc”.isInstanceOf[String]
re0: Boolean = true
```

Cast

```
3.asInstanceOf[Double]
res0: Double = 3.0
```

Runtime Representation

```
classOf[String]
res7: java.lang.Class[String] = class java.lang.String
```

Import

```
import java.awt._ // All classes under java.awt
import java.io.File
import java.io.File._ // Import all File’ static methods
import java.util.{Map, HashMap} // only the 2 classes
```

Narrow import:

```
def dolt() = {
  import java.math.BigDecimal.{ONE}
  println(ONE)
}
```

Rename import:

```
import java.math.BigDecimal.{
  ONE  => _,              // Exclude ONE
  ZERO => JAVAZERO        // Rename it
}
println(JAVAZERO)
```

import statements are relative, not absolute. To create an absolute path, start with _root_

```
import _root_.scala.collectionsjcl._
```

Packages

File names don’t have to match the type names, the package structure does not have to match the directory structure. So, you can define packages in files independent of their “physical” location.

Traditional:

```
package com.xtech.scala
```

Nested:

```
package com{
  package scala { class A }
  package util    { class B } }
```

Tuples

Are immutable and can contain different types of elements.

```
val nena = (99, "Luftballons",”1983”)
println(nena._1)
println(nena._2) .... println(nena(0)) (not same Type in list)
```

_ Usage Summary

Curried functions

```
def twice(op: Double => Double) (x: Double) = op(op(x))
twice(_ + 1) (5)
res8: Double = 7.0
twice(x => x + 1)(5) // More verbose
```

Existential types

Labeling something that is unknown:

```
class Marshaller[T] { def marshall(t:T) = {println(t)} }
new Marshaller[String]
res1: Marshaller[String] = Marshaller@7896b1b8
res1.isInstanceOf[Marshaller[_]]
res4: Boolean = true
same as:
.isInstanceOf[T forSome {type T <: Marshaller[String]}]
```

Function literals

```
someNumbers.filter(_ > 0)
```

Partially applied functions

```
def sum(a: Int, b: Int, c: Int) = a + b + c
val a = sum _
a: (Int, Int, Int) => Int = <function>
val b = sum(1, _: Int, 3)
b: (Int) => Int = <function>
b(2)
res10: Int = 6
```

Import statements

```
import com.xtech.cf._
```

Match expressions

```
case _ => “default value” // Default case value
```

Initialization

```
var age: Int = _ // age initialized to 0
```

Setter

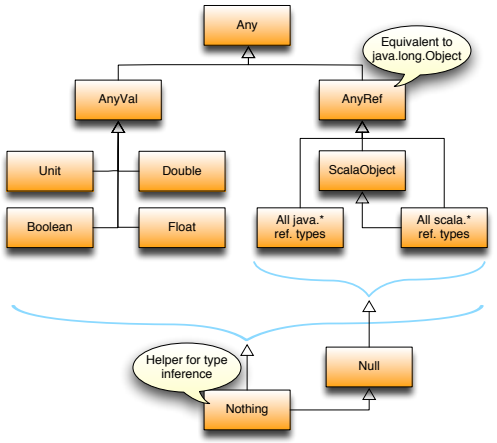
Redefined a setter method:

```
def age_ = (a: Int) { if(girl) age = a - 5 else age = a }
```

Classes, Objects & Actors

CROSSINGTECH

Class Hierachy



Definition

Simple class:

```
class ChecksumAccumulator {
  private var sum = 0
  def add(b: Byte): Unit = sum += b
  def checksum(): Int = ~(sum & 0xFF) + 1
}
```

Constructor

The default constructor (primary constructor) is defined by the body class and parameters are listed after the class name. Other constructors (auxiliary constructor) are defined by the function definition "this()":

```
class Rational(n: Int, d: Int) {
   require(d != 0)
   val numer: Int = n
   val denom: Int = d
   def this(n: Int) = this(n, 1) // auxiliary constructor
}
```

To hide the constructor make it private:

```
class Rational private(n: Int, d: Int)
```

Getter / Setter

Once a val or var is defined within a class the corresponding accessor methods are generated. The generated methods use the same privilege as the field.
Only the getter is generated in case of val.
The methods don't follow the JavaBean nomenclature.

To generate JavaBean getter and setter add the annotation:

```
@scala.reflect.BeanProperty var level: Int = _
```

Abstract

```
abstract class Document {
  def footNotes: Array[String] // abstract method
  var nbOfPages : Int          // abstract Field
  type paper                   // abstract type
}
```

Inheritance

```
class A extends B
```

Call super constructor

```
class A(param: String) extends B(param: String)
```

Singleton / Static

Singleton objects are objects with only on instance in the whole JVM.

There is no static in Scala, instead use the companion object to support class-level operation and properties. A companion is a singleton

```
class Book private (title: String)
object Book {
  val favorites= """Java Puzzlers", "Design Patterns"""

  def apply(title: String) = {
    println("Book construction ...");new Book(title)
  }
  def main(args: Array[String]){ ... }
}
printf("My favorites are : %s\n", Book.favorites)
My favorites are : "Java Puzzlers", "Design Patterns"
Book("Digital Fortress")
Book construction ...
res1: Book = Book@2012a961
```

Variance

Covariance: Ability to accept sub-classes.
"**T <: Pet**" or "**+T**" : (as T extends Pet)

Contra-variance: Ability to accept base classes
"**T >: Cat**" or "**-T**" : (as T is superType of Cat)

Traits

A traits is like a java interface at the difference that it's possible to implements methods and fields on it. Traits can be reused into classes by mixing the trait to the class or by extending it.

Definition

```
trait Saxo {
  def play() {
    println("Nice sound!")
  }
}
```

Extends

```
class Alto extends Saxo {
  override def toString = "Alto"
}
```

With

```
class Instrument
class Baryton extends Instrument {
  override def toString = "Baryton"
}
val baryton = new Baryton() with Saxo
```

Ordered Traits

The Ordered trait defines <, >, <=, and >= just by implementing one method, compare.

```
class Rational(n: Int, d: Int) extends Ordered[Rational]{
   // ...
  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)
}
```

Mixing

Once a trait is mixed into a class, you can alternatively call it a mixin.Traits are a way to inherit from multiple class-like constructs, but they differ in important ways from the multiple inheritance present in many languages. With traits, the method called is determined by a linearization of the classes and traits that are mixed into a class.

Linearization algorithm

1. Put the actual type of the instance as the first element.
2. Starting with the right most parent type and working left,compute the linearization of each type, appending its linearization to the cumulative linearization. (Ignore ScalaObject, AnyRef, and Any for now.)
3. Working from left to right, remove any type if it appears again to the right of the current position.
4. Append ScalaObject, AnyRef, and Any.

```
class C1 {def m = List("C1")}
trait T1 extends C1 {override def m ={ "T1" :: super.m}}
trait T2 extends C1 {override def m ={ "T2" :: super.m}}
trait T3 extends C1 {override def m ={ "T3" :: super.m}}
class C2 extends T2 {override def m ={ "C2" :: super.m}}
class C extends C2 with T1 with T2 with T3{
  override def m ={ "C" :: super.m}
}
```

# Linearization	Description
1 C	+ type of the instance.
2 C, T3, C1,	+ farthest on the right (T3)
3 C, T3, C1, T2, C1	+ T2
4 C, T3, C1, T2, C1, T1, C1	+ T1
5 C, T3, C1, T2, C1, T1, C1, C2, T2, C1	+ C2
6 C, T3, T2, T1, C2, T2, C1	- duplicates C1 but last
7 C, T3, T1, C2, T2, C1	- duplicates T2 but last
8 C, T3, T1, C2, T2, C1, ScalaObject, AnyRef, Any	done.

SelfType

Redefines the type of this. Must be a subclass of all the self type of all its base class.

```
class Animal {this: Dog with Friend => ... }
```

Actors

```
import scala.actors._
object SimpleActor extends Actor {
 def act() {
  for (i <- 1 to 5) {
    println("Do it!")
    Thread.sleep(1000)
  }
 }
}
```

To Start it:

```
SimpleActor.start()
```

To start a thread immediately use the utility method actor:

```
import scala.actors._
val seriousActor2 = actor {
 for (i <- 1 to 5)
  println("Do it!.")
}
```

Send message to Actor;

```
import scala.actors._
val echoActor = actor {
 while (true) {
  receive {
    case msg => println("received message: "+ msg)
  }
 }
}
```

To send a message:

```
echoActor ! "Hello"
received message: hi there
```

To use the current thread use self:

```
self ! "hello"
self.receive { case x => x }
res6: Any = hello
self.receiveWithin(1000) { case x => x }
res7: Any = TIMEOUT
```

Change Scheduler:

Run it on the main Thread

```
trait SingleThread extends Actor{
override protected def scheduler() =
                        new SingleThreadScheduler
}
```

Run all actors in the Main thread:

```
Scheduler.impl = new SingleThreadScheduler
```

Thread reuse

Writing an actor to use react instead of receive is challenging, but pays off in performance. Because react does not return, the calling actor's call stack can be discarded, freeing up the thread's resources for a different actor. At the extreme, if all of the actors of a program use react, then they can be implemented on a single thread.

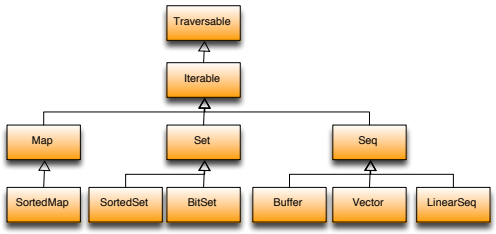
As empiric rule:
- Actors that are message-heavy are better implemented with "while(true)/receive" (Hogging a thread).
- Actors with non trivial work are better implemented with "loop/react".

```
object NameResolver extends Actor {
 import java.net.{InetAddress, UnknownHostException}
 def act() {
  react {
   case (name: String, actor: Actor) =>
                    actor ! getIp(name)
                    act()
   case "EXIT" => println("Exiting.") // quit
   case msg => println("Unhandled message: "+ msg)
                    act()
  }
 }
 def getIp(name: String): Option[InetAddress] = {
    try {
     Some(InetAddress.getByName(name))
    }catch {
     case _:UnknownHostException => None
    }
 }
}
```

Collections

CROSSINGTECH

Collection



The main trait is traversable, which is the supertrait of both mutable and immutable variations of sequences (Seq), sets, and maps. Sequences are ordered collections, such as arrays and lists. Sets contain at most one of each object, as determined by the == method. Maps contain a collection of keys mapped to values.

First try with immutable and switch to mutable only if needed.

JAVA <-> Scala Conversion

```
import scala.collection.JavaConversions._
```

Sets and Maps

Immutable Set (default if no explicit import):

```
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
```

Mutable Set:

```
import scala.collection.mutable.Set
val movieSet = Set("Hitch", "Poltergeist")
movieSet += "Shrek"
println(movieSet)
```

Immutable Map (default if no explicit import):

```
import scala.collection.immutable.HashMap
var hashMap = HashMap(1 -> "one", 2 -> "two")
println(hashMap.get(1))
```

Mutable Map:

```
import scala.collection.mutable.Map
val treasureMap = Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))
```

Conversion immutable to mutable

```
import scala.collection.mutable
var mutaSet = mutable.Set.empty ++ immutableSet
```

Conversion mutable to immutable

```
val immu = Map.empty ++ muta
```

Map sample

Immutable

```
val nums = Map("i" -> 1, "ii" -> 2)
```

`nums + ("vi" -> 6)`	Map(i -> 1, ii -> 2, vi -> 6)
`nums - "ii"`	Map(i -> 1)
`nums ++ List("iii" -> 3, "v" -> 5)`	Map(i -> 1, ii -> 2, iii -> 3, v -> 5)
`nums -- List("i", "ii")`	Map()
`nums.size`	2
`nums.contains("ii")`	true
`nums("ii")`	2
`nums.keys`	Iterator over the strings "i" and "ii"
`nums.keySet`	Set(i, ii)
`nums.values`	Iterator over the integers 1 and 2
`nums.isEmpty`	false

Mutable

```
val wd = scala.collection.mutable.Map.empty[String, Int]
```

`wd += ("one" -> 1)`	Map(one -> 1)
`wd -= "one"`	Map()
`wd ++= List("one" -> 1, "two" -> 2, "three" -> 3)`	Map(one -> 1, two -> 2, three -> 3)
`wd --= List("one", "two")`	Map(three -> 3)
`wd.getOrElseUpdate (k, v)`	return the value for the key 'k'. If doesn't exists update wd with the mapping k->v and return v.
`wd.transform( (s,i) => i + 1)`	Map(one -> 2)

TreeSet / TreeMap

```
val ts = TreeSet(9, 3, 1, 8, 0, 2, 7, 4, 6, 5)
scala.collection.immutable.SortedSet[Int] =
        Set(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 -> 'x')
scala.collection.immutable.SortedMap[Int,Char] =
        Map(1 -> x, 3 -> x, 4 -> x)
```

Enumeration

```
object Color extends Enumeration {
val Red, Green, Blue = Value
}
```

Enumeration with value:

```
object Direction extends Enumeration {
  val Up = Value("goUp")
  val Down = Value("goDown")
}
Direction.Up.id
res0: Int = 0
Direction(1)
res1: Direction.Value = goDown
```

Lists

Class List provides fast access to the head of the list, but not the end. Thus, when you need to build a list by appending to the end, you should consider building the list backwards by prepending elements to the front, then when you're done, calling reverse to get the elements in the order you need.

Another alternative, which avoids the reverse operation, is to use a ListBuffer (see next section)

Creation:

```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThree = "one" :: "two" :: "three" :: Nil
```

Concatenation (":::"):

```
val oneTwoThreeFour = oneTwo ::: threeFour
```

Prepends ("::" pronounced "cons"):

```
val twoThreeFour = 2 :: threeFour
```

Operation on List:

Basics:

```
val nums = Set(1, 2, 3)
```

`nums + 5`	Set(1, 2, 3, 5)
`nums - 3`	Set(1, 2)
`nums ++ List(5, 6)`	Set(1, 2, 3, 5, 6)
`nums -- List(1, 2)`	Set(3)
`nums & Set(1, 3, 5, 7)`	Set(1, 3)
`nums.size`	3
`nums.contains(2)`	TRUE

```
import scala.collection.mutable
val words = mutable.Set.empty[String]
```

`words += "thx"`	Set(thx)
`words -= "thx"`	Set()
`words ++= List("α", "β", "γ")`	Set(α, β, γ)
`words --= List("α", "β")`	Set(α)
`words.clear`	Set()

ListBuffer

List used to append values in an optimized way (see general remark of Lists) and to avoid stack overflow.

```
val buf = new ListBuffer[Int]
buf += 1
buf += 2
3 +: buf
buf.toList
List[Int] = List(3, 1, 2)
```

Lists sample

```
val truth = "Fly" :: "is" :: "fun" :: Nil
```

From Traversable trait:

`truth.foreach(print)`	Flyisfun
`truth.head`	Fly
`truth.isEmpty`	false
`List.unzip(zippedTruth)`	(List(0, 1, 2), List(Fly, is, fun))
`List.flatten( List(List('f','l'),List('y'), List('.')))`	List(f, l, y, .)
`truth.count(s => s.length == 3)`	2
`truth.drop(2)`	List(fun)
`truth.exists(s => s == "is")`	true
`truth.filter(s => s.length == 3)`	List(Fly, fun)
`truth.forall(s => s.endsWith("y"))`	false
`truth.tail`	List(is, fun)
`truth.init`	List(Fly, is)
`truth.last`	fun
`truth.length`	3
`truth.map(s => s + "!")`	List(Fly!, is!, fun!)
`truth.mkString(",")`	Fly,is,fun
`truth.remove(s => s.length == 3)`	List(is)
`truth.reverse`	List(fun, is, Fly)
`truth.sort((s,t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)`	List(fun, Fly, is)
`truth.indices`	List(0, 1, 2)
`truth.toArray`	Array(Fly, is, fun)
`truth flatMap (_.toList)`	List(F, l, y, i, s, f, u, n)
`truth partition (_.length == 2)`	((List(is),List(Fly, fun))
`truth find (_.charAt(0) == 'a')`	None
`truth takeWhile (_.charAt(0).toLowerCase != 'i')`	List(Fly)
`truth dropWhile (_.charAt(0).toLowerCase != 'i')`	List(is, fun)
`truth forall (_.length > 2)`	false
`truth exists (_.charAt(0) == 'i')`	true
`truth.foldRight("!")(_ + _)`	Flyisfun!
`truth.reduceRight (_ + _)`	Flyisfun
`truth.foldRight(List[String]()) {(x, list) => ("<"+x+">") :: list}`	List(<Fly>, <is>, <fun>)
`truth.foldLeft("Yes,")(_ + _)`	Yes,Flyisfun
`List(1,2,3) reduceLeft(_ + _)`	6
`List.range(9, 1, -3)`	List[Int] = List(9, 6, 3)
`List.make(5, 'a')`	List(a, a, a, a, a)
`List.concat( List(), List('b'), List('c'))`	List(b, c)

From Iterable traits:

`truth.dropRight(2)`	List(Fly)
`truth.takeRight(2)`	List(is, fun)
`truth.zipWithIndex`	List((Fly,0), (is,1), (fun,2))
`truth.indices zip truth`	List((0,Fly), (1,is), (2,fun))
`truth.grouped(2)`	Iterator: List(Fly, is), List(fun)
`truth.sliding(2)`	Iterator: List(Fly, is), List(is, fun)
`truth.sameElements( List("Fly", "is", "fun") )`	true

Arrays, Queues, Controls

CROSSINGTECH

Queues

Mutable and immutable first-in-first-out sequence.

Immutable

```
import scala.collection.immutable.Queue
val empty = new Queue[Int]
val has1 = empty.enqueue(1)
val has123 = has1.enqueue(List(2, 3))
val (element, has23) = has123.dequeue
element: Int = 1 has23: scala.collection.immutable.Queue
[Int] = Queue(2,3)
```

Mutable

```
import scala.collection.mutable.Queue
val queue = new Queue[String]
queue += "a"
queue ++= List("b", "c")
queue
 scala.collection.mutable.Queue[String] = Queue(a, b, c)
queue.dequeue
res0: String = a
queue
res1: scala.collection.mutable.Queue[String] = Queue(b, c)
```

Stacks

Mutable and immutable last-in-first-out sequence.

Mutable

```
import scala.collection.mutable.Stack
val stack = new Stack[Int]
stack.push(1)
stack.push(2)
stack.top
res0: Int = 2
stack
res1: scala.collection.mutable.Stack[Int] = Stack(1, 2)
stack.pop
res2: Int = 2
stack
res3: scala.collection.mutable.Stack[Int] = Stack(1)
```

Arrays

Creation:

```
val greetStrings: Array[String] = new Array[String](3)
val greetStrings = new Array[String](3)
val greetStrings = Array("Hello",",","," world!\n")
```

Access:

```
greets(0) = "Hello" or greets.update(0, "Hello")
greets(1) = ", "
greets(2) = "world!\n"
for (i <- 0 to 2) print(greets(i))
```

explode array

```
def max(values: Int*) = values.foreach(print)
max(Array(1,2,3,4,5): _*) // ':_*' tell compiler to pass
12345                       each elements
```

ArrayBuffer

An ArrayBuffer is like an array, except that you can additionally add and remove elements from the beginning and end of the sequence.

```
import scala.collection.mutable.ArrayBuffer
val buf = new ArrayBuffer[Int]()
buf += 1
buf += 2
```

Control Structures

The only control structure are:

```
if, while, for, try, match
```

IF

```
println(if (!args.isEmpty) args(0) else "default.txt")
```

while (imperative Style)

```
var i = 0
while (i < args.length) {
  println(args(i))
  i += 1
}
```

FOR

```
for (arg <- args) println(arg)
for (i <- 0 to 5) print(i)          012345
for (i <- 0 until  5) print(i)      01234
```

FILTERING

```
for ( file <- filesHere
    if file.isFile;
    if file.getName.endsWith(".scala")
) println(file)
```

If you add more than one filter on a generator, the filter's if clauses must be separated by semicolons. This is why there's a semicolon after the "if file.isFile"

NESTED

```
def fileLines(file: java.io.File) =
  scala.io.Source.fromFile(file).getLines.toList

def grep(pattern: String) =
  for (
    file <- filesHere
    if file.getName.endsWith(".scala"); // <-- semi-colon
    line <- fileLines(file)
    trimmed = line.trim // Mid-stream variable bindings
    if trimmed.matches(pattern)
) println(file +": "+ trimmed)
grep(".*gcd.*")
```

Return new collection:

```
for clauses yield body
```

Generates a collection with elements of each iteration.

```
var validUsers = for {
    user          <- newUserProfiles
    userName      <- user get "userName"
    name          <- user get "name"
    email         <- user get "email"
    bio           <- user get "bio"
  } yield new User(userName, name, email, bio)
```

TRY

```
try {
    val f = new FileReader("input.txt") // Use and close file
} catch {
    case ex: FileNotFoundException => // missing file
    case ex: IOException => // Handle other I/O error
}
```

FINALLY

Used only to close opened resources.

```
val file = new FileReader("input.txt")
try {
    // Use the file
} finally {
    file.close() // Be sure to close the file
}
```

MATCH

```
firstArg match {
    case "salt"    => println("pepper")
    case "chips"   => println("salsa")
    case "eggs"    => println("bacon")
    case _         => println("huh?")
}
```

FOREACH

(functional Style to print Main Args):

```
args.foreach((arg: String) => println(arg))
args.foreach(arg => println(arg))
args.foreach(println)
```

Functional Programming & XML

Functions

Scala has first-class functions. Not only can you define functions and call them, but you can write down functions as unnamed literals and then pass them around as values.

General Definition:
```
def name(x: Type, y: Type)[: Type] = { ... }
```

Function literal (closed term):
```
(x: Type, y: Type) => x + y
```

Function literal (closure or open term):
```
var more = 10
(x: Type, y: Type) => x + y + more
```

Save function in a variable:
```
var increase = (x: Int) => { println("Add 1"); x + 1}
increase: (Int) => Int = <function>
increase(10)
Add1
res0: Int = 11
```

Placeholder:
```
numbers.filter(x => x > 0)
numbers.filter(_ > 0)
```

Partially applied function:
```
someNumbers.foreach(x => println(x))
someNumbers.foreach(println _)
someNumbers.foreach(println ) // all same result
```

knowing the function:
```
def sum(a: Int, b: Int, c: Int) = a + b + c
sum(1, 2, 3)
res0: Int = 6
val a = sum _
a(1, 2, 3)
res1: Int = 6
val b = sum(1, _: Int, 3)
b(5) // same as sum(1, 5, 3)
res15: Int = 9
```

Repeated parameters
```
def echo(args: String*) = for (arg <- args) println(arg)
```

Cast an array as repeated parameters
```
val arr = Array("What's", "up", "doc?")
echo(arr: _*)
```

void (java) = Unit (Scala)

No Return statement:
```
def max2(x: Int, y: Int) = if (x > y) x else y
```

A Function could have a symbol as name. This allow to add for eg the operator "+" just by defining def + (that: Type)

Parameterized Methods

Scala's parameterized types are similar to Java and C# generics and C++ templates.
```
onInOut[+Ti, +To]( infoInput:Ti ): To
```

Named parameter / Default

Possibility to get the parameter by name instead of their position. Furthermore, you can assign a default value
```
def draw(x: Int  y: Int, dblBuff: Boolean = false) = { ... }
draw(dblBuff=true, x=0, y=10)
```

Implicit

This is a standard function definition starting with implicit. Once declared like that the compiler can use it to perform type conversion.
```
implicit def intToString(x: Int) = x.toString
```

Rules:

Scope	An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conversion.
Non-Ambiguity	An implicit conversion is only inserted if there is no other possible conversion to insert.
One-at-a-time	Only one implicit is tried.
Explicits-First	Whenever code type checks as it is written, no implicits are attempted.

Improve the code

Using closure to reduce code duplication.
```
object FileMatcher {
private def filesHere = (new java.io.File(".")).listFiles
private def filesMatching(matcher: String => Boolean) =
  for (file <- filesHere; if matcher(file.getName))
    yield file
def filesEnding(query: String) =
        filesMatching(_.endsWith(query))
def filesContaining(query: String) =
        filesMatching(_.contains(query))
def filesRegex(query: String) =
        filesMatching(_.matches(query))
}
```

Simplifying code

Avoid loops to search an elements.
```
def hasNeg(nums: List[Int]) = nums.exists(_ < 0)

def hasOdd(nums: List[Int]) = nums.exists(_ % 2 == 1)
```

Currying

Currying is the technique of transforming a function that takes more than one parameter into a function that takes multiple parameter lists, the primary use for currying is to specialize functions for particular types of data.
```
def multiplier(i: Int)(factor: Int) = i * factor
val byFive = multiplier(5) _
val byTen = multiplier(10) _
```

It's possible to curry a function:
```
val f = (x: Double, y: Double, z: Double) => x * y / z
val fc = f.curry
```

Control structures

This is used to implement patterns like "*loan pattern*":
```
def withPrintWriter(file: File) (op: PrintWriter => Unit) {
  val writer = new PrintWriter(file)
  try {
    op(writer)
  } finally {
    writer.close()
  }
}
```

to call this code
```
val file = new File("date.txt")
withPrintWriter(file) {
writer => writer.println(new java.util.Date)
}
```

By Name parameters

A by-name parameter is specified by omitting the parentheses that normally accompany a function parameter. Once defined like that the parameter is not evaluated until it's called within the function.
```
def myWhile(conditional: => Boolean)(f: => Unit) {
 if (conditional) {
   f
  myWhile(conditional)(f)
 }
}
```

To use this code
```
var count = 0
myWhile(count < 5) {
 println("still awesome")
 count += 1
}
```

XML

```
val myXML =
<html>
 <head>
  <script type="text/javascript">
   document.write("Hello")</script>
  <script type="text/javascript">
   document.write("world!")</script>
 </head>
 <body id="bID">some Text</body>
</html>
myXML \ "body"
res0: scala.xml.NodeSeq =
<body id="bID">some Text</body>
(myXML \ "body").text
res1: String = some Text
myXML \\ "script"
res2: scala.xml.NodeSeq =
<script type="text/javascript">
 document.write(&quot;Hello&quot;)</script>
<script type="javascript">
 document.write(&quot;world!&quot;)</script>
(myXML \\ "script")(0) \ "@type"
res3: scala.xml.NodeSeq = text/javascript
```

Build an XML
```
val simple = <a> {3 + 4} </a>
res3: scala.xml.Elem = <a> 7 </a>
val xml = scala.xml.XML.loadString("<test>evt</test>")
xml: scala.xml.Elem = <test>event</test>
```

CDATA
```
val body = <body> {PCData(in.getBodyasTxt)} </body>
```

Serialization
```
abstract class Plane{
   val description: String
   val year: Int
   val licence: String
   override def toString = description
   def toXML =
   <plane>
    <desc>{description}</desc>
    <year>{year}</year>
    <licence>{licence}</licence>
   </plane>
   }
defined class Plane
val piper = new Plane {
   val description = "Versatile Plane"
   val year = 1967
   val licence = "HB-PNJ"
   }
piper: Plane = Versatile Plane
piper.toXML
res0: scala.xml.Elem =
<plane>
      <year>1967</year>
      <licence>HB-PNJ</licence>
</plane>
```

Deserialization
```
def fromXML(node: scala.xml.Node): Plane =
   new Plane {
    val year = (node \ "year").text.toInt
    val licence = (node \ "licence").text
   }
fromXML: (scala.xml.Node)Plane
```

Save to file
```
scala.xml.XML.saveFull("pa28.xml",
                       node, "UTF-8", true, null)
```

Load from file
```
val loadnode = xml.XML.loadFile("pa28.xml")
```

XML and Pattern

A pattern embedded in {} can use the full Scala pattern language, including binding new variables, performing type tests, and ignoring content using the _ and _* patterns
```
def proc(node: scala.xml.Node): String =
 node match {
   case <a>{contents}</a> => "It's an a: "+ contents
   case <b>{contents}</b> => "It's a b: "+ contents
   case _ => "It's something else."
}
```

If the tag has children then use the "_*" notation:
```
case <a>{contents @ _*}</a> => "It's an a: "+ contents
```

notice the "@" is a pattern with a variable binding.

Case Classes & Patterns

CROSSINGTECH

Case Classes

To create a case class add the keyword case on the class Definition:

```
case class Var(name: String)                    // get
case class Config(var name: String = "Me") // get/set
```

Effects:

1/ No needs to add new to create a class instance:

```
val v = Var("x")
```

2/ All arguments in the parameter list are maintained as fields:

```
v.name
```

3/ Compiler adds "natural" implementations of methods *toString*, *hashCode*, and *equals:*

```
println(v)
```

Var(x)

You can have secondary constructors in case classes, but they wont overload the apply method generated that has the same argument list. You'll have to use new to create instances with those constructors.

Copy Method

On a case class a copy method is generated which allow to create a modified copy of an existing instance.

The definition is

```
case class A[T](a: T, b: Int) {
// def copy[T'](a: T' = this.a, b: Int = this.b): A[T'] =
//                                          new A[T'](a, b)
}
val a1: A[Int] = A(1, 2)
val a2: A[String] = a1.copy(a = "someString")
```

Pattern Matching

General definition:

```
selector match { alternatives }
```

Match example

```
def matchOn(shape: Shape) =
  shape match {
    case Circle(center, radius) =>
          println("Circle: center = "+center+", radius =
                  "+radius)
    case Rectangle(ll, h, w) =>
          println("Rectangle: lower-left = "+ll+", height =
                  "+h+", width = "+w)
    case Triangle(p1, p2, p3) =>
          println("Triangle: point1 = "+p1+", point2 =
                  "+p2+", point3 = "+p3)
    case _ => println("Unknown shape!"+shape)
}
```

Wildcard Pattern

Wildcards can also be used to ignore parts of an object that you do not care about.

```
expr match {
  case BinOp(_, _, _) =>
     println(expr +"is a binary operation")
  case _ => println("It's something else")
}
```

Constant Pattern

```
def describe(x: Any) = x match {
    case 1                  => "one"
    case true               => "truth"
    case "hello" | "Ciao"   => "hi!"
    case i: Int             => "scala.Int"
    case Nil                => "the empty list"
    case _                  => "something else"
}
```

Variable Pattern

```
expr match {
    case 0 => "zero"
    case <tag>{ t }</tag> => t
    case somethingElse => "not zero: "+ somethingElse
}
```

A variable pattern matches any object, just like a wildcard. Unlike a wildcard, Scala binds the variable to whatever the object is.

Sequence Pattern

```
expr match {
    case List(0, _, _) => println("found it")
    case _ =>
}
```

If you want to match against a sequence without specifying how long it can be, you can specify _* as the last element of the pattern.

```
expr match {
    case List(0, _*) => println("found it")
    case _ =>
}
```

Tuple Pattern

```
def tupleDemo(expr: Any) = expr match {
    case (a, b, c) => println("matched "+ a + b + c)
    case _ =>
}
```

Applied on List;

```
val List(a, b, c) = fruit
```

a: String = apples
b: String = oranges
c: String = pears

Typed Pattern

```
def generalSize(x: Any) = x match {
    case s: String => s.length
    case m: Map[_, _] => m.size
    case _ => -1
}
```

Variable-Binding pattern

```
item match {
    case (id, p @ Person(_, _, Manager)) =>
              format("%s is overpaid.\n", p)
    case (id, p @ Person(_, _, _)) =>
              format("%s is underpaid.\n", p)
    case _  =>
}
```

Sealed Classes

If you know that the case class hierarchy is unlikely to change and you can define the whole hierarchy in one file. In this situation, you can add the sealed keyword to the declaration of the common base class. When sealed, the compiler knows all the possible classes that could appear in the match expression, because all of them must be defined in the same source file.
So, if you cover all those classes in the case expressions (either explicitly or through shared parent classes), then you can safely eliminate the default case expression.

```
sealed abstract class HttpMethod()
case class Connect(body: String) extends HttpMethod
case class Delete  (body: String) extends HttpMethod
case class Get     (body: String) extends HttpMethod
case class Head    (body: String) extends HttpMethod
case class Options (body: String) extends HttpMethod
case class Post    (body: String) extends HttpMethod
case class Put     (body: String) extends HttpMethod
case class Trace   (body: String) extends HttpMethod
```

No default case is necessary (otherwise -> error)

Option Type

As everything is an object in Scala, instead of returning null from a method then use the object None.If the return is not null then return Some(x) where is the actual value.

```
def show(x: Option[String]) = x match {
    case Some(s) => s
    case None => "?"
}
```

Patterns in variable definitions

```
val myTuple = (123, "abc")
val (number, string) = myTuple
```

number: Int = 123 string: java.lang.String = abc

Case sequences as partial functions

```
react {
    case (name: String, actor: Actor) =>
              actor ! getip(name) act()
     case msg =>
              println("Unhandled message: "+ msg) act()
}
val second: PartialFunction[List[Int],Int] = {
    case x :: y :: _ => y
}
```

Partial Function

A partial function of type PartialFunction[A, B] is a unary function where the domain does not necessarily include all values of type A. The function isDefinedAt allows to test dynamically if a value is in the domain of the function.
PartialFunction trait defines a method *orElse* that takes another *PartialFunction.*

```
val truthier: PartialFunction[Boolean, String] = {
    case true => "truthful" }
val fallback: PartialFunction[Boolean, String]= {
    case x => "sketchy" }
val tester = truthier orElse fallback
println(tester(1 == 1)) println(tester(2 + 2 == 5))
```

Patterns in for expressions

Using Tuple pattern:

```
for ((country, city) <- capitals)
        println("The capital of "+ country +" is "+ city)
```

Using Option type:

```
val results = List(Some("apple"), None, Some("orange"))
for (Some(fruit) <- results) println(fruit)
```

apple
orange

Extractor

An extractor in Scala is an object that has a method called unapply as one of its members.

```
object EMail {
// The injection method (optional)
  def apply(user: String, domain: String) =
            user +"@"+ domain
// The extraction method (mandatory)
  def unapply(str: String): Option[(String, String)] = {
            val parts = str split "@"
            if (parts.length == 2) Some(parts(0), parts(1))
            else None
  }
}
```

The unapply method is called an extraction and can be used in pattern matching;

```
val x: Any = ...
x match { case EMail(user, domain) => ... }
```

Regular Expressions

the syntax is inherited from JAVA.

```
import scala.util.matching.Regex
val Decimal = new Regex("""(-)?(\d+)(\.\d*)?""")
// Or more simpler
"""(-)?(\d+)(\.\d*)?""".r
```

Decimal: scala.util.matching.Regex = (-)?(\d+)(\.\d)?*

Searching:

```
val input = "-1.0 to 99 by 3"
for (s <- Decimal findAllIn input) println(s)
```

-1.0
99
3

```
Decimal findFirstIn input
```

res1: Option[String] = Some(-1.0)

```
Decimal findPrefixOf input
```

res2: Option[String] = Some(-1.0)

Extracting:

```
val Decimal(sign, integerpart, decimalpart) = "-1.23"
```

sign: String = -
integerpart: String = 1
decimalpart: String = .23

```
val Decimal(sign, integerpart, decimalpart) = "1.0"
```

sign: String = null
integerpart: String = 1
decimalpart: String = .0