# Différence Scala-JAVA

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| **Val s : String = "Hello"** | **const String s = "Hello" ;** |
| **var s :String = "Hello"** | **String s = "Hello";** |
| **s(4)  s.apply(4) //o** | **s[4] ;//o** |
| **+=1** | **++** |
| **-=1** | **--** |
| **"Bonjour".sorted**  **[pas de parenthèses si la méthode ne nécessite pas d’argument]** | **"Bonjour".sorted() ;** |
| **import scala.math.\_** | **import scala.math.\* ;** |
| **scala.math.sqrt(2)** | **pas d’équivalent** |
| **val s = if (x > 0) 1 else -1** | **If(x>0){s=1 ;}**  **else{s=-1 ;}** |
| **If(x>0) 1 else**  **//si la condition n’est pas vérifier alors l’expression vaut Unit** |  |
| **if (x > 0) 1 else -1** | **x > 0 ? 1 : -1 ;** |
| **val distance = { val dx = x - x0; val dy = y - y0; sqrt(dx \* dx + dy \* dy) }**  **distance has value and the type of red expression** |  |
| **{ r = r \* n; n -= 1 }**  **Expression has value Unit()** |  |
| **for (i <- 1 to n)**  **r = r \* i** | **for(int i=1 ; i<=n ;i++){**  **r = r \* i ;**  **}** |
| **val s = "Hello"**  **var sum = 0**  **for (i <- 0 to s.length - 1)**  **sum += s(i)**  ****  **var sum = 0**  **for (ch <- "Hello") sum += ch** |  |
| **Pas break** |  |
| **Multiples generators :**  **for (i <- 1 to 3; j <- 1 to 3) print(f"${10 \* i + j}%3d") // Prints 11 12 13 21 22 23 31 32 33** |  |
| **for (i <- 1 to 3; j <- 1 to 3 if i != j) print(f"${10 \* i + j}%3d") // Prints 12 13 21 23 31 32** |  |
| **for (i <- 1 to 10) yield i % 3**  **// Yields Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)** |  |
| **Lazy values** |  |
| **You can think of lazy values as halfway between val and def. Compare :**  **val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString**  **// Evaluated as soon as words is defined**  **lazy val words = scala.io.Source.fromFile("/usr/share/dict/words").mkString**  **// Evaluated the first time words is used**  **def words = scala.io.Source.fromFile("/usr/share/dict/words").mkString**  **// Evaluated every time words is used** |  |
| **for(i<-10 to (0,-1))** | **for(int i=10 ; i>-1 ;i--)** |
| **for(i <- 0 until 10)** | **for(int i=0 ; i<10 ; i++)** |
| **for(0 until 10 by 2) //0 2 4 6 8** | **for(int i=0 ;i<5 ; i++) 2\*i** |
| **for(0 until 10 by -1) //9 8 7 6 5 4 3 2 1 0** | **for(int i=9 ; i>-1 ;i--)** |
| **Tableau de taille variable :** | |
| **import scala.collection.mutable.ArrayBuffer**  **val tab = ArrayBuffer[Type]()** | **import java.util.ArrayList ;**  **ArrayList<Type> tab = new ArrayList<Type>()** |
| **Ajouter des éléments :**  **tab += (0,1,2,3,4,5)**  **tab ++= ArrayBuffer(6,7,8,9)** | **Ajouter des éléments :**  **tab.addAll(new ArrayList<Int>(0,1,2,3))** |
| **for(i <- tab.indices)**  **for(i <- tab.indices.reverse** |  |
| **Transformez en tableau de taille fixe : tab.Array** |  |
| **Array comprehension :**  **val a = Array(2, 3, 5, 7, 11)**  **val result = for (elem <- a) yield 2 \* elem**  **// result is Array(4, 6, 10, 14, 22)**  ****  **val result = a.map{2\*\_}** |  |
| **Array comprehension :**  **val a = Array(2, 3, 5, 7,11)**  **for (elem <- a if elem % 2 == 0) yield 2 \* elem**  ****  **a.filter(\_ % 2 == 0).map(2 \* \_)**  **or even**  **a filter { \_ % 2 == 0 } map { 2 \* \_ }** |  |
| **Array comprehension :**  **val positionsToRemove = for (i <- a.indices if a(i) < 0) yield i**  **for (i <- positionsToRemove.reverse) a.remove(i)** |  |
| **Array attribut : max, min, sum, sorted, sortWith(func)** |  |
| **a.mkString(" and ") // « 2 and 3 and 5 and 7 and 11»** | **Python :** " and ".join(a) |
| **a.mkString("<", ",", ">") // "<1,2,7,9>"** |  |
| **Tableau multi-dimensionnel** | |
| **val matrix = Array.ofDim[Double](3, 4) // Three rows, four columns**  **To access an element, use two pairs of parentheses:**  **matrix(row)(column) = 42**  **You can make ragged arrays, with varying row lengths:**  **val triangle = new Array[Array[Int]](10)**  **for (i <- triangle.indices)**  **triangle(i) = new Array[Int](i + 1)** |  |
| **Map** | |
| **val scores = scala.collection.mutable.Map[String, Int]()** |  |
| **val scores = Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)**  ****  **val scores = scala.collection.mutable.Map("Alice" -> 10, "Bob" -> 3, "Cindy" -> 8)**  ****  **val scores = Map(("Alice", 10), ("Bob", 3), ("Cindy", 8))** |  |
| **val bobsScore = scores("Bob")**  **If the map doesn’t contain a value for the requested key, an exception is thrown** | **scores.get("Bob")** |
| **val bobsScore = if (scores.contains("Bob")) scores("Bob") else 0**  ****  **val bobsScore = scores.getOrElse("Bob", 0)**  **If the map contains the key "Bob", return the value; otherwise, return 0.** |  |
| 1. **scores("Bob") = 10**   **// Updates the existing value for the key "Bob" (assuming scores is mutable)**  **et**   1. **scores("Fred") = 7**   **// Adds a new key/value pair to scores (assuming it is mutable)**  **(1., 2.)scores += ("Bob" -> 10, "Fred" -> 7)** | **scores.put("Bob",10)** |
| **scores -= "Alice"**  **Remove the key Alice** |  |
| **val newScores = scores + ("Bob" -> 10, "Fred" -> 7)**  **// New map with update** |  |
| **var scores=…**  **scores = scores + ("Bob" -> 10, "Fred" -> 7)**  ****  **scores += ("Bob" -> 10, "Fred" -> 7)** |  |
| **scores = scores – Alice**  ****  **scores -= Alice** |  |
| **for ((k, v) <- *map*)** |  |
| **scores.keySet**  **// A set such as Set("Bob", "Cindy", "Fred", "Alice")**  **for (v <- scores.values) println(v) // Prints 10 8 7 10** |  |
| **To reverse a map—that is, switch keys and values—use**  **for ((k, v) <- *map*) yield (v, k)** |  |
| **visit the keys in sorted order**  **val scores = scala.collection.mutable.SortedMap("Alice" -> 10,**  **"Fred" -> 7, "Bob" -> 3, "Cindy" -> 8)** |  |
| **If you want to visit the keys in insertion order, use a LinkedHashMap. For example,**  **val months = scala.collection.mutable.LinkedHashMap("January" -> 1,**  **"February" -> 2, "March" -> 3, "April" -> 4, "May" -> 5, ...)** |  |
| **import scala.collection.JavaConversions.mapAsScalaMap**  **val scores: scala.collection.mutable.Map[String, Int] =**  **new java.util.TreeMap[String, Int]** |  |
| **get a conversion from java.util.Properties to a Map[String,**  **String]:**  **import scala.collection.JavaConversions.propertiesAsScalaMap**  **val props: scala.collection.Map[String, String] = System.getProperties()** |  |
| **Scala map to a method that expects a Java map, provide**  **the opposite implicit conversion :**  **import scala.collection.JavaConversions.mapAsJavaMap**  **import java.awt.font.TextAttribute.\_ // Import keys for map below**  **val attrs = Map(FAMILY -> "Serif", SIZE -> 12) // A Scala map**  **val font = new java.awt.Font(attrs) // Expects a Java map** |  |
| **val t = (1, 3.14, "Fred")**  **access its components with the methods \_1, \_2, \_3**  **val second = t.\_2 // Sets second to 3.14**  **Unlike array or string positions, the component positions of a tuple start with 1, not 0.** |  |
| **val (first, second, third) = t // Sets first to 1, second to 3.14, third to "Fred"**  **You can use a \_ if you don’t need all components:**  **val (first, second, \_) = t** |  |
| **Classes** | |
| **Class Person{**  **var age=0**  **//Le setter et getter sont automatiquement crées si déclarer var**  **Sinon si déclarer val, seul le getter est uniquement créer**  **}**  **val p = new Person //<=> new Person()**  **p.age //<=> p.getAge() en JAVA**  **p.age\_ //<=> p.setAge() en JAVA** |  |
| **class Person(val name: String, val age: Int) {**  **// Parameters of primary constructor in (...)**  **...**  **}** | **public class Person { // This is Java**  **private String name; private int age; public Person(String name, int age) {**  **this.name = name; this.age = age;**  **}**  **public String name() { return this.name; } public int age() { return this.age; }**  **...**  **}** |

**Avertissement :**

Occasionally, the () notation conflicts with another Scala feature: implicit parameters. For example, the expression "Bonjour".sorted(3) yields an error because the sorted method can optionally be called with an ordering, but 3 is not a valid ordering.You can use parentheses:

("Bonjour".sorted)(3) or call apply explicitly: "Bonjour".sorted.apply(3)

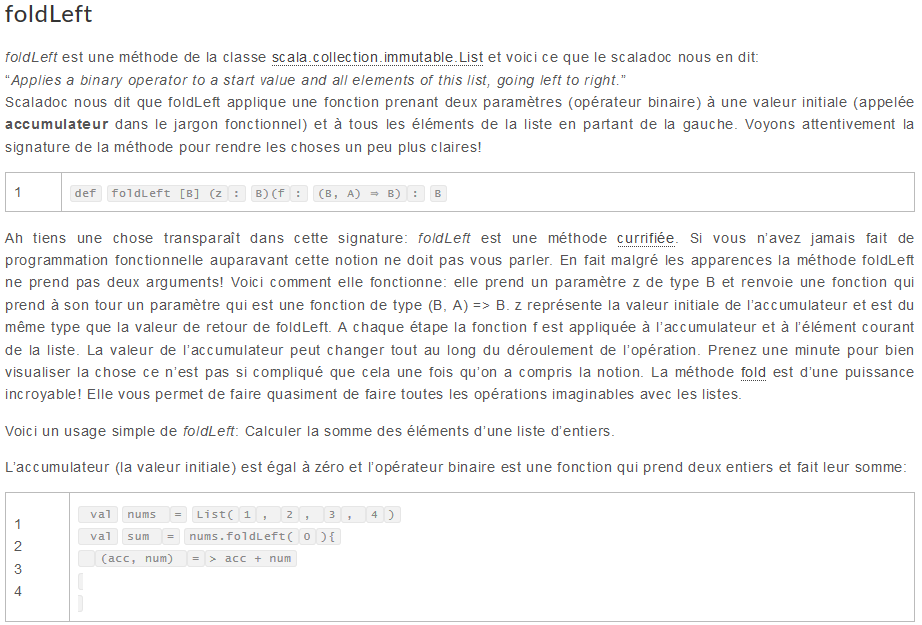
**AIDE SCALA** :

Scala possède un interpréteur.

Pour obtenir de l’aide on peut faire ex: Taper 3. et Press Tab Key

Vous obtiendez une liste de métles opérations disponible pour l’objet 3

## FoldLeft



## Class

In Scala (as well as in Java or C++), a method can access the private fields of all

objects of its class. For example,

**class Counter {**

**private var value = 0**

**def increment() { value += 1 }**

**def isLess(other : Counter) = value < other.value**

**// Can access private field of other object**

**}**

Accessing other.value is legal because other is also a Counter object.

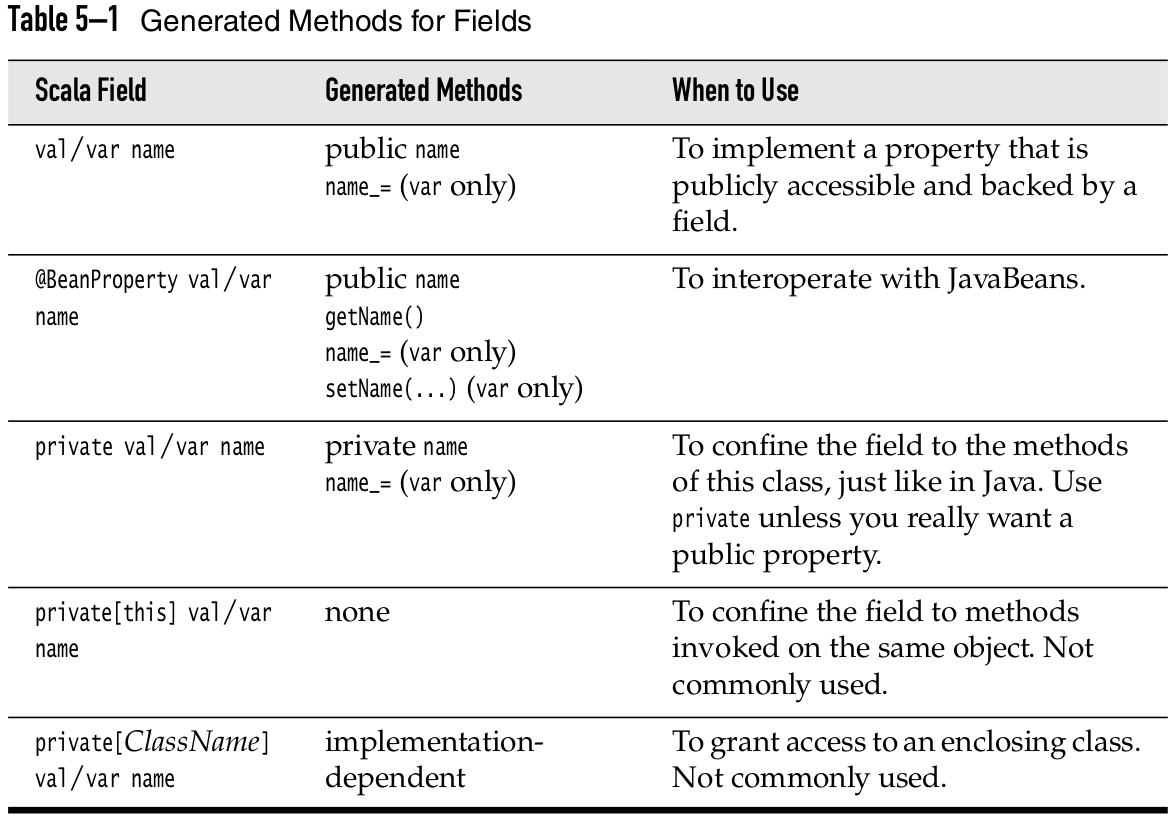
Scala allows an even more severe access restriction with the private[this] qualifier:

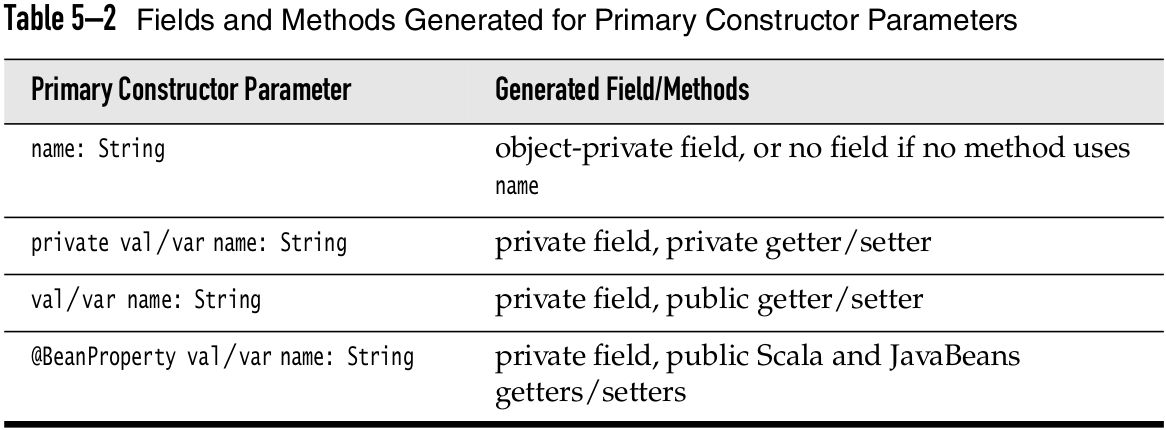
**private[this] var value = 0 // Accessing someObject.value is not allowed**

Now, the methods of the Counter class can only access the value field of the current

object, not of other objects of type Counter . This access is sometimes called

object-private, and it is common in some OO languages such as SmallTalk.

****

****

### Nested Class

In Scala, you can nest just about anything inside anything. You can define func-

tions inside other functions, and classes inside other classes. Here is a simple

example of the latter:

**import scala.collection.mutable.ArrayBuffer**

**class Network {**

**class Member(val name: String) {**

**val contacts = new ArrayBuffer[Member]**

**}**

**private val members = new ArrayBuffer[Member]**

**def join(name: String) = {**

**val m = new Member(name)**

**members += m**

**m**

**}**

**}**

Consider two networks:

**val chatter = new Network**

**val myFace = new Network**

In Scala, each instance has its own class Member , just like each instance has its own

field members . That is, chatter.Member and myFace.Member are different classes.