Every value is an object and every operation is a message send.

**PACKAGE**
Java style:
```scala
class MyPackage {
  def method() = 
}
```

```
package com.

// scoping: curly brace delimited
package com {
  package tedneward {
    package scala {
      package demonstration {
        object App {
          // just to show nested importing
          def main(args : Array[String]) : Unit =
          {
            System.out.println("Howdy, from packaged code!")
          }
        }
      }
    }
  }
}
```

**IMPORT**
```
import java.math.BigInteger
import java.lang.*
```

```
import p._ // imports all members of p
import p.x
import p.{x => a}
import p.{x, y}
import p1.p2.z
import p1._, p2._
```

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

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

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

**CLASS**
```
abstract class Person(name: String, age: int) {
  // secondary constructor
def this(name: String) {
    this(name, 1); // call the "primary" constructor
  }
}
```

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

**INNER CLASS**
```
class Graph {
  class Node {
    var connectedNodes: List[Node] = Nil
    def connectTo(node: Node) {
      if (connectedNodes.find(node.equals).isEmpty) {
        connectedNodes = node :: connectedNodes
      }
    }
  }
}
```
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!
}

note that a node type is prefixed with its outer instance, can't mix instances

METHODS
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() = 

can override as contra/covariant (different return type)
'='>' separates the function's argument list from its body
def re = real // method without arguments

OPERATORS
all operators are functions on a class
operators 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 :: z)

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)

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

ANNOTATIONS
to come

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: same as in Java
for loop:
  if to is a method in Int that produces a Range object
  for (i <- 1 to 10 if 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:
• 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

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit
http://creativecommons.org/licenses/by-nc-sa/3.0/; or, (b) send a letter to Creative Commons, 171 2nd Street, Suite 300, San Francisco, California, 94105, USA.