ScalaAcademy

EDV Pragmatic Scala - EMP LVC, 64352231 Course

Himanshu Kesarvani

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 doubleVar: Double = 0.0
  var booleanVar: Boolean = false
  var stringVar: String = "hello world"
  // var ...


  // 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 = 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 = 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 = 0

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

  // 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:
    // ...

    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
myList.head
myList.last

// 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_.java.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")
    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
}


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...

abstract 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:

  var 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 {
  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)


  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

```java
// === 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")
}
```

```scala
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)
}
```

Classes - Additional constructor

```
class Cat(age: Int) {
  // This is an additional constructor:
  def this(ageStr: Int) = {
    this(ageStr.toInt)
  }
}
new Cat(3) // Calling the main constructor
new Cat("3") // Calling the additional constructor
// NOTE: Additional constructors must always end-up calling the main constructor!
// (They can call another constructor which will then call the main constructor though)
```

```
class Cat(val tailSize: Double) {

  def this() = {
    this(1.0)
  }
}

class Dog(val earSize: Double) {
  // TASK: Create a no-arguments constructor which creates a Dog with earSize 0.5:

}

object Scratchpad {

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

  val dog = new Dog()
  println("Dog's earSize: " + dog.earSize)

  ensure(new Dog().earSize == 0.5)

  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 - Private main constructor

```
class Cat private (val age: Int, val tailSize: Double) {
  def this(age: Int) = {
    this(age, age * 12939.72/2939.284-3832)
  }
}

val cat1 = new Cat(6) // OK
val cat2 = new Cat(6, 823.93) // COMPILE ERROR!
```

```
// Now I can only use the no-arguments constructor to construct a Cat:
class Cat private (val tailSize: Double) {

  def this() = {
    this(1.0)
  }
}

// TASK: Make the main constructor for the Dog private:
class Dog(val earSize: Double) {

  def this() = {
    this(0.5)
  }
}

object Scratchpad {

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

  val dog = new Dog()
  println("Dog's earSize: " + dog.earSize)

  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 and inheritance 1

```
abstract class Animal {
  val age: Int
}

class Cat(val age: Int, val tailSize: Double) extends Animal

// TASK: Make the Dog an Animal and add the age:
class Dog(val earSize: Double, val age: Int) extends Animal

object Scratchpad {

  val cat = new Cat(3, 7.3)
  println("Cat's tailSize: " + cat.tailSize)

  val dog = new Dog(age = 4, earSize = 5.2)
  println("Dog's earSize: " + dog.earSize)

  val test: Animal = dog

  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 and inheritance 2

```
abstract class Animal(age: Int) {
  def isOld = age > 10
}

class Cat(val age: Int, val tailSize: Double) extends Animal(age)

// TASK[1/2]: Run the code
// TASK[2/2]: Update the code for the Dog:
class Dog(val age: Int, val earSize: Double) extends Animal(age)

object Scratchpad {

  val cat = new Cat(3, 7.3)
  println("Cat's tailSize: " + cat.tailSize)

  val dog = new Dog(4, 5.2)
  println("Dog's earSize: " + dog.earSize)

  val test: Animal = dog

  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)
}
```

Objects

Objects - Basics

```
new Zoo() // makes no sense - only one in the application
```

```
// Tutorial: objects only have one instance in the application.
// For example the Zoo object:
object Zoo {
  val address = "Zoo Street, 2103, Animal City"
  val phone = "102-123-1314"
}

object Scratchpad {

  // TASK: Get the Zoo's address:
  // val zooAddress = ...


  println("zooAddress: " + zooAddress)

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

Objects - Inheritance

```
abstract class Animal(val age: Int)

class Cat(age: Int, tailSize: Double) extends Animal(age)
class Dog(age: Int, earsSize: Double) extends Animal(age)

// Kitty is a singleton (and a Cat, and an Animal):
object Kitty extends Cat(3, 1.23)

// TASK: Create a dog object, "Ace", which extends Dog, with age 5, and earsSize 3.21:
// ...

object Scratchpad {
  def main(args: Array[String]): Unit = {
    val test1: Animal = Ace
    val test2: Dog = Ace
    ensure(Ace.age == 5)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Case Classes

Case Classes - The apply method 1

```
object Tree {
  def apply(centimeters: Double) {
    println("Growing "+centimeters+" centimeters")
  }
}
// This:
Tree.apply(1.23)
// Is equivalent to:
Tree(1.23)

// Also works on instances (this is not specific of objects!):
cat.apply("miau")
// Is equivalent to:
cat("miau")
```

```
// Tutorial: apply is a special method!
// You can call it as: Tree.apply(1.23),
// but you can also call it simply as: Tree(1.23)
object Tree {
  def apply(centimeters: Double): Unit = {
    println("Growing " + centimeters + " centimeters")
  }
}

object Rock {
  // TASK [1/2]: change the name of the method throwRock to apply:
  def apply(distance: Double): Unit = {
    println("Throwing rock at " + distance + " meters")
  }
}

object Scratchpad {
  println("Growing tree:")
  Tree(1.23)

  println("Throwing rock:")
  // TASK [2/2]: Correct here:
  Rock(123.45)

  def main(args: Array[String]): Unit = {
    Rock(1.23)//Testing
  }
}
```

Case Classes - Companion Objects 1

```
class Tree(val name: String, val size: Double)

// Tutorial: This is called a Companion Object of the class Tree:
//  - It has the same name,
//  - And it is defined in the same file.
object Tree {

}

// TASK: Do a similar construct (class+companion object) for Rock,
// with  name (String) and weight (Double):

// class Rock ...
class Rock(val name: String, val weight: Double)

// object Rock ...

object Rock{

}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    println(new Tree("Tall tree", 1.23).name)
    println(new Rock("Heavy rock", 3.21).name)
    println(new Rock("Heavy rock", 3.21).weight)
    val test = Rock
  }
}
```

Case Classes - Companion Objects 2

```
class Tree(val name: String, val size: Double) {

  private var hiddenI1 = 10
}

object Tree {
  val tree = new Tree("tall tree", 1.23)
  tree.hiddenI1 += 100 // OK!
}

object SomewhereElse {
  val tree = new Tree("tall tree", 1.23)
  tree.hiddenI1 += 100 // ERROR!
}
```

```
class Tree(val name: String, val size: Double) {
  // This is a private attribute:
  private var growthSpeed = 1.2
}

// Tutorial: This is called a Companion Object of the class Tree:
//  - It has the same name,
//  - And it is defined in the same file.
object Tree {
  def speedupGrowth(tree: Tree): Unit = {
    // It can access the private fields:
    tree.growthSpeed = 3.5
  }
}

class Rock(val name: String, val weight: Double) {
  // TASK [1/2]: Add a private String attribute 'rockType', with value 'Basalt':
  // private...
}

object Rock {
  // TASK [2/2]: Add a method 'changeRockType', which receives a rock and changes the 'rockType' to "Quartz":
  // def...
}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    Rock.changeRockType(new Rock("Heavy rock", 1.23))
    println("Changed rock type")
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Case Classes - The apply method 2

```
// Looks like creating a tree without the new:
      Tree("tall tree", 1.23)
  new Tree("tall tree", 1.23)
// In reality we're doing this:
Tree.apply("tall tree", 1.23)
```

```
class Tree(val name: String, val size: Double)

object Tree {
  // We can use the apply to create trees:
  // We can now do:
  //   val tree = Tree("Tall tree", 1.23)
  // instead of:
  //   val tree = new Tree("Tall tree", 1.23)
  def apply(name: String, size: Double): Tree = {
    new Tree(name, size)
  }
}

class Rock(val name: String, val weight: Double)

object Rock {
  // TASK: Create an apply method for the rock:
  def apply(name: String, weight: Double): Rock = {
    new Rock(name, weight)
  }
}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    println("Creating a tree")
    Tree("Tall tree", 1.23)
    println("Creating a rock")
    val rock: Rock = Rock("Heavy rock", 3.21)
  }
}
```

Case Classes - toString

```
// Overriding:
abstract class SuperClass { def name: String = "no name" }
// MUST use the keyword override
class SubClass extends SuperClass { override def name: String = "hello" }

// Implementing:
abstract class SuperClass { def name: String }
// MAY use the keyword override
class SubClass extends SuperClass { override def name: String = "hello" }
// A very good idea! you get an error if you're not really overriding anything!
```

```
class Tree(val name: String, val size: Double) {
  // Like in Java, we can override the toString, like this:
  override def toString = "Tree(" + name + "," + size + ")"
}

object Tree {
  def apply(name: String, size: Double): Tree = new Tree(name, size)
}

class Rock(val name: String, val weight: Double) {
  // TASK: Override the toString like the one in the Tree:
  // override...

  override def toString = "Rock(" + name + "," + weight + ")"

}

object Rock {
  def apply(name: String, weight: Double): Rock = new Rock(name, weight)
}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    println(Tree("Tall tree", 1.23)) // Tree(Tall tree,1.23)
    println(Rock("Heavy rock", 3.21))
    assert(Rock("Heavy rock", 3.21).toString == "Rock(Heavy rock,3.21)")
  }
}
```

Case Classes - equals and hashCode

```
// Override the equals:
new Tree("tall tree", 1.23).equals(new Tree("tall tree", 1.23))
// Use the ==:
new Tree("tall tree", 1.23) == new Tree("tall tree", 1.23)
// There are differences for numbers and when doing "null == ..." (equals crashes, == doesn't)

// Override the hashCode:
new Tree("tall tree", 1.23).hashCode
// Use the ##:
new Tree("tall tree", 1.23).##
// There are differences for numbers

// Get's the Class<Cat> representation of the class Cat in JAVA:
Class<Cat> catClass = Cat.class;
// The same thing in SCALA:
val catClass: Class[Cat] = classOf[Cat]

// Using the "isInstanceOf" and "asInstanceOf" is insecure because we can always get the types wrong, ex:
// [...]
    if(other.isInstanceOf[Dog]) {
      val otherTree = other.asInstanceOf[Tree]
// [...]
// And it only crashes in run-time. The compiler cannot see the bug in compile time!

// Scala reference equality (equivalent to the '==' in Java):
new Tree("tall tree", 1.23) eq new Tree("tall tree", 1.23)
// There is also the 'ne' (not equals)



// isInstanceOf / asInstanceOf:
abstract class Animal
class Cat(age: Int) extends Animal
class Dog(name: String) extends Animal

val cat: Cat = new Cat()
val dog: Dog = new Dog()

val animal: Animal = cat

if(animal.isInstanceOf[Cat]) {
  println("it's a cat, age: " + animal.asInstanceOf[Cat].age)
}
```

```
class Tree(val name: String, val size: Double) {

  // Like in Java we can override the equals and hashCode:

  override def equals(other: Any) = {
    if(other.isInstanceOf[Tree]) {
      val otherTree = other.asInstanceOf[Tree]
      name == otherTree.name && size == otherTree.size
    } else {
      false
    }
  }

  override def hashCode(): Int = name.## + size.## * 41

  override def toString = "Tree(" + name + "," + size + ")"
}

object Tree {
  def apply(name: String, size: Double): Tree = new Tree(name, size)
}

class Rock(val name: String, val weight: Double) {
  // TASK: Override the equals and hashCode for the rock.
  // It should use the rock's name and weight:

  // override def equals...

  override def equals(other: Any) = {
    if(other.isInstanceOf[Rock]) {
      val otherRock = other.asInstanceOf[Rock]
      name == otherRock.name && weight == otherRock.weight
    } else {
      false
    }
  }

  // override def hashCode...

  override def hashCode(): Int = name.## + weight.## * 41

  override def toString = "Rock(" + name + "," + weight + ")"
}

object Rock {
  def apply(name: String, weight: Double): Rock = new Rock(name, weight)
}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    val tree1 = Tree("Tall tree", 1.23)
    val tree2 = Tree("Tall tree", 1.23)
    val rock1 = Rock("Heavy rock", 3.21)
    val rock2 = Rock("Heavy rock", 3.21)
    println("Equals trees are equal? " + (tree1 == tree2))
    println("Equals rocks are equal? " + (rock1 == rock2))
    ensure(tree1 == tree2)
    ensure(rock1 == rock2)
    ensure(rock1 != 123)
    ensure(tree1.hashCode == tree2.hashCode)
    ensure(rock1.hashCode == rock2.hashCode)
    ensure(Rock("Heavy rock", 3.21) != Rock("Heavy rockXYZ", 3.21))
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```

Case Classes - The copy method

```
var tree = new Tree("tall tree", 12.3)

Logging.logTreeState(System.currentTimeMillis, tree)

tree.name = "new name" // ERROR! - it's a val!

tree = tree.copy(name = "new name") // ok
```

```
class Tree(val name: String, val size: Double) {

  // Tutorial: Let's add a method called copy which returns a copy of the Tree
  // setting new values to the parameters, if you pass them:
  def copy(name: String = name, size: Double = size) = new Tree(name, size)

  override def equals(other: Any) = {
    if(other.isInstanceOf[Tree]) {
      val otherTree = other.asInstanceOf[Tree]
      otherTree.name == name && otherTree.size == size
    } else {
      false
    }
  }

  override def hashCode(): Int = name.hashCode + size.hashCode * 41

  override def toString = "Tree(" + name + "," + size + ")"
}

object Tree {
  def apply(name: String, size: Double) = new Tree(name, size)
}

class Rock(val name: String, val weight: Double) {

  // TASK: Implement the copy for the Rock, like the one in the Tree:
  // def copy(...

  def copy(name: String = name, weight: Double = weight): Rock = new Rock(name, weight)

  override def toString = "Rock(" + name + "," + weight + ")"

  override def equals(other: Any) = {
    if(other.isInstanceOf[Rock]) {
      val otherRock = other.asInstanceOf[Rock]
      otherRock.name == name && otherRock.weight == weight
    } else {
      false
    }
  }

  override def hashCode(): Int = name.hashCode + weight.hashCode * 41
}

object Rock {
  def apply(name: String, weight: Double) = new Rock(name, weight)
}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    val rock = Rock("Heavy rock", 1.23)
    val heavierRock: Rock = rock.copy(weight = 3.21, name = "Heavier rock")
    ensure(rock.copy().name == "Heavy rock")
    ensure(rock.copy(weight = 3.21).weight == 3.21)
    println("Rock: "+rock)
    println("Heavier rock: "+heavierRock)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
}
```

Case Classes - Basics

```
// Tutorial: With a case class all this is already done for you:
//   ·"name" and "size" are vals (public) by default now,
//   ·A companion object is created on the background, with the apply as we've seen before:
//     ·So you can do: Tree("my tree", 28.1) to create a new tree
//   ·The equals, hashCode, are already implemented, testing the parameters: if "name" and "size" are the same, the trees are the same.
//   ·The toString is implemented as seen before (ex.: "Tree(my tree,28.1)")
//   ·There is a "copy" method as seen before.
case class Tree(name: String, size: Double)


// TASK: transform the Rock into a case class:
case class Rock(val name: String, val weight: Double)

object Rock {

}

object Scratchpad {
  def main(args: Array[String]): Unit = {
    println("A rock: " + new Rock("Small rock", 3.21))
    new Rock("Small rock", 3.21).copy()
  }
}
```

Collections

List - Create

```
val strings: List[String] = List[String]("a", "b", "c")
// The type on the left can be inferred:
val strings               = List[String]("a", "b", "c")
// The same goes for the [String] on the right:
val strings               = List("a", "b", "c")
// When we do List(1,2) we're actually calling the apply method:
val strings               = List.apply[String]("a", "b", "c")
// It is receiving a variable number of arguments (like we can do in Java!)
// So nothing "special" here.
```

```
object Scratchpad {

  // A list 'strings' of type string, with "a", "b", "c":
  val strings: List[String] = List.apply[String]("a", "b", "c")

  //TASK: Create a list 'numbers' of type Int, with 1, 2, 3:
  // val...
  val numbers = List(1, 2, 3)
  println("numbers = " + numbers)

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(numbers == List(1,2,3))
    println("Your solution seems correct.")
  }
}
```

List - Empty

```
object Scratchpad {

  // This is an empty list of Strings:
  var strings1 = List[String]()
  // This is also an empty list of Strings:
  var strings2: List[String] = Nil

  // TASK: Do the same for numbers1 and numbers2 (type Int):
  // var numbers1...
  // var numbers2...

  println("numbers1 = " + numbers1)
  println("numbers2 = " + numbers2)

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    val test1: List[Int] = numbers1
    val test2: List[Int] = numbers2
    ensure(numbers1 == Nil)
    ensure(numbers2 == Nil)
    numbers1 = List(123)
    numbers2 = List(123)
    println("Your solution seems correct.")
  }
}
```

List - Prepend

```
// In Scala we can have method names with symbols:
  def myMethod(): Int = 123
  def ::(): Int = 123
  def ::  : Int = 123

          l5
          ||
          \/
         "xy"
          ||
          \/
    "c" -> "b" -> "a" -> Nil
    /\     /\     /\     /\
    ||     ||     ||     ||
  theList  l2   myList  lst1
```

```
object Scratchpad {

  var strings = List[String]()
  var strings2 = strings
  // Prepending strings to the list 'strings':
  strings = strings.::("a") // List(a)
  strings = strings.::("b") // List(b, a)
  strings = strings.::("c") // List(c, b, a)
  println("strings: " + strings)
  // Now the list has "c","b","a"

  var numbers = List[Int]()
  //TASK: Prepend 1,2,3 in the same way:
  numbers = numbers.::(1)
  numbers = numbers.::(2)
  numbers = numbers.::(3)

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(numbers == List(3,2,1))
    println("numbers = " + numbers)
    println("Your solution seems correct.")
  }
}
```

List - Creation

```
object Scratchpad {

  // Tutorial: We can create a list by prepending elements to an initially empty one:
  val strings = List[String]().::("a").::("b").::("c")
  println("strings: " + strings) // Prints: List(c, b, a)

  // TASK: prepend 1,2,3 to the list chaining the operations:
  val numbers = List[Int]().::(1).::(2).::(3)

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(numbers == List(3,2,1))
    println("numbers = " + numbers)
    println("Your solution seems correct.")
  }
}
```

List - Creation 2

```
  "a" :: "b" :: "c" :: Nil
  // Does not translate to this:
  "a".::("b").::("c").::(Nil)
  // But to this:
  ("a")::.(("b")::.(("c")::.Nil)) // methods ending with ":" are right-associative!
```

```
object Scratchpad {

  // Tutorial: We can simplify this:
  // Note: the :: method is right associative, which is why the a,b,c are on the left.
  var strings = "a" :: "b" :: "c" :: List[String]()
  println("strings: " + strings)

  // TASK: remove the dots and parenthesis, and reverse the order:
  val numbers = (1) :: (2) :: (3) :: List[Int]()


  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(numbers == List(1,2,3))
    println("numbers = " + numbers)
    println("Your solution seems correct.")
  }
}
```

List - Methods 1

```
scala> "a" :: "b" :: "c" :: Nil ++ "d" :: "e" :: "f" :: Nil
res: List[java.io.Serializable] = List(a, b, c, List(d), e, f)

scala> "a" :: "b" :: "c" :: Nil ::: "d" :: "e" :: "f" :: Nil
res: List[String] = List(a, b, c, d, e, f)
```

```
object Scratchpad {

  val strings = "a" :: "b" :: "c" :: Nil
  val numbers = 1 :: 2 :: 3 :: Nil

  val stringsFirst = strings.head // a (Throws an exception if there are no elements!)
  val stringsLast = strings.last // c (Throws an exception if there are no elements!)
  val stringsPlusDEF  = strings ::: List("d", "e", "f") // List(a,b,c,d,e,f)
  val stringsPlusDEF2 = strings ++  List("d", "e", "f") // List(a,b,c,d,e,f)
  val stringsRest = strings.tail // List(b,c) (Throws an exception if there are no elements!)
  val stringsOnlyFirst2 = strings.take(2) // List(a,b)
  val stringsExceptFirst2 = strings.drop(2) // List(c)
  val stringsIsEmpty = strings.isEmpty // false
  val stringsIsNotEmpty = strings.nonEmpty // true
  val stringsMax = strings.max // c (Throws an exception if there are no elements!)
  val stringsMin = strings.min // a (Throws an exception if there are no elements!)
  val stringsSize = strings.size // 3
  //
  val stringsArray = strings.toArray // Array(a,b,c)
  val stringsBuffer = strings.toBuffer // Buffer(a,b,c)
  val stringsIndexedSeq = strings.toIndexedSeq // IndexedSeq(a,b,c)
  val stringsIterable = strings.toIterable // Iterable[String]

  // TASK: complete these:
  val numbersFirst =
  val numbersLast =
  val numbersPlusFourFiveSix =
  val numbersPlusFourFiveSix2 =
  val numbersRest =
  val numbersOnlyFirst2 =
  val numbersExceptFirst2 =
  val numbersIsEmpty =
  val numbersIsNotEmpty =
  val numbersMaximum =
  val numbersMininum =
  val numbersSize =
  val numbersArray =
  val numbersBuffer =
  val numbersIndexedSeq =
  val numbersIterable =

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

  ensure(numbersFirst == 1)
  ensure(numbersLast == 3)
  ensure(numbersPlusFourFiveSix == List(1,2,3,4,5,6))
  ensure(numbersPlusFourFiveSix2 == List(1,2,3,4,5,6))
  ensure(numbersRest == List(2,3))
  ensure(numbersOnlyFirst2 == List(1,2))
  ensure(numbersExceptFirst2 == List(3))
  ensure(numbersIsEmpty == false)
  ensure(numbersIsNotEmpty == true)
  ensure(numbersMaximum == 3)
  ensure(numbersMininum == 1)
  ensure(numbersSize == 3)
  println("Your solution seems correct.")

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

Functions

Functions 1

```
// Calling the function:
addOneF.apply(123) // 124
addOneF(123) // 124



 def addOneMethod(i: Int) = i + 1
 def powMethod(i: Int) = i * i

 val addOneF = (i: Int) => i + 1
 val powF = (i: Int) => i * i

 var mathOpF: (Int) => Int = powF // OK
 var mathOpMethod: ??????????? = ???addOneMethod???

 mathOpF.apply(input)
```

```
object Scratchpad {

  // Tutorial: a function which receives a string and returns the string with
  // an '!' appended:
  val appendExclF: (String) => String = (s: String) => { s + "!" }


  // TASK: Create an addOneF, which receives an Int and returns the Int plus 1:
  // val addOneF...

  val addOneF: (Int) => Int = (i: Int) => {i + 1}

  println("appendExclF(\"hello world\")=" + appendExclF("hello world"))
  println("addOneF(1)=" + addOneF(1))

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(addOneF(0) == 1)
  }
}
```

Functions 2

```
object Scratchpad {

  // Tutorial: a function which receives 2 strings and returns them concatenated
  // with a '-' in the middle:
  val concatStringsF: (String, String) => String =
    (s1: String, s2: String) => { s1 + "-" + s2 }


  // Create an sumNumbersF, which receives two Ints and returns the sum:
  // val sumNumbersF...

  val sumNumbersF: (Int, Int) => Int = (i1: Int, i2: Int) => {i1 + i2}

  println("concatStringsF(\"hello\", \"world\")=" + concatStringsF("hello", "world"))
  println("sumNumbersF(1, 1)=" + sumNumbersF(1, 1))

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(sumNumbersF(1, 1) == 2)
    println("Your solution seems correct.")
  }
}
```

Functions - Simple syntax

```
  val appendExclF: (String) => String = (s: String) => { s + "!" }
  val appendExclF                     = (s: String) => { s + "!" }
  val appendExclF                     = (s: String) =>   s + "!"

  val appendExclF: (String) => String = (s: String) => { s + "!" }
  val appendExclF: (String) => String = (s: String) =>   s + "!"
  val appendExclF:  String  => String = (s: String) =>   s + "!"
  val appendExclF:  String  => String = (s         ) =>   s + "!"
  val appendExclF:  String  => String =  s           =>   s + "!"
```

```
object Scratchpad {

  // Tutorial: we can simplify the syntax:
  val appendExclF: (String) => String = (s: String) => { s + "!" }
  val appendExclF2: String => String = (s) => s + "!"
  val appendExclF3: String => String = s => s + "!"

  // TASK: Simplify the syntax:
  val addOneF: Int => Int = n => n + 1

  println("appendExclF(\"hello world\")=" + appendExclF("hello world"))
  println("addOneF(1)=" + addOneF(1))

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(addOneF(0) == 1)
  }
}
```

Functions - Placeholder Syntax 1

```
object Scratchpad {

  val appendExclF: String => String = s => s + "!"
  // Tutorial: you can use an underscore to put the input argument,
  // and there is no need for the curly braces:
  val appendExclF2: String => String = _ + "!"


  // TASK: use the placeholder syntax:
  val addOneF: Int => Int = _ + 1

  println("appendExclF(\"hello world\")=" + appendExclF("hello world"))
  println("addOneF(1)=" + addOneF(1))

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(addOneF(0) == 1)
  }
}
```

Functions - Placeholder Syntax 2

```
val concatStringsF3: (String, String) => String = _ + "-" + _
// I can omit the types on the left like this:
val concatStringsF3                             = (_: String) + "-" + (_: String)
```

```
object Scratchpad {

  val concatStringsF:  (String, String) => String = (s1: String, s2: String) => {s1 + "-" + s2}
  val concatStringsF2: (String, String) => String = (s1, s2) => s1 + "-" + s2
  // Tutorial: we can also use 2 underscores, for 2 arguments:
  val concatStringsF3: (String, String) => String = _ + "-" + _


  // TASK: Modify the sumNumbersF to use the underscore:
  val sumNumbersF: (Int, Int) => Int = _ + _

  println("concatStringsF(\"hello\", \"world\")=" + concatStringsF("hello", "world"))
  println("sumNumbersF(1, 1)=" + sumNumbersF(1, 1))

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(sumNumbersF(1, 1) == 2)
    println("Your solution seems correct.")
  }
}
```

Functions - From methods

```
val addOneF1: Int => Int =          n => n + 1
val addOneF2: Int => Int =               _ + 1 // <- Underscore notation

val other_F1: Int => Int =          n => 123
val other_F2: Int => Int =          _ => 123 // <- Ignoring the argument:


val shortestF1: (String, String) => String = shortest
val shortestF2                             = shortest _ // Same as shortestF1
val shortestF3                             = shortest   // ERROR
val shortestString                         = shortest("abcd", "xyz") // xyz
```

```
object Scratchpad {

  def shortest(s1: String, s2: String): String = if(s1.length < s2.length) s1 else s2

  val shortestF1: (String, String) => String = shortest
  val shortestF2 = shortest _

  def max(n1: Int, n2: Int): Int = if(n1 > n2) n1 else n2

  // TASK: Complete maxF1 and maxF2:
  //val maxF1...
  //val maxF2...

  println("maxF1(1,2): " + maxF1(1,2))
  println("maxF2(1,2): " + maxF2(1,2))

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    val _test1: (Int,Int) => Int = maxF1
    val _test2: (Int,Int) => Int = maxF2
    ensure(maxF1(1,2) == 2)
    ensure(maxF2(2,1) == 2)
  }
}
```

JVM Implementation

```
// This function:
val addOneF: (Int) => Int = (i: Int) => { i + 1 }

// Translates into:
val addOneF: Function1[Int, Int] = new Function1[Int, Int] {
  def apply(i: Int): Int = i + 1
}

// Java (just pseudocode, may not compile):
Function1<Int, Int> addOneF = new Function1<Int, Int> {
  public Int apply(Int i) {
    return i + 1;
  }
};

// The function interface (trait) is more or less this:

interface Function1<T1, R> {
  public R apply(T1 arg1);
}

abstract class Function1[-T1, +R] {
  def apply(v1: T1): R
}

// CLOSURES:
// In Scala we can mutate a variable outside the function, in Java we can't:

final int i = 0;
Function1<Integer, Integer> addOneF = new Function1<Integer, Integer> {
  public Integer apply(Integer j) {
    i = i + 1; // ERROR! - Cannot mutate a variable outside
    return i;
  }
};

var i = 0
val addOneF: (Int) => Int = (j: Int) => {
  i += 1 // Works!
  i
}

// Translates into something like:
var i: IntRef.create(0) = IntRef.create(0)
val addOneF: (Int) => Int = (j: Int) => {
  i.elem = i.elem + 1 // Works!
  i.elem
}
```

```
object Scratchpad {

  // This function:
  val addOneF: (Int) => Int = (i: Int) => { i + 1 }

  // Translates into:
  val addOneF: Function1[Int, Int] = new Function1[Int, Int] {
    def apply(i: Int): Int = i + 1
  }

  // Function1<Int, Int> addOneF = new Function1<Int, Int>() {
  //   public Int apply(Int i) {
  //     return i + 1;
  //   }
  // };

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(addOneF(0) == 1)
  }
}
```

List - Methods 2a

```
object Scratchpad {

  val strings = "a" :: "b" :: "c" :: Nil
  val numbers = 1 :: 2 :: 3 :: Nil

  val stringsCountBs =        strings.count((string: String) => string == "b") // 1
  val stringsUntilC =         strings.takeWhile(string => string != "c")       // List(a,b)
  val stringsDropAllFirstAs = strings.dropWhile(_ == "a")                      // List(b,c)
  val stringsExistsB =        strings.exists(_ == "b")                         // true
  val stringsIsAllB =         strings.forall(s => s == "b")                    // false
  val stringsFilterOutBs =    strings.filterNot((s: String) => s == "b")       // List(a,c)
  val stringsOnlyBs =         strings.filter((s: String) => s == "b")          // List(b)

  // TASK: complete these:
  val numbersCountTwos = numbers.count((number: Int) => number == 2)
  val numbersExistsTwo = numbers.exists(_ == 2)

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    ensure(numbersCountTwos == 1)
    ensure(numbersExistsTwo == true)

    println("numbersCountTwos: "+numbersCountTwos)
    println("numbersExistsTwo: "+numbersExistsTwo)
  }
}
```

Methods - Hierarchy

```
object Scratchpad {

  //         Traversable
  //             ||
  //             \/
  //           Iterable
  //         //   ||   \\
  //        \/    \/    \/
  //       Seq   Set    Map
  //      //||    ||    ||
  //     // ||    ||    \/
  //    \/  \/    ||  SortedMap
  //Idx.Seq Linear||
  //             //\\
  //            //  \\
  //           \/    \/
  //      SortedSet  BitSet

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

Tuples

Tuples 1

```
object Scratchpad {

  // Tutorial: This is a tuple with an string and a boolean:
  val tupleA: Tuple3[String, Boolean, Int] = new Tuple3[String, Boolean, Int]("a", false, 123)
  // Prints the string:
  println("tupleA._1: " + tupleA._1)
  // and the boolean:
  println("tupleA._2: " + tupleA._2)
  // and the int:
  println("tupleA._3: " + tupleA._3)

  // TASK: create a tupleB with an Int (value: 123) and a Boolean (value: true):
  val tupleB: Tuple2[Int, Boolean] = new Tuple2[Int, Boolean](123, true)

  println("tupleB._1: " + tupleB._1)
  println("tupleB._2: " + tupleB._2)

  def main(args: Array[String]): Unit = {
    val integer: Int = tupleB._1
    val boolean: Boolean = tupleB._2
    ensure(tupleB._1 == 123)
    ensure(tupleB._2 == true)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
}
```

Tuples 2

```
val tupleA: (String, Boolean) = ("a", false)
// Works:
val (str, bool) = tupleA
// (Btw, also works):
val (str: String, bool: Boolean) = tupleA
```

```
object Scratchpad {

  // Tutorial: we can write it like this:
  val tupleA: (String, Boolean) = ("a", false)

  // TASK: do the same for tupleB:
  val tupleB: (Int, Boolean) = (123, true)

  def main(args: Array[String]): Unit = {
    println("tupleA: " + tupleA)
    println("tupleB: " + tupleB)
  }
}
```

Tuples 3

```
object Scratchpad {

  // Tutorial: to create a *tuple of 2 elements*, we can use the special notation:
  //   <element1> -> <element2>
  // instead of:
  //   (<element1>, <element2>)
  val tupleA: (String, Boolean) = "a" -> false

  // TASK: use the '->' to create a tuple with 123 (Int) and true (Boolean):
  // val tupleB: (Int, Boolean) = ...

  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") = if(!b) throw new Exception(mesg)
  def main(args: Array[String]): Unit = {
    println("tupleA: " + tupleA)
    println("tupleB: " + tupleB)
    ensure(tupleB._1 == 123)
    ensure(tupleB._2 == true)
  }
}
```

Methods - Immutable Collections

```
scala> List("a", "b", "c") == List("a", "b", "c")
res1: Boolean = true

scala> Array("a", "b", "c") == Array("a", "b", "c")
res0: Boolean = false

// Example of mutable collections:
import scala.collection.mutable.Map
import scala.collection.mutable.Set
import scala.collection.mutable.ListBuffer
```

```
object Scratchpad {

  //         Traversable
  //              ||
  //              \/
  //           Iterable
  //         //   ||   \\
  //        \/    \/    \/
  //       Seq   Set    Map
  //      //||    ||    ||
  //     // ||    ||    \/
  //    \/  \/    ||  SortedMap
  //Idx.Seq Linear||
  //             //\\
  //            //  \\
  //           \/    \/
  //      SortedSet  BitSet

  import scala.collection.immutable._
  val seq: Seq[String] = Seq("a", "b", "c")

  val vector: IndexedSeq[String] = Vector("a", "b", "c")

  // Java array:
  val array: Array[String] = Array("a", "b", "c")

  val list: LinearSeq[String] = List("a", "b", "c")
  val stream: LinearSeq[String] = Stream("a", "b", "c")

  val hashSet: Set[String] = HashSet("a", "b", "c")
  val treeSet: SortedSet[String] = TreeSet("a", "b", "c")

  val hashMap: Map[String, String] = HashMap(("a", "A"), ("b", "B"))
  val treeMap: Map[String, String] = TreeMap("a" -> "A", "b" -> "B")
  val listMap: Map[String, String] = ListMap("a" -> "A", "b" -> "B")

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

Misc

Scynchronization

```
  val lock1 = new Object()
  val lock2 = new java.io.File("")

  // Tutorial: we can synchronize some code with any object:
  Scratchpad.synchronized( println("synchronized1") )
  lock1.synchronized({println("synchronized2")})
  lock2.synchronized {
    println("synchronized2")
  }
```

```
object Scratchpad {

  var i1 = 0L

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

    val lock1 = new AnyRef()

    val threads = (1 to 5).toList.map(n => {
      new Thread() {
        override def run() {
          println(s"Thread $n")
          // TASK: Synchronize the i1 increment:
          (1 to 1000000).foreach(_ => i1 = i1 + 1)
        }
      }
    })
    threads.foreach(_.start())
    threads.foreach(_.join())

    println("i1=" + i1)
    ensure(i1 == 5000000)
  }
  def ensure(b: Boolean, mesg: String = "Solution isn't totally correct.") =
    if(!b) throw new Exception(mesg)
}
```