Développement Orienté Objets

Les bases de Kotlin

Arnaud Lanoix Brauer
Arnaud.Lanoix@univ-nantes.fr

IUT de Nantes
Départemement informatique

Sommaire

1. Introduction
2. Les variables
3. Les structures de contrôles
4. Les fonctions
5. Les tableaux

Java

- Langage de programmation orienté objet datant de 1995 (héritant de C++ (entre autre))
- Développé par Sun Microsystems, puis par Oracle ; en partie open-source
- Multiplateforme via l'utilisation d'une machine virtuelle : la JVM (Java Virtual Machine)
- Langage d'exécution en partie interprété, en partie compilé : le bytecode
- Gestion mémoire simplifiée via l'utilisation d'un Garbage Collector (Ramasse-miettes)
- Toujours en évolution : Java SE 17 (sept. 2021)
- `https://www.oracle.com/java/technologies/`

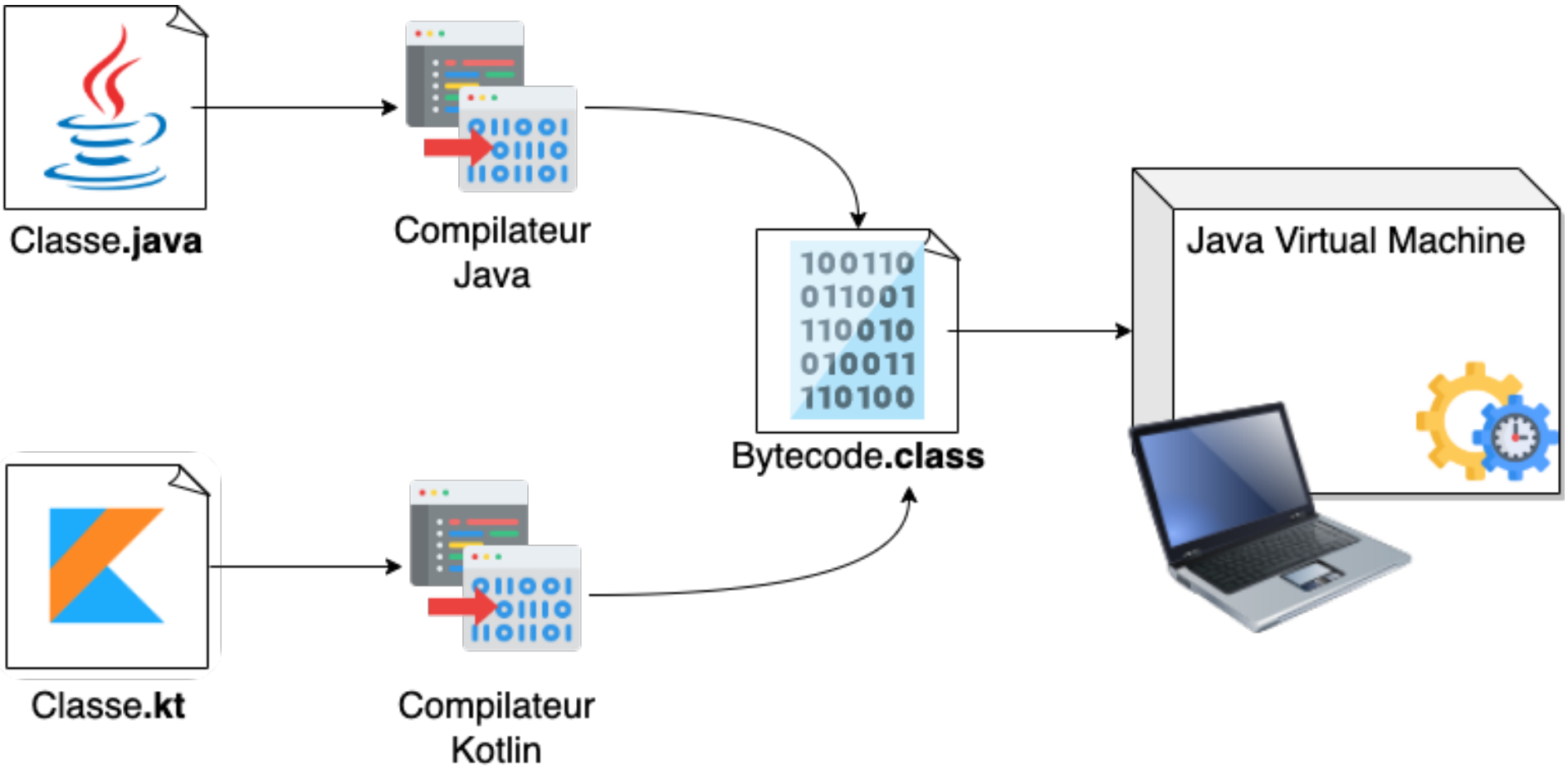
Kotlin

- Langage de programmation orienté objet **et** fonctionnel
- Développé à partir de 2010 par JetBrains et de nombreux autres contributeurs (complètement open-source)
- 100 % interopérable avec Java
  - Même langage compilé : le bytecode
  - Même machine virtuelle : la JVM (Java Virtual Machine)
  - Toujours multiplatforme
- Philosophie : *"plus concis, plus pragmatique, plus sûr que Java"*
- Langage "recommandé" par Google pour le développement Android à partir de 2019
- Kotlin également compatible avec *Javascript (JS)*[1], du code natif, etc.
- `https://kotlinlang.org/`

vs.

1. rien à voir avec Java

Kotlin vs. Java



Le classique "HelloWorld" : `Hello.kt`

```kotlin
// 1er programme Kotlin

fun main() {
    println("*** Hello students !!!! ***")
}
```

Le classique "HelloWorld" : `Hello.kt`

```
// 1er programme Kotlin

fun main() {
    println("*** Hello students !!!! ***")
}
```

- Compilation via la commande `kotlinc` (dans un terminal) :
  `kotlinc src/Hello.kt -d bin`

Le classique "HelloWorld" : `Hello.kt`

```
// 1er programme Kotlin

fun main() {
    println("*** Hello students !!!! ***")
}
```

- Compilation via la commande `kotlinc` (dans un terminal) :
  `kotlinc src/Hello.kt -d bin`
- Résultat de la compilation dans le dossier :
  `bin/HelloKt.class`

Le classique "HelloWorld" : `Hello.kt`

```
// 1er programme Kotlin

fun main() {
    println("*** Hello students !!!! ***")
}
```

- Compilation via la commande `kotlinc` (dans un terminal) :
  `kotlinc src/Hello.kt -d bin`
- Résultat de la compilation dans le dossier :
  `bin/HelloKt.class`
- Exécution via le lancement de la machine virtuelle Java :
  `kotlin -cp bin HelloKt` (ou `java -cp bin HelloKt`)
- Affichage dans le terminal :
  `*** Hello students !!!! ***`

Sommaire

1. Introduction
2. Les variables
   - Déclarer des variables
   - Utiliser des variables
3. Les structures de contrôles
4. Les fonctions
5. Les tableaux

Déclarer des variables : `var` ou `val`

```
val monNom : String = "Arnaud Lanoix"
var monAge : Int = 42
```

En Kotlin, on manipule deux sortes de variables :

- Des variables classiques, dite *muables*, grâce à `var` pour "variable"
- Des variables immuables, grâce à `val` pour "valeur", c-à-d des variables non-modifiables, une fois initialisées (= "en lecture seulement")

Déclarer des variables : `var` ou `val`

```
val monNom : String = "Arnaud Lanoix"
var monAge : Int = 42
```

En Kotlin, on manipule deux sortes de variables :

- Des variables classiques, dite *muables*, grâce à `var` pour "variable"
- Des variables immuables, grâce à `val` pour "valeur", c-à-d des variables non-modifiables, une fois initialisées (= "en lecture seulement")

Nommage des variables

- le premier caractère est une lettre minuscule
- utiliser uniquement des caractères alphanumériques (=lettres ou chiffres)
- utiliser la notation *"lowerCamelCase"*, c-à-d utiliser des lettres majuscules uniquement pour séparer les mots et faciliter la lecture

Les variables immuables : `val`

Une variable "immuable" ne veut pas dire que la variable doit forcément être initialisée dès sa déclaration, mais dans le même bloc de code :

```
{
  val monNom : String
  ...
  if (suisJeLeProf()) {
    monNom = "Arnaud Lanoix"
  }
  else {
    monNom = "Etudiant inconnu"
  }
}
```

Les variables immuables : `val`

Une variable "immuable" ne veut pas dire que la variable doit forcément être initialisée dès sa déclaration, mais dans le même bloc de code :

```
{
  val monNom : String
  ...
  if (suisJeLeProf()) {
    monNom = "Arnaud Lanoix"
  }
  else {
    monNom = "Etudiant inconnu"
  }
}
```

ATTENTION : variables "immuables" $\neq$ constantes

Une variable `val` prend une valeur **dynamiquement** à l'exécution, contrairement à une constante :

```
const val MAX : Int = 10_000_000
```

Les types primitifs

Type	occ. mém. (bits)	min	max
Les nombres entiers			
`Byte`	8	-128	127
`Short`	16	-32_768	32_767
`Int`	32	-2_147_483_648	2_147_483_647
`Long`	64	-9_223_372_036_854_775_808	9_223_372_036_854_775_807
Les nombres flottants			
`Float`	32		
`Double`	64		
Les caractères			
`Char`	16		
`String`	variable	= séquence de caractères [2]	
Les booléens			
`Boolean`	8	`true` (vrai) ou `false` (faux)	

2. On y reviendra

Variables et types primitifs

```
var nbEtudiants : Int = 113
var argent : Long = 1_000L
val age : Byte = 10
var uneLettre : Char = 'a'
var estOk : Boolean = true // false
var unPrenom : String = "Totoro"
val resultat : Double = 99_999.999999999
var valeur : Float = 87.345f
```

L'inférence de type

Indiquer le type d'une variable n'est pas forcément nécessaire : le compilateur déduit automatiquement le type des variables, quand c'est possible.

```
val monNom : String = "Arnaud Lanoix"
var monAge : Int = 42
var autreAge : Short = 6
```

(presque) équivalent à

```
val monNom = "Arnaud Lanoix"
var monAge = 42
var autreAge = 6
```

Attention, sans autre précision, les "entiers inférés" sont des `Int`, les "flottants" des `Double`, ...

Affichage (dans le terminal)

On affichera du texte dans le terminal via les fonctions `print` et `println`

```
fun main() {
    val premiereLettre = 'a'
    val nbLettres = 5
    print("Le mot 'alpha' commence par un ")
    println(premiereLettre)
    print("Le mot 'alpha' contient ")
    print(nbLettres)
    println(" lettres")
}
```

Interpolation de chaînes de caractères

On peut substituer directement une variable `x` par sa valeur, via `$x` :

```
fun main() {
  val premiereLettre = 'a'
  val nbLettres = 5
  println("Le mot 'alpha' commence par un $premiereLettre
      et contient $nbLettres lettres")
}
```

On peut également interpréter une expression, via `${...}` :

```
fun main() {
  val nbConsonnes = 3
  val nbVoyelles = 2
  println("Le mot 'alpha' contient
    ${nbConsonnes + nbVoyelles} lettres")
}
```

Opérations sur les types primitifs

- Addition, soustraction, multiplication, division, modulo

```
var nb = 8
var add = nb + 2
println(add)          // 10
var min = add - nb
println(min)          // 2
var mult = 5 * min
println(mult)         // 10
var div = mult / 4
println(div)          // 2
var reste = 5 % min
println(reste)        // 1
var div2 = mult / 4.0
println(div2)         // 2.5
```

- Incréments

```
var compteur = 1
compteur++
compteur++
println(compteur)     // 3
compteur--
println(compteur)     // 2
compteur += 2
println(compteur)     // 4
compteur -= 3
println(compteur)     // 1
compteur *= 7
println(compteur)     // 7
compteur /= 2
println(compteur)     // 3
```

Comparaisons sur les types primitifs

- Opérateurs de comparaison
  - Egalité `==`
  - différence `!=`
  - Supérieur `>`
  - Supérieur ou égal `>=`
  - Inférieur `<`
  - Inférieur ou égal `<=`
- Opérateurs logiques
  - "et" `&&`
  - "ou" `||`
  - négation `!`
- Appartenance `in`

```
val grand = 5
val petit = 2
var cond : Boolean

cond = grand == petit
println("$cond") // false
cond = grand == 5
println("$cond") // true
cond = grand != petit
println("$cond") // true
cond = grand >= petit
println("$cond") // true
cond = (grand < petit) && (petit > 10)
println("$cond") // false
cond = (grand > petit) || !(petit > 10)
println("$cond") // true
cond = grand in 4..Int.MAX_VALUE
println("$cond") // true
```

Le package `kotlin.math`

propose de nombreuses fonctions et constantes mathématiques

- la valeur de π : `PI`
- Valeur absolue d'un nombre : `abs(x : Double)`
- Arrondi à l'entier supérieur : `ceil(x : Double)`
- la racine carrée : `sqrt(x : Double)`
- le plus grand de deux nombres : `max(a : Double, b : Double)`
- ...

Il est nécessaire d'importer le package pour avoir accès à ces fonctions :

```
import kotlin.math.*
```

Documentation détaillée :
`https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.math/`

Sommaire

1. Introduction
2. Les variables
3. Les structures de contrôles
   - Les conditionnelles
   - Les boucles
4. Les fonctions
5. Les tableaux

La condition `if... else...`

```
var cptAbs : Int = ...

// Survenue d'une nouvelle abs
var justifiee : Boolean = ...

if (!justifiee) {
  println("Abs comptabilisee")
  cptAbs += 1
}

if (justifiee)
  println("rien a faire")
```

```
if (cptAbs >= 5) {
  println("Echec($cptAbs abs)")
}
else if (cptAbs == 4) {
  println("Alerte rouge($cptAbs Abs)")
  println("* alerter tuteur *")
}
else if (cptAbs in 1..3)
  println("Attention($cptAbs abs)")
else
  println("Pas d'absence")
```

- On peut imbriquer les `if... else...`
- On peut se passer des `{...}` si le bloc d'instructions ne contient qu'une instruction

La condition `when...`

pour simplifier l'imbrication des `if... else...`

```
    when {
      cptAbs >= 5 -> println("Echec ($cptAbs abs)")
      cptAbs == 4 -> {
            println("Alerte rouge ($cptAbs abs)")
            println("* alerter tuteur *")
      }
      cptAbs in 1..3 -> println("Attention ($cptAbs abs)")
      else -> println("Pas d'absence")
    }
```

On peut préciser sur quelle variable porte le `when` :

```
  when (cptAbs) {
    in 5.. Int.MAX_VALUE -> println("Echec ($cptAbs abs)")
    4 -> {
          println("Alerte rouge ($cptAbs abs)")
          println("* alerter tuteur *")
    }
    in 1..3 -> println("Attention ($cptAbs abs)")
    else -> println("Pas d'absence")
  }
```

`if... else...` avec retour de valeur

```
var a : Int = ...
var b : Int = ...
var max = if (a >= b) {
    println("$a plus grand que $b")
    a // la derniere instruction du bloc est retournee
}
else if (a <b) {
    println("$a plus petit que $b")
    b
}
else b
println(max)
```

`when...` avec retour de valeur

```
max = when {
   a > b -> a
   a < b -> b
   else -> {
      println("$a egal $b")
      a
   }
}
println(max)
```

La boucle `while`

```
var cptRebourd = 10
println("Depart dans...")

while (cptRebourd >= 0) {
  println(cptRebourd)
  cptRebourd--
}

println("Go !!!")
```

- Attention aux boucles infinies
- les boucles `while` sont à utiliser quand on ne peut pas "prévoir" le nombre d'itérations
- dans l'exemple, on devrait (plutôt) utiliser une boucle `for`

La boucle `for`

```
println("Depart dans...")
for (cpt in 10 downTo 0) {
  println(cpt)
}
println("Go !!!")
```

```
println("Depart a 10...")
for (cpt in 0 until 10  step 2) {
  println(cpt)
}
println("Go !!!")
```

On doit préciser

- la variable d'itération ; ici `cpt`
- la valeur "initiale"
- l'ordre d'itération : décrémental `downTo` ou incrémental `until`
- la valeur "limite" incluse si `downTo`, exclue si `until`
- le pas d'itération `step` – facultatif si `1`

Boucle `for` sur un intervalle

```
println("Depart a 10...")
for (cpt in 0..10 step 1) {
  println(cpt)
}
println("Go !!!")
```

- Dans ce cas, la valeur "limite" est incluse
- le pas d'itération `step` est également facultatif si `1`

Sommaire

1 Introduction

2 Les variables

3 Les structures de contrôles

4 Les fonctions

5 Les tableaux

Déclarer des fonctions

Une fonction est un morceau de code réutilisable qui réalise une tâche précise.

Une fonction est définie par :

- le mot-clef `fun`
- un nom
- (éventuellement) des paramètres et leurs types
- (éventuellement) un résultat typé et renvoyé (`return`)

```
fun log(msg : String, niv : Int) {
  println("*** LOG($niv):$msg ***")
}

fun mult(a : Int, b : Double, c : Double) : Double {
  var resultat = a * b * c
  return resultat
}
```

Déclarer des fonctions

Une fonction est un morceau de code réutilisable qui réalise une tâche précise.

Une fonction est définie par :

- le mot-clef `fun`
- un nom
- (éventuellement) des paramètres et leurs types
- (éventuellement) un résultat typé et renvoyé (`return`)

```
fun log(msg : String, niv : Int) {
  println("*** LOG($niv):$msg ***")
}

fun mult(a : Int, b : Double, c : Double) : Double {
  var resultat = a * b * c
  return resultat
}
```

Les paramètres sont immuables

Impossible de faire cela :

```
fun inc(x: Int) : Int {
  return x++  // error: val cannot be reassigned
}
```

Appeler des fonctions

On doit passer des valeurs ou des variables comme arguments de la fonction appelée et on stocke l'éventuel résultat.

```
fun main() {
  log("azerty", 3)
  log("qwerty", 1)
  val x = mult(2, 3.0, 1.0)
  println(x)  //  6.0
  val y = mult(2, x, x)
  println(y)  // 72.0
}
```

Affecter des valeurs par défaut aux paramètres

= rendre optionnel certains paramètres afin de pouvoir les omettre lors des appels de la fonction

```
fun log(msg : String = "hello", niv : Int = 1) {
  println("*** LOG($niv):$msg ***")
}
```

```
fun main() {
log("azerty", 3)
log("qwerty", 1)
log("dvorak")
// log(5) ne compile pas
log()
}
```

Affecter des valeurs par défaut aux paramètres

= rendre optionnel certains paramètres afin de pouvoir les omettre lors des appels de la fonction

```
fun log(msg : String = "hello", niv : Int = 1) {
  println("*** LOG($niv):$msg ***")
}
```

```
fun main() {
log("azerty", 3)
log("qwerty", 1)
log("dvorak")
// log(5) ne compile pas
log()
}
```

`log(5)` : le compilateur échoue
"the integer literal does not conform to the expected type String"

Modifier l'ordre d'appel des arguments

- possible si l'on précise les noms de chaque paramètre
- compatible avec les paramètres par défaut (permet de lever les ambiguités)

```
fun log(msg : String = "hello", niv : Int = 1) {
  println("*** LOG($niv):$msg ***")
}
```

```
fun main() {
  log(niv = 7, msg = "qwertz")
  log(msg = "colemak")
  log(niv = 4)
}
```

Fonction en écriture raccourcie

```
fun mult(a : Int, b : Double, c : Double) : Double {
  return a * b * c
}
```

peut s'écrire de manière raccourcie

```
fun mult2(a : Int, b : Double, c : Double) = a * b * c
```

- uniquement possible si le corps de la fontion `{...}` ne compte qu'une instruction
- on peut omettre le type de retour (= inférence de type)

Aller plus loin avec les fonctions

On reviendra plus tard sur

- la surcharge de fonctions
- les fonctions récurcives `tailrec`
- les fonctions `infix`
- ...

Sommaire

1. Introduction
2. Les variables
3. Les structures de contrôles
4. Les fonctions
5. Les tableaux

Déclarer un tableau

Deux possibilités :

1. un tableau prérempli

```
val notes = arrayOf(12.0, 7.0, 10.5, 8.2, 17.8)
val matieres = arrayOf("Info", "Math", "Anglais", "Eco", "Comm")
```

2. un tableau vide

```
val notes0 = arrayOfNulls<Double>(4)
val matieres0 = arrayOfNulls<String>(10)
```

- dans le cas 2. il faut déclarer le type des éléments contenus `<...>` et la taille du tableau
- dans le cas 2. toutes les cases du tableau contiennent la valeur `null` [3]
- le type des tableaux est `Array<Double>` et `Array<String>`
- la taille du tableau est définitivement fixée par le nombre d'éléments contenus

3. On y reviendra

Accéder aux cases d'un tableau

```
val matieres = arrayOf("Info", "Math", "Anglais", "Eco", "Comm")
```

Classiquement, les tableaux sont indicés de 0 à taille du tableau - 1.
Le tableau `matieres` contient 5 cases indicées de 0 à 4.

indice	0	1	2	3	4
valeur	`"Info"`	`"Math"`	`"Anglais"`	`"Eco"`	`"Comm"`

On accède aux valeurs d'un tableau via `[...]` :

```
val mat = matieres[0]
println(mat)
println(matieres[2])
matieres[0] = "Droit"
matieres[2] = "Russe"
println(matieres[2])
```

Accéder aux cases d'un tableau

```
val matieres = arrayOf("Info", "Math", "Anglais", "Eco", "Comm")
```

Classiquement, les tableaux sont indicés de 0 à taille du tableau - 1.
Le tableau `matieres` contient 5 cases indicées de 0 à 4.

Avant

indice	0	1	2	3	4
valeur	"Info"	"Math"	"Anglais"	"Eco"	"Comm"

On accède aux valeurs d'un tableau via `[...]` :

```
val mat = matieres[0]
println(mat)
println(matieres[2])
matieres[0] = "Droit"
matieres[2] = "Russe"
println(matieres[2])
```

Après

indice	0	1	2	3	4
valeur	"Droit"	"Math"	"Russe"	"Eco"	"Comm"

Parcourir un tableau

```
val notes = arrayOf(12.0, 7.0, 10.5, 8.2, 17.8)
```

- La taille d'un tableau est stockée dans la propriété `size`

```
var nbNotes = notes.size
```

- Un parcours indicé s'écrit donc ainsi :

```
for (indice in 0 until notes.size) {
  println(notes[indice])
}
```

Parcourir un tableau

```
val notes = arrayOf(12.0, 7.0, 10.5, 8.2, 17.8)
```

- La taille d'un tableau est stockée dans la propriété `size`

```
var nbNotes = notes.size
```

- Un parcours indicé s'écrit donc ainsi :

```
for (indice in 0 until notes.size) {
  println(notes[indice])
}
```

Foreach

On peut aussi parcourir un tableau ainsi :

```
for (note in notes) {
  println(note)
}
```

Tableaux en paramètre d'une fonction

Il est nécessaire de préciser le type du tableau en paramètre de la fonction : `Array<Double>` par exemple

On peut modifier le contenu des cases d'un tableau passé en paramètre d'une fonction.

```
fun moyenne(tab: Array<Double>) : Double {
  var moy = 0.0
  for (note in tab) {
    moy += note
  }
  return moy / tab.size
}
fun changer(tab : Array<Double>,
  note : Double, pos : Int) {
  tab[pos] = note
}
```

```
fun main() {
  val notes = arrayOf(0.0, 10.0)
  val res = moyenne(notes)
  println("moyenne : $res") // 5.0
  changer(notes, 20.0, 0)
  println("moyenne : $res") // 10.0
}
```

Aller plus loin

On reviendra plus tard sur

- les tableaux de types primitifs
- les tableaux multidimensionnels
- les variables *nullable* qui peuvent être `null`