COMPTE RENDU N 1

Nom: Karouit

Prenom: Khadija

Groupe:DEV OM202

Année de formation:2025-2026

Sommaire:

Introduction:

Ce TP a pour but d’explorer les mécanismes d’instanciation différée en Kotlin, à travers l’utilisation des mots-clés lazy et lateinit. Ces outils permettent de retarder la création d’un objet jusqu’à ce qu’il soit réellement nécessaire, ce qui améliore la gestion des ressources et la performance des applications. Les exercices proposés couvrent des cas concrets comme le chargement de services, la simulation de connexions à une base de données, et l’exécution de tâches en parallèle avec des threads.

Exercice 1 – Opérations sur deux nombres

1. **Demander à l'utilisateur de saisir deux nombres.**
2. **Calculer et afficher**

```
package TP9

fun main(){
    print("veuillez entrer a")
    val a =readLine()!!.toInt()
    print("veuillez entrer b")
    val b=readLine()!!.toInt()
    val somme=a + b
    println("somme de a et b est:$somme")
    val soustraction=a-b
    println("la soustratction de et b est:$soustraction")
    val division =a/b
    if(b!=0){
        println("la division de a et  b est :$division")
    }else{
        println("veuillez entrer un nombre valide ")
    }
    val multiplication=a*b
    println("la multipliucation de a et b est:$multiplication")
```

Figure1:Demande et calcule

- ➔ Demande à l'utilisateur d'entrer deux entiers a et b. Lit les valeurs avec readLine()!!.toInt().
- ➔ Calcule et affiche :
- ➔ la somme a + b
- ➔ la soustraction a - b
- ➔ la multiplication a * b
- ➔ la division a / b (seulement si b ≠ 0, sinon affiche un message d'erreur)

```
if(a>b){
    println("$a est superieur a $b")
}else{
    println("$b est superieur a $a")
}
if(somme%2==0){
    println("$somme est paire")
}else{
    println("$somme est impaire")
}
```

Figure2:verification

- ➔ Compare les deux nombres aet b :Affiche lequel est supérieur.
- ➔ Vérifie si la somme de a et b est paire ou impaire :Utilise l'opérateur % pour tester la divisibilité par 2.

```
C:\Users\dell\.jdks\openjdk-24.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\Intel
veuillez entrer a50
veuillez entrer b90
somme de a et b est:140
la soustratction de et b est:-40
la division de a et  b est :0
la multipliucation de a et b est:4500
90 est superieur a 50
140 est paire

Process finished with exit code 0
```

Figure3:affichage

Exercice 2 – Calcul de moyenne et évaluation

1-Demander à l'utilisateur de saisir les notes des trois examens.

```
package TP9

fun main(){
    print("donner note1")
    val note1=readLine()!!.toInt()
    print("donner note2")
    val note2=readLine()!!.toInt()
    print("donner note3")
    val note3=readLine()!!.toInt()
```

Figure4:saisir les notes

On demande à l'utilisateur d'entrer trois notes : `note1`, `note2` et `note3`. On lit chaque note avec `readLine()!!.toInt()` et on les stocke dans des variables.

2-Calculer et afficher la moyenne des trois notes

```
val moyenne=(note1+note2+note3)/3
println("moyenne est :$moyenne")
```

Figure5:la moyenne

4-Afficher un message personnalisé selon la performance

```
if(moyenne>=80){
    println("Reussi avec mention excellente")

}else if(moyenne>=50){
    println("Reussi")

}else{
    println("Echoue")
}
```

Figure6:affichage du message

```
C:\Users\dell\.jdks\openjdk-24.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Editi
donner note117
donner note219
donner note36
moyenne est :14
Echoue

Process finished with exit code 0
```

Figure7:execution

Exercice 3 – Threads avec Runnable

1-Une classe implémente l'interface Runnable

2-La méthode run() affiche un message chaque second

```
class MonThread : Runnable {  3 Usages
    override fun run() {
        repeat( times = 5) { i ->
            println("Message du thread ${Thread.currentThread().name} : ${i + 1}")
            Thread.sleep( millis = 1000)
        }
    }
}
```

Figure8:interface Runnable

- ➔ On crée une classe `MonThread` qui implémente l'interface `Runnable`. On redéfinit la méthode `run()` pour exécuter une tâche répétée 5 fois.
- ➔ À chaque répétition :
- ➔ On affiche le nom du thread et le numéro de l'itération.
- ➔ On met le thread en pause pendant 1 seconde avec `Thread.sleep(1000)`.

3-Plusieurs threads sont créés et démarrés :

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

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

Figure9:Création et démarrage

```
Message du thread Thread-0 : 2
Message du thread Thread-2 : 2
Message du thread Thread-0 : 3
Message du thread Thread-2 : 3
Message du thread Thread-1 : 3
Message du thread Thread-0 : 4
Message du thread Thread-2 : 4
Message du thread Thread-1 : 4
Message du thread Thread-2 : 5
Message du thread Thread-0 : 5
Message du thread Thread-1 : 5

Process finished with exit code 0
```

Figure10:execution

Exercice 4 – Chargement différé avec `lazy`

1-Simulation du chargement d'une configuration complexe

```
class Configuration {  2 Usages
    init {
        println("Chargement de la configuration...")
        Thread.sleep( millis = 2000)
        println("Configuration chargee avec succes.")
    }

    fun utiliser() {  1 Usage
        println("La configuration est maintenant utilisée.")
    }
}
```

Figure11:classe Configuration

- ➔ On crée une classe `Configuration` avec un bloc `init` exécuté automatiquement à l'instanciation. On affiche un message de chargement, puis on attend 2 secondes avec `Thread.sleep(2000)`.
- ➔ Une fois le délai écoulé, on affiche que la configuration est chargée avec succès.
- ➔ La méthode `utiliser()` affiche que la configuration est en cours d'utilisation.

2-Déclaration d'une propriété lazy dans une classe App

Figure12:classe App

- ➔ On crée une classe `App` avec une propriété `config` de type `Configuration`. On utilise `by lazy` pour que l'objet `Configuration` soit créé uniquement au premier accès.
- ➔ La méthode `lancer()` :
- ➔ Affiche "`Application lancée.`"
- ➔ Accède à `config`, ce qui déclenche son initialisation (et exécute le bloc `init` de `Configuration`)
- ➔ Appelle la méthode `utiliser()` sur `config`

3-Initialisation de la connexion

```
fun main() {
    val app = App()
    println("Avant d'utiliser la configuration.")
    app.lancer()
}
```

Figure13:utilisation

4-Exécution:

```
C:\Users\dell\.jdks\openjdk-24.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edit
Avant d'utiliser la configuration.
Application lancée.
Accès à la configuration...
Chargement de la configuration...
Configuration chargee avec succes.
La configuration est maintenant utilisée.

Process finished with exit code 0
```

Figure14:execution

Exercice 5 : Instanciation différée avec lazy

1-Définition d'une fonction calculLourd() simulant un traitement complexe

```
fun calculLourd(): Int {  1 Usage
    println(" Demarrage du calcul couteux...")
    Thread.sleep( millis = 2000)
    println("Calcul termine.")
    return 42
}
```

Figure15:classe calculLourd

- ➔ On lance un calcul simulé avec le message "`Démarrage du calcul coûteux...`". On met le programme en pause pendant 2 secondes avec `Thread.sleep(2000)` pour simuler un traitement long.
- ➔ On affiche "`Calcul terminé.`" une fois le délai écoulé.
- ➔ On retourne la valeur 42 comme résultat du calcul.

2:Déclaration d'une propriété lazy pour stocker le résultat du calcul

```
val resultat: Int by lazy {  1 Usage
    calculLourd()
}
```

Figure16: propriété lazy

On déclare une variable resultat avec by lazy, ce qui signifie que la fonction calculLourd() ne sera exécutée qu'au premier accès à resultat. Cela permet de retarder le calcul coûteux jusqu'à ce qu'il soit vraiment nécessaire.

3-Affichage du résultat uniquement lors du premier accès à la propriété

```kotlin
fun main() {
    println("Programme demarre.")
    println("Le resultat sera affiche maintenant...")


    println("Resultat du calcul : $resultat")
}
```

Figure17: utilisation

4-Exécution

```
C:\Users\dell\.jdks\openjdk-24.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025
Programme demarre.
Le resultat sera affiche maintenant...
 Demarrage du calcul couteux...
Calcul termine.
Resultat du calcul : 42

Process finished with exit code 0
```

Figure18: execution

Exercice 6 : Initialisation tardive avec lateinit

1-La classe `UtilisateurService` simule un service avec un bloc `init`

```kotlin
class UtilisateurService {  2 Usages
    init {
        println( " Initialisation du service utilisateur...")
        Thread.sleep( millis = 1000)
        println("Service utilisateur pret.")
    }

    fun afficherUtilisateur() {  1 Usage
        println(" Utilisateur affiche.")
    }
}
```

Figure19: classe UtilisateurService

- ➔ On crée une classe UtilisateurService qui simule l'initialisation d'un service. Le bloc init s'exécute automatiquement à la création d'un objet :
  - ◆ On affiche "Initialisation du service utilisateur...".
  - ◆ On attend 1 seconde avec Thread.sleep(1000) pour simuler un chargement.
  - ◆ On affiche "Service utilisateur prêt." une fois l'attente terminée.
- ➔ La méthode afficherUtilisateur() affiche "Utilisateur affiché." quand elle est appelée.

2-La classe Application contient une propriété lateinit pour un objet UtilisateurService

```
class Application {  1 Usage

    lateinit var service: UtilisateurService  3 Usages


    fun initialiserService() {  1 Usage
        println(" Initialisation du service...")
        service = UtilisateurService()
    }


    fun utiliserService() {  2 Usages
        if (::service.isInitialized) {
            service.afficherUtilisateur()
        } else {
            println(" Le service n'est pas encore initialise.")
        }
    }
}
```

Figure20: lateinit

- ➔ On déclare une variable `service` avec lateinit, ce qui permet une initialisation tardive. On initialise `service` dans la méthode initialiserService() en créant un objet UtilisateurService.
- ➔ On utilise `::`service.isInitialized pour vérifier si le service est prêt avant de l'utiliser.
- ➔ Si le service est initialisé, on appelle afficherUtilisateur() ; sinon, on affiche un message d'erreur.

3-Utilisation

```kotlin
fun main() {
    val app = Application()

    println(" Avant l'initialisation du service")
    app.utiliserService()

    println(" Initialisation du service maintenant")
    app.initialiserService()

    println("Utilisation du service après initialisation")
    app.utiliserService()
}
```

Figure21: utilisation

4-Exécution

```
C:\Users\dell\.jdks\openjdk-24.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025
Programme demarre.
Le resultat sera affiche maintenant...
 Demarrage du calcul couteux...
Calcul termine.
Resultat du calcul : 42

Process finished with exit code 0
```

Figure22: execution

Exercice 7:Simulation de connexion à une base de données

1-Classe DatabaseConnection

```
package TP9

class DatabaseConnection {  2 Usages
    init {
        println(" Connexion à la base de donnees en cours...")
        Thread.sleep( millis = 1500)
        println(" Connexion etablie.")
    }

    fun executerRequete(sql: String) {  1 Usage
        println(" Requete executee : $sql")
    }
}
```

Figure23: classe DatabaseConnection

➔ La classe DatabaseConnection simule une connexion avec un délai de 1,5 seconde. La méthode executerRequete(`sql`) affiche la requête SQL exécutée.

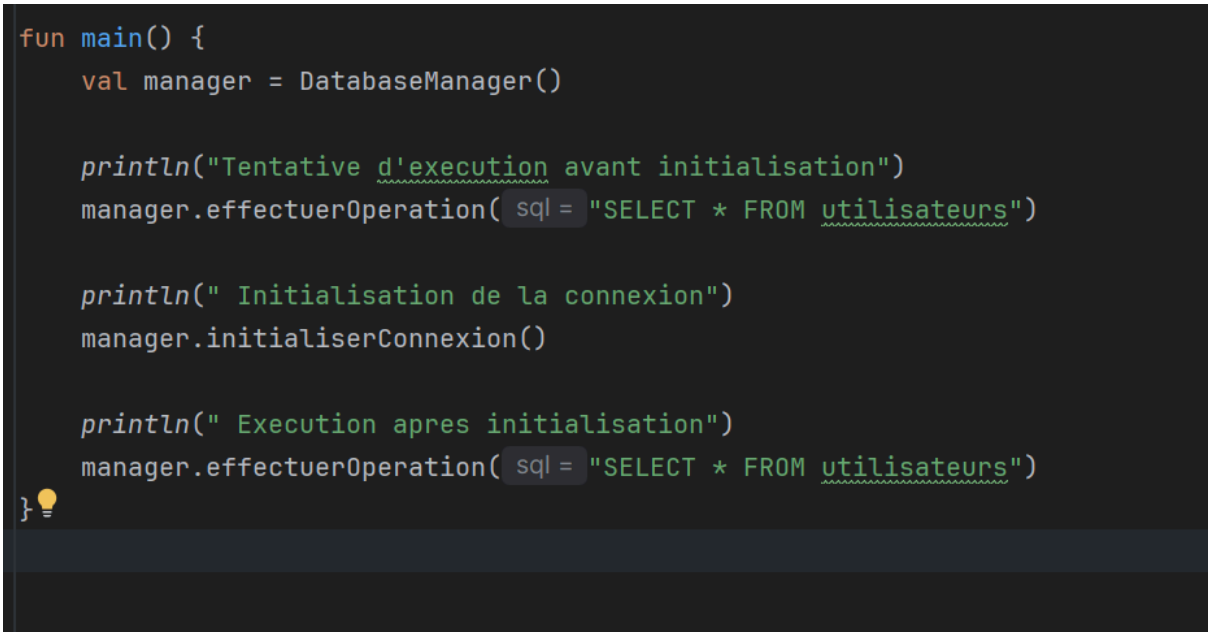
2-Classe DataManager

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

    fun initialiserConnexion() {  1 Usage
        println(" Initialisation du service de base de donnees...")
        connexion = DatabaseConnection()
    }

    fun effectuerOperation(sql: String) {  2 Usages
        if (::connexion.isInitialized) {
            connexion.executerRequete(sql)
        } else {
            println(" Connexion non initialisee. Impossible d'executer la requete.")
        }
    }
}
```

Figure24: classe DataManager

- ➔ La classe DatabaseManager utilise lateinit pour initialiser la connexion plus tard. initialiserConnexion() crée l'objet DatabaseConnection.
- ➔ effectuerOperation(sql) vérifie si la connexion est prête avant d'exécuter la requête SQL.

3-Initialisation:



```
fun main() {
    val manager = DatabaseManager()

    println("Tentative d'execution avant initialisation")
    manager.effectuerOperation( sql = "SELECT * FROM utilisateurs")

    println(" Initialisation de la connexion")
    manager.initialiserConnexion()

    println(" Execution apres initialisation")
    manager.effectuerOperation( sql = "SELECT * FROM utilisateurs")
}
```

Figure25: initialisation

4-Exécution:

```
C:\Users\dell\.jdks\openjdk-24.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025
Tentative d'execution avant initialisation
 Connexion non initialisee. Impossible d'executer la requete.
 Initialisation de la connexion
 Initialisation du service de base de donnees...
 Connexion à la base de donnees en cours...
 Connexion etablie.
 Execution apres initialisation
 Requete executee : SELECT * FROM utilisateurs

Process finished with exit code 0
```

Figure26: execution

CONCLUSION:

Ce TP a permis de comprendre comment Kotlin offre un contrôle précis sur le cycle de vie des objets grâce à lazy et lateinit. En différant l'instanciation, on évite les calculs inutiles et on optimise l'utilisation de la mémoire. Les exercices ont également mis en évidence l'importance de vérifier l'état d'un objet avant son utilisation, ce qui renforce la fiabilité du code. Ces pratiques sont essentielles pour développer des applications mobiles robustes et bien structurées.