

Scala Reference Guide

Packages

`import scala.collection._` wildcard to import everything from the collection library

`import scala.collection.Vector` specific import for the Vector class

`import scala.collection.{Vector, Sequence}` import multiple classes

`package pkgname` declare a package

Operators

`x op y` is `x.op(y)` infix notation where `op` can be `+`, `-`, `*`, `/`, `%`

`x op` is `x.op()` postfix notation

`x == y` compares two objects (calls equals method)

There is no `++`, `--` in Scala

Symbols

`;` optional end of line

`->` returns a two element tuple for a key, value pair

`<-` assign to in a for comprehension

`=>` used in function literals to separate arguments from the function body

`::` cons operator

`//` single-line comment

`/*...*/` multiline comment

Relational Operators

`||` or

`&&` and

`!` not

Comparison

`==` equals

`<` less than

`>` greater than

`<=` less than or equal to

`>=` greater than or equal to

Lambda Expression

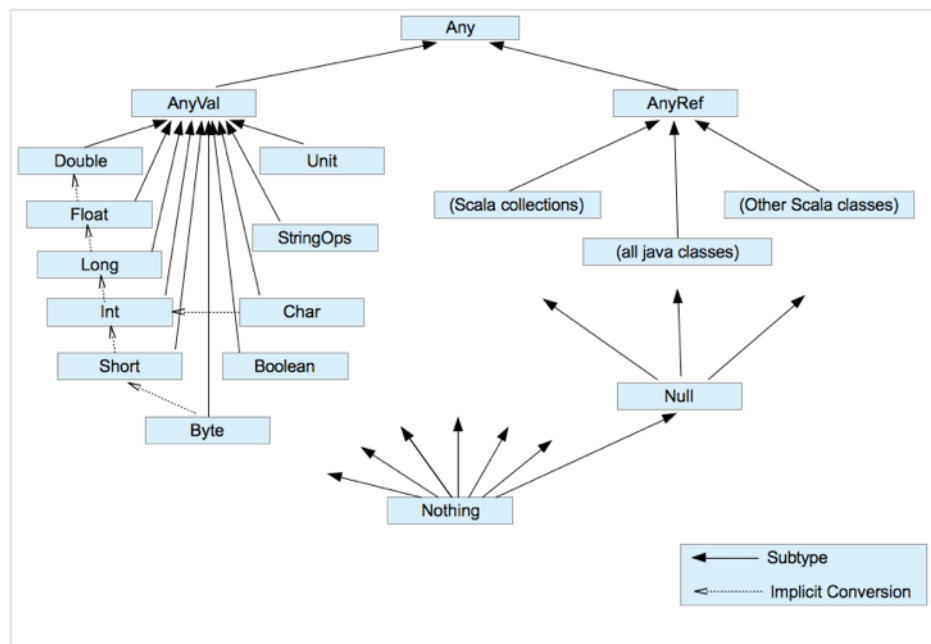
<code>(x:Int) => x * x</code>	anonymous function to square x
<code>(1 to 5).map(2 * _)</code>	anonymous function using bound infix method, multiplies 1,2,3,4,5 by 2
<code>val x = (1 to 5).map {2 * _</code>	multiplies each value by 2
<code>println(x)</code>	print x
<code>x }</code>	returns x (Vector (2, 4, 6, 8, 10))
<code>(1 to 10) filter { _ % 2==0}</code>	only returns even numbers; creates vector (2, 4, 6, 8, 10)
<code>(1 to 10) filter { _ % 2==0} map { _ * 2}</code>	multiplies all even values by 2; creates vector (4, 8, 12, 16, 20)

Variables

<code>var</code>	creates a mutable variable
<code>var myVar:Int</code>	creates a mutable integer variable
<code>val</code>	creates an immutable variable
<code>val myVal:String</code>	creates an immutable String variable (or <code>val myVal = "Monday"</code>)

Data Types

Byte
Short
Int
Long
Float
Double
Boolean
String
Char
Unit
Null
Nothing
Any
AnyRef



Functions

```
def f(x:Int) = {...}    define function f, with parameter x, an integer; no return type specified
def times3(x:Int) = 3 * x
val f = (x:Int) => 3 * x anonymous function call
def message(x:Int){    //function returns unit since it has no = sign; prints Hello world x times
  for(i<-(1 to x)) println("Hello World")}
def message(x:String, intro:String ="Dear") { //use a default value for intro
  println(intro + "," + x)}
def f(x: R)            call by value
def f(x: => R)         call by name (reference)
def sum(xs:Int*):Int = { //return type required for recursive functions
  // * indicates variable number of args
  var r = 0
  for(x <- xs) r += x
  r}
def sum(xs:Int*):Int = //same results as above
  if(xs.length == 0) 0 else xs.head + sum(xs.tail : _*)
```

Data Structures

```
(1,2,3)                tuple literal
var(a,b,c) = (1,2,3)   tuple unpacking via pattern matching
var xs = List(1,2,3)   creates an immutable list called xs
xs(0)                 access the element at location zero, indexing
4::List(3,2,1)        adds 4 to the front of the list creating List(4,3,2,1)
1 to 10                range of numbers from 1 to 10 inclusive
1 until 10            range of numbers from 1 to 9, excludes upper bound
val list = List.range(1,11) creates a List of values excluding the upper bounds
```

Decision Statements

`if(expr that evaluates to true/false) println("true") else println("false")`

Loops

`while(expr) {...}` execute a body of code while the expr is true
`do{...} while(expr)` execute a body of code at least once, continue while expr is true
`for(x <- myList) println(x)` print all values of x from the List called myList
`for(x <- myList if x%2 == 0) yield x*10` for comprehension
`for(x <- 1 to 10) {...}`

Pattern Matching

```
val x = r match {  
  case '0' => ... //match a value  
  case ch if someProperty(ch) => ... //add a guard to the match criteria  
  case e: Employee => ... //match runtime type  
  case (x,y) => ... //destructures pairs  
  case Some(v) => ... //case classes have extractors  
  case 0 :: tail => ... //infix notation for extractors yielding a pair  
  case _ => ... //default case
```

Escape Sequences

`\b` backspace
`\t` tab
`\n` newline
`\r` carriage return
`\"` double quote
`\'` single quote
`\\` backslash