References


Reading

Chapters 4 & 6
Program & Scripts
Script

Script.scala

```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
RunMe.scala

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

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?
Classes
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(var a: Int, val b: Int, c: Int) {
    override def toString() = "Foo: " + a + " " + b + " " + c
}

val x = new Foo(1,2,3)
x.a = 4
println(x.a)
println(x.b)
println(x.b) //Compile error
println(x)
println(x)
x.b = 5   //Compile error
x.c = 6   //Compile error

C is a class parameter as it does not have a val or var in its declaration. The difference between a field and a class parameter is that class parameters can not be accessed outside of the class, while fields can.
class Foo(var a: Int, val b: Int, c: Int) {
    override def toString() = "Foo: " + a + " " + b + " " + c

def test(x: Foo) {
    x.a;
    x.b;
    x.c;       //Compile Error
}
}
Access Control

class Foo(private var a: Int, val b: Int, c: Int) {
    override def toString() = "Foo: " + a + " " + b + " " + c
}

Optional period and ()

class Test {
    def bar(x: Int) = println(x)
}

def fooBar(x: Int) = println(x)

test.bar(3) // OK
fooBar(3)   // Compile error
Cascading

val data = List(2, -2, 3, -4, -5, 6)
def isNegative(x: Int) = x < 0

data.filter(isNegative).foreach(println)

data filter isNegative foreach println
Parameterless Methods

class Foo {
    var x = 0
    def a: Int = x
    def b(): Int = x
}

var result = 0
val test = new Foo
test.x = 1
result = test.x
result = test.a
result = test.b
result = test.b()
result = test.a() // Compile error
class Foo {
  var x = 0
  def a: Int = x
  def a_=(b: Int) { x = b }
}

var result = 0
val test = new Foo
  test.x = 1
result = test.x
  test.a = 2
result = test.a
Uniform Access Principle

Changing a field to a method should not affect client code

class Square( l: Int) {
    var length = l
    var area = length * length
}

val test = new Square(5)
test.area
test.length

class Square( l: Int) {
    var length = l
    def area() = length * length
}

val test = new Square(5)
test.area()
test.length
Field & Methods Name clash

class Foo {
    var x: Int = 0
    def x(a: Int) {x = a}
    def x(): Int = 4               // Compile error
    def x_=(a: Int) {x = a}       // Compile error
}
Overloading Method Names

Methods can have the same name if arguments differ in number or type

```scala
class Foo {
    def bar() = 5
    def bar = 5 // Compile error
    def bar(a: String) = 5
    def bar(a: Int) = 5
    def bar(a: Int): String = {"cat"} // Compile error
}
```
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)
Output
Start
End

val two = new Fraction(2)
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
}

There are no operators in Scala. * is a valid name for a method. Why should there be different rules for operators and methods?
Using the Operators

```scala
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
```
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)
val 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)
new JustOne // compile error
Companion objects & classes

```scala
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)
```

Companion objects and classes must be defined in the same source file
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
Magic
Where is println defined?

What other functions can we call?

```scala
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

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

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

RunMe.scala

```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)

val result = 10!
def factorial(n: BigInt): BigInt = {
  if (n <= 1)
    1
  else
    n * factorial(n - 1)
}
def repeatWhile(condition: => Boolean)(code: => Unit) {
    while (condition) {
        code
    }
}

var x = 0
repeatWhile (x < 4) {
    println(x)
    x += 1
}

Output
0
1
2
3
Blocks as Arguments

```scala
var y = 1;

def bar(x: Int) = {
  println("In Bar");
  println(x)
}

def foo(x: => Int) = {
  println("In foo");
  println(x)
}
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar( y + 1 )</td>
<td>In Bar 2</td>
</tr>
<tr>
<td>bar({y + 1})</td>
<td>In Bar 2</td>
</tr>
<tr>
<td>bar( { println(&quot;Call Bar&quot; ); y + 1 } )</td>
<td>Call Bar In Bar 2</td>
</tr>
<tr>
<td>bar( println(&quot;Call Bar&quot; ); y + 1)</td>
<td>Compile Error</td>
</tr>
<tr>
<td>bar{ println(&quot;Call Bar&quot; ); y + 1}</td>
<td>Call Bar In Bar 2</td>
</tr>
</tbody>
</table>
Scala's by name parameter passing is not the same as what other languages call pass by name. In Scala pass by name is a form of lazy evaluation. The expression passed as an argument is not evaluated until it is used in the function.
By Name Verses First Class Functions

Sigh, now we have two different mechanisms to pass code as an argument to a function. Wouldn't life be easier if they would work together
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)

Why all this syntax flexibility
Domain-Specific Language (DSL)

programming language dedicated to a particular problem domain

spreadsheet formulas
YACC grammars
UNIX shell scripts
ColdFusion scripting language
Why DSL

Hides implementation details

Expresses programs in terms of the domain

Helps developers understand domain

Domain experts can help verify implementation
Scala Example

```scala
val payrollCalculator = rules { employee =>
    employee salary_for 2.weeks minus_deductions_for { gross =>
        federalIncomeTax is (25. percent_of gross)
        stateIncomeTax is (5. percent_of gross)
        insurancePremiums are (500. in gross.currency)
        retirementFundContributions are (10. percent_of gross)
    }
}

val buck = Employee(Name("Buck", "Trends"), Money(80000))
val jane = Employee(Name("Jane", "Doe"), Money(90000))

List(buck, jane).foreach { employee =>
    val check = payrollCalculator(employee)
    format("%s %s: %s\n", employee.name.first, employee.name.last, check)
}
```