

scala cheat sheet

functional programming

1. functions are first-class values
2. immutable data, no side effects

variables

```
var x = 1      // mutable
val x = 1      // immutable
val s = "foo"  // implicit type
val i: Int = 1 // explicit type
```

loops

```
while (i < foo) {
  println(i)
}

do {
  // stuff
} while (condition)

for (arg <- args) println(arg)
for (i <- 0 to 5) println(i)
for (i <- 0 until 10 by 2)
  println(i)

for (
  file <- files
  if file.isFile
  if file.getName.endsWith(".a")
) doSomething(file)

args.foreach(arg => println(arg))
args.foreach(println(_))
args.foreach(println)
```

control structures

```
if (a == b) foo()
if (a == b) foo else bar
if (a == b) {
  foo
} else if (a == c) {
  bar
} else {
  baz
}

'==' is not reference equality

foo match {
  case "a" => doA()
  case "b" => doB()
  case _   => doDefault
}
```

functions

```
def hello(s: String): Unit = {
  println("Hello " + s)
}

def hello: Unit = println("Hello")
def hello = println("Hello")
// not recommended
def hello { println("Hello") }

// variable length args
def foo(args: String*) = {
  args.foreach(...)

  // named args
  point(x=10, y=20)

  // can call without a dot or parens if
  // it takes only one param
  var x = f.add(10)
  var x = f add 10

  // function literal syntax
  (x: Int, y: Int) => x + y

  - functions return last value computed
    by method
  - if name ends in `:`, invoke on right
    operand
  - `+` is a method on Int, String
  - apply(), update()
```

data types

```
val names = Array("Al", "Bob")
val names = new Array[String](5)
val names = Map(99676 -> "AK")

val names = List("Al", "Bob")
val names2 = "Joe" :: names
```

```
// tuples
val things = (100, "Foo")
println(things._1)
println(things._2)
```

collections

```
Sequence, List, Array, ListBuffer,
ArrayBuffer, StringOps, Set, Map,
TreeSet, Stream, Vector, Stack,
Queue, Range, BitSet, ListMap, more

- Tuples can hold different objects
- Mutable and immutable collections
- traits: Traversable, Iterable, Seq,
  IndexedSeq, LinearSeq, Buffer
```

try, catch, finally

```
try {
  something
} catch {
  case ex: IOException => // handle
  case ex: FileNotFoundException =>
    // handle
} finally { doStuff }
```

classes and objects

```
package foo.bar

import java.io.File
import java.io._
import java.io.{Foo, File => Bar}

class A { ... }
class A (s: String) { ... }
class A (val s: String) { ... }
class A (private val s: String) { ... }
class A (var s: String) { ... }

class Person (s: String) {
  require(name != "Joe")
  val name: String = s
  private val a = "foo"
}

class Bird extends Animal with Wings {
  override val foo = true
}

// singleton objects
object Foo {
  def main(args: Array[String]) = {
    args.foreach(println)
  }
}

object Bob extends Person { ... }
```

```
// abstract class
abstract class Person {
  // method with no implementation
  def walk: Unit
}
```

```
class Employee(name: String)
extends Person {
  ...
}
```

```
// sealed class - no new subclasses
// unless defined in current file
sealed abstract class Foo { ... }
case class Bar(s: String) extends Foo
```

traits

```
traits like class except:
1 - no class params
2 - super dynamically bound

trait Talks {
  def speak() {
    println("Yada yada yada...")
  }
}

class A { ... }
trait T1 { ... }
trait T2 { ... }
class B extends T1 { ... }
class C extends A with T1 with T2
```

scripts

```
println(args(0))
println(args.toList)
args.foreach(println)
scala foo.scala

#!/bin/sh
exec scala "$@" "$@"
!#
object Hello {
  def main(args: Array[String]) {
    args.foreach(println)
  }
}

// Application trait
object Hello extends Application {
  args.foreach(println)
}
```

underscore

```
"think of _ as a blank that needs
to be filled in"

strings.map(_.toUpperCase())
(1 to 10).map(_*2)

args.foreach(println(_))
args.foreach(println)

numbers.filter(_ < 10)

// for each element in the array
println(array: _*)
```

scala cheat sheet

case classes

```
abstract class Expr
case class Var(name: String) extends Expr
case class Num(num: Double) extends Expr
```

scala adds syntactic conveniences:

- 1) adds a factory method with the name of your class
- 2) all args in param list implicitly get a val, and become fields
- 3) add implementations of toString, hashCode, and equals
- 4) adds a copy method

examples:

- 1) val v = Var("x")
- 2) v.name
- 3) println(v) (shows toString), '==' works
- 4) v.copy

see <http://www.scala-lang.org/node/107>

case, match

```
selector match { choices }
_ is the 'wildcard pattern'

def f(x: Int): String = x match {
  case 1|2|3 => "1-2-3"
  // default
  case _ => "huh?"
}

def f(x: Any): String = x match {
  case 1 => "one"
  case "2" => "two"
  // typed pattern
  case i:Int => "got an int"
  case s:String => "got a string"
  case _ => // do nothing
}
```

pattern matching:

1. constant pattern ("a", 10)
2. variable pattern (x)
3. wildcard pattern (_)
4. constructor pattern (Foo("-", e))
5. typed pattern (see above)

isInstanceOf and asInstanceOf are discouraged

actors

```
import scala.actors._
object Foo extends Actor {
  def act() {
    // your logic here
  }
}

import scala.actors.Actor._
val hiActor = actor {
  while(true) {
    receive {
      case msg =>
        // your logic
    }
  }
}

object Foo extends Actor {
  def act() {
    react {
      case ...
    }
  }
}

// send message
hiActor ! "hello"

notes:
1. share-nothing, message-passing model
2. receive, receiveWithin
3. react is more efficient
4. don't block when processing messages (helper actor)
5. prefer immutable messages
6. make messages self-contained
```

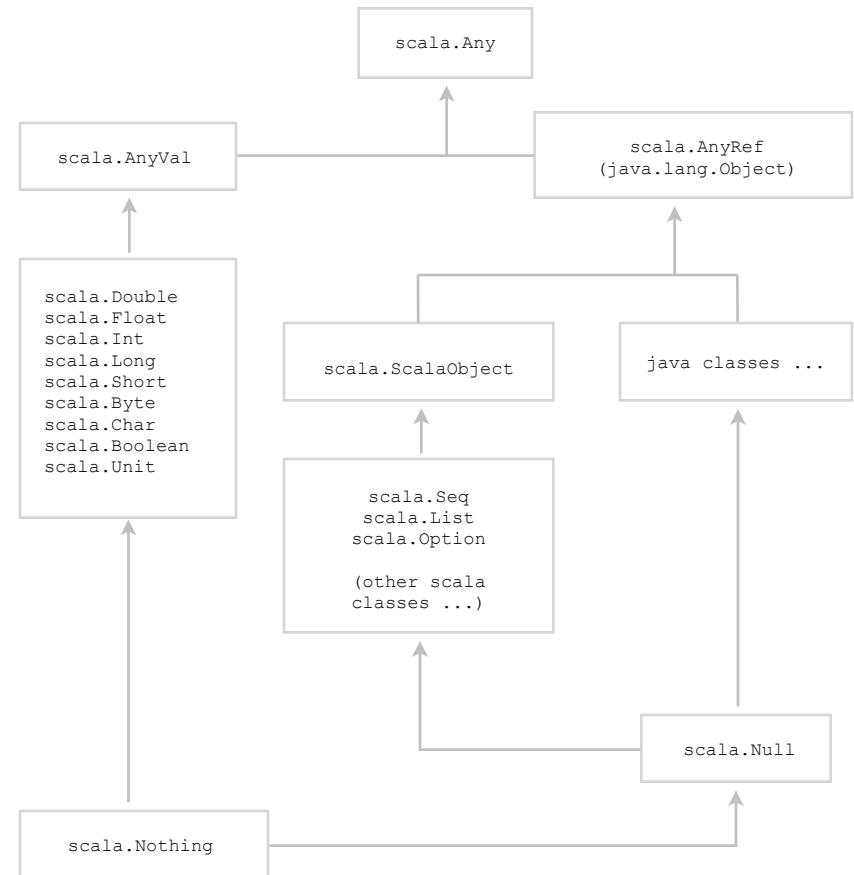
much more

```
// type alias
type D = Double

// anonymous function
(x:D) => x + x

// lisp cons
var x = 1 :: List(2,3)

var(a,b,c) = (1,2,3)
val x = List.range(0,20)
```



<http://www.scala-lang.org/node/71>

more information

<http://scala-lang.org>

<http://www.scala-lang.org/docu/files/collections-api/collections.html>

Mailing lists: <http://www.scala-lang.org/node/199>

Lift framework: <http://liftweb.net>