

Développement avec Kotlin: Exercices Pratiques

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1	Configuration de l'environnement	1
2	Compilation et Exécution en ligne de commande	1
2.1	Fichier unique	1
2.2	Compilation avec dépendances	2
2.3	Comment Kotlin compile-t-il ?	2
2.3.a	Structure de compilation	2
2.3.b	Compilation des fichiers	3
2.3.c	Configuration du point d'entrée	3
3	FitTracker App - Exercices Pratiques	3
3.1	1. Introduction aux bases	3
3.2	2. Classes	4
3.3	3. Control Flow	4
3.4	4. Fonctions	5
3.5	5. Collections	6
3.6	6. Scope Functions	6
3.7	7. Null Safety	7
3.8	8. Coroutines	7
3.9	9. Héritage et Interfaces	8
3.10	10. Sealed Class, Gestion des Erreurs	9
3.11	11. Génériques	10
4	Koans Kotlin	11
5	Un vrai projet Kotlin	11
5.1	IntelliJ IDEA	11
5.2	Dans VS Code	11
5.3	Structure de projet	12
6	Rest Client App - Exercices Pratiques	12

1 Configuration de l'environnement

Pour ces exercices, vous pouvez utiliser :

- Kotlin Playground
- IntelliJ IDEA
- Terminal avec SDK :

```
sdk install kotlin
```

2 Compilation et Exécution en ligne de commande

Le plus simple pour commencer est d'utiliser Kotlin Playground

Ensuite, pour compiler et exécuter un fichier Kotlin :

2.1 Fichier unique

i Compilation basique

```
# Compiler un fichier Kotlin
kotlinc MonFichier.kt -d MonFichier.jar

# Exécuter le fichier compilé
java -jar MonFichier.jar
```

2.2 Compilation avec dépendances

⚠️ Projet avec bibliothèques

```
# Compiler avec inclusion des
dépendances runtime
kotlinc      -cp      "lib/*"
MonFichier.kt -include-runtime -
d MonFichier.jar

# Compiler plusieurs fichiers
kotlinc Fichier1.kt Fichier2.kt -
include-runtime -d MonProjet.jar
```

2.3 Comment Kotlin compile-t-il ?

2.3.a Structure de compilation

i 1. Organisation du code

- **Fichiers (.kt)**
 - Peuvent contenir plusieurs classes/fonctions
 - Nom du fichier libre (mais conventionnellement PascalCase)
 - Extension .kt
- **Packages**
 - Déclarés en début de fichier
 - Structurent le namespace
 - Correspondent aux dossiers

💡 2. Point d'entrée

• Fonction `main()`

```
// Version simple
fun main() {
    println("Hello")
}

// Version avec arguments
fun main(args: Array<String>) {
    println("Hello ${args[0]}")
}
```

• Fichier avec plusieurs `main()`

```
// filepath: /home/jovyan/
work/local/src/main/kotlin/
com/example/App.kt
package com.example

// Point d'entrée principal
fun main() {
    println("Main app")
}

// Point d'entrée secondaire
dans un object
object DevTools {
    @JvmStatic
    fun main(args:
Array<String>) {
        println("Dev tools")
    }
}
```

2.3.b Compilation des fichiers

! 3. Processus

```
// filepath: /home/jovyan/
work/local/src/main/kotlin/com/
example/Demo.kt
package com.example

// Une classe par fichier
(recommandé)
class Demo {
    fun hello() = println("Hello")
}

// Plusieurs classes possible
class Helper {
    fun help() = println("Help")
}

// Top-level functions (dans le
package)
fun topLevel() = println("Top
level")
```

Le compilateur : 1. Crée un fichier `.class` pour chaque classe 2. Crée un fichier `*Kt.class` pour les fonctions top-level 3. Ajoute les métadonnées Kotlin.

2.3.c Configuration du point d'entrée

Dans les cas simples, le point d'entrée est la fonction `main()`. Dans les cas plus complexes, il sera défini par Gradle ou Maven.

3 FitTracker App - Exercices Pratiques

3.1 1. Introduction aux bases

i Variables et Types

i Note

En Kotlin, la déclaration des variables se distingue de Java par plusieurs aspects clés. Kotlin utilise les mots-clés `val` pour les variables immuables (similaire à `final` en Java) et `var` pour les variables mutables. Le type est déclaré après le nom de la variable, et grâce à l'inférence de type, il peut souvent être omis. Les constantes sont déclarées avec `const val` au niveau du fichier ou d'un objet companion. Cette approche rend le code plus concis et sûr, tout en maintenant une excellente lisibilité.

Créez un nouveau fichier `BasicExercise.kt` et :

1. Déclarez les variables suivantes pour un exercice :
 - Nom (non modifiable)
 - Durée en minutes (modifiable)
 - Calories brûlées (modifiable)
2. Définissez une constante pour la durée minimale d'entraînement (15 minutes)
3. Utilisez l'inférence de type pour simplifier vos déclarations

3.2 2. Classes

i Note

Les classes en Kotlin offrent une approche moderne et concise de la programmation orientée objet. Contrairement à Java, Kotlin propose des data classes qui automatisent la génération des méthodes utilitaires (`equals`, `hashCode`, `toString`, `copy`), des classes avec constructeur primaire intégré, et une syntaxe simplifiée pour les propriétés. Les classes Kotlin sont `final` par défaut pour favoriser l'encapsulation, et la visibilité par défaut est `public`. L'héritage nécessite le mot-clé `open`, encourageant ainsi une conception réfléchie de la hiérarchie des classes. Les propriétés peuvent être déclarées directement dans le constructeur primaire, réduisant considérablement le boilerplate code par rapport à Java.

i POO

Dans un fichier `Model.kt`, créez les classes suivantes :

1. Une data class `BasicExercise` avec :
 - Type (enum class)
 - Nom
 - Durée
 - Calories
 - Date
2. Une classe `Workout` contenant :
 - Liste d'exercices
 - Méthodes de calcul (durée et calories totales)
3. Une classe `User` avec :
 - Profil utilisateur
 - Historique des workouts

3.3 3. Control Flow

i Note

Kotlin offre des structures de contrôle modernes et expressives qui améliorent la lisibilité et la sécurité du code. Le `when` remplace avantageusement le `switch` de Java avec une syntaxe plus flexible et la possibilité de retourner des valeurs. Les boucles en Kotlin combinent la simplicité des itérateurs avec la puissance des expressions lambda. Les collections peuvent être parcourues de multiples façons, du `for` traditionnel aux fonctions d'ordre supérieur comme `forEach`. L'API des séquences permet une gestion efficace des itérations, particulièrement pour les grandes collections. Cette approche fonctionnelle du contrôle de flux rend le code plus concis et moins sujet aux erreurs.

i Conditions et Boucles

Dans `WorkoutLogic.kt`, implémentez :

Compiler en utilisant `Model.kt` pour les classes de données.

1. Une fonction `calculateIntensity` qui retourne :
 - « Facile » si < 100 calories
 - « Moyen » si entre 100 et 200 calories
 - « Difficile » si > 200 calories
2. Une fonction `categorizeExercise` utilisant `when` pour classifier :
 - Cardio (course, vélo, natation)
 - Force (squat, pompes, tractions)
 - Flexibilité (yoga, étirements)
3. Une fonction calculant les calories totales d'une liste d'entraînements.
4. Parcours de collections Dans `WorkoutLogic.kt`, implémentez différentes façons de parcourir une liste d'exercices :
 - Utilisation d'un `for` traditionnel
 - Utilisation de `forEach`
 - Utilisation de `forEachIndexed`
 - Parcours avec gestion d'index manuels
5. Itérateur personnalisé Créez un itérateur spécialisé dans `WorkoutIterator.kt` qui :
 - Implémente l'interface `Iterator`
 - Filtre les exercices selon leur intensité
 - Permet de parcourir les exercices par type
 - Utilise les séquences Kotlin

3.4 4. Functions

i Note

Les fonctions en Kotlin offrent une flexibilité et une expressivité supérieures à Java. Les extension functions permettent d'ajouter des méthodes à des classes existantes sans les modifier, les paramètres nommés et par défaut réduisent la nécessité de surcharges, et les fonctions d'ordre supérieur permettent de manipuler les fonctions comme des valeurs de première classe. Cette approche fonctionnelle, combinée avec la concision de la syntaxe et l'inférence de type, rend le code plus lisible et plus maintenable tout en réduisant significativement le boilerplate code.

i Fonctions et Extensions

Dans un fichier `Extensions.kt`, créez des fonctions pour implémenter :

1. Une extension function sur `List<Exercise>` calculant la moyenne des calories
2. Une fonction `createExercise` avec des paramètres par défaut pour :
 - Type
 - Durée
 - Intensité
3. Une fonction d'ordre supérieur filtrant les exercices par type. Le critère de filtrage sera une lambda.

3.5 5. Collections

i Note

Les collections en Kotlin offrent une API riche et intuitive, fortement inspirée de la programmation fonctionnelle. Contrairement à Java, Kotlin fait une distinction claire entre les collections mutables et immutables, et propose des opérations de transformation puissantes comme `map`, `filter`, `groupBy` et `maxBy`. Ces opérations peuvent être chaînées de manière fluide et sont optimisées pour la performance. L'API des collections Kotlin simplifie considérablement le traitement des données tout en réduisant les risques d'erreurs grâce à son typage fort et ses opérations null-safe.

i Manipulation de Données

Dans un fichier `DataManipulation.kt`, créez des fonctions pour : Travaillez avec une liste d'exercices pour :

1. Filtrer les exercices > 20 minutes avec `map/filter`
2. Grouper les exercices par type avec `groupBy`
3. Trouver l'exercice le plus intense avec `maxBy`

3.6 6. Scope Functions

i Note

Les scope functions (`let`, `run`, `with`, `apply`, et `also`) sont des fonctions spécifiques à Kotlin qui permettent d'exécuter un bloc de code dans le contexte d'un objet. Chaque scope function a un cas d'utilisation particulier : `with` pour grouper des opérations sur un objet, `let` pour le traitement des nullables, `apply` pour la configuration d'objets, `run` pour la combinaison des deux précédents, et `also` pour les effets secondaires. Ces fonctions améliorent la lisibilité du code en évitant la répétition et en clarifiant l'intention du développeur, tout en maintenant un scope bien défini pour les opérations.

i Contexte et Configuration

Dans un fichier `ScopeFunctions.kt`, créez des scope functions pour :

1. Configurer un nouveau workout avec `with`
2. Traiter un exercice optionnel avec `let`
3. Initialiser un profil utilisateur avec `apply`

3.7 7. Null Safety

i Note

La gestion des valeurs nullables est l'une des fonctionnalités phares de Kotlin, conçue pour éliminer le risque de `NullPointerException`. Contrairement à Java, Kotlin différencie explicitement les types nullables (marqués par `?`) des types non-nullables. Le langage fournit des opérateurs sécurisés comme `?.` (safe call), `?:` (Elvis), et `!!` (non-null assertion) pour manipuler les valeurs potentiellement nulles. Cette approche force les développeurs à gérer explicitement les cas de nullité dès la compilation, rendant le code plus sûr et plus prévisible.

i Gestion des Nullables

Dans un fichier `NullSafety.kt`, créez des fonctions pour :

1. Compléter la classe `Exercise` avec un coach optionnel
2. Une fonction utilisant l'opérateur Elvis pour la durée
3. Une fonction `safe` pour accéder aux détails d'exercice

3.8 8. Coroutines

i Note

Les coroutines sont la solution de Kotlin pour la programmation asynchrone, offrant une alternative élégante aux callbacks et aux promesses. Elles permettent d'écrire du code asynchrone de manière séquentielle, ce qui le rend plus lisible et plus facile à maintenir. Les coroutines utilisent des `suspending functions` qui peuvent être pausées et reprises sans bloquer le thread principal, offrant ainsi une excellente performance et une meilleure utilisation des ressources. Avec les scopes et les dispatchers, Kotlin fournit un contrôle précis sur l'exécution des tâches asynchrones, tout en simplifiant la gestion des erreurs et l'annulation des opérations en cours.

i Asynchrone

Dans un fichier `AsyncTasks.kt`, créez des fonctions pour :

1. Une fonction de chargement asynchrone d'historique. Cette fonction simule un chargement réseau avec un délai de 2 secondes. Exécutez la, puis écrivez une boucle qui affiche « Chargement... » toutes les 500ms jusqu'à ce que le chargement soit terminé.
2. Un compteur de temps pour exercice en cours. Cette fonction simule un exercice en cours avec un délai de 1 seconde. Exécutez la, puis écrivez une boucle qui affiche le temps écoulé toutes les 30 jusqu'à ce que l'exercice soit terminé.
3. Un gestionnaire d'exercices simultanés. Cette fonction simule plusieurs exercices en parallèle en simulant leur durée. Appeler cette fonction dans le programme principal pour afficher les exercices en cours et afficher le nombre de calories brûlées à la fin pour chaque exercice.

3.9 9. Héritage et Interfaces

i Note

L'héritage en Kotlin est plus restrictif qu'en Java pour favoriser une meilleure conception. Les classes sont `final` par défaut et nécessitent le mot-clé `open` pour être héritées. Les interfaces peuvent contenir des implémentations par défaut, et Kotlin supporte la délégation de classe comme alternative à l'héritage. Les propriétés et méthodes doivent également être explicitement marquées comme `open` pour être surchargées, encourageant ainsi une conception plus réfléchie de la hiérarchie des classes. Les membres non abstraits d'une classe abstraite sont `final` par défaut, et les classes internes sont `static` par défaut.

La délégation de classe est une alternative à l'héritage qui permet à une classe d'utiliser les membres d'une autre classe sans hériter de celle-ci. Cela permet de réutiliser du code sans créer de dépendances inutiles entre les classes. Kotlin fournit une syntaxe concise pour la délégation de classe, ce qui rend le code plus lisible et plus maintenable.

i Héritage et Polymorphisme

Dans un fichier `ExerciseHierarchy.kt`, implémentez (dans un sous-package pour éviter les confusions avec les classes précédentes) :

1. Interface de base
 - Créez une interface `Exercise` avec :
 - Une propriété `name` en lecture seule
 - Une propriété `duration` en lecture seule
 - Une méthode `calculateCalories()`
2. Classe abstraite
 - Créez une classe abstraite `AbstractExercise` qui :
 - Implémente l'interface `Exercise`
 - Déclare toutes les propriétés et méthodes comme abstraites
 - Ajoute une méthode `describe()` ouverte à la surcharge
3. Exercices spécifiques
 - Créez deux classes concrètes :
 1. `CardioExercise` avec :
 - Distance parcourue (km)
 - Intensité (1-10)
 - Calcul des calories : `duration * intensity * 3.5`
 2. `StrengthExercise` avec :
 - Nombre de répétitions
 - Poids (kg)
 - Calcul des calories : `duration * weight * 2.1`
4. Système de suivi
 - Créez une interface `ExerciseTracker` avec :
 - `startExercise()`
 - `pauseExercise()`
 - `stopExercise()`
 - `getCurrentStatus()`

5. Définition d'un système de suivi

1. Implémentez les exercices
2. Utilisez les interfaces `AbstractExercise` et `ExerciseTracker`
3. Réalisez les tests
4. Interface le cycle de vie complet des exercices

3.10 10. Sealed Class, Gestion des Erreurs

i Note

Les sealed classes en Kotlin représentent une hiérarchie de classes restreinte où toutes les sous-classes doivent être déclarées dans le même fichier que la classe scellée. C'est une forme de restriction plus souple que `enum` mais plus stricte qu'une classe abstraite classique. Cette fonctionnalité est particulièrement utile pour représenter un ensemble limité d'états ou de types, comme dans le cas de la gestion des résultats d'opérations.

i Note

Kotlin améliore la gestion des erreurs par rapport à Java avec le type `Result`, les expressions `try` et une meilleure gestion des exceptions. Le type `Result` encapsule un succès ou un échec, permettant une gestion plus fonctionnelle des erreurs. Les expressions `try` peuvent retourner une valeur, et les exceptions vérifiées ne sont pas obligatoires. Cette approche rend le code plus sûr et plus expressif tout en réduisant le boilerplate code.

Le `try` en Kotlin est une expression, ce qui signifie qu'il peut retourner une valeur. Cela permet de gérer les erreurs de manière plus concise et expressive. Les exceptions vérifiées ne sont pas obligatoires en Kotlin, ce qui simplifie la gestion des erreurs et améliore la lisibilité du code. Les coroutines offrent également une gestion des erreurs plus efficace, avec des fonctions de sécurité pour les appels asynchrones.

i Exercices de Gestion d'Erreurs

Dans un fichier `ErrorHandling.kt`, implémentez et lisez en détail :

```
package
fr.univtlb.bruno.exercices.basic

import
kotlinx.coroutines.runBlocking

// Sealed class to represent the
// result of an operation, which can
// be either a success or an error
sealed class ExerciseResult<out
T> {
    // Success case holding the
    // data
    data class Success<T>(val data:
T) : ExerciseResult<T>()
    // Error case holding the
    // exception
    data class
Error(val exception: Exception) :
ExerciseResult<Nothing>()
}

// Function to validate
// an exercise, returning an
// ExerciseResult
fun validateExercise(exercise:
Exercise):
ExerciseResult<Exercise> = when {
    exercise.duration < 0 -> ExerciseResult.Error(IllegalArgumentException("Duration
cannot be negative"))
    exercise.name.isBlank() -> ExerciseResult.Error(IllegalArgumentException("Name
cannot be blank"))
    else ->
ExerciseResult.Success(exercise)
}

// Extension function for handling
// success cases
fun <T> ExerciseResult<T>.onSuccess(action:
(T) -> Unit): ExerciseResult<T> {
    if
(this is ExerciseResult.Success)
        action(data)
    return this
}
```

3.11 11. Génériques

i Note

Les génériques en Kotlin améliorent le système de type en permettant d'écrire du code qui fonctionne avec différents types tout en maintenant la sécurité du typage. Kotlin ajoute des fonctionnalités avancées comme la variance déclarée (`out/in`), les contraintes de type (`where`), et les projections de type (`*`). Ces concepts permettent une meilleure flexibilité et sécurité dans la conception des APIs.

i Note

La covariance (`out`) permet à un type générique de préserver la relation d'héritage du type qu'il contient. En d'autres termes, si `B` hérite de `A`, alors `Container<B>` peut être utilisé là où `Container<A>` est attendu.

i Note

La covariance (`in`) permet à un type générique de consommer des valeurs de type générique. En d'autres termes, si `B` hérite de `A`, alors `Container<A>` peut être utilisé là où `Container<B>` est attendu.

i Exercices sur les Génériques

Dans un fichier `Generics.kt`, implémentez :

```
import fr.univtl.bruno.exercices.basic.inheritance.Exercise
import fr.univtl.bruno.exercices.basic.inheritance.CardioExercise
import fr.univtl.bruno.exercices.basic.inheritance.StrengthExercise
```

```
// 1. Container générique avec
// covariance
// T est covariant,
// on peut donc utiliser
// Container<StrengthExercise> là où
// Container<Exercise> est attendu
// Par exemple, si
// Container<StrengthExercise>
// hérite de Container<Exercise>,
// on peut utiliser
// Container<StrengthExercise> là où
// Container<Exercise> est attendu
// Cela permet de récupérer des
// objets de type StrengthExercise
// dans un container de type Exercise
interface Container<out T> {
    fun get(): T
}
```

```
// 2. Logger générique avec
// contravariance
// T est contravariant, on peut
// donc utiliser Logger<Exercise> là
// où Logger<StrengthExercise> est
// attendu
// Par exemple, si
// Logger<Exercise> hérite de
// Logger<StrengthExercise>, on peut
// utiliser Logger<Exercise> là
// où Logger<StrengthExercise> est
// attendu
// Cela permet de logger des
// objets de type Exercise dans un
// logger de type StrengthExercise
interface Logger<in T> {
```

```
    fun log(item: T)
```

```
}
```

```
// 3. Classe de gestion
// d'exercices générique
// T est un type générique qui
```

4 Koans Kotlin

Les Koans Kotlin sont des exercices interactifs conçus pour apprendre les idiomes Kotlin progressivement. Disponibles sur Kotlin Koans Online, ils couvrent les concepts essentiels du langage.

5 Un vrai projet Kotlin

5.1 IntelliJ IDEA

💡 Configuration recommandée

1. Créer un projet Kotlin

- File → New → Project
- Sélectionner « Kotlin »
- Choisir « JVM » et « Gradle »

2. Exécution

- Cliquer sur le bouton ▶ à côté de `main()`
- Ou utiliser `Shift + F10`

5.2 Dans VS Code

💡 Configuration recommandée

1. Installer les extensions :

- « Kotlin » de mathiasfrohlich
- « Kotlin Language » de fwcd
- « Code Runner » de Jun Han

2. Créer un fichier `.kt` et ajouter une fonction `main` :

```
fun main() {
    println("Hello Kotlin!")
}
```

3. Cliquer sur ▶ en haut à droite ou utiliser `Ctrl+Alt+N`

5.3 Structure de projet

! Organisation recommandée

```
monprojet/
├─ src/
│   ├── main/
│   │   └─ kotlin/
│   │       └─ MonFichier.kt
│   └─ test/
│       └─ kotlin/
│           └─ MonFichierTest.kt
└─ build/
    └─ libs/
        └─ monprojet.jar
```

```
implementation("io.ktor:ktor-
serialization-kotlinx-json:2.3.7")
implementation("org.jetbrains.kotlinx:kotlinx-
serialization-json:1.6.0")
implementation("org.jetbrains.kotlinx:kotlinx-
coroutines-core:1.7.3")
}
```

Créer les classes qui représentent les entités dans l'application clientes. La bibliothèque Ktor utilise `kotlinx.serialization` pour la sérialisation/désérialisation des objets JSON.

```
package com.example.api

import
kotlinx.serialization.Serializableizable

@Serializableizable
data class Exercise(
    val id: Int,
    val name: String,
    val duration: Int,
    val calories: Int
)
```

Créer une classe cliente pour effectuer des requêtes HTTP vers l'API REST.

6 Rest Client App - Exercices Pratiques

Créer un projet Kotlin pour une application de gestion de clients REST dans IntelliJ IDEA.

Nous allons utiliser Ktor pour effectuer des requêtes HTTP vers une API REST. Ktor est un framework client et serveur pour créer des applications web et mobiles.

ajouter les dépendances suivantes dans le fichier `build.gradle.kts` :

```
plugins {
    kotlin("jvm") version "1.9.0"
    kotlin("plugin.serialization")
version "1.9.0"
}

dependencies {
    implementation("io.ktor:ktor-
client-core:2.3.7")
    implementation("io.ktor:ktor-
client-cio:2.3.7")
    implementation("io.ktor:ktor-
client-content-negotiation:2.3.7")
}
```

```
package com.example.api

import io.ktor.client.*
import io.ktor.client.engine.cio.*
import io.ktor.client.plugins.contentnegotiation.*
import
io.ktor.serialization.kotlinx.json.*
import io.ktor.client.request.*
import io.ktor.client.statement.*
import io.ktor.http.*

class ExerciseApiClient(private val
baseUrl: String = "http://localhost:
8080") {
```

```

        private val client =
            HttpClient(CIO) {
                install(ContentNegotiation) {
                    json()
                }
            }

            suspend fun getExercises():
            List<Exercise> {
                return client.get("$baseUrl/
                exercises").body()
            }

            suspend fun getExercise(id: Int):
            Exercise {
                return client.get("$baseUrl/
                exercises/$id").body()
            }

            suspend
            fun createExercise(request:
            CreateExerciseRequest): Exercise {
                return client.post("$baseUrl/
                exercises") {
                    contentType(ContentType.Applicati
                    setBody(request)
                }.body()
            }

            suspend fun
            updateExercise(id: Int, request:
            CreateExerciseRequest): Exercise {
                return client.put("$baseUrl/
                exercises/$id") {
                    contentType(ContentType.Applicati
                    setBody(request)
                }.body()
            }

            suspend fun deleteExercise(id:
            Int): Boolean {
                val response
                = client.delete("$baseUrl/exercises/

```

```

                $id")
                return response.status ==
                HttpStatusCode.NoContent
            }

            fun close() {
                client.close()
            }
        }
    }
}

```

```

package com.example

import com.example.api.*
import
kotlinx.coroutines.runBlocking

fun main() = runBlocking {
    val client = ExerciseApiClient()

    try {
        // Créer un exercice
        val newExercise =
        client.createExercise(
            CreateExerciseRequest(
                name = "Course",
                duration = 30,
                calories = 300
            )
        )
        println("Exercice créé:
        $newExercise")

        // Récupérer tous les exercices
        val exercises =
        client.getExercises()
        println("Liste des exercices:
        $exercises")

        // Récupérer un exercice
        spécifique
        val exercise =
        client.getExercise(newExercise.id)
    }
}

```

```

        println("Détail de l'exercice:
$exercice")

        // Mettre à jour un exercice
        val updatedExercise =
client.updateExercise(
    newExercise.id,
    CreateExerciseRequest(
        name = "Course rapide",
        duration = 45,
        calories = 450
    )
)
        println("Exercice mis à jour:
$updatedExercise")

        // Supprimer un exercice
        val deleted = client.deleteExercise(newExercise.id)
        println("Exercice supprimé:
$deleted")

    } catch (e: Exception) {
        println("Erreur: ${e.message}")
    } finally {
        client.close()
    }
}

```