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
Devoir n°1

Nom: Boulahya

Prénom: Asma

Groupe: DEVOAM 202

Année de formation:2025/2026

Introduction..... 3
Exercice1..... 4
Exercice2:..... 5
Exercice3:..... 6
Exercice4:..... 7
Exercice5:..... 9
Exercice6:..... 9
Exercice7:..... 11

Introduction

Dans le cadre du module Kotlin (M202), ce TP a pour objectif de mettre en pratique des concepts essentiels du langage Kotlin liés aux opérations arithmétiques, à la gestion des conditions, ainsi qu'à la programmation concurrente à travers les threads. Les exercices proposés permettent également de découvrir et d'appliquer des notions avancées comme l'utilisation de lazy pour l'initialisation différée des variables et lateinit pour l'initialisation tardive d'objets.

Ce travail pratique vise donc à renforcer notre compréhension de la syntaxe et des fonctionnalités de Kotlin en réalisant différents programmes : calculs et vérifications logiques, gestion de la moyenne des notes, création et exécution de threads, simulation de services et de connexions, ainsi que l'optimisation des ressources grâce aux initialisations différées.

À travers ce TP, nous développons non seulement notre maîtrise du langage Kotlin, mais aussi notre capacité à concevoir des programmes fiables, modulaires et efficaces dans un contexte proche du développement mobile.

Exercice1: Opérations arithmétiques et vérifications logiques

```kotlin
fun main() {
    print("Entrez le premier nombre: ")
    val a = readLine()!!.toInt()
    print("Entrez le deuxième nombre: ")
    val b = readLine()!!.toInt()

    println("Addition: ${a + b}")
    println("Soustraction: ${a - b}")
    println("Multiplication: ${a * b}")
    if (b != 0) {
        println("Division: ${a / b}")
    } else {
        println("Erreur: Division par zéro")
    }

    println("Le premier est-il supérieur au second ? ${a > b}")
    println("La somme est ${if ((a + b) % 2 == 0) "paire" else "impaire"}")
}
```

Figure 1: Code de l'exercice 1 – opérations arithmétiques en Kotlin

Explication :

Cet exercice consistait à demander deux nombres à l'utilisateur puis à effectuer diverses opérations (addition, soustraction, multiplication et division avec gestion d'erreur). Nous avons aussi vérifié si le premier nombre est supérieur au second et si la somme des deux nombres est paire ou impaire.

Résultat :

```
Entrez le premier nombre: 12
Entrez le deuxième nombre: 2
Addition: 14
Soustraction: 10
Multiplication: 24
Division: 6
Le premier est-il supérieur au second ? true
La somme est paire
```

Figure2: Résultat de l'exécution de l'exercice 1

Exercice2: Calcul de la moyenne et gestion des résultats

```kotlin
fun main() {
    println("Entrez les trois notes: ")
    val n1 = readLine()!!.toDouble()
    val n2 = readLine()!!.toDouble()
    val n3 = readLine()!!.toDouble()

    val moyenne = (n1 + n2 + n3) / 3
    println("Moyenne = $moyenne")

    when {
        moyenne >= 80 -> println("Réussi avec mention excellente")
        moyenne >= 50 -> println("Réussi")
        else -> println("Échoué")
    }
}
```

Figure3: Code de l'exercice 2 – calcul de la moyenne et affichage du résultat

Explication:

L'objectif de cet exercice est de calculer la moyenne de trois notes saisies par l'utilisateur. En fonction de cette moyenne, nous déterminons si l'étudiant est **réussi** ou **échoué**, et ajoutons des mentions selon la

performance (réussi, réussi avec mention excellente, échoué). Cet exercice illustre l'usage des **structures conditionnelles (`if`, `when`)** pour traiter des cas différents.

Résultat :

```
Entrez les trois notes:
88
89
69
Moyenne = 82.0
Réussi avec mention excellente
```

Figure4: Résultat de l'exécution de l'exercice 2

Exercice3: Utilisation des threads avec Runnable

```
class Tache : Runnable {  3 Usages
    override fun run() {
        for (i in 1 ≤ .. ≤ 5) {
            println("Message du thread ${Thread.currentThread().name}")
            Thread.sleep( millis = 1000)
        }
    }
}
```

```
fun main() {
    val t1 = Thread( task = Tache())
    val t2 = Thread( task = Tache())
    val t3 = Thread( task = Tache())

    t1.start()
    t2.start()
    t3.start()
}
```

Figure5: Code de l'exercice 3 – création et exécution de threads

Explication:

L'objectif de cet exercice était de créer une classe implémentant `Runnable`, qui affiche un message toutes les secondes. Plusieurs threads ont été créés et démarrés, ce qui permet de comprendre la **programmation concurrente** et la gestion des tâches parallèles.

Résultat:

```
Message du thread Thread-0
Message du thread Thread-1
Message du thread Thread-2
Message du thread Thread-0
Message du thread Thread-2
Message du thread Thread-1
Message du thread Thread-1
Message du thread Thread-2
```

Figure6: Résultat de l'exécution de l'exercice 3

Exercice4: Classe Configuration avec lazy

```
class Configuration {  2 Usages
    init {
        println("Chargement de la configuration...")
    }
}

class App {  1 Usage
    val config: Configuration by lazy {  1 Usage
        Configuration()
    }
}

fun main() {
    val app = App()
    println("L'application démarre...")
    println("Utilisation de la configuration: ${app.config}")
}
```

Figure7: Code de l'exercice 4 – simulation du chargement de la configuration avec lazy

Explication:

Cet exercice simule le chargement d'une configuration complexe. La classe App possède une propriété **lazy** pour initialiser la configuration seulement quand elle est utilisée. Cela montre comment **optimiser l'utilisation des ressources** en retardant l'initialisation d'objets coûteux jusqu'à leur premier accès.

Résultat:

```
L'application démarre...
Chargement de la configuration...
Utilisation de la configuration: Configuration@4eec7777
```

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

Exercice5: Calcul coûteux avec lazy

```kotlin
fun calculCouteux(): Int {  1 Usage
    println("Calcul en cours...")
    return (1 ≤ .. ≤ 1_000_000).sum()
}

val resultat by lazy { calculCouteux() }  1 Usage

fun main() {
    println("Programme lancé")
    println("Résultat du calcul: $resultat")
}
```

Figure9: Code de l'exercice 5 – fonction de calcul coûteux utilisant lazy

Explication:

Cet exercice simulait un calcul long. L'utilisation de la propriété lazy a permis de différer le calcul jusqu'à ce que le résultat soit nécessaire. Ainsi, nous avons vu l'importance du concept d'évaluation paresseuse en Kotlin.

Résultat:

```
Programme lancé
Calcul en cours...
Résultat du calcul: 1784293664
```

Figure10: Résultat de l'exécution de l'exercice 5

Exercice6: Utilisation de lateinit pour un service

```
class UtilisateurService {  2 Usages
    fun afficherMessage() {  1 Usage
        println("Service utilisateur actif !")
    }
}

class Application {  1 Usage
    lateinit var service: UtilisateurService  3 Usages

    fun initService() {  1 Usage
        service = UtilisateurService()
    }

    fun utiliserService() {  2 Usages
        if (::service.isInitialized) {
            service.afficherMessage()
        } else {
            println("Service non initialisé")
        }
    }
}
```

```
fun main() {
    val app = Application()
    app.utiliserService()
    app.initService()
    app.utiliserService()
}
```

Figure11: Code de l'exercice 6 – implémentation de la classe Application avec lateinit

Explication:

Nous avons créé une classe UtilisateurService et une classe Application qui contient une propriété lateinit. Le service est initialisé dans une méthode spécifique, puis utilisé dans une autre. Cet exercice illustre comment retarder l'initialisation d'un objet jusqu'à un moment choisi.

Résultat:

```
Service non initialisé
Service utilisateur actif !
```

Figure12: Résultat de l'exécution de l'exercice 6

Exercice7: Connexion à une base de données

```
class DatabaseConnection {  2 Usages
    fun connecter() {  1 Usage
        println("Connexion à la base de données établie")
    }
}
```

```
class DatabaseManager {  1 Usage
    lateinit var connection: DatabaseConnection  3 Usages

    fun initConnexion() {  1 Usage
        connection = DatabaseConnection()
        connection.connecter()
    }

    fun executerRequete() {  2 Usages
        if (::connection.isInitialized) {
            println("Exécution de la requête...")
        } else {
            println("Connexion non initialisée")
        }
    }
}

fun main() {
    val manager = DatabaseManager()
    manager.executerRequete()
    manager.initConnexion()
    manager.executerRequete()
}
```

Figure13: Simulation d'une connexion à une base de données

Explication:

Enfin, cet exercice consistait à créer une classe DatabaseConnection et une classe DatabaseManager avec une propriété lateinit pour gérer la connexion. Avant d'exécuter des requêtes, nous vérifions si la connexion a bien été initialisée. Cet exercice illustre la sécurité et le contrôle d'état dans les applications qui interagissent avec une base de données.

Résultat:

```
Connexion non initialisée
Connexion à la base de données établie
Exécution de la requête...
```

- **Figure14:** Résultat de l'exécution de l'exercice 7(avant et après initialisation de la connexion)

Conclusion

À travers ce TP9, j'ai pu explorer plusieurs notions fondamentales du langage Kotlin. Les premiers exercices m'ont permis de manipuler les opérations arithmétiques, les structures conditionnelles et la saisie utilisateur, ce qui a renforcé ma logique de programmation. Ensuite, j'ai découvert la programmation concurrente avec les threads, ce qui m'a aidée à mieux comprendre l'exécution parallèle des tâches.

J'ai également travaillé sur des concepts avancés de Kotlin comme les propriétés lazy et lateinit, qui sont très utiles pour optimiser l'utilisation des ressources et gérer l'initialisation différée ou tardive des objets.

En résumé, ce TP m'a permis de mettre en pratique aussi bien les bases du langage que des fonctionnalités plus avancées. Cela m'a aidée à améliorer mes compétences et à progresser dans la conception de programmes plus efficaces et mieux structurés, utiles dans le développement mobile.