=Scala=

CHEAT SHEET v.0.1

"Every value is an object & every operation is a message send."

PACKAGE

Java style:

```
package com.mycompany.mypkg
```

applies across the entire file scope

Package "scoping" approach: curly brace delimited

```
package com
{
  package mycompany
  {
    package scala
    {
      package demo
      {
        object HelloWorld
        {
          import java.math.BigInteger
          // just to show nested importing
          def main(args : Array[String]) :
          Unit =
          { Console.println("Hello there!")
          }
        }
      }
    }
  }
}
```

IMPORT

```
import p._         // imports all members of p
// (this is analogous to import p.* in Java)

import p.x         // the member x of p
import p.{x => a}  // the member x of p renamed
                   // as a
import p.{x, y}    // the members x and y of p
import p1.p2.z     // the member z of p2,
                   // itself member of p1
import p1._, p2._  // is a shorthand for import
                   // p1._; import p2._
```

implicit imports:

the package `java.lang`
the package `scala`
and the object `scala.Predef`

Import anywhere inside the client Scala file, not just at the top of the file, for scoped relevance, see example in Package section.

VARIABLE

```
var var_name: type = init_value;
```

eg. `var i : int = 0;`

default values:

```
private var myvar: T = _ // "_" is a default
value
```

`scala.Unit` is similar to `void` in Java, except Unit can be assigned the () value.

```
unnamed2: Unit = ()
```

default values:

0 for numeric types
false for the Boolean type
() for the Unit type
null for all object types

CONSTANT

Prefer val over var.

form: `val var_name: type = init_value;`

```
val i : int = 0;
```

STATIC

No static members, use Singleton, see Object

CLASS

Every class inherits from `scala.Any`

2 subclass categories:

`scala.AnyVal` (maps to java.lang.Object)
`scala.AnyRef`

form: `abstract class(pName: PType1, pName2: PType2...) extends SuperClass`

with optional constructor in the class definition:

```
class Person(name: String, age: int) extends
Mammal {
  // secondary constructor
  def this(name: String) {
    // calls to the "primary" constructor
    this(name, 1);
  }
  // members here
}
```

predefined function `classOf[T]` returns Scala class type T

OBJECT

A concrete class instance and is a singleton.

```
object RunRational extends Application
{
  // members here
}
```

MIXIN CLASS COMPOSITION

Mixin:

```
trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) {
    while (hasNext) f(next)
  }
}
```

Mixin Class Composition:

The first parent is called the superclass of Iter, whereas the second (and every other, if present) parent is called a mixin.

```
object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0))
      with RichIterator
    val iter = new Iter
    iter foreach println
  }
}
```

note the keyword "with" used to create a mixin composition of the parents StringIterator and RichIterator.

TRAITS

Like Java interfaces, defines object types by specifying method signatures, can be partially implemented. See example in Mixin.

GENERIC CLASS

```
class Stack[T] {
  // members here
}
```

Usage:

```
object GenericsTest extends Application {
  val stack = new Stack[Int]
  // do stuff here
}
```

note: can also define generic methods

INNER CLASS

example:

```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if
(connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
  // members here
}
```

usage:
```
object GraphTest extends Application {
  val g: Graph = new Graph
  val n1: g.Node = g.newNode
  val n2: g.Node = g.newNode
  n1.connectTo(n2)      // legal
  val h: Graph = new Graph
  val n3: h.Node = h.newNode
  n1.connectTo(n3)      // illegal!
}
```
Inner classes are bound to the outer object, so a node type is prefixed with its outer instance and can't mix instances.

CASE CLASSES
See http://www.scala-lang.org/node/107 for info.

METHODS/FUNCTIONS
Methods are Functional Values and Functions are Objects

form: `def name(pName: PType1, pName2: PType2...) : RetType`

use `override` to override a method
```
override def toString() = "" + re + (if (im < 0) "" else "+") + im + "i"
```
Can override for different return type.

"=>" separates the function's argument list from its body
```
def re = real // method without arguments
```
Anonymous:

(function params) | rt. arrow | function body
```
(x : int, y : int) => x + y
```

OPERATORS
All operators are functions on a class.

Have fixed precedences and associativities:
```
(all letters)
|
^
&
< >
= !
:
+ -
/ %
*
(all other special characters)
```
Operators are usually left-associative, i.e. x + y + z is interpreted as (x + y) + z, except operators ending in colon ':' are treated as right-associative.

An example is the list-consing operator "::". where, `x :: y :: zs` is interpreted as `x :: (y :: zs)`.

eg.
```
def + (other: Complex) : Complex = {
  //....
}
```

Infix Operator:

Any single parameter method can be used :
```
System exit 0
Thread sleep 10
```

unary operators - prefix the operator name with "unary_"
```
def unary_~ : Rational = new Rational(denom, numer)
```

The Scala compiler will try to infer some meaning out of the "operators" that have some predetermined meaning, such as the += operator.

ARRAYS
arrays are classes
```
Array[T]
```
access as function:
```
a(i)
```
parameterize with a type
```
val hellos = new Array[String](3)
```

MAIN
```
def main(args: Array[String])
```
return type is `Unit`

ANNOTATIONS
See http://www.scala-lang.org/node/106

ASSIGNMENT
=
```
protected var x = 0
```
<-

`val x <- xs` is a generator which produces a sequence of values

SELECTION
The else must be present and must result in the same kind of value that the if block does
```
val filename =
  if (options.contains("configFile"))
    options.get("configFile")
  else
    "default.properties"
```

ITERATION
Prefer recursion over looping.

while loop: similar to Java

for loop:
```
// to is a method in Int that produces a Range object
for (i <- 1 to 10; i % 2 == 0) // the left-arrow means "assignment" in Scala
  System.out.println("Counting " + i)
```
`i <- 1 to 10` is equivalent to:

`for (i <- 1.to(10))`

`i % 2 == 0` is a filter, optional

`for (val arg <- args)`

maps to `args foreach (arg => ...)`

More to come...

REFERENCES
The Busy Developers' Guide to Scala series:

- "Don't Get Thrown for a Loop", IBM developerWorks
- "Class action", IBM developerWorks
- "Functional programming for the object oriented", IBM developerWorks

Scala Reference Manuals:

- "An Overview of the Scala Programming Language" (2. Edition, 20 pages), scala-lang.org
- A Brief Scala Tutorial, scala-lang.org
- "A Tour of Scala", scala-lang.org

"Scala for Java programmers", A. Sundararajan's Weblog, blogs.sun.com

"First Steps to Scala", artima.com