Cités des Métiers et des Compétences
La formation nouvelle génération

مكتب التكوين المهني و إنعاش الشغل
Office de la Formation Professionnelle et de la Promotion du Travail

Compte-rendu
Tp n°11

Nom: Boulahya

Prénom: Asma

Groupe: DEVOAM 202

Année de formation:2025/2026

Sommaire

Introduction

Dans le cadre de ce TP de développement mobile avec **Kotlin**, nous avons abordé plusieurs notions fondamentales du langage et leur application pratique. L'objectif principal était de renforcer notre compréhension de la **programmation fonctionnelle et orientée objet** en Kotlin, tout en explorant des concepts essentiels tels que les **fonctions anonymes, les lambdas, les filtres sur les collections, la gestion des exceptions** et la **programmation asynchrone simple**.

À travers une série d'exercices progressifs, nous avons appris à :

- Manipuler des **fonctions anonymes et lambdas** pour effectuer des calculs et filtrer des données.
- Utiliser les **structures de contrôle** pour tester des conditions (pair/impair).
- Travailler avec des **listes et collections** afin d'appliquer des traitements comme le filtrage des nombres pairs, impairs ou supérieurs à une valeur donnée.
- Mettre en place des **mécanismes de gestion d'erreurs** grâce aux blocs try/catch pour éviter les erreurs courantes comme la division par zéro ou les conversions incorrectes.
- Définir et utiliser une **exception personnalisée**, renforçant ainsi la robustesse du code.

Ce TP nous a donc permis d'allier théorie et pratique afin de mieux comprendre comment développer des programmes fiables, réutilisables et lisibles en Kotlin, tout en appliquant de bonnes pratiques de programmation.

Exercice1: Utilisation d'une fonction calculate avec lambda

```
fun calculate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  3 Usages
    return operation(a, b)
}

fun main() {
    val addition = calculate( a = 10,  b = 5) { x, y -> x + y }
    val soustraction = calculate( a = 10,  b = 5) { x, y -> x - y }
    val multiplication = calculate( a = 10,  b = 5) { x, y -> x * y }

    println("Addition : $addition")
    println("Soustraction : $soustraction")
    println("Multiplication : $multiplication")
}
```

Figure 1: code de la fonction calculate

Cet exercice nous a permis de créer une fonction générique calculate qui prend deux entiers et une **lambda** en paramètre. Cela montre comment passer une fonction comme argument afin d'effectuer différentes opérations (addition, soustraction, multiplication).

```
Addition : 15
Soustraction : 5
Multiplication : 50
```

Figure2: Résultat de l'exécution avec les trois opérations

Exercice2: Manipulation des listes et filtrage avec des lambdas

```kotlin
fun main() {
    val nombres = List( size = 10) { (1 ≤ .. ≤ 20).random() } // liste de 10 nombres aléatoires
    println("Liste originale : $nombres")

    val pairs = nombres.filter { it % 2 == 0 }
    val impairs = nombres.filter { it % 2 != 0 }
    val superieursA10 = nombres.filter { it > 10 }

    println("Nombres pairs : $pairs")
    println("Nombres impairs : $impairs")
    println("Nombres > 10 : $superieursA10")
}
```

Figure3: Liste originale affichée

Ici, nous avons généré une **liste de nombres entiers aléatoires** puis appliqué différentes lambdas pour filtrer :

- les nombres pairs,
- les nombres impairs,
- les nombres supérieurs à une valeur donnée (ex : 10).
  Cet exercice illustre la puissance des **fonctions d'ordre supérieur** et de la manipulation des collections en Kotlin.

```
Liste originale : [18, 3, 14, 15, 20, 5, 12, 2, 1, 11]
Nombres pairs : [18, 14, 20, 12, 2]
Nombres impairs : [3, 15, 5, 1, 11]
Nombres > 10 : [18, 14, 15, 20, 12, 11]
```

Figure4: Résultat après filtrage (pairs, impairs, > 10)

Exercice3: Fonction anonyme pour la somme d'une liste

```kotlin
fun main() {
    val liste = listOf(10, 20, 30, 40, 50, 60)

    val somme = fun(nums: List<Int>): Int {
        var total = 0
        for (n in nums) total += n
        return total
    }

    println("La somme est : ${somme(liste)}")
}
```

Figure5: Déclaration de la fonction anonyme.

Nous avons créé une **fonction anonyme** qui prend une liste d'entiers et retourne la somme de ses éléments. Cela met en pratique les bases des fonctions anonymes et la manipulation d'une collection d'entiers.

```
La somme est : 210
```

Figure6: Résultat de la somme

Exercice4: Vérification pair/impair avec fonction anonyme

```kotlin
fun main() {
    val estPair = fun(nombre: Int): Boolean {
        return nombre % 2 == 0
    }

    val nombres = listOf(3, 4, 7, 10, 15)
    for (n in nombres) {
        if (estPair(n)) {
            println("$n est pair")
        } else {
            println("$n est impair")
        }
    }
}
```

Figure7: Fonction anonyme estPair

Dans cet exercice, une **fonction anonyme** a été définie pour vérifier si un nombre est pair (true) ou impair (false). Nous avons ensuite testé plusieurs nombres et affiché le résultat pour chacun. Cet exercice illustre l'utilisation des fonctions anonymes dans un contexte pratique.

```
3 est impair
4 est pair
7 est impair
10 est pair
15 est impair
```

Figure8: Résultat de l'exécution de l'exercice 4

Exercice5: Division et gestion des erreurs avec try/catch

```
fun divide(dividende: Int, diviseur: Int): Int? {  2 Usages
    return try {
        dividende / diviseur
    } catch (e: ArithmeticException) {
        println("Erreur : Division par zéro !")
        null
    }
}

fun main() {
    println(divide( dividende = 10,  diviseur = 2)) // Affiche 5
    println(divide( dividende = 10,  diviseur = 0)) // Affiche le message d'erreur
}
```

Figure9: Code de la fonction divide

Nous avons créé une fonction divide qui prend un dividende et un diviseur en paramètres. Grâce au bloc try/catch, nous avons pu gérer l'exception de **division par zéro**. Cela démontre comment renforcer la **robustesse** d'un programme.

```
5
Erreur : Division par zéro !
null
```

Figure10: Résultat avec un diviseur valide et un cas de division par zéro

Exercice6: Conversion de chaîne en entier avec exception personnalisée

```kotlin
class NegativeNumberException(message: String) : Exception(message)  1 Usage

fun convertToInt(chaine: String): Int {  3 Usages
    return try {
        val valeur = chaine.toInt()
        if (valeur < 0) {
            throw NegativeNumberException( message = "Nombre négatif non autorisé: $valeur")
        }
        valeur
    } catch (e: NumberFormatException) {
        println("Erreur : La chaîne '$chaine' n'est pas un nombre valide.")
        0
    }
}

fun main() {
    println(convertToInt( chaine = "123"))     // OK
    println(convertToInt( chaine = "-45"))     // Déclenche NegativeNumberException
    println(convertToInt( chaine = "abc"))     // NumberFormatException
}
```

Figure11: Code de la fonction convertToInt et de l'exception personnalisée

Cet exercice nous a appris à :

1. Convertir une chaîne en entier avec toInt().
2. Gérer les erreurs de conversion grâce à try/catch.
3. Définir une **exception personnalisée** NegativeNumberException pour interdire les nombres négatifs.

```
123
Exception in thread "main" NegativeNumberException: Nombre négatif non autorisé: -45
    at EX6Kt.convertToInt(EX6.kt:8)
    at EX6Kt.main(EX6.kt:19)
    at EX6Kt.main(EX6.kt)
```

Figure12: Résultat avec différents cas de test

Conclusion

Ce TP nous a permis de mettre en pratique plusieurs concepts fondamentaux de la programmation en **Kotlin**. À travers des exercices variés, nous avons appris à utiliser les **fonctions anonymes et lambdas** pour écrire du code plus concis et réutilisable, à manipuler des **listes et collections** avec des filtres, et à vérifier des conditions simples comme la parité des nombres. Nous avons également exploré la **gestion des exceptions** grâce aux blocs try/catch, ce qui nous a permis de traiter des erreurs comme la division par zéro ou la conversion de chaînes non valides. Enfin, la création d'une **exception personnalisée** a renforcé notre compréhension de la robustesse et de la sécurité du code.

En résumé, ce TP a consolidé nos bases en **programmation fonctionnelle et orientée objet** avec Kotlin, tout en développant des réflexes de **bonne pratique** indispensables pour le développement d'applications mobiles fiables et maintenables