University Mohammed VI Polytechnic

Lab: Introduction to Scala
Professor : Fahd KALLOUBI
Year : 2025/2026
Major : CS3 – College of computing

1. Introduction

1.1 Eléments de syntaxe

1. Exécuter les différents bouts de code ci-dessous et commenter les résultats obtenus

```
val (x: Int, y: String) = Pair(7, "Ayushi")

val (x, y) = Pair(7, "Ayushi")
```

```
for (i <- (1 to 100)) {
  println(i)
}
```

```
var total = 0
for (element <- (1 to 100))
  total += element
```

```
var total = 0
for (element <- (1 to 100))
  if(element % 2 ==0)
    total += element
```

```
var lb = 1
val ub = 100
var totalEven = 0
var totalOdd = 0
while(lb <= ub) {
  if(lb % 2 == 0)
    totalEven += lb
  else
    totalOdd += lb
  lb += 1
}
```

2. Enregistrer le fichier ci-dessous en tant que « HelloWorld.scala », puis ouvrez la fenêtre d'invite de commande et accédez au répertoire dans lequel le fichier est enregistré. La commande ‘scalac’ est utilisée pour compiler le programme Scala et génère quelques fichiers de classe dans le répertoire en cours. L'un d'eux s'appellera HelloWorld.class. Ce bytecode s’exécutera sur la machine virtuelle Java (JVM) à l’aide de la commande ‘scala’.

```
object HelloWorld {
   /* mon premier programm scala
   * qui affichera Hello, World
   */
   def main(args: Array[String]) {
      println("Hello, world!") // Hello, world
```

```
  }
}
```

- ✓ **Sensibilité à la casse** - Scala est sensible à la casse, ce qui signifie que l'identificateur Hello et hello aurait une signification différente dans Scala
- ✓ **Noms de classe** - Pour tous les noms de classe, la première lettre doit être en majuscule. Si plusieurs mots sont utilisés pour former un nom de classe, la première lettre de chaque mot intérieur doit être en majuscule.
- ✓ **Noms de méthodes** - Tous les noms de méthodes doivent commencer par une lettre minuscule. Si plusieurs mots sont utilisés pour former le nom de la méthode, la première lettre de chaque mot intérieur doit être en majuscule.
- ✓ **Nom du fichier programme** - Le nom du fichier programme doit correspondre exactement au nom de l'objet. Lors de la sauvegarde du fichier, vous devez l'enregistrer avec le nom de l'objet
- ✓ **def main (args: Array [String])** - Le traitement du programme Scala commence à partir de la méthode main () qui est une partie obligatoire de chaque programme Scala.
- ✓ Scala est un langage orienté ligne où les instructions peuvent être terminées par des points-virgules (;) ou des retours à la ligne. Un point-virgule à la fin d'une instruction est généralement facultatif.

```
val s = "hello"; println(s)
```

3. Enregistrer les fichiers ci-dessous avec le même nom de l'objet (i.e. Demo.scala) et exécutez les

```
object Demo {
  val greeting: String = "Hello, world!"

  def main(args: Array[String]) {
    println( greeting )
  }
}
```

```
object Demo {
  def main(args: Array[String]) {
    var palindrome = "Dot saw I was Tod";
    var len = palindrome.length();

    println( "String Length is : " + len );
  }
}
```

```
object Demo {
  def main(args: Array[String]) {
    var str1 = "Dot saw I was ";
    var str2 =  "Tod";

    println("Dot " + str1 + str2);
  }
}
```

```
object Demo {
   def main(args: Array[String]) {
      var floatVar = 12.456
      var intVar = 2000
      var stringVar = "Hello, Scala!"
// semblable à la fonction printf en C
      var fs = printf("The value of the float variable is " + "%f, while the value of the
integer " + "variable is %d, and the string" + "is %s", floatVar, intVar, stringVar);

      println(fs)
   }
}
```

```
// Le «s» littéral permet l'utilisation de variable directement dans le traitement d'une
//chaîne, lorsque vous y ajoutez le préfixe «s».
object Demo {
   def main(args: Array[String]) {
      val name = "James"

      println(s"Hello, $name")
      println(s"1 + 1 = ${1 + 1}")
   }
}
```

1.2 Classes et objets

1. Enregistrer le fichier ci-dessous en tant que « Demo.scala », puis exécuter le

```
import java.io._

class Point(val xc: Int, val yc: Int) {
   var x: Int = xc
   var y: Int = yc

   def move(dx: Int, dy: Int) {
      x = x + dx
      y = y + dy
      println ("Point x location : " + x);
      println ("Point y location : " + y);
   }
}

object Demo {
   def main(args: Array[String]) {
      val pt = new Point(10, 20);

      // Move to a new location
      pt.move(10, 10);
   }
}
```

2. Exécuter aussi le code ci-dessous

```
import java.io._

class Point(val xc: Int, val yc: Int) {
  var x: Int = xc
  var y: Int = yc

  def move(dx: Int, dy: Int) {
    x = x + dx
    y = y + dy
    println ("Point x location : " + x);
    println ("Point y location : " + y);
  }
}

class Location(override val xc: Int, override val yc: Int,
  val zc :Int) extends Point(xc, yc){
  var z: Int = zc

  def move(dx: Int, dy: Int, dz: Int) {
    x = x + dx
    y = y + dy
    z = z + dz
    println ("Point x location : " + x);
    println ("Point y location : " + y);
    println ("Point z location : " + z);
  }
}
// Demo est un objet singleton
object Demo {
  def main(args: Array[String]) {
    val loc = new Location(10, 20, 15);

    // Move to a new location
    loc.move(10, 10, 5);
  }
}
```

1.3 Les fonctions

1. Définition et appel d'une fonction : l'exemple suivant illustre la définition d'une fonction et son appel

```
object Demo {
  def main(args: Array[String]) {
    println( "Returned Value : " + addInt(5,7) );
  }
  def addInt( a:Int, b:Int ) : Int = {
    var sum:Int = 0
    sum = a + b
    return sum
  }
```

```
}
```

2. Une fermeture est une fonction dont la valeur de retour dépend de la valeur d'une ou de plusieurs variables déclarées en dehors de cette fonction.

```
object Demo {
  def main(args: Array[String]) {
    println( "multiplier(1) value = " +  multiplier(1) )
    println( "multiplier(2) value = " +  multiplier(2) )
  }
  var factor = 3
  val multiplier = (i:Int) => i * factor
}
```

1.4 Les Tableaux et les collections

1. Exécuter le fichier ci-dessous, et regarder les résultats obtenus

```
object Demo {
  def main(args: Array[String]) {
    var myList = Array(1.9, 2.9, 3.4, 3.5)
    // Afficher les éléments du tableau
    for ( x <- myList ) {
      println( x )
    }
    // la somme des éléments
    var total = 0.0;
    for ( i <- 0 to (myList.length - 1)) {
      total += myList(i);
    }
    println("Total is " + total);
    // trouver le maximum
    var max = myList(0);
    for ( i <- 1 to (myList.length - 1) ) {
      if (myList(i) > max) max = myList(i);
    }
    println("Max is " + max);
  }
}
}
```

1.5 Les Traits

1. Exécuter le fichier ci-dessous : Ici, il est nécessaire de connaître deux méthodes importantes de Scala, qui sont utilisées dans l'exemple suivant.

- obj.isInstanceOf [Point] Pour vérifier le type d'objet et Point sont identiques ne sont pas.
- obj.asInstanceOf [Point] signifie le transtypage exact en prenant le type d'objet et renvoie le même obj que le type de point.

```
trait Equal {
  def isEqual(x: Any): Boolean
  def isNotEqual(x: Any): Boolean = !isEqual(x)
}
```

```
class Point(xc: Int, yc: Int) extends Equal {
  var x: Int = xc
  var y: Int = yc

  def isEqual(obj: Any) = obj.isInstanceOf[Point] && obj.asInstanceOf[Point].x == y
}

object Demo {
  def main(args: Array[String]) : Unit = {
    val p1 = new Point(2, 3)
    val p2 = new Point(2, 4)
    val p3 = new Point(3, 3)

    println(p1.isNotEqual(p2))
    println(p1.isNotEqual(p3))
    println(p1.isNotEqual(2))
  }
}
```

1.6 Pattern Matching

1. Le Pattern Matching est la deuxième caractéristique la plus largement utilisée de Scala, après les valeurs des fonctions et les fermetures. Scala fournit un excellent support pour la correspondance de modèle, dans le traitement des messages. Un pattern matching comprend une séquence d'alternatives, chacune commençant par de mot-clé **case**. Chaque alternative comprend un **pattern** et une ou plusieurs expressions, qui seront évaluées si le pattern correspond. Un symbole de flèche **=>** sépare le pattern des expressions.

```
object Demo {
  def main(args: Array[String]) {
    println(matchTest(3))
  }

  def matchTest(x: Int): String = x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
  }
}
```

2. La pattern matching en utilisant les classes de cas : Les classes de cas sont des classes spéciales utilisées dans le pattern matching avec des expressions de cas. Syntaxiquement, ce sont des classes standard avec un modificateur spécial : case.

```
object Demo {
  def main(args: Array[String]) {
    val alice = new Person("Alice", 25)
    val bob = new Person("Bob", 32)
    val charlie = new Person("Charlie", 32)

    for (person <- List(alice, bob, charlie)) {
      person match {
```

```
        case Person("Alice", 25) => println("Hi Alice!")
        case Person("Bob", 32) => println("Hi Bob!")
        case Person(name, age) => println(
          "Age: " + age + " year, name: " + name + "?")
      }
    }
  }
  case class Person(name: String, age: Int)
}
```

1.7 E/S en scala

Scala peut utiliser n'importe quel objet Java et java.io.File est l'un des objets qui peuvent être utilisés dans la programmation Scala pour lire et écrire des fichiers.

1. Ecriture dans un fichier

```
import java.io._

object Demo {
  def main(args: Array[String]) {
    val writer = new PrintWriter(new File("test.txt" ))

    writer.write("Hello Scala")
    writer.close()
  }
}
```

2. Lecture d'une ligne à partir de la console

```
object Demo {
  def main(args: Array[String]) {
    print("Please enter your input : " )
    val line = Console.readLine

    println("Thanks, you just typed: " + line)
  }
}
```

3. Lecture du contenu d'un fichier

```
import scala.io.Source

object Demo {
  def main(args: Array[String]) {
    println("Following is the content read:" )

    Source.fromFile("Demo.txt" ).foreach {
      print
    }
  }
}
```

Exercice 1 :

Ci-dessous la définition d'une fonction qui calcule la somme des nombres dans un intervalle donné :

```
def sum(lb: Int, ub: Int)={
        var total =0
        for (element <- lb to ub) {
          total += element
        }
        total
        // return n'est pas obligatoire
}
```

a. En se basant sur cette définition, essayez de définir les fonctions suivantes :
   1. La somme des nombres au carré dans un intervalle donné
   2. La somme des nombres au cube dans un intervalle donné
   3. La somme des nombres multipliés par 2 dans un intervalle donné

Scala est un langage purement fonctionnel alors il permet l'appel d'une fonction au paramètres d'une autre. Pour ce faire, nous allons réécrire l'exemple en haut d'une façon fonctionnelle.

```
// en paramètre : une fonction qui prend un Int et retourne un Int plus l'intervalle [lb,ub]
def sum2(func: Int => Int,  lb: Int, ub: Int)={
        var total =0
        for (element <- lb to ub) {
          total += func(element)

        }
        total
        // return n'est pas obligatoire
}
```

```
def id(i: Int)= i // une fonction qui prend un int et retourne un int
```

Appel de la fonction qui calcule la somme des nombres contenus dans l'intervalle [0,10]

```
sum2(id, 1, 10) // équivalent à sum(1,10)
```

b. Essayez de refaire la question a. en utilisant que des fonctions et en les appelant dans la fonction sum2.

Exercice 2 :

Nous allons créer une classe « Order » et une fonction toString() qui est une redéfinition de la méthode toString() de la classe Object :

```
class Order(orderId: Int, orderDate: String, orderCustomerId: Int, orderStatus: String) {
        println("Je suis dans le constructeur")
        override def toString = "Order("+ orderId+ ","+ orderDate + ","+
        orderCustomerId + ","+ orderStatus +")"
}
```

Essayez d'exécuter la commande « :javap -p Order » pour voir la définition de la classe en Java. Qu'est-ce que constatez ?

Création d'un objet Order :

```
var order = new Order(1, "2013-10-01 00:00:00.00",100, "COMPLETE")
```

Essayez d'afficher un attribut de l'objet « order »

```
Order.orderId
```

Vous allez remarquer que orderId n'est pas un membre de la classe Order.

Nous allons maintenant passer des variables de classe au lieu des arguments.

```
class Order(val orderId: Int, val  orderDate: String, val orderCustomerId: Int, val
orderStatus: String) {
        println("Je suis dans le constructeur")
        override def toString = "Order("+ orderId+ ","+ orderDate + ","+
        orderCustomerId + ","+ orderStatus +")"
}
```

Essayez d'exécuter la commande « :javap -p Order » pour voir la définition de la classe en Java. Qu'est-ce que constatez ?

Création d'un objet de type Order.

```
var order = new Order(1, "2013-10-01 00:00:00.00",100, "COMPLETE")
```

Essayez d'afficher un attribut de l'objet « order », vous allez remarquer que cela est possible parce que les accesseurs ont été ajoutés en déclarant les attributs de classe comme variables immutables (i.e. val)

```
Order.orderId // équivalente à Order.orderId(), mais en scala une fonction sans
//paramètres est appelé sans  parenthèses
```

Maintenant, Essayez de modifier la valeur d'un attribut de l'objet « order », Qu'est-ce que constatez ?

```
Order.orderId = 2
```

1. Comment peut-on ajouter les modificateurs ?

Exercice 3 :

Nous voulons avoir la somme de tous les éléments pairs au carré dans un intervalle donné (dans ce cas c'est [1, 100]), en utilisant le pattern MapReduce.

Premièrement nous allons créer une liste de 1 jusqu'à 100.

```
val l = (1 to 100).toList
```

Ensuite, nous allons filtrer les nombres pairs, en utilisant la méthode filter de la classe List.

```
val f = l.filter(ele => ele % 2 == 0)
```

Pour chaque élément, nous allons calculer sa racine en utilisant la méthode map de la classe List.

```
val m = f.map(rec => rec * rec)
```

Finalement, pour la somme de tous les éléments de la liste obtenue, nous faisons appel à la méthode reduce de la classe List.

```
val r=m.reduce((total, element) => total+element)
```

Le même résultat obtenu en utilisant la méthode reduce peut être obtenu en utilisant les deux méthodes suivantes :

```
var total =0
for( e <- m) total + = e
```

```
Val r= m.sum
```

Maintenant, nous allons appliquer ce processus sur des données réelles. Pour cela, nous allons lire un fichier « part-00000 » et en extraire des informations pour appliquer une opération MapReduce.

```
val orderItems = Source.fromFile("part-00000").getLines.toList
```

Comme vous pouvez le remarquer chaque ligne contient un enregistrement d'une commande qui représente les colonnes suivantes : OrderId, ProductId, ….

```
scala> orderItems(0)
res8: String = 1,1,957,1,299.98,299.98
```

Nous allons extraire que les commandes qui concerne la produit numéro 2.

```
val orderItemsFilter = orderItems.filter(orderItem => orderItem.split(",")(1).toInt == 2)
```

Ensuite, nous allons extraire, de chaque enregistrement, le 5ème élément qui représente le prix en faisant appel à la fonction map.

```
val orderItemsMap = orderItemsFilter.map(orderItem => orderItem.split(",")(4).toFloat)
```

- Implémenter la somme des prix en utilisant la méthode reduce.

Exercice 4 :

Configuration :

- sbt version 1.3.3
- scala version 2.11.0

Dans l'éditeur Itellij IDEA, créer un nouveau projet nommé « OrderRevenuePrice ». Ensuite, dans le dossier « src/main/scala/ », créer un objet Scala nommé « OrderRevenue » dans lequel

vous mettez le code que vous avez créé en haut portant sur le fichier « part-00000 » (ajouter le fichier dans la racine du projet).

- Exécuter le projet « Run –> Run OrderRevenue »
- Pour exécuter le projet en utilisant « sbt » : taper cette commande dans la racine du projet

```
sbt “runMain OrderRevenue”
```

- Pour générer le fichier jar et l’exécuter (toujours dans la racine) :

```
sbt package
```

Puis,

```
Scala target/scala-2.11/orderrevenue_2.11-0.1.jar OrderRevenue
```

Changer le code de l’objet pour qu’on puisse passer l’id de la commande en argument :

```
import scala.io.Source
object OrderRevenue {
  def main(args: Array[String]) = {
    val orderId = args(1).toInt
    val orderItems = Source.fromFile("part-00000").getLines
    val orderRevenue = orderItems.filter(oi => oi.split(",")(1).toInt ==
orderId).
      map(oi => oi.split(",")(4).toFloat).
      reduce((t, v) => t + v)
    println(orderRevenue)
  }
}
```

1. Générer le fichier jar et exécuter l’objet avec un argument de votre choix.