Cités des Métiers et des Compétences
La formation nouvelle génération

OFPPT
La voie de l'avenir

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

Sommaire

Introduction

Le présent devoir a pour objectif de consolider les compétences acquises en **programmation orientée objet (POO)** à travers le langage **Kotlin**. Tout au long de ce travail, nous avons mis en œuvre plusieurs concepts essentiels tels que **l'héritage**, **l'encapsulation**, **le polymorphisme**, et **les classes abstraites**. Les deux exercices proposés visent à appliquer ces principes dans des contextes réels:

- la **gestion d'une bibliothèque**, et
- la **gestion d'un parc automobile**.

Ce devoir m'a permis de mieux comprendre la logique des objets, les relations entre les classes et la manière d'organiser un projet orienté objet de manière claire et réutilisable. Il constitue une étape importante dans l'apprentissage du développement mobile, où Kotlin est un langage de base pour la création d'applications Android structurées et maintenables.

Exercice1: Système de gestion de bibliothèque

Question 1 : Création des classes Personne, Utilisateur, Livre et Emprunt

```
open class Personne(val nom: String, val prenom: String, val email: String) {  1 Usage  1 Inheritor
    open fun afficherInfos() {
        println("Nom : $nom, Prénom : $prenom, Email : $email")
    }
}

// Classe Livre
class Livre(val titre: String, val auteur: String, val isbn: String, var nombreExemplaires: Int) {  6 Usages
    fun afficherDetails() {  1 Usage
        println("Livre : $titre | Auteur : $auteur | ISBN : $isbn | Stock : $nombreExemplaires")
    }

    fun disponiblePourEmprunt(): Boolean = nombreExemplaires > 0  1 Usage

    fun mettreAJourStock(nouveauStock: Int) {  2 Usages
        nombreExemplaires = nouveauStock
    }
}
```

Figure 1: Déclaration des classes Personne et Livre

```
// Classe Emprunt
class Emprunt(  2 Usages
    val utilisateur: Utilisateur,
    val livre: Livre,
    val dateEmprunt: String,
    var dateRetour: String? = null
) {
    fun afficherDetails() {  1 Usage
        println("Emprunt : ${livre.titre} par ${utilisateur.prenom} ${utilisateur.nom} le $dateEmprunt")
        println("Date retour : ${dateRetour ?: "Non encore retourné"}")
    }

    fun retournerLivre(dateRetourEffectif: String) {  1 Usage
        dateRetour = dateRetourEffectif
        livre.mettreAJourStock( nouveauStock = livre.nombreExemplaires + 1)
        println("Livre '${livre.titre}' retourné le $dateRetour")
    }
}
```

Figure 2: Déclaration de la classe Emprunt

```
// Classe Utilisateur
class Utilisateur(nom: String, prenom: String, email: String, val idUtilisateur: Int) : 6 Usages
    Personne(nom, prenom, email) {

    val emprunts = mutableListOf<Emprunt>() 3 Usages

    fun emprunterLivre(livre: Livre, dateEmprunt: String) { 2 Usages
        if (livre.disponiblePourEmprunt()) {
            emprunts.add(Emprunt( utilisateur = this, livre, dateEmprunt))
            livre.mettreAJourStock( nouveauStock = livre.nombreExemplaires - 1)
            println("${prenom} ${nom} a emprunté '${livre.titre}'")
        } else {
            println("Le livre '${livre.titre}' n'est pas disponible !")
        }
    }

    fun afficherEmprunts() { 2 Usages
        println("\n Emprunts de $prenom $nom :")
        emprunts.forEach { it.afficherDetails() }
    }
}
```

Figure 3: Déclaration de la classe Utilisateur

Chaque classe possède un **constructeur** pour initialiser ses propriétés, et des **méthodes** pour afficher les informations ou mettre à jour les données. Cette structure met en avant la notion d'**héritage** et la **réutilisation du code**.

- Personne représente les informations d'un individu.
- Utilisateur hérite de Personne et peut effectuer des emprunts.
- Livre contient les données sur les ouvrages et leur disponibilité.
- Emprunt lie un utilisateur à un livre avec des dates précises.

Chaque classe joue un rôle précis et communique avec les autres pour simuler la gestion d'une bibliothèque réelle.

Question 2 : Création de la classe abstraite GestionBibliotheque et de la classe Bibliotheque

```kotlin
// Classe abstraite GestionBibliotheque
abstract class GestionBibliotheque {  1 Usage  1 Implementation
    val utilisateurs = mutableListOf<Utilisateur>()  1 Usage
    val livres = mutableListOf<Livre>()  3 Usages

    abstract fun ajouterUtilisateur(utilisateur: Utilisateur)  2 Usages  1 Implementation
    abstract fun ajouterLivre(livre: Livre)  2 Usages  1 Implementation
    abstract fun afficherTousLesLivres()  1 Usage  1 Implementation
}

// Classe Bibliotheque
class Bibliotheque : GestionBibliotheque() {  1 Usage
    override fun ajouterUtilisateur(utilisateur: Utilisateur) {  2 Usages
        utilisateurs.add(utilisateur)
    }

    override fun ajouterLivre(livre: Livre) {  2 Usages
        livres.add(livre)
    }

    override fun afficherTousLesLivres() {  1 Usage
        println("\n Liste des livres :")
        livres.forEach { it.afficherDetails() }
    }
```

Figure 4: Code de la classe abstraite et de la classe Bibliothèque

```kotlin
    fun rechercherLivreParTitre(titre: String): Livre? {
        return livres.find { it.titre.equals( other = titre, ignoreCase = true) }
    }
}
```

Figure 5: Suite

La classe GestionBibliotheque définit la structure abstraite du système.
La classe Bibliotheque concrétise cette structure avec des méthodes réelles pour gérer la liste des livres et des utilisateurs.
C'est ici que la logique principale de la bibliothèque est centralisée.

Question 3 : Programme principal

```kotlin
fun main() {
    val biblio = Bibliotheque()

    val livre1 = Livre( titre = "Kotlin Débutant", auteur = "Dupont", isbn = "ISBN001", nombreExemplaires = 3)
    val livre2 = Livre( titre = "Android Studio", auteur = "Martin", isbn = "ISBN002", nombreExemplaires = 2)

    val user1 = Utilisateur( nom = "Boulahya", prenom = "Asma", email = "asma@example.com", idUtilisateur = 1)
    val user2 = Utilisateur( nom = "Ali", prenom = "Sara", email = "sara@example.com", idUtilisateur = 2)

    biblio.ajouterLivre(livre1)
    biblio.ajouterLivre(livre2)
    biblio.ajouterUtilisateur( utilisateur = user1)
    biblio.ajouterUtilisateur( utilisateur = user2)

    user1.emprunterLivre(livre1, dateEmprunt = "2025-10-04")
    user2.emprunterLivre(livre2, dateEmprunt = "2025-10-04")

    biblio.afficherTousLesLivres()
    user1.afficherEmprunts()
    user2.afficherEmprunts()

    user1.emprunts[0].retournerLivre( dateRetourEffectif = "2025-10-10")
}
```

Figure 6: Code principal d'exécution de la bibliothèque

Ce programme instancie des objets, ajoute des livres et des utilisateurs, et simule plusieurs emprunts.

Il montre comment les classes interagissent entre elles pour reproduire le fonctionnement d'une vraie bibliothèque.

```
Asma Boulahya a emprunté 'Kotlin Débutant'
Sara Ali a emprunté 'Android Studio'

 Liste des livres :
Livre : Kotlin Débutant | Auteur : Dupont | ISBN : ISBN001 | Stock : 2
Livre : Android Studio | Auteur : Martin | ISBN : ISBN002 | Stock : 1

 Emprunts de Asma Boulahya :
Emprunt : Kotlin Débutant par Asma Boulahya le 2025-10-04
Date retour : Non encore retourné

 Emprunts de Sara Ali :
Emprunt : Android Studio par Sara Ali le 2025-10-04
Date retour : Non encore retourné
Livre 'Kotlin Débutant' retourné le 2025-10-10
```

Figure 7: Résultat d'exécution de l'Exercice 1 — Gestion de la bibliothèque

Exercice2: Système de gestion du parc automobile

Question 1 : Création des classes Vehicule, Voiture et Moto

```kotlin
// Exception personnalisée
class VehiculeIndisponibleException(message: String) : Exception(message)  1 Usage
class VehiculeNonTrouveException(message: String) : Exception(message)  1 Usage

// Classe abstraite Vehicule
abstract class Vehicule(  5 Usages  2 Implementations
    val immatriculation: String,
    val marque: String,
    val modele: String,
    var kilometrage: Int,
    var disponible: Boolean = true
) {
    open fun afficherDetails() {  3 Usages  2 Overrides
        println("[$immatriculation] $marque $modele - $kilometrage km - ${if (disponible) "Disponible" else "Indisponible"}")
    }

    fun estDisponible(): Boolean = disponible  2 Usages
    fun marquerIndisponible() { disponible = false }  1 Usage
    fun marquerDisponible() { disponible = true }  1 Usage
    fun mettreAJourKilometrage(km: Int) { kilometrage = km }  1 Usage
}
```

Figure 8: Code de la classe Véhicule

```
class Voiture(  2 Usages
    immatriculation: String,
    marque: String,
    modele: String,
    kilometrage: Int,
    var nombrePortes: Int,
    var typeCarburant: String
) : Vehicule(immatriculation, marque, modele, kilometrage) {
    override fun afficherDetails() {
        super.afficherDetails()
        println("→ $nombrePortes portes | Carburant : $typeCarburant")
    }
}

// Moto
class Moto(  1 Usage
    immatriculation: String,
    marque: String,
    modele: String,
    kilometrage: Int,
    var cylindree: Int
) : Vehicule(immatriculation, marque, modele, kilometrage) {
    override fun afficherDetails() {
        super.afficherDetails()
        println("→ Cylindrée : $cylindree cm³")
    }
}
```

Figure9: Code des classes Voiture et Moto(suite)

On modélise les véhicules du parc automobile :

- Vehicule est une classe abstraite commune à tous les types de véhicules.
- Voiture et Moto héritent de Vehicule et ajoutent leurs attributs spécifiques. Chaque sous-classe adapte la méthode afficherDetails() selon son type.

Question 2 : Classes Conducteur et Reservation

```
// Conducteur
class Conducteur(val nom: String, val prenom: String, val numeroPermis: String) {  4 Usages
    fun afficherDetails() {
        println("Conducteur : $prenom $nom | Permis : $numeroPermis")
    }
}
```

Figure10: Code de la classe Conducteur

```
// Réservation
class Reservation(  2 Usages
    val vehicule: Vehicule,
    val conducteur: Conducteur,
    val dateDebut: String,
    val dateFin: String,
    val kilometrageDebut: Int,
    var kilometrageFin: Int? = null
) {
    fun cloturerReservation(kilometrageRetour: Int) {
        kilometrageFin = kilometrageRetour
        vehicule.mettreAJourKilometrage( km = kilometrageRetour)
        vehicule.marquerDisponible()
        println("✅ Réservation clôturée pour ${vehicule.marque} - km final : $kilometrageRetour")
    }

    fun afficherDetails() {  1 Usage
        println("🚗 Réservation de ${vehicule.marque} par ${conducteur.prenom} ${conducteur.nom} du $dateDebut au $dateFin")
    }
}
```

Figure11: Code de la classe Reservation(suite)

Ces classes gèrent les conducteurs et les réservations. Chaque réservation associe un conducteur à un véhicule sur une période donnée, avec suivi du kilométrage.

Question 3 : Gestion du parc automobile et des exceptions

```
// Parc Automobile
class ParcAutomobile {  1 Usage
    val vehicules = mutableListOf<Vehicule>()  4 Usages
    val reservations = mutableListOf<Reservation>()  2 Usages

    fun ajouterVehicule(vehicule: Vehicule) = vehicules.add(vehicule)  3 Usages

    fun supprimerVehicule(immatriculation: String) {
        vehicules.removeIf { it.immatriculation == immatriculation }
    }
```

Figure12: Code de la classe ParcAutomobile et exceptions

```
fun reserverVehicule(  2 Usages
    immatriculation: String,
    conducteur: Conducteur,
    dateDebut: String,
    dateFin: String
) {
    val vehicule = vehicules.find { it.immatriculation == immatriculation }
        ?: throw VehiculeNonTrouveException( message = "❌ Véhicule $immatriculation introuvable")

    if (!vehicule.estDisponible())
        throw VehiculeIndisponibleException( message = "🚫 Véhicule $immatriculation déjà réservé")

    vehicule.marquerIndisponible()
    val reservation = Reservation(vehicule, conducteur, dateDebut, dateFin,  kilometrageDebut = vehicule.kilometrage)
    reservations.add(reservation)
    println("✅ Réservation confirmée pour ${conducteur.prenom} ${conducteur.nom}")
}

fun afficherVehiculesDisponibles() {  2 Usages
    println("\n🚗 Véhicules disponibles :")
    vehicules.filter { it.estDisponible() }.forEach { it.afficherDetails() }
}
```

Figure13: Code de la classe ParcAutomobile et exceptions

```
    fun afficherReservations() {  1 Usage
        println("\n📅 Réservations en cours :")
        reservations.forEach { it.afficherDetails() }
    }
}
```

Figure13: Code de la classe ParcAutomobile et exceptions

Cette classe gère tout le parc automobile : ajout, suppression, affichage et réservation. Les exceptions personnalisées permettent une gestion propre des erreurs (véhicule non trouvé ou déjà réservé).

Question 4 : Programme principal

```
fun main() {
    val parc = ParcAutomobile()

    val v1 = Voiture( immatriculation = "A1111", marque = "Toyota", modele = "Yaris", kilometrage = 50000, nombrePortes = 5, typeCarburant = "Essence")
    val v2 = Moto( immatriculation = "M2222", marque = "Yamaha", modele = "R3", kilometrage = 15000, cylindree = 321)
    val v3 = Voiture( immatriculation = "A3333", marque = "Tesla", modele = "Model 3", kilometrage = 20000, nombrePortes = 4, typeCarburant = "Électrique")

    parc.ajouterVehicule( vehicule = v1)
    parc.ajouterVehicule( vehicule = v2)
    parc.ajouterVehicule( vehicule = v3)

    val cond1 = Conducteur( nom = "Boulahya", prenom = "Asma", numeroPermis = "PERM1234")
    val cond2 = Conducteur( nom = "Ali", prenom = "Sara", numeroPermis = "PERM5678")

    parc.afficherVehiculesDisponibles()

    try {
        parc.reserverVehicule( immatriculation = "A1111", conducteur = cond1, dateDebut = "2025-10-04", dateFin = "2025-10-10")
        parc.reserverVehicule( immatriculation = "M2222", conducteur = cond2, dateDebut = "2025-10-05", dateFin = "2025-10-12")
    } catch (e: Exception) {
        println(e.message)
    }

    parc.afficherReservations()
    parc.afficherVehiculesDisponibles()
}
```

Figure14: Exécution du programme principal — Parc automobile

Ce programme crée plusieurs véhicules et conducteurs, puis gère les réservations. Les exceptions permettent de détecter les erreurs en toute sécurité.

```
 Véhicules disponibles :
[A1111] Toyota Yaris - 50000 km - Disponible
→ 5 portes | Carburant : Essence
[M2222] Yamaha R3 - 15000 km - Disponible
→ Cylindrée : 321 cm³
[A3333] Tesla Model 3 - 20000 km - Disponible
→ 4 portes | Carburant : Électrique
 Réservation confirmée pour Asma Boulahya
 Réservation confirmée pour Sara Ali

 Réservations en cours :
 Réservation de Toyota par Asma Boulahya du 2025-10-04 au 2025-10-10
 Réservation de Yamaha par Sara Ali du 2025-10-05 au 2025-10-12

 Véhicules disponibles :
[A3333] Tesla Model 3 - 20000 km - Disponible
→ 4 portes | Carburant : Électrique
```

Figure15: Résultat d'exécution de l'Exercice 2 — Gestion du parc automobile

Conclusion

Ce devoir m'a permis de comprendre en profondeur les bases de la **programmation orientée objet avec Kotlin**.
En réalisant ces deux projets, j'ai appris à organiser le code à l'aide de classes bien structurées, à utiliser l'héritage, les exceptions personnalisées et les objets interconnectés.
Ces notions sont essentielles pour le développement **mobile Android**, où les classes et objets interagissent constamment.

Grâce à cette pratique, je me sens plus à l'aise pour concevoir des projets complets, propres et évolutifs.