ScalaAcademy

Pragmatic Scala Course

abhiram devineni

Table Of Contents

Pragmatic Scala

Hello World

```
// No need for semicolons (";"), usually (unless we have multiple statements in the same line)
// We can have multiple public classes on the same file (and the name of the classes doesn't need to match the name of the file)
// Package names don't need to match folder structure
```

```scala
// The unavoidable hello world!

object Scratchpad {

  def main(args: Array[String]): Unit = {
    println("Hello world!")
  }
}
```

Variables

Primitive Types

```scala
object Scratchpad {

  // Line comment
  /*
   * Multi-line comment
   */
  /**
   * Scala Doc
   */

  // Tutorial:
  // A variable is defined like this:
  // var intVar: Int = 0
  // This is a variable called intVar, of type integer, with value 0.
  //
  // Likewise a double var is defined like:
  // var doubleVar: Double = 0.0

  // TASK:
  //   create the variables:
  //    · doubleVar
  //    · booleanVar
  //    · stringVar
  //   with the types:
  //    · Double
  //    · Boolean
  //    · String
  //   and the values:
  //    · 0.0
  //    · false
  //    · "hello world"
  var intVar: Int = 0
  // var ...
  var doubleVar: Double = 0.0
  var booleanVar: Boolean = false
  var stringVar: String = "hello world"



  // Complete list of types:
  // Byte  Short  Int
  // Long  Float  Double
  // Char  String Boolean

  def main(args: Array[String]): Unit = {
    println("intVar: " + intVar)
    println("doubleVar: " + doubleVar)
    println("booleanVar: " + booleanVar)
    println("stringVar: " + stringVar)
    val test1: Double = doubleVar
    val test2: Boolean = booleanVar
    val test3: String = stringVar
    ensure(doubleVar == 0.0)
    ensure(booleanVar == false)
    ensure(stringVar == "hello world")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Type inference

```
// Not magic! :)
var myDoubleVar = 0 // This is an Int!!
var myDoubleVar = 0.0 // Now this is a Double
var myDoubleVar = 0d // This is also a Double
var myDoubleVar = (0: Double) // Also works

// Multi-line comments:
/*
var intVar: Int = 0
*/
// Line comments:
//var intVar: Int = 0
```

```
object Scratchpad {

  // Tutorial: the compiler automatically infers this is an Int:
  var intVar = 0

  // TASK: Remove the types from these variables:
  var doubleVar  = 0.0
  var booleanVar = false
  var stringVar = "hello world"

  def main(args: Array[String]): Unit = {
    println("intVar: " + intVar)
    println("doubleVar: " + doubleVar)
    println("booleanVar: " + booleanVar)
    println("stringVar: " + stringVar)
  }
}
```

Variables (Vals)

```
// Constants code style:
val PI_CONSTANT = 3.14 // Java Style
val PiConstant = 3.14 // Scala Style
```

```
object Scratchpad {
  def main(args: Array[String]): Unit = {

    // Tutorial: A 'val' is an immutable variable, it cannot be changed.

    val Pi = 3.14 // <-- Pi cannot be changed

    // We will calculate the area of a circle with radius = 2cm

    // TASK[1/2]: create a val, 'radius', type Double, with value 2:
    // val...
    val radius: Double = 2

    // TASK[2/2]: create a val, 'area', type Double, with the area of the circle.
    // (Area of the circle = Pi*radius*radius)
    // val...
    val area: Double = Pi*radius*radius

    println("Area of the circle is: " + area)
    ensure(radius == 2)
    ensure(area == 12.56)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Lazy val

```
// A lazy val is a variable which is only evaluated when it is used, the first time it is used.

// Regarding having the curly braces on the right-hand-side (RHS) of the equals:
// We can have this in Java:
  int myInteger = 123;
  int myInteger = 123 + 0;
  int myInteger = (123 + 0);
  int myInteger = (123 + 0) * 1;
  int myInteger = ((123 + 0) * 1);
// -> i.e.: we just need anything which is an integer on the RHS of the equals
// In Scala it's the same, but curly braces work the same as parenthesis:
  val myInteger: Int = {{123} + {0}*1}
  val myInteger: Int = {123 + 0}
  val myInteger: Int = (123 + 0)
  val myInteger: Int = 123 + 0
  val myInteger: Int = 123
// Now: a particular thing about curly braces is that we can have multiple statements inside:
  val myInteger: Int = {println("hello"); 123}
  val myInteger: Int = {val myV = 123; myV}
// The value of the curly braces is the value of the last statement (123, in both previous cases)
```

```
object Scratchpad {

  lazy val value1 = {println("Accessing value 1"); 11}
  lazy val value2 = {println("Accessing value 2"); 22}

  println("First time:")
  println(value2)
  println(value1)

  println("Second time:")
  println(value2)
  println(value1)

  def main(args: Array[String]): Unit = { }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods

Methods - Simple

```
// **RETURNING VALUES**
// When we have a variable, we just need to have some value on the right-hand-side (RHS) of the equals of the correct type:
var myString: String = //[Some String Here!]
var myString: String = "hello"
var myString: String = ("hello" + "")
var myString: String = {"hello" + ""}
var myString: String = {"hello" + ""}.toUpperCase
var myString: String = API.getToken()
// Etc...
// With methods in Scala it's the same thing:
def getLog(): String = //[Some String Here!]
def getLog(): String = "hello"
def getLog(): String = ("hello" + "")
def getLog(): String = {"hello" + ""}
def getLog(): String = {"hello" + ""}.toUpperCase
def getLog(): String = API.getToken()
// Or, in multiple lines...:
def getLog(): String = {
  API.getToken()
}
// We also have the "return" keyword, but it's not a good practice to use it:
def getLog(): String = return API.getToken()

// **"Ternary" Operator Behavior**
// If we want to return something conditionally we can just use an "If":
def getLog(): String = if(Deploy.inProduction) "812duweuf" else "test"
// Because the "If" in Scala has a value, just like the ternary operator in Java:
// Ex.: Java version:
String myToken = (Deploy.inProduction) ? "812duweuf" : "test";
// Corresponding Scala version:
val myToken: String = if(Deploy.inProduction) "812duweuf" else "test"
```

```
object Scratchpad {

  var log: String = ""

  def logMessage(s: String): Unit = { // (Unit is the equivalent to "void" in Java)
    log = log + "\n" + s
  }

  // This is not a "getter" like there is in Java!:
  def getLog(): String = {
    return log
  }

  // TASK [1/3]: Create a var 'counter' (Int), initialize it with zero:
  // var...
  var counter: Int = 0

  // TASK [2/3]: Create a method incrementCounter, which increments the counter by an amount 'amount' (Int):
  // def ...
  def incrementCounter(amount: Int): Unit = {
    counter = counter + amount
  }

  // TASK [3/3]: Create a method count, which returns the value of the counter:
  // def ...
  def count() = counter



  def main(args: Array[String]): Unit =  {
    println("count(): " + count())
    ensure(count() == 0)
    println("incrementCounter(10)")
    incrementCounter(10)
    println("count(): " + count())
    ensure(count() == 10)
    println("incrementCounter(5)")
    incrementCounter(5)
    println("count(): " + count())
    ensure(count() == 15)
    val test1: Int = count()
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods - Pass by name 1

```
// A pass-by-name argument is identified by the arrow pointing to the type:
// def passByNameArgAgain(n: => Int): Unit = ...
// And is evaluated only when it is used, each time it is used.

// Simple "Profile" method example:
def value1(): Int = { println("[Computing 1...]"); 1 }

val myVar = profile(  value1()  )

def profile(n: => Int): Int = {
  val start = System.currentTimeMillis()
  val rslt = n + 0 // The "n" is only evaluated here (and we don't really need the "+0")
  println("Took " + (System.currentTimeMillis() - start) + "ms")
  rslt
}
```

```
object Scratchpad {
  def value1(): Int = { println("[Computing 1...]"); 1 }
  def value2(): Int = { println("[Computing 2...]"); 2 }
  def value3(): Int = { println("[Computing 3...]"); 3 }
  def value4(): Int = { println("[Computing 4...]"); 4 }

  normalArg(value1() + 0)
  passByNameArg(value2() + 0)
  passByNameArgIgnored(value3() + 0)
  passByNameArgAgain(value4() + 0)

  def passByNameArgAgain(n: => Int): Unit = {
    println("ENTERED passByNameArgAgain")
    println("N: " + n)
    println("N: " + n)
    println()
  }
  def passByNameArgIgnored(n: => Int): Unit = {
    println("ENTERED passByNameArgIgnored")
    println()
  }
  def passByNameArg(n: => Int): Unit = {
    println("ENTERED passByNameArg")
    println("N: " + n)
    println()
  }
  def normalArg(n: Int): Unit = {
    println("ENTERED normalArg")
    println("N: " + n)
    println()
  }

  def main(args: Array[String]): Unit = {}
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if (!b) throw new Exception(mesg)
}
```

Methods - Calling 1

```
// It makes sense to call many methods of one argument with this style:
profile {
  processSomething()
}
// (Equivalent to:)
profile(processSomething())

// Curiosity - the profile:
def profile[T](code: => T): T = {
  val start = System.currentTimeMillis()
  val rslt = code
  println("Took " + (System.currentTimeMillis() - start) + "ms")
  rslt
}

// Inside the curly braces you can have multiple statements:
profile {
  println("hello")
  processSomething()
}
```

```
object Scratchpad {

  var log: String = ""
  def logMessage(s: String): Unit = log = log + "\n" + s
  def getLog(): String = log

  var counter = 0
  def incrementCounter(amount: Int): Unit = if(counter < 100) counter = counter + amount
  def count() = counter


  def main(args: Array[String]): Unit = {
    // Tutorial: we can pass the arguments in curly braces instead of parens:
    logMessage {
      "Hello world"
    }

    // TASK: Call incrementCounter with 50 inside curly braces:
    // ...
    incrementCounter {
      50
    }

    println("count: " + count)
    ensure(count == 50)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods - Pass by name 2

```
object Scratchpad {

  val LogLevel = 3
  var log: String = ""

  // TASK: Make the mesg be a pass by name argument:
  def logMessage(lvl: Int, mesg: => String): Unit = {
    if(lvl >= LogLevel) log = log + "\n" + mesg
  }

  def getLog(): String = log

  logMessage(1, {throw new Exception("Your solution is not correct"); ""})

  def main(args: Array[String]): Unit = { println("Your solution looks correct.")}
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods - No paren

```
/*
Style rule for parenthesis:
 - Method with side effects: use parenthesis
 - Method without side effects: don't use parenthesis
*/

// With parenthesis:
println()
System.exit()
file.delete()
socket.connect()
// Without:
myList.size

// We can call a method defined with parenthesis with or without them:
def method1(): Unit = {}
method1() // ok
method1 // ok
// (Please keep in mind the style conventions, in any case)

// If it was defined without parenthesis, we cannot add them!:
def method2: Unit = {}
method2 // ok
method2() // compile error!
```

```
object Scratchpad {

  val LogLevel = 3
  var log: String = ""

  def logMessage(lvl: Int, s: => String): Unit = {
    if(lvl >= LogLevel) {
      log = log + "\n" + s
    }
  }

  // We can have methods without parenthesis:
  def getLog: String = {
    log
  }
  println("getLog: " + getLog)

  var counter = 0
  def incrementCounter(amount: => Int): Unit = {
    counter = counter + amount
  }
  // TASK [1/2]: Remove the parenthesis:
  def count = counter
  // TASK [2/2]: Remove the parenthesis here too:
  println( "count: " + count)

  def main(args: Array[String]): Unit = {
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods - Infix Notation

```
// Imports the class TimeZone:

import java.util.TimeZone

// Imports everything in the java.util package:

import java.util._

// Imports TimeZone and Locale:

import java.util.{TimeZone, Locale}

// Imports the TimeZone, with the name TZ:

import java.util.{TimeZone => TZ}

// Imports everything except the TimeZone:

import java.util.{_, TimeZone => _}

// Works:

import java.util
import      util.TimeZone

new util.TimeZone()


// Problem:

import java.util.java

import java.util.Locale // ups!

import _root_.util.Locale

// We can import anywhere in the file.

// That is, if you can have 'println("hello")' there,

// you can also have 'import java.util.TimeZone'.

val tz1 = TimeZone.getTimeZone("Europe/London")

// Imports the *method* getDisplayName of the tz1:

// (Also works for variables)

import tz1.getDisplayName

// Uses the method getDisplayName of tz1:

println(getDisplayName)

// Equivalent:

val sql = SQL.select("id").from("users").where("age > 20")

val sql = SQL select "id" from "users" where "age > 20"
```

```
import java.util.TimeZone

object Scratchpad {
  def main(args: Array[String]): Unit = {

    // Tutorial: we can call methods without the '.' and the parenthesis:
    val tz1 = TimeZone.getTimeZone("Europe/London")
    // Same:
    val tz2 = TimeZone getTimeZone "Europe/London"

    // TASK: Uncomment and change this into the infix notation:
    val tz3 = TimeZone getTimeZone "Europe/Belfast"


    ensure(tz3.getDisplayName == "Greenwich Mean Time")
    println("Timezone 3: " + tz3.getDisplayName)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods - Default arguments

```
object Scratchpad {

  var log: String = ""

  // Tutorial: The default log message is now "Error!":
  def logMessage(s: String = "Error!"): Unit = log = log + "\n" + s
  def getLog(): String = log

  logMessage()
  println("getLog: " + getLog)

  var counter = 0
  // TASK: Make the default amount be 1:
  def incrementCounter(amount: Int = 1): Unit = {
    counter = counter + amount
  }
  def count = counter

  incrementCounter(3)
  incrementCounter() // Should increment by 1
  incrementCounter(2)
  println("count: " + count)
  ensure(count == 6)

  def main(args: Array[String]): Unit = {
    incrementCounter()
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Methods - Named arguments

```
// Example with a factory method:
  def createUser(
                 accessLevel: Int = 3,
                 salary: Double = 1000,
                 firstName: String = "",
                 lastName: String = "",
                 email: String = ""
                 // ...
                 ): User = {
    // ...
  }

  createUser(firstName = "John", lastName = "Doe")
  createUser(
             firstName = "John",
             lastName = "Doe"
            )
```

```
object Scratchpad {

  val LogLevel = 3
  var log: String = ""

  def logMessage(lvl: Int = 3, mesg: => String = "Error!"): Unit = {
    if(lvl >= LogLevel) log = log + "\n" + mesg
  }

  def getLog(): String = log

  // Tutorial: we can use the argument names:
  // TASK [1/2]: switch the arguments positions:
  logMessage(mesg = "divide by 0", lvl = 4 )

  // TASK [2/2]: call logMessage, with mesg 'not found', and the default level:
  // logMessage...
  logMessage(mesg = "not found")

  println("getLog:\n" + getLog())

  ensure(log == "\ndivide by 0\nnot found")
  println("Your solution looks correct.")
  def main(args: Array[String]): Unit = { }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Classes

Classes - Intro

```
// private[this]:
class Cat {
  private[this] var tailSize: Double = 0.0

  // Works with private, doesn't work with private[this]:
  def compareTails(other: Cat): Boolean = tailSize > other.tailSize
  // With private this only the this.tailSize works, not the other.tailSize.
}

// package access:
class Cat {
  // now the tailSize is only accessible in the package "animals":
  private[animals] var tailSize: Double = 0.0
}
```

```
class Cat {
  // Tutorial: public by default:
  // Can also be: protected, private, ...
  var tailSize: Double = 0.0
  def sound(): Unit = println("miau")
}


// TASK[1/2]: Create a class Dog with an "earSize" (Double), starting with 0:
// class...
class Dog{
  var earSize: Double = 0.0
}


object Scratchpad {

  val cat = new Cat()

  // TASK [2/2]: Create a Dog dog:
  // val
  val dog = new Dog()

  def main(args: Array[String]): Unit = {
    val test = new Dog()
    val test2: Dog = dog
    val test3: Double = dog.earSize
    ensure(dog.earSize == 0.0)
    println("Your solution seems correct")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Classes - Inheritance

```
abstract class Animal {
  // Abstract classes in Scala can have concrete and abstract def/var's/val's:

  def name1: String = "Animal"
  def name2: String

  var name3 = "Animal"
  var name4: String

  val name3 = "Animal"
  val name4: String
}
```

```
abstract class Animal { var age: Int = 0 }

class Cat extends Animal {
  var tailSize: Double = 0.0
}

class Dog extends Animal {
  var earSize: Double = 0.0
}

// TASK [1/3]: Create an abstract class Vehicle, with a var manufacturer (String, empty string):
// abstract class...
class Vehicle{
  var manufacturer: String = ""
}

// TASK [2/3]: Car should inherit from Vehicle:
class Car extends Vehicle{
  var wheelSize = 0.0
}

// TASK [3/3]: Create a class Plane (which extends Vehicle), with a var maxHeight (Double, 0.0):
// class ...
class Plane extends Vehicle{
  var maxHeight: Double = 0.0
}


object Scratchpad {
  def main(args: Array[String]): Unit = {
    val test: Vehicle = new Car()
    val test2: Vehicle = new Plane()
    val test3: Double = new Plane().maxHeight
    ensure(new Plane().maxHeight == 0.0)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Classes - Constructors 0

```
class Cat {
  var tailSize: Double = 0.0
}

// TASK[1/2]: Create a class Dog with an "earSize" (Double), starting with 0:
// class...
class Dog {
   var earSize: Double = 0.0
}



object Scratchpad {

  val cat = new Cat()
  cat.tailSize = 7

  // TASK[2/2]: Create a new dog, and set it's earSize to 5:
  val dog = new Dog()
  dog.earSize = 5

  val test: Dog = dog
  ensure(test.earSize == 5)
  test.earSize += 0

  def main(args: Array[String]): Unit = {
    println("Your solution seems correct")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Classes - Constructors

```
class Cat {
  private var tailSize: Double = 0.0

  def this(_tailSize: Double) = {
    this() // Note: We'll look at this later
    tailSize = _tailSize
    println("Creating a Cat")
  }
}

// TASK[1/2]: Update the class Dog to use the constructor:
// (print "Creating a Dog" in the constructor)
class Dog {
  private var earSize: Double = 0.0

  def this(_earSize: Double) = {
    this()
    earSize = _earSize
  }
}

object Scratchpad {

  val cat = new Cat(7)

  // TASK[2/2]: Update this code:
  val dog = new Dog(5.0)



  val test: Dog = dog
  //ensure(test.earSize == 5)
  //test.earSize += 0

  def main(args: Array[String]): Unit = {
    println("Your solution seems correct")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Classes - Main constructor

```
// === Main constructor ==
// Java version:
public class Cat {

  private final int age;

  public Cat(int age) {
    this.age = age;
    System.out.println("hello");
  }
}

// Roughly equivalent in Scala (using the "main constructor"):
class Cat(val age: Int) { // Default: private, val
  // Notice the age in the main constructor is *both* an argument of a constructor, *and*
  // a variable of the class
  println("hello")
}
```

```
class Cat(tailSize: Double) {
  println("Creating a Cat")
}

// TASK: Update the class Dog to use the main constructor:
class Dog (earSize: Double) {
    println("Creating a Dog")
}



object Scratchpad {

  val cat = new Cat(7)

  val dog = new Dog(5)

  def main(args: Array[String]): Unit = {
    println("Your solution seems correct")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Classes - Constructor access level

```
class Cat(              age: Int) // private,   immutable

class Cat(          val age: Int) // public ,   immutable
class Cat(  private val age: Int) // private,   immutable (DEFAULT)
class Cat(protected val age: Int) // protected, immutable

class Cat(          var age: Int) // public ,   mutable
class Cat(  private var age: Int) // private,   mutable
class Cat(protected var age: Int) // protected, mutable
```

```
class Cat(val tailSize: Double) {} // <- you don't need these empty curly braces

// TASK[1/2]: Make the earSize public and immutable:
class Dog(val earSize: Double) {

  // TASK[2/2]: Fix this:
  def sound(what: Double) = println("Dog says: " + what)
}

object Scratchpad {

  val cat = new Cat(7)
  println("Cat's tailSize: " + cat.tailSize)

  trait T { val earSize: Double }
  val dog = new Dog(5) with T
  println("Dog's earSize: " + dog.earSize)
  dog.sound(123)

  def main(args: Array[String]): Unit = {
    println("Your solution seems correct")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Misc

Multiline Strings

```
// Example with SQL:
val sql = """CREATE TABLE table_name
            |(
            |  column_name1 data_type(size),
            |  column_name2 data_type(size),
            |  column_name3 data_type(size),
            |  ....
            |)""".stripMargin
```

```
object Scratchpad {

  println(
    """Shall I compare thee to a summer's day?
      |Thou art more lovely and more temperate:""".stripMargin
  )

  // TASK: print:
  // Rough winds do shake the darling buds of May,
  // And summer's lease hath all too short a date:
  // println...
  println(
    """Rough winds do shake the darling buds of May,
       |And summer's lease hath all too short a date:""".stripMargin
  )

  def main(args: Array[String]): Unit = {}
}
```