# Scala

## Les Bases

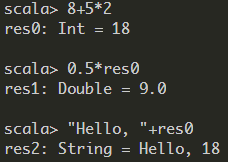
Lancer l’interpréteur en tapant dans un terminal : scala

Taper ensuite : 8\*5+2



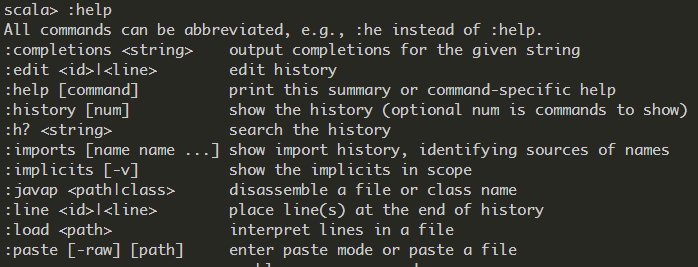
Comme on peut le voir par défaut le résultat est stocké dans une variable appelée res0.

Autres commandes :



Comme on pouvait s’y attendre on retrouve un comportement similaire à JAVA.

On peut obtenir de l’aide de l’interpréteur en tapant :help



Une autre commande utile est :warnings ou :w cette commande permet d’obtenir les avertissements du compilateur

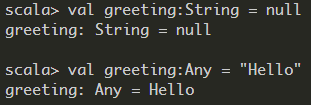


### Déclarer une variable

Contrairement à JAVA, mais tous comme Python en Scala le type d’une variable peut être déduit en fonction de son initialisation, on peut néanmoins toujours si on le souhaite initialiser une variable comme en JAVA :

On notera la façon particulière de déclarer le type d’une variable et l’absence de ‘;’.

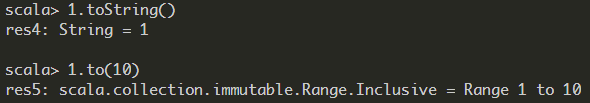


#### Types de données

Il existe différents types de données en Scala :

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Byte** | 8 bit signed value. Range from -128 to 127 |
| **Short** | 16 bit signed value. Range -32768 to 32767 |
| **Int** | 32 bit signed value. Range -2147483648 to 2147483647 |
| **Long** | 64 bit signed value. -9223372036854775808 to 9223372036854775807 |
| **Float** | 32 bit IEEE 754 single-precision float |
| **Double** | 64 bit IEEE 754 double-precision float |
| **Char** | 16 bit unsigned Unicode character. Range from U+0000 to U+FFFF.  Use **ˈ ˈ** to declare it. |
| **String** | A sequence of Chars. Use " " to declare it |
| **Boolean** | Either the literal true or the literal false |
| **Unit** | Corresponds to no value |
| **Null** | null or empty reference |
| **Nothing** | The subtype of every other type; includes no values |
| **Any** | The supertype of any type; any object is of type Any (like Object in JAVA) |
| **AnyRef** | The supertype of any reference type |

Contrairement à JAVA les ***types*** sont aussi des ***classes***, il n’y a pas de différence entre ***types primitives*** et ***classes***, de ce fait on peut directement invoquer les fonctions liées aux classes associées à ces types :



#### Déclarer plusieurs variables en même temps :

val xmax, ymax = 100 // Sets xmax and ymax to 100

var greeting, message: String = null // greeting and message are both strings, initialized with null

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| val s:String = "Hello" | const String s = "Hello" ; |
| var s:String = "Hello" | String s = "Hello"; |

### Les opérateurs

a+b est un raccourci de a.+(b) ici + est le nom de la méthode. En Scala, contrairement à Java, on peut définir des méthodes avec des symboles.

En général, on peut écrire :

a method b raccourci de a.method(b) où method est une fonction qui prend 2 paramètres un implicite l’autre explicite. Autre exemple, on peut écrire : 1 to 10 au lieu de 1.to(10)

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| +=1 | ++ |
| -=1 | -- |
| **BigInt et BigDecimal** | |
| val x:BigInt = 1234567890  x \* x \* x  // Yields 1881676371789154860897069000 | BigInteger x = new BigInteger("1234567890");  x.multiply(x).multiply(x); |

### Appeler une méthode

En général, on peut écrire :

a method b raccourci de a.method(b) où method est une fonction qui prend 2 paramètres un implicite l’autre explicite. Autre exemple, on peut écrire : 1 to 10 au lieu de 1.to(10)

On a déjà vu comment appeler une méthode sur un objet, exemple : "Hello".intersect("World")

Noter que contrairement à Java, **en** **Scala si la méthode n’a pas de paramètres on peut omettre les parenthèses**, exemple : "Bonjour".sorted // Yields the string "Bjnooru"

#### Import de package

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| import scala.math.\_  Si le package est préfixé par scala, on peut écrire : import math.\_ | import scala.math.\* ; |

Même si l’on importe pas un package, on peut utiliser ses méthodes en écrivant pckge.method(), exemple : scala.math.sqrt(2) //Yields 1.4142135623730951 et min(3, Pi) // Yields 3.0

#### La méthode apply

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| val s = "Hello"  s(4) // équivaut à **s.apply(4)**  // Yields ‘o’ | String s = "Hello" ;  s.charAt(4) // Yields ‘o’ ; |

Comme on peut le voir s(4) est un raccourci de s.apply(4).

**Pourquoi est-ce que l’on n’utilise pas les [] ?**

On peut voir une séquence **s** d’élément de type ***T*** comme une fonction mathématique qui va de {0,1,…,n-1} à ***T*** qui fait correspondre(**map**) i à ***s(i)***, i ième élément de la séquence.

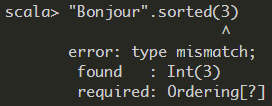
##### **Créer un objet**

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| val x = BigInt("1234567890")  // x: scala.math.BigInt = 1234567890 | BigInteger x = new BigInteger("1234567890"); |
| val y = Array(1,2,3,4)  // y: Array[Int] = Array(1, 2, 3, 4) | int[] arr = new int[4];  for (int i = 0; i < arr.length; i++){  arr[i]=i+1;  }  **ou**  int arr[] = {1,2,3,4}; |

En effet val x = BigInt("1234567890") est un raccourci BigInt.apply("1234567890"), on n’a pas besoin de ***new*** pour créer l’objet grâce à **apply**.

##### **Avertissement**

Occasionnellement, il arrive que la ***notation*** () soit en conflit avec une autre fonctionnalité de Scala : ***les paramètres implicites***. Exemple :

 Ce code produit une erreur car la méthode **sorted** peut être appelée de façon optionnelle avec un ordre de tri, or 3 n’est pas un ordre valide de tri.

**Solution : ("Bonjour".sorted)(3)** ou **"Bonjour".sorted.apply(3)**

### Keep In Mind

Keep these tips in mind:

• Remember to look into **RichInt**, **RichDouble**, and so on, if you want to know how

to work with **numeric types**. Similarly, to work with **strings**, look into **StringOps**.

• The mathematical functions are in the *package* **scala.math**, not in any class.

• Sometimes, you’ll see functions with funny names. For example, **BigInt** has a

method **unary\_-**. This is how you define the prefix

negation operator -x.

• Methods can have functions as parameters. For example, the **count** method in

**StringOps** requires a function that returns true or false for a **Char**, specifying

which characters should be counted:

def count(**p**: **(Char) => Boolean**) : **Int**

You supply a function, often in a very compact notation, when you call the

method. As an example, the call s.count(\_.isUpper) counts the number of uppercase

characters.

• You’ll occasionally run into classes such as **Range** or **Seq[Char]**. They mean what

your intuition tells you—a range of numbers, a sequence of characters. You

will learn all about these classes as you delve more deeply into Scala.

• In Scala, you use square brackets for type parameters. A **Seq[Char]** is a sequence

of elements of type **Char**, and **Seq[A]** is a sequence of elements of some type **A**.

• There are many slightly different types for sequences such as **GenSeq**, **GenIterable**,

**GenTraversableOnce**, and so on. The differences between them are rarely important.

When you see such a construct, just think “sequence.” For example, the

**StringOps** class defines a method

def containsSlice[B](that: GenSeq[B]): Boolean

This method tests whether the string contains with a given sequence. If you

like, you can pass a **Range**:

"Bierstube".containsSlice(**'r'.to('u')**)

// Yields true since the string contains Range('r', 's', 't', 'u')

• Don’t get discouraged that there are so many methods. It’s the Scala way to

provide lots of methods for every conceivable use case. When you need

to solve a particular problem, just look for a method that is useful. More often

than not, there is one that addresses your task, which means you don’t have

to write so much code yourself.

• Some methods have an **“implicit” parameter**. For example, the **sorted** method

of **StringOps** is declared as

def sorted[B >: Char](implicit ord: math.Ordering[B]): String

That means that an ordering is supplied “implicitly,” using a mechanism that

• Finally, don’t worry if you run into the occasional indecipherable incantation,

such as the [B >: Char] in the declaration of sorted. The expression B >: Char

means **“any supertype of Char”**.

## Différence Scala-JAVA

|  |  |
| --- | --- |
| **SCALA** | **JAVA** |
| s(4) <-> s.apply(4) //o | s[4] ;//o |
| "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--) |
| import scala.util.control.\_  val loop = new Breaks  loop.breakable{  for(i←1 to 10 by 2){  println("Value of i: "+i)  if(i==5) loop.break  }  } | for(int i=1 ; i<10 ; i+2){  System.out.println("Value of i: "+i) ;  if(i==5) break ;  } |
|  |  |
| 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)

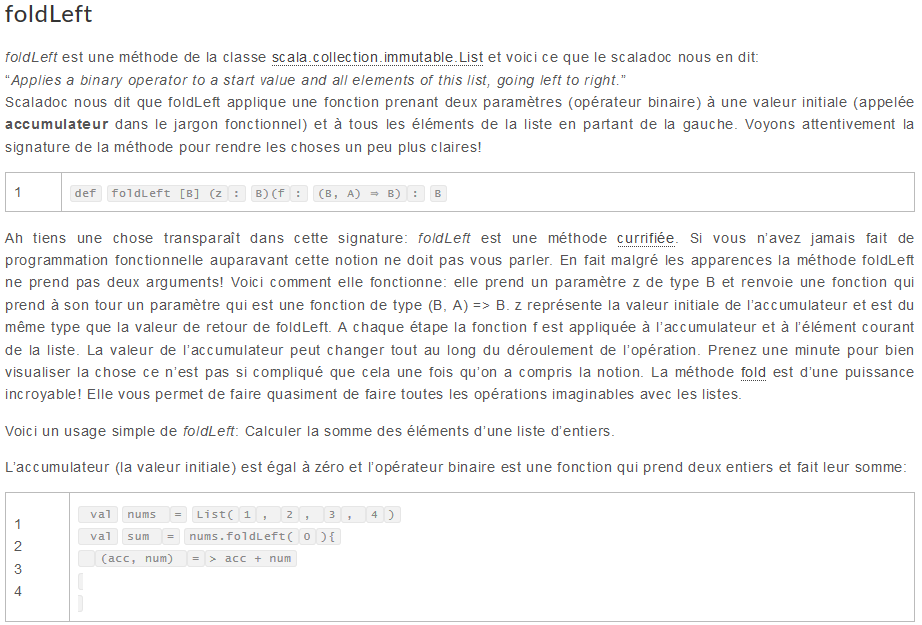
**AIDE SCALA** :

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.

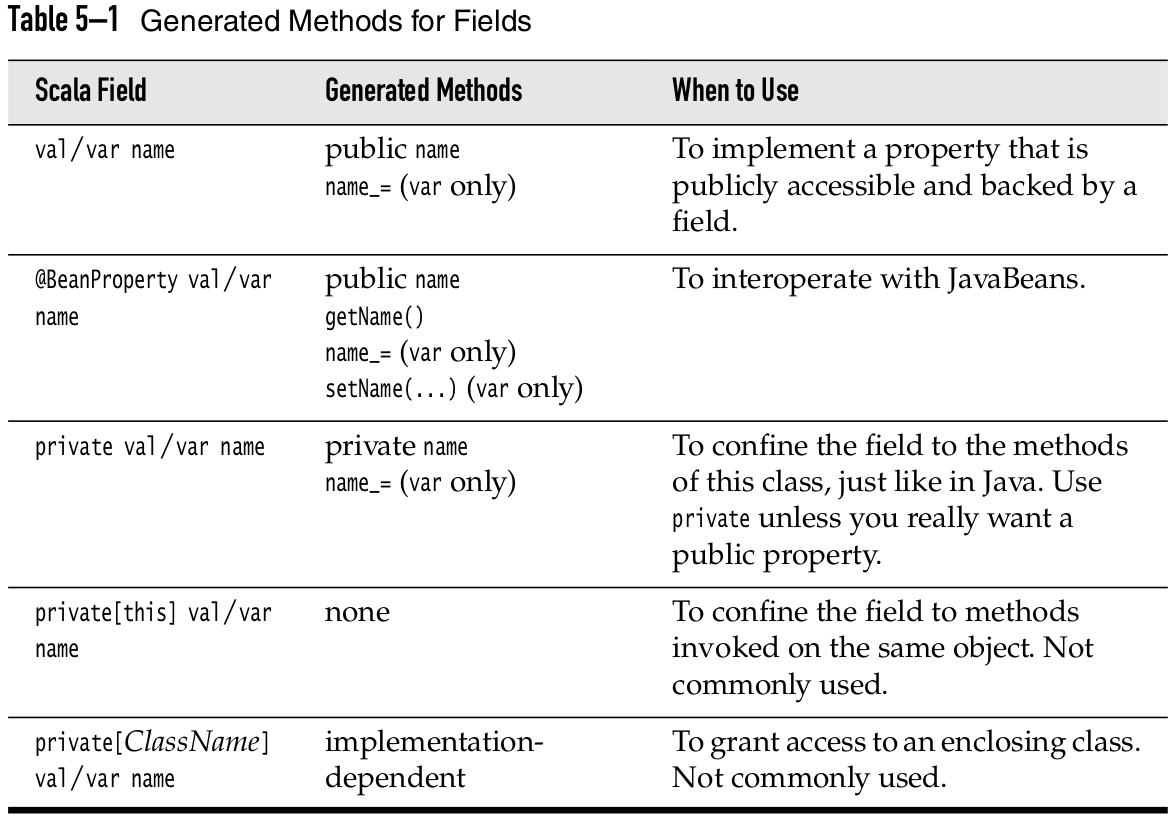
Scala allows an even more severe access restriction with the private[this] qualifier:

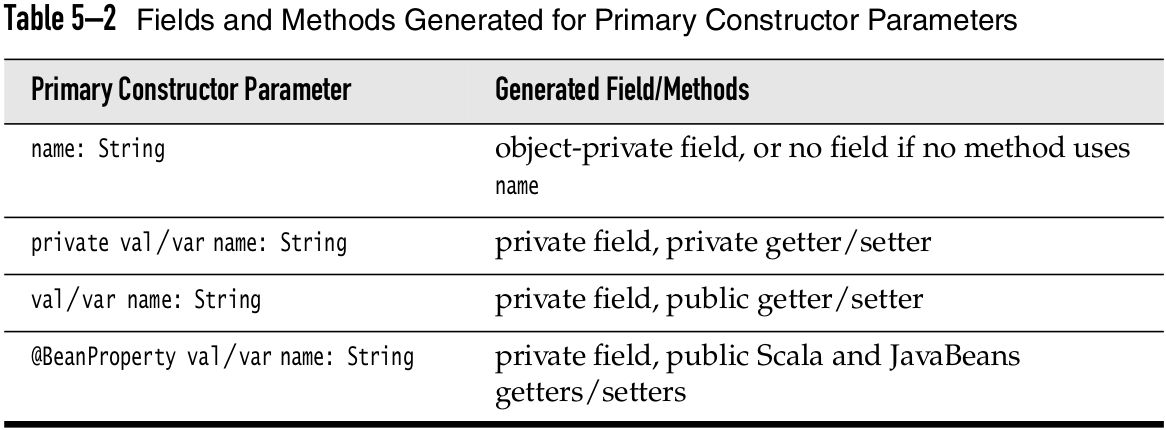
**private[this] var value = 0 // Accessing someObject.value is not allowed**

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.