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

Compte Rendu N°4

Devoir N°1

Année scolaire :2025/2026

Nom : sifedin

Prenom : Bammarouf

DEVOAM201

Introduction

Dans le cadre du module Kotlin (M202), ce devoir vise à mettre en pratique les concepts fondamentaux de la programmation orientée objet à travers deux projets distincts mais complémentaires. Le premier exercice consiste à développer un système de gestion de bibliothèque permettant de gérer les livres, les utilisateurs et leurs emprunts. Le second exercice porte sur la création d'un système de gestion d'un parc automobile, intégrant la réservation de véhicules par des conducteurs, avec une attention particulière portée à l'héritage, au polymorphisme et à la gestion des exceptions.

Ces deux projets ont été réalisés en langage Kotlin, en respectant les principes de conception orientée objet tels que l'encapsulation, l'abstraction, l'héritage et la modularité. L'objectif principal est de renforcer la compréhension des structures de classes, des relations entre objets, et de la logique métier dans un contexte applicatif réaliste.

Exercice 1 – Gestion de bibliothèque :

Partie 1 :

```
// Classe de base Personne
open class Personne(val nom: String, val prenom: String, val email: String) {  1 Usage  1 Inheritor
    fun afficherInfos() {  2 Usages
        println("Nom: $nom, Prénom: $prenom, Email: $email")
    }
}
```

Figure 1 : Classe `Personne`

Création d'une classe de base `Personne` avec les propriétés `nom`, `prénom`, et `email`, ainsi qu'une méthode `afficherInfos()` pour afficher les informations.

➢ Cette classe sert de fondation pour les utilisateurs, en regroupant les données personnelles communes.

Partie 2 :

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

    val emprunts = mutableListOf<Emprunt>()  3 Usages

    fun emprunterLivre(livre: Livre, dateEmprunt: String) {  2 Usages
        if (livre.disponiblePourEmprunt()) {
            livre.mettreAJourStock( nouveauStock = livre.nombreExemplaires - 1)
            val emprunt = Emprunt( utilisateur = this, livre, dateEmprunt)
            emprunts.add(emprunt)
            println("Livre emprunté avec succès.")
        } else {
            println("Le livre n'est pas disponible.")
        }
    }

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

Figure 2 : Classe `Utilisateur` (hérite de `Personne`)

Ajout de la propriété `idUtilisateur` et d'une liste `emprunts`. J'ai aussi créé les méthodes `emprunterLivre()` et `afficherEmprunts()`.

➢ Elle permet de gérer les emprunts d'un utilisateur et d'hériter des infos de base via `Personne`.

Partie 3 :

```
class Livre(val titre: String, val auteur: String, val isbn: String, var nombreExemplaires: Int) {  8 Usages
    fun afficherDetails() {  1 Usage
        println("Titre: $titre, Auteur: $auteur, ISBN: $isbn, Exemplaires: $nombreExemplaires")
    }

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

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

Figure 3 : Classe `Livre`

Définition des propriétés du livre (`titre`, `auteur`, `isbn`, `nombreExemplaires`) et des méthodes pour afficher les détails, vérifier la disponibilité, et mettre à jour le stock.

➢ Elle représente les livres disponibles dans la bibliothèque et gère leur état.

Partie 4 :

```
class Emprunt(val utilisateur: Utilisateur, val livre: Livre, val dateEmprunt: String) {  2 Usages
    var dateRetour: String? = null  2 Usages

    fun afficherDetails() {  2 Usages
        println("Emprunt - Utilisateur: ${utilisateur.nom}, Livre: ${livre.titre}, Date Emprunt: $dateEmprunt, Da
    }

    fun retournerLivre(date: String) {  1 Usage
        dateRetour = date
        livre.mettreAJourStock( nouveauStock = livre.nombreExemplaires + 1)
    }
```

Figure 4 : Classe `Emprunt`

Création d'une classe qui relie un `Utilisateur` à un `Livre`, avec les dates d'emprunt et de retour. La méthode `retournerLivre()` met à jour le stock et la date de retour.

➢ Elle permet de suivre chaque emprunt individuellement.

Partie 5 :

```
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
}
```

Figure 5 : Classe abstraite `GestionBibliotheque`

Déclaration des listes `utilisateurs` et `livres`, et des méthodes abstraites pour ajouter et afficher.

➢ Elle structure la gestion sans être instanciée directement.

Partie 6 :

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

Figure 6 : Classe `Bibliotheque` (hérite de `GestionBibliotheque`)

Implémentation des méthodes abstraites et ajout de `rechercherLivreParTitre()`.

➢ Elle représente le système complet de la bibliothèque.

Exercice 2 – Gestion de parc automobile :

Partie 1 :

```
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: $immatriculation, Marque: $marque, Modèle: $modele, Km: $kilometrage, Disponib
    }

    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 1 : Classe abstraite `Vehicule`

Création d'une classe abstraite avec les propriétés communes (`immatriculation`, `marque`, `modele`, `kilometrage`, `disponible`) et des méthodes pour gérer l'état du véhicule.

➢ Elle sert de base pour tous les types de véhicules.

Partie 2 :

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

Figure 2 : `Voiture` (hérite de `Vehicule`)

Ajout des propriétés `nombrePortes` et `typeCarburant`, et redéfinition de `afficherDetails()`.

➢ Elle permet de gérer les voitures avec leurs spécificités.

Partie 3 :

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

Figure 3 : Classe `Moto` (hérite de `Vehicule`)

Ajout de la propriété `cylindree` et redéfinition de `afficherDetails()`.

➢ Elle permet de gérer les motos avec leurs caractéristiques propres.

Partie 4 :

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

Figure 4 : Classe `Conducteur`

Création d'une classe simple avec `nom`, `prenom`, et `numeroPermis`, plus une méthode d'affichage.

➢ Elle représente les personnes qui réservent les véhicules.

Partie 5 :

```
class Reservation(  2 Usages
    val vehicule: Vehicule,
    val conducteur: Conducteur,
    val dateDebut: String,
    val dateFin: String,
    val kilometrageDebut: Int
) {
    var kilometrageFin: Int? = null  2 Usages

    fun clotureReservation(kilometrageRetour: Int) {  1 Usage
        kilometrageFin = kilometrageRetour
        vehicule.mettreAJourKilometrage( km = kilometrageRetour)
        vehicule.marquerDisponible()
    }
```

Figure 5 : Classe `Reservation`

Création d'une classe qui relie un `Conducteur` à un `Véhicule`, avec les dates et le kilométrage. La méthode `clotureReservation()` met à jour le véhicule.

➢ Elle permet de suivre les réservations et de gérer les retours.

Partie 6 :

```
class ParcAutomobile {  1 Usage
    val vehicules = mutableListOf<Vehicule>()  4 Usages
    val reservations = mutableListOf<Reservation>()  3 Usages

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

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

    fun reserverVehicule(  4 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 non disponible.")
        }
```

```
        vehicule.marquerIndisponible()
        val reservation = Reservation(vehicule, conducteur, dateDebut, dateFin, kilometrageDebut = vehicule.kilom
        reservations.add(reservation)
        println("Réservation effectuée avec succès.")
    }

    fun afficherVehiculesDisponibles() {  1 Usage
        println("Véhicules disponibles :")
        vehicules.filter { it.estDisponible() }.forEach { it.afficherDetails() }
    }

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

Figure 6 : Classe `ParcAutomobile`

Gestion des listes de véhicules et de réservations, avec des méthodes pour ajouter, supprimer, réserver, et afficher.

➢ Elle centralise la gestion du parc automobile.

Partie 7 :

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

Figure 5 : Exceptions personnalisées

Création de deux classes d'exception : `VehiculeIndisponibleException` et `VehiculeNonTrouveException`.

➢ Elles permettent de gérer les erreurs de réservation de manière propre et contrôlée.

Main :

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

    // Question 1 : Création de véhicules
    val voiture1 = Voiture( immatriculation = "123ABC", marque = "Peugeot", modele = "208", kilometrage = 15000
    val voiture2 = Voiture( immatriculation = "456DEF", marque = "Tesla", modele = "Model 3", kilometrage = 800
    val moto1 = Moto( immatriculation = "789GHI", marque = "Yamaha", modele = "MT-07", kilometrage = 5000, cyl

    parc.ajouterVehicule( vehicule = voiture1)
    parc.ajouterVehicule( vehicule = voiture2)
    parc.ajouterVehicule( vehicule = moto1)

    // Question 2 : Ajout de conducteurs
    val conducteur1 = Conducteur( nom = "Ali", prenom = "Kacem", numeroPermis = "PERM123")
    val conducteur2 = Conducteur( nom = "Nora", prenom = "Bennani", numeroPermis = "PERM456")
```

```
// Question 3 : Réservations
try {
    parc.reserverVehicule( immatriculation = "123ABC", conducteur1, dateDebut = "2025-10-05", dateFin = "2025-1
    parc.reserverVehicule( immatriculation = "789GHI", conducteur2, dateDebut = "2025-10-06", dateFin = "2025-1
} catch (e: Exception) {
    println("Erreur : ${e.message}")
}

// Question 4 : Gestion des exceptions
try {
    parc.reserverVehicule( immatriculation = "123ABC", conducteur2, dateDebut = "2025-10-07", dateFin = "2025-1
} catch (e: Exception) {
    println("Exception capturée : ${e.message}")
}

try {
    parc.reserverVehicule( immatriculation = "000XYZ", conducteur1, dateDebut = "2025-10-08", dateFin = "2025-1
} catch (e: Exception) {
    println("Exception capturée : ${e.message}")
}

// Question 5 : Affichage
parc.afficherVehiculesDisponibles()
parc.afficherReservations()

// Clôture d'une réservation
val reservation = parc.reservations.first()
reservation.cloturerReservation( kilometrageRetour = 15200)
reservation.afficherDetails()
```

Conclusion

À travers la réalisation de ces deux systèmes, nous avons pu consolider notre maîtrise des concepts clés de la programmation orientée objet en Kotlin. Le projet de bibliothèque nous a permis de manipuler des objets liés à la gestion documentaire et aux interactions utilisateur, tandis que le projet de parc automobile a introduit des notions plus avancées telles que le polymorphisme et la gestion des erreurs via des exceptions personnalisées.

Ces exercices ont non seulement renforcé notre capacité à structurer un projet logiciel complet, mais ont également illustré l'importance de la clarté du code, de la réutilisabilité des classes, et de la robustesse des systèmes. Ils constituent une base solide pour aborder des projets plus complexes dans le domaine du développement mobile.