Kotlin

Sommaire

1. Avant-propos

Kotlin est un langage multi-paradigmes (procédurale, POO et fonctionnel), cependant il ne laisse pas totalement le choix du paradigme. La logique globale était d'avoir un langage permettant de produire du code plus rapidement (syntaxe plus courte que **Java**) mais sécurisé. Pour arriver à ses fins **Kotlin** s'appuie sur un **typage fort** et une logique de **programmation fonctionnelle**.
Il en résulte qu'il n'est pas aisé de segmenter totalement l'étude du langage, il faut aborder de front plusieurs éléments du langage pour comprendre certains mécanismes de sa syntaxe.

2. Présentation & Origines

2.1. Les origines

Kotlin est un langage relativement relativement récent (2011). Il est dévellopé par l'éditeur **JetBrains**. Le nom du langage vient de l'île de Kotlin située en Russie. En effet, l'équipe de développement était basée à St-Pétesbourg et s'était tout simplement inspirée du nom de l'île pour baptiser son nouveau langage.

L'objectif de **Kotlin** est d'offrir une alternative au langage **Java** tout en permettant le fonctionnement du code **Java** existant.
Pour ce faire, **Kotlin** va être compilé en byte code pour la **JVM**. L'avantage est double :

- **Kotlin** hérite naturellement du caractère multiplateforme de **Java**;
- Les anciens programmes compilés à partir de **Java** demeurent opérationnels, ce qui augure une existante parallèle des 2 langages afin de d'offrir une transition en douceur de **Java** vers **Kotlin**.

Sources : Page Wikipédia de Kotlin

2.2. Pourquoi choisir Kotlin ?

Kotlin se veut être un langage moderne afin de faciliter le développement d'applications. Il garde les bases syntaxiques de **Java** mais en offrant les facilités de langages comme **C#** ou **Python** entres autres.

Kotlin est appréciable par ses qualités intrinsèques :

- Programmation avec plusieurs paradigmes :
  - **Procédural;**
  - **Orienté Objet (POO);**
  - **Fonctionnel**.
- La syntaxe est plus concise là ou **Java** était un peu verbeu.
- Simplification et réduction du temps de développement car **Kotlin** permet de réaliser des automatisations. La génération des accesseurs et mutateurs pour les attributs des classes est une très bonne illustration des gains de productivité et d'ergonomie permis par **Kotlin**.
- **Kotlin** renforce les bonnes pratiques et la fiabilité du code développé.

Kotlin est déjà un langage de référence :

En 2017 **Google** adopte officiellement **Kotlin** comme second langage de dévellopement pour son OS **Android**. En 2019 **Google** fait passer **Kotlin** comme langage recommandé pour le développement sous **Android**.

Du côté serveur, le framework **Spring** supporte officiellement **Kotlin** en 2017.

3. Références

- Le site officiel : Accueil Kotlin
- Documentation officielle : Documentation Kotlin
- Le playground pour tester du code en ligne : Playground

4. Comment coder du Kotlin ?

Le plus simple est d'utiliser **IntelliJ IDEA** de **JetBrains**. C'est l'**IDE** de prélidiction pour **Java** et encore plus pour développer **Kotlin**.

Il existe 2 versions de cet **IDE**, la version gratuite (*Community Edition*) et la version payante (*Ultimate*).

Le téléchargement des 2 versions est disponible sur la page d'accueil d'**IntelliJ** : IntelliJ IDEA

Le compilateur sera fourni et prise en charge par **IntelliJ**.
Android Studio prend également en charge **Kotlin** pour le développement d'application

En cas d'utilisation d'un autre **IDE** ou éditeur de texte, il faudra installer le compilateur qui est mis à disposition sur **GitHub**. La documentation officielle consacre une partie sur l'utilisation du compilateur.

- Compilateur **Kotlin** : Releases sur GitHub
- Documentation du compilateur : Doc du compilateur

Conseil : Les fabricants de Kotlin mettent à disposition une sandbox en ligne, dans laquelle vous pourrez tester tous les exemples.

4.1. Point d'entrée du programme

Tout comme en **C/C++** c'est la fonction `main()` qui est sert de point d'entrée pour l'exécution d'un programme.

La définition d'une fonction se fait avec le mot clé **fun**

Hello World en Kotlin

```
fun main(){
    println("Hello World !")
}
```

4.2. Conventions pour la rédaction du code

Comme d'autres de langages, Kotlin préconise de bonnes pratiques sur la formatage du code (nombre d'espace ,nommage des identifiants, etc).

- Convention de nommage des variables, utiliser la convention **camelCase**, une notation consistant à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot. La première lettre du premier mot étant en minuscule exemple : **maVariable**
- Placer des espaces entre les opérandes et opérateurs :
  - `5 + 3 * 2 / 4`
  - `a + 3 * b`
- Placer un espace entre la paire ouvrant et le mot clé d'une structure de contrôle (`if`, `while`, `for`, et `when`,) :
  - `if (a == 2)`
  - `while (x > 2)`
  - `for (i in 1..5)`
- Pas d'espace entre la parenthèse ouvrant et l'identifiant d'une fonction, méthode, etc :
  - `fun exemple()`
  - `classe UneClasse(val x Int)`
- NE PAS placer un espace avant `:`, mais TOUJOURS placer un espace après
  - déclaration d'une variable et de son type.

```
val myNumber: Long = 40_000
```

4.3. Commentaires

4.3.1. Commentaires de fin de ligne

Ce type de commentaire est placé en fin d'une ligne d'instruction ou seul mais sur une seule ligne. Ils sont introduits par un double slash : `//`

Exemples de commentaires simples :

```
// Affichage des informations :
println("Texte affiché dans le terminal.")
// println ajoute le saut de ligne
```

4.3.2. Commentaires multiligne

On retrouve la même syntaxe que pour les langages comme C/C++/Java. On utilise la combinaison `/` `en ouverture de commentaire et/` en fermeture de commentaire.

Exemple :

```
/* Utilisation d'un commentaire multiligne :
1ère ligne...
2e ligne
...
et ligne de fin. */
```

5. Introduction à Kotlin avec des exemples

5.1. Variables

Kotlin connaît deux types de variables : **les variables immuables**, qui sont en lecture seule, sont introduites par **val**. **Les autres variables**, dont la valeur est modifiable au fil du programme, sont introduites par **var**.

```
val nom = "John"
var age = 22
```

Contrairement au nom, qui est fixe, l'âge peut être adapté, par exemple dans une fonction.

Dans cet exemple, Kotlin a déterminé seul le type de valeur des variables. Il est également possible d'indiquer individuellement ces types de base. **val nom: String = "John"**

5.2. Types de base (basic types)

Kotlin travaille avec certains types de variables et de classes. Chaque type est un objet, ce qui distingue quelque peu Kotlin de Java.

5.2.1. Codage des valeurs numériques :

Tous les types numériques héritent de classe **Number**.

Table 1. Codage des nombres entiers :

MOT CLE DU TYPE	TAILLE MEMOIRE (bits)	VALEUR MINI	VALEUR MAXI
Byte	8	-128	127
Short	16	-32 768	32 767
Int	32	-2^{31}	$2^{31} - 1$
Long	64	-2^{63}	$2^{63} - 1$

Rappel :

En **Java/Kotlin** l'occupation maximale en mémoire (taille) ne dépend pas de la cible (OS & machine).

Table 2. Codage des nombres à virgule flottante :

MOT CLE DU TYPE	TAILLE MEMOIRE (bits)	BITS MANTISSE	BITS EXPOSANT
Float	32	24	8
Double	64	53	11

La comparaison de valeurs numériques en Kotlin ne peut se faire que **si les deux valeurs sont strictement du même type !** Deux valeurs indentiques mais de types différents ne seront pas considérés comme égaux ! Voir exemple ci-dessous.

Exemple : Comparaison du type de 2 variables :

```
val quinze_Int: Int = 15
val quinze_Long: Long = 15

// Vérification des types :
println("quinze_Int is Int : ${quinze_Int is Int}")
println("quinze_Long is Long : ${quinze_Long is Long}")
```

Tout se passe bien et on obtient le bon résultat dans la console.

Console :

```
quinze_Int is Int : true
quinze_Long is Long : true

Process finished with exit code 0
```

Ajoutons maintenant une instruction de comparaison d'égalité stricte.

Ajout de la comparaison des valeurs des 2 variables :

```
    println("quinze_Int == quinze_Long : ${quinze_Int == quinze_Long}")
```

Et là nous obtenons une erreur. .Console :

```
Kotlin: Operator '==' cannot be applied to 'Int' and 'Long'
```

On ne peut réaliser de comparaison d'égalité entre un objet de la classe Int et un objet de la classe Long.

Par contre on peut appliquer des comparaisons <, <=, > et >=.

Quels types privilègier pour les valeurs numériques ? : Pour les valeurs entières il est conseillé d'utiliser le type **Int** et pour les décimaux le type **Double**.

Dans Kotlin, vous pouvez utiliser des nombres sans aucune balise : le compilateur comprend que ce sont des valeurs numériques. Les virgules sont réalisées à l'aide de points. Afin de permettre une meilleure lisibilité, les séparateurs de milliers peuvent être représentés à l'aide de tirets.

```
val myNumber: Long = 40_000
```

Il est possible de convertir un nombre d'un type en nombre d'un autre type.

```
val myInt = 600
val myLong= myInt.toLong()
```

La commande **toLong** convertit la valeur « Int » en valeur « Long ». La commande fonctionne de façon analogue pour les autres types de nombres.

5.2.2. Booléens & caractères

On trouve 2 autres types que sont les **Booléens** et les **Caractères** :

Table 3. Codage des booléens et caractères

MOT CLE DU TYPE	DESCRIPTION
Boolean	Ne prend que 2 valeurs dites booléennes : **true** ou **false**
Char	Stock un caractère unique.

En raison du typage fort de **Kotlin** on ne peut pas réaliser de tests logiques comme `true == 1` ou `false == 0` Le compilateur considéra cela comme une erreur.

Pour les caractères, Kotlin met également à disposition le type de données spécifique Character : **Char**. Pour initialiser la variable plutôt que de d'utiliser des guillemets doubles, on utilise des guillemets simples.

```
val lettre: Char = 'a'
```

5.2.3. String

Un string est un ensemble de mots ou des phrases complètes, autrement dit, une chaîne de caractères. Pour utiliser un string dans Kotlin, placez le texte entre des guillemets doubles. Si vous souhaitez intégrer plusieurs lignes de texte, il est nécessaire d'ajouter trois guillemets doubles au début et à la fin (raw string).

```
val myString = "Ce string comporte une seule ligne."
val myLongString = """Ce string s'étend
sur plusieurs lignes."""
```

Comme dans de nombreux langages de programmation, Kotlin permet l'utilisation de caractères d'échappement : une barre oblique inversée permet de désigner un caractère ne faisant pas partie du string et devant être traité comme un caractère de contrôle. À l'inverse, une barre oblique inversée permet également d'insérer dans le string des caractères ayant normalement une autre signification dans Kotlin. Les caractères d'échappement suivants sont possibles :

1. \t : tabulation
2. \b : retour arrière
3. \n : nouvelle ligne
4. \r : retour chariot
5. \' : guillemets simples
6. \" : guillemets doubles
7. \\ : barre oblique inversée
8. \$: symbole dollar

Dans les strings, le symbole dollar sert à indiquer une balise. Il est possible de la définir comme variable lors d'une étape préalable. La balise est alors remplacée par une véritable valeur dans l'édition.

```
val author = "Sandra"
val myString = "Ce texte a été écrit par $author"
```

5.3. L'entrée standard avec readLine() et readln()

Kotlin dispose de la fonction readLine() pour permettre la lecture de valeur dans la console. La fonction readLine() retourne systématique la saisie sous la forme d'un String (idem en Python avec la fonction input() La fonction readLine() peut retourner le type null si la touche entrée est frappée sans aucune entrée préalable. Il existe depuis la version 1.6 de Kotlin une variante court de readLine() qui est readln().

en Python

```
nb = int (input("entrer un nombre"))
```

en Kotlin

```
println("entrer un nombre ")
val nb1 = readln().toInt()

println("entrer un nombre 2")
val nb2 = readln().toInt()
println("l'addition de $nb1 + $nb2 = ${nb1 + nb2} ")
```

Les fonctions **readline()** et **readln()** ne fonctionnent pas sur la sandbox en ligne.

5.4. Arrays Kotlin

Dans Kotlin, un array est une collection de données. Vous pouvez construire un array avec arrayOf() ou Array(). La première de ces fonctions est simple :

```
val tabEleves = arrayOf("adriEn","ahmeD","bertrand","éric","oliv","tom")

tabEleves[1] = "hamed"// on remplace ahmed par hamed

tabEleves.sort() // classement alphabétique

for(eleve in tabEleves){ // parcours du tableau
    println(eleve)
```

```
}
```

Si on souhaite limiter l'array à un type, il suffit de l'indiquer dans la fonction.

```
val myArray2 = arrayOf<Int>(10, 20, 30)
```

5.4.1. Déclarations & affectations sur les collections

Les collections sont utilisées pour stocker et manipuler des groupes d'objets ou de données. Plusieurs types de collections sont disponibles avec Kotlin, notamment:

- Listes - Collections ordonnées d'éléments permettant des doublons.
- Set : Collections non ordonnées d'éléments uniques.
- Map – Collections de paires clé-valeur, où chaque clé est unique.

A la différence des **Array**, les collections pourront être initialisées comme étant immuable ou mutables. C'est la fonction d'initialisation qui va affecter ce caractère à la liste instanciée. Ici le caractère de mutabilité concerne bien les valeurs stockées dans la liste. La mutabilité de la référence quant à elle repose toujours sur les mots clés **var** et **val**.
Nous opterons pour des références immuables dans nos prochains exemples.

5.4.2. Les listes

Voici un exemple de création et d'utilisation d'une liste :

```
val fruits = listOf("cerise", "banane", "orange", "pomme", "papaye")

    // Accéder à un élement de la liste
    println("First fruit: ${fruits[0]}") // affiche l'élément se trouvant à l'indice 0
    println("Last fruit: ${fruits.last()}") // affiche le dernier élément de la liste
    println("first fruit: ${fruits.first()}") // affiche le premier élément
    println(fruits.get(2)) // affiche l'élément se trouvant à l'indice 2


    //Parcourir la liste
    for ( fruit in fruits)
        println(fruit)

    //filtrer la liste
    val filtered = fruits.filter { it.startsWith("p") }
    println("Filtered list: $filtered")
```

Dans cet example, nous créons une liste de fruits en utilisant la fonction listOf, qui prend un nombre variable d'éléments et retourne une liste immuable. Nous montrons ensuite comment accéder aux éléments de la liste en utilisant l'indexation ou des fonctions spécifiques aux collections, ensuite nous montrons comment parcourir la boucle et enfin comment utiliser la

fonction de filtrage pour créer une nouvelle liste ne contenant que les éléments qui commencent par la lettre « p ».

Les autres types de collection de Kotlin peuvent être utilisés de manière similaire, avec des fonctions et des méthodes spécifiques adaptées aux caractéristiques uniques de chaque type. En utilisant ces types de collection, vous pouvez facilement gérer des groupes de données dans vos programmes.

Lorsque vous avez des opérations d'ajout, de suppression, d'insertion etc sur une collection vous devez préciser le caractère mutable de la collection dans le cas contraire toutes ces opérations provoqueront des erreurs de compilation.

```
val fruits = mutableListOf("cerise", "banane", "orange", "pomme", "papaye")

    fruits.add("poire") // ajoute poire à la fin de la liste
    fruits.removeAt(1) // supprime l'élément à l'index 1 : banane
    fruits.add(3,"clémentine") // ajoute clémentine à l'index 3
```

L'affichage du contenu d'une collection peut se faire en passant directement l'identifiant de la liste en argument de la fonction `print()` ou `println()`

```
println(fruits)
```

Le type Map (tableau associatif en PHP ou dictionnaire en Python)

Le type Map est un tableau associatif, c'est-à-dire que dans un tableau sont stockées des valeurs. Chaque valeur est associée à une clé pour permettre son accès au lieu d'utiliser un numéro d'indice.

```
    // Création d'un map :
    var tabAssoImmuable = mapOf(0 to "Zero", 1 to "Une", 2 to "Deux", 3 to "Trois")
    var tabAssoMutable = mutableMapOf(0 to "Zero", 1 to "Une", 2 to "Deux", 3 to
"Trois")

    // Exploitation :
    println("Exemple avec tabAssoImmuable : $tabAssoImmuable")
    println("tabAssoImmuable.keys : ${tabAssoImmuable.keys}")
    println("tabAssoImmuable.values : ${tabAssoImmuable.values}")

    // Accès par clé :
    println("Valeur à la clée 2 : tabAssoImmuable[2] = ${tabAssoImmuable[2]}")

    // Vérification présence clé :
    println("La clée 1 est-elle dans tabAssoImmuable : ${1 in tabAssoImmuable}")
    println("La clée 7 est-elle dans tabAssoImmuable : ${7 in tabAssoImmuable}")

    // Vérification présence valeur :
    println("La valeur \"Deux\" est-elle dans tabAssoImmuable : ${"Deux" in
```

```
tabAssoImmuable.values}")
```

Console :

```
Exemple avec tabAssoImmuable : {0=Zero, 1=Une, 2=Deux, 3=Trois}
tabAssoImmuable.keys : [0, 1, 2, 3]
tabAssoImmuable.values : [Zero, Une, Deux, Trois]
Valeur à la clée 2 : tabAssoImmuable[2] = Deux
La clée 1 est-elle dans tabAssoImmuable : true
La clée 7 est-elle dans tabAssoImmuable : false
La valeur "Deux" est-elle dans tabAssoImmuable : true

Process finished with exit code 0
```

6. Les structures/expressions conditionnelles

L'indentation est primordiale avec Python car elle sert à déterminer les blocs qui constituent votre code là où d'autres langages comme Kotlin privilégient les accolades **{ }** pour spécifier ces blocs. Lorsque l'indentation n'est pas nécessaire, elle est quand même utilisée pour une meilleure lisibilité du programme, car l'oubli d'une accolade provoquera une erreur.

Le elif en Python est remplacé par else if → sinon si. En Kotlin, vous devez mettre la ou les **conditions entre parenthèses**.

Les opérateurs de comparaison en Kotlin et PHP sont identiques :

- == (égal à) : renvoie True si les deux valeurs sont égales, False sinon.
- **!**= (différent de) : renvoie True si les deux valeurs sont différentes, False sinon.
- > (strictement supérieur à) : renvoie True si la première valeur est strictement supérieure à la seconde, False sinon.
- < (strictement inférieur à) : renvoie True si la première valeur est strictement inférieure à la seconde, False sinon.
- >= (supérieur ou égal à) : renvoie True si la première valeur est supérieure ou égale à la seconde, False sinon.
- < = (inférieur ou égal à) : renvoie True si la première valeur est inférieure ou égale à la seconde, False sinon.
- || : OU logique Vérifie qu'une des conditions est réalisée
- **&&** : ET logique Vérifie que toutes les conditions sont réalisées

Kotlin n'accepte pas les notations **or** et **and** vous devez utiliser pour le **or** : || et pour le **and** : &&

6.1. Syntaxe des expressions conditionnelles `if/else` en Kotlin

Les mots clés utilisés sont identiques que dans les langages de type C/C++/Java : **if** /**else if** et **else**

```
val heure = 22
if (heure < 10) {
  println("Good morning.")
} else if (heure < 20) {
  println("Good day.")
} else {
  println("Good evening.")
}
```

Avec Kotlin, vous pouvez utiliser une structure conditionnelle comme une expression conditionnelle et affectée une valeur à une variable

```
val heure = 20
val politesse = if (heure < 18) {
    "Good day."
  } else {
     "Good evening."
  }
println(politesse)
```

Quand vous utilisez ue structure conditionnelle comme une expression conditionnelle, il doit obligatoirement y avoir un **else**. Vous pouvez même afficher directement l'expression.

```
val heure = 20
println(if (heure < 18) "Good day." else "Good evening.")
```

6.2. Expression conditionnelle `when`

L'expression conditionnelle remplace la structure conditionnelle switch case que l'on retrouve dans de nombreux langage. On obtient le même résultat avec une syntaxe plus concise.

Exemple :

```
val day = 4

val result = when (day) {
  1 -> "Monday"
  2 -> "Tuesday"
  3 -> "Wednesday"
  4 -> "Thursday"
  5 -> "Friday"
```

```
  6 -> "Saturday"
  7 -> "Sunday"
  else -> "Invalid day."
}
println(result)

// Outputs "Thursday" (day 4)
```

Il est également possible d'associer plusieurs valeurs en les séparant avec une virgule, ou une plage de valeurs avec la notation **1..10** qui signifie pour des valeurs de allant de de 1 à 10

Exemple :

```
val resultat = when (classeEleve){
        "maternelle", "ce2", "ce1","cm2","cm1" -> "primaire"
        "6e","5e","4e","3e"-> "collège"
        "seconde", "première", "terminale" -> "lycée"
        else -> "classe inconnue"
    }
println(resultat)
```

6.3. Convertir une structure conditionnelle en expression conditionnelle

Kotlin préconise l'utilisation d'expression conditionnelle car elle est souvent plus lisible et elle évite les répétitions d'instructions inutiles. Exemple :

Structure conditionnelle :

```
val CouleurFeu = "Vert"

    if (CouleurFeu == "Rouge") {
        println("Stop")
    } else if (CouleurFeu == "Orange") {
        println("Slow")
    } else if (CouleurFeu == "Vert") {
        println("Go")
    } else {
        println("Couleur non valide")
    }
```

Expression conditionnelle :

```
val CouleurFeu = "Rouge"

val resultat = when (
    if (CouleurFeu == "Rouge") "Stop"
    else if (CouleurFeu == "Orange") "Slow"
```

```
    else if (CouleurFeu == "Vert")  "Go"
    else "Couleur non valide"

println(resultat)
```

When :

```
val couleurFeu = "Rouge"

val resultat = when (couleurFeu) {
    "Rouge" -> "Stop"
    "Orange" -> "Slow"
    "Vert" ->  "Go"
    else -> "Couleur non valide"
}
println(resultat)
```

7. Les structures itératives

7.1. Généralités sur les structures itératives en Kotlin

Kotlin propose plusieurs possibilités pour réaliser des structures itératives. On retrouve les instructions suivantes :

- La boucle `while` et `do while` (boucle non bornée) : Kotlin ne propose pas d'innovation, c'est une classique boucle dont la fin dépend d'une proposition logique.
- La boucle `for (…in…)` : C'est une boucle de type `for each` qui permet d'itérer sur des collections et des plages dont la syntaxe est très proche du **for de Python**
- La boucle `forEach` : C'est une boucle spécifique pour les **collections**, on peut aussi l'utiliser avec des plages.

7.2. Boucle `while`

Rien de particulier pour ce type de structure itérative. La syntaxe est classique :

Syntaxe de la boucle `while` :

```
    while (condition) {
        // Instructions à itérer
    }
```

Exemple d'une boucle de comptage :

```
    var i: Int = 0
    while( i < 10) {
        println("Iteration while n°$i")
```

```
        i++
    }
```

7.3. Boucle `do while`

On reste encore classique au niveau de la syntaxe :

Syntaxe de la boucle do while :

```
    do{
        // Instruction à itérer...
    }while (condition)
```

Exemple d'une boucle de comptage avec `do while` :

```
    var i: Int = 0
    do{
        println("Iteration do while n°$i")
        i++
    } while (i < 10)
```

7.4. Boucle `for (…in…)`

C'est la boucle de prédilection pour réaliser des itérations bornées. On peut également l'appliquer au contenu de collections, mais dans ce cadre la boucle forEach offre de meilleures performances.

Syntaxe de la boucle `for (…in…)` :

```
    for (item in itérable) {
        // Instruction à itérer
    }
```

Exemples sur des collections (`List`, `Map` et `Set`)

```
    val liste = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    val ensemble = setOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    val tabAssociatif = mapOf(1 to "un", 2 to "deux", 3 to "trois", 4 to "quatre")

    for (elmt in liste) {
        println(elmt)
    }
    for (elmt in ensemble) {
        println(elmt)
    }

    for (elmt in tabAssociatif) {
        println(elmt)
```

Nous pouvons appliquer la boucle `for (…in…)` à une plage.

Exemple de boucles `for (…in…)` sur une plage :

```
for (i in 0..10) {print(i)}
```

avec cette syntaxe la dernière valeur est comprise, la valeur de **10 sera donc affichée** alors qu'en **Python la borne de fin n'est pas comprise**.

La boucle peut également s'appliquer sur une plage de lettres.

Exemple sur une plage de lettres :

```
for (lettre in 'a'..'z') {
    print("$lettre - ")
}
```

Nous pouvons parcourir en sens inverse une plage, mais cela implique à utiliser le mot clé `downTo` à la place des 2 points `..`

Il faut impérativement utiliser `downTo` en effet vous aurez une erreur avec un intervalle comme : `10..1` au lieu de `10 downTo 1`. La première valeur n'est pas comprise donc le 10 ne sera pas affichée.

Utilisation de `downTo` :

```
for (i in 10 downTo 1) {println(i)}
```

Comme en Python nous pouvons également préciser le `step` (pas) sans précision comme en Python il est de **1** ou **-1**.

Utilisation de `step` :

```
for (i in 10 downTo 1 step 2) {println(i)}
```

Il existe une variante avec `until` qui remplace `..` La différence est qu'on atteint pas la dernière valeur.

Utilisation de `until` :

```
for (i in 0 until 10) {print("$i - ")}
```

Résultat :

```
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
```

Il faut être vigilant une plage ne peut être que croissante de 1 à 10 par exemple et la dernière valeur est comprise. Alors qu'avec `until` la valeur de fin n'est pas comprise.

7.5. Les boucles avec `break` et `continue`

Il est parfois intéressant de pouvoir interrompre une boucle ou de sauter une itération. Ces contrôles sur les boucles se réalisent avec les instructions : `break` ou `continue`

7.5.1. Utilisation de `break`

L'instruction `break` permet d'interrompre l'exécution d'une boucle. Les règles sont les suivantes :

- La boucle est interrompue aussitôt que le break est exécuté.
- Si le break est contenu dans une boucle imbriquée, l'interruption ne s'applique que sur la boucle la plus proche.

Exemple de l'instruction `break` :

```
    for (i in 1..2) {
        for (j in 0..5) {
            if (j == 3) break
            print(j)
        }
        println()
    }
```

Résultat, la boucle secondaire n'atteint jamais sa fin :

```
012
012
```

7.5.2. Utilisation de `continue`

L'instruction `continue` permet de faire sauter des itérations sur une boucle.

Exemple de l'instruction `continue` :

```
    for (i in 1..2) {
        for (j in 0..5) {
            if (j == 3) continue
            print(j)
        }
        println()
    }
```

Résultat, la boucle secondaire saute un tour pour j == 3 :

```
01245
01245
```

7.6. Boucle forEach

La boucle `forEach` est particulièrement intéressante avec les collections.

Syntaxe de `forEach`

```
    iterable.foreach {lambda}
```

Exemples sur des collections :

```
    val liste = listOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    val ensemble = setOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    val tabAssociatif = mapOf(1 to "un", 2 to "deux", 3 to "trois", 4 to "quatre")

    liste.forEach {println(it)}
    ensemble.forEach {println(it)}
    tabAssociatif.forEach {println(it)}
```

Exemple sur une plage :

```
    (0..10).forEach {println(it)}
```

7.7. Choix entre for (…in…) et forEach

La boucle forEach offre de meilleures performances pour réaliser des itération sur des collections. Par contre pour réaliser des boucles à partir d'une plage c'est la boucle for(…in…) qui est plus intéressante en terme de performance.

8. Les fonctions

Une fonction est un sous-programme permettant d'exécuter un ensemble d'instruction en un seul appel. Une fonction est composée d'une entête et d'un corps. Dans l'entête de la fonction se définit un identificateur, les paramètres de la fonction et le type de retour de la fonction.

Le corps de la fonction est délimité par des **accolades** dans lesquelles vous pouvez définir l'ensemble des instructions à exécuter.

8.1. Déclaration d'une fonction dans Kotlin.

Une fonction est déclarée en Kotlin en utilisant le mot clé fun. Voir l'exemple suivant.

```
fun hello() {
    println("Hello guest")
}
```

8.2. Appel d'une fonction dans Kotlin

Dans Kotlin une fonction est appelée comme dans d'autre langage de programmation comme Python ou PHP,par son nom, une parenthèse ouvrante, des arguments éventuels et la parenthèse fermante. Voir l'exemple suivant de la fonction hello précédent.

```
hello()
```

8.3. Le type de retour d'une fonction

Une fonction peut aussi retourner une valeur. Pour retourner une valeur utilisez le mot clé return. Voir l'exemple suivant

```
fun somme(a: Int, b: Int): Int {
    return a + b
}
```

8.4. Les valeurs par défaut des paramètres d'une fonction dans Kotlin.

Les paramètres de fonction peuvent avoir des valeurs par défaut, qui sont utilisées lorsqu'aucune valeur n'est définie pour l'argument correspondant lors de l'appelle de la fonction. Voir l'exemple suivant.

```
fun somme(a: Int = 3, b: Int = 4): Int {
    return a + b
}
fun main() {
    val s = somme()
    println("Le calcul de la somme est : $s")
}
```

Dans cet exemple la fonction somme est appelée sans passer d'arguments, ce sont les valeurs par défaut des paramètres de la fonction somme qui sont utilisées.

8.5. Les arguments nommés

Lorsqu'une fonction à beaucoup de paramètre, cela peut devenir difficile lors de l'appel de d'une

fonction d'associer une valeur à un argument, pour cela Kotlin vous permet de nommer un à plusieurs arguments.

Lorsque vous nommez les arguments dans un appel de fonction, vous pouvez modifier l'ordre dans lequel ils sont spécifiés. Voir l'exemple suivant

```
fun somme(a: Int = 3, b: Int = 4) {
    println("La somme de $a et $b est ${a + b}")
}
fun main() {
    somme(b = 10, a = 5)
}
```

8.6. Le type de retour Unit(vide en Kotlin)

Les fonctions en Kotlin doivent spécifier leur type de retour. Si la fonction ne retourne aucune valeur, sont type de retour est Unit (None en Python). Voir l'exemple suivant.

```
fun somme(a: Int = 3, b: Int = 4, c: Int = 3, d: Int = 12): Unit {
    println("La somme de $a, $b, $c et $d  est ${a + b + c + d}")
}
```

8.7. Les fonctions d'expression unique en Kotlin

Lorsqu'une fonction contient une seule expression, les accolades peuvent être omises. Le corps de la fonction est spécifié après le signe =. Voir l'exemple suivant

```
fun somme(a: Int, b: Int) = a + b
```