Reference


Reading

Chapters 4 & 6
More Functions
def sum(a: Int, b:Int, c:Int) = {
    println("Start")
    a + b + c
}

val partialSum = sum(1,_, 5)
println("Before call")
val result = partialSum(2)
println(result)
def sum(a: Int, b:Int, c:Int) = {
    println("Start")
    a + b + c
}

val partialSum = sum(_:Int,_: Int,5)
println("Before call")
val result = partialSum(1,2)
println( result)
def sum(a: Int, b:Int, c:Int) =  a + b + c

val newSum = sum _
newSum(1,2,3)

def passSum(x: ((Int, Int, Int) => Int)): Int = {x(1,1,3)}

passSum(sum _)

passSum(sum)
Curried Functions

def curriedSum(x: Int)(y: Int) = x + y

val result = curriedSum(1)(2)

val partialSum = curriedSum(1)_
partialSum(2)

val noSumYet = curriedSum _
noSumYet(1)(2)
Classes
Sample Class

class Fraction {
    var numerator = 0
    private var denominator = 0

    def set(x: Int) = {this.denominator = x}

    override def toString() = numerator + "/" + denominator
}

val test = new Fraction
  test.numerator = 10
  test.set(3)
  println(test)
Protection Levels

private
   Same as Java/C++
   Accessible only in the class
protected
   Accessible in class
   Accessible in subclasses
   Not accessible other places
public
   Accessible in any class or function
   with reference to object
   Default protection level
Any

Root of class hierarchy

!=  
==

asInstanceOf
equals
hashCode
isInstanceOf
toString
class Foo(test: Int) {
    def getTest() = test
    override def toString() = "Foo: " + test
}

val x = new Foo(10)
println(x)
Constructors

class Fraction(n: Int, d: Int) {
    println("Start")
    private var numerator = n
    private var denominator = d

def this(x: Int) = {this(x,1); println("auxiliary")}

    override def toString() = numerator + "/" + denominator

    println("End")
}

val test = new Fraction(1,2)
val two = new Fraction(2)

Output
Start
End

Output
Start
End
auxiliary
class Fraction(n: Int, d: Int) {
  private var numerator = n
  private var denominator = d

  def this(x: Int) = {this(x,1); println("auxiliary")}

  def *(that: Int) = new Fraction(numerator*that, denominator)

  def *(that: Fraction) = new Fraction(numerator*that.numerator,
                                      denominator*that.denominator)

  override def toString()= numerator + "/" + denominator
}
Using the Operators

val halve = new Fraction(1,2)
var one = halve * 2
println(one)        //prints 2/2

val two = new Fraction(2)
one = halve * two
println(one)        //prints 2/2
But

val halve = new Fraction(1,2)
var one = 2 * halve  // Compile Error
Implicit Conversions

implicit def intToFraction(x: Int) = new Fraction(x)

val test: Fraction = 2

val halve = new Fraction(1,2)

var one = 2 * halve
println(one)  // Prints 2/2

The name "intToFraction" is just to help humans. The compiler only cares about the argument type and the return type. We could name function foo and it would work fine.
class Fraction(n: Int, d: Int) {
    require(d != 0)

    private var numerator = n
    private var denominator = d

    def this(x: Int) = {this(x,1); println("auxiliary")}
    def *(that: Int) = new Fraction(numerator*that, denominator)
    def *(that: Fraction) = new Fraction(numerator*that.numerator,
                                           denominator*that.denominator)
    override def toString() = numerator + "/" + denominator
}

new Fraction(1,0) //Causes exception
No static fields or methods

Use singleton objects
Singleton Objects

object JustOne {
  private var x = 0
  def getX(): Int = x
  def setX(x: Int) = this.x = x

  override def toString() = "JustOne: " + x
}

JustOne.setX(10)
println(JustOne.getX())
println(JustOne)
println(JustOne) // compile error
new JustOne // compile error
Companion objects & classes

class Fraction(n: Int, d: Int) {
    private var numerator = n
    private var denominator = d

    override def toString() =
        numerator + "/" + denominator
}

object Fraction {
    def zero() = new Fraction(0,1)
    def unity() = new Fraction(1,1)
}

val a = Fraction.zero()
val b = new Fraction(1,2)
Scala Application

object StartHere {
  def main(args: Array[String]) {
    for (arg <- args)
      println(arg)
  }
}

StartHere.main(Array("this", "is", "a", "test"))

Output
  this
  is
  a
  test
object Foo {
    private val x = 3

    println("Before main")

    def main(args: Array[String]) {
        println("In Main")
    }

    println("After main")
}

Foo.main(Array("test"))

Output
Before main
After main
In Main
Program & Scripts
Script

Script.scala

```
var x = 2
var y = 3
println(x + y)
```

Al pro 34-> scala Script.scala
5
Al pro 35->scalac Script.scala
Script.scala:1: error: expected class or object definition
var x = 2
^
Script.scala:2: error: expected class or object definition
var y = 3
^
Script.scala:3: error: expected class or object definition
println(x + y)
^
three errors found
Program

RunMe.scala

object RunMe {
  def main(args : Array[String]) : Unit = {
    var x = 2
    var y = 3
    println(x + y)
  }
}

Al pro 36->scalac RunMe.scala
Al pro 37->scala RunMe
6
Al pro 38->scala RunMe.scala
Al pro 39->

You can run this in Eclipse
Application Trait

Foo.scala

object Foo extends Application {
  private val x = 3
  println("Why use main")
  println("If you don use the args?")
}

Al pro 39->scalac Foo.scala
Al pro 40->scala Foo
Why use main
If you don use the args?
Some Old Magic Explained
Some New Magic
Where is println defined?

What other functions can we call?

object Foo extends Application{
    private val x = 3
    println("Why use main")
    println("If you don use the args?")
}
import with Script

Foo.scala

object Foo {
    def bar(x: String) {
        println("You gave me: " + x)
    }

    def bar() {
        println("Hi")
    }
}

sample.scala

import Foo.bar

Foo.bar()
Foo.bar("test")
bar()
bar("You dont need the full name")

Al pro 53->scala sample.scala
Hi
You gave me: test
Hi
You gave me: You dont need the full name
import with Program

Foo.scala

object Foo {
  def bar(x: String) {
    println("You gave me: " + x)
  }

  def bar() {
    println("Hi")
  }
}

RunMe.scala

import Foo._

object RunMe extends Application {
  bar()
  bar("from RunMe")
}

Al pro 55->scalac Foo.scala
Al pro 56->scalac RunMe.scala
Al pro 57->scala RunMe
Hi
You gave me: from RunMe
Al pro 58->
Where is println defined?

object Predef

Part of standard Scala library

Is imported in all Scala files

defines many methods
def factorial(n: BigInt): BigInt = {
    def factorial(n: BigInt, accumulator: BigInt): BigInt = {
        if (n <= 1)
            accumulator
        else
            factorial(n - 1, n * accumulator)
    }
    factorial(n, 1)
}

class Factorial(n :BigInt) {
    def !():BigInt = factorial(n)
}

implicit def intToFactorial(x: Int) = new Factorial(x)
Why Not Simple factorial?

def factorial(n: BigInt): BigInt = {
    if (n <= 1)
        1
    else
        n * factorial(n - 1)
}
class Repeat(code: => Unit) {
  def until(condition: => Boolean) = {
    while (!condition) { code }
  }

  def when(condition: => Boolean) = {
    while (condition) { code }
  }
}

def repeat(code: => Unit) = new Repeat(code)

var x = 0
repeat {
  println(x)
  x += 1
} when (x < 5)

var y = 0
repeat {
  println(y)
  y += 1
} until (y == 3)