

Mise à niveau : Java et Git

Révision des fondamentaux Java, Maven, Git, JDBC et JUnit

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-05

Table des matières

1 FitTrack Pro - Application de Suivi Sportif	2	4.3 Étape 3.2 : Interface <code>Activite</code>	9
1.1 Compétences visées	2	4.4 Étape 3.3 : Classe abstraite <code>Exercice</code>	9
2 Partie 1 : Configuration de l'environnement	3	4.5 Étape 3.4 : Classes concrètes	10
2.1 Objectifs	3	4.6 Étape 3.5 : Classe <code>Seance</code> refactorisée	11
2.2 Étape 1.1 : Vérification des outils	3	4.7 Étape 3.6 : Tests complets	12
2.3 Étape 1.2 : Création du projet Maven	3	4.8 Étape 3.7 : Commit	13
2.4 Étape 1.3 : Configuration du <code>pom.xml</code>	3	4.9 ☒ Validation Partie 3	14
2.5 Étape 1.4 : Structure du projet	4	5 Partie 4 : Pattern Strategy	14
2.6 Étape 1.5 : Initialisation Git	4	5.1 Objectifs	14
2.7 Étape 1.6 : Connexion à GitHub	5	5.2 Étape 4.1 : Théorie du Pattern Strategy	14
2.8 ☒ Validation Partie 1	5	5.3 Étape 4.2 : Interface <code>CalculStrategy</code>	14
3 Partie 2 : Premier modèle objet simple	5	5.4 Étape 4.3 : Implémentations concrètes	14
3.1 Objectifs	5	5.5 Étape 4.4 : Modification de l'interface <code>Activite</code>	15
3.2 Étape 2.1 : Classe <code>ExerciceSimple</code> ..	5	5.6 Étape 4.5 : Mise à jour des classes d'exercice	16
3.3 Étape 2.2 : Premier test unitaire	6	5.7 Étape 4.6 : Ajout du calcul de calories dans <code>Seance</code>	17
3.4 Étape 2.3 : Exécution des tests	6	5.8 Étape 4.7 : Tests du pattern Strategy	18
3.5 Étape 2.4 : Classe <code>SeanceSimple</code>	7	5.9 Étape 4.8 : Commit	20
3.6 Étape 2.5 : Test de <code>SeanceSimple</code>	8	5.10 ☒ Validation Partie 4	20
3.7 Étape 2.6 : Commit Git	8	6 Partie 5 : Patterns Factory et Builder (1h15)	20
3.8 ☒ Validation Partie 2	9	6.1 Objectifs	20
4 Partie 3 : Introduction aux interfaces	9		
4.1 Objectifs	9		
4.2 Étape 3.1 : Théorie - Pourquoi les interfaces ?	9		

6.2	Étape 5.1 : Factory Pattern	20
6.3	Étape 5.2 : Builder Pattern pour Seance	22
6.4	Étape 5.3 : Tests des patterns Factory et Builder	24
6.5	Étape 5.4 : Application de démonstration	26
6.6	Étape 5.5 : Exécution	27
6.7	Étape 5.6 : Commit	27
6.8	☒ Validation Partie 5	28
7	Partie 6 : Persistance JDBC - Base H2 . . .	28
7.1	Objectifs	28
7.2	Étape 6.1 : Configuration de la base de données	28
7.3	Étape 6.2 : Interface DAO	29
7.4	Étape 6.3 : Implémentation H2SeanceDAO	30
7.5	Étape 6.4 : Tests d'intégration	33
7.6	Étape 6.5 : Exécution des tests . . .	36
7.7	Étape 6.6 : Application complète avec persistance	36
7.8	Étape 6.7 : Commit final	37
7.9	☒ Validation Partie 6	37
8	Partie 7 : Merge et documentation	38
8.1	Objectifs	38
8.2	Étape 7.1 : Préparation du merge . .	38
8.3	Étape 7.2 : Création du README . .	38
8.4	Étape 7.3 : Tag de version	39
8.5	☒ Validation Partie 7	40
9	Partie 8 : Extensions optionnelles (Bonus)	40
9.1	Extension 1 : Rapports et statistiques	40
9.2	Extension 2 : Export CSV	40
9.3	Extension 3 : Interface console interactive	41
10	Conclusion	43
10.1	Prochaines étapes possibles	43

1 FitTrack Pro - Application de Suivi Sportif

Bienvenue dans ce TP progressif de mise à niveau en Java, Maven et Git. Vous allez développer **FitTrack Pro**, une application console de planification et suivi de séances sportives.

FitTrack Pro est une application qui permet de créer des séances d'entraînement en combinant différents exercices (cardio comme la course ou le vélo, et musculation comme les pompes ou squats). Chaque exercice a une durée définie, et l'application calcule le total de calories brûlées selon l'intensité de votre effort. Toutes vos séances sont automatiquement sauvegardées dans une base de données, permettant de consulter l'historique.

Il s'agit d'une projet de base que vous allez construire étape par étape. Vous commencerez par créer les classes de base, puis vous ajouterez des fonctionnalités avancées comme le pattern Strategy pour le calcul des calories, la persistance avec JDBC, et enfin des tests unitaires avec JUnit 5. **Vous pourrez le compléter avec des fonctionnalités supplémentaires selon votre niveau.**

Ce TP est conçu pour être **très progressif** : chaque étape consolide vos acquis avant d'introduire de nouveaux concepts.

1.1 Compétences visées

- ☒ Programmation orientée objet (POO) en Java
- ☒ Gestion de projet avec Maven
- ☒ Versionnage avec Git
- ☒ Tests unitaires avec JUnit 5
- ☒ Persistance de données avec JDBC
- ☒ Design patterns classiques

2 Partie 1 : Configuration de l'environnement

2.1 Objectifs

- Créer un projet Maven
- Initialiser un dépôt Git
- Comprendre la structure d'un projet Java

2.2 Étape 1.1 : Vérification des outils

Vérifiez que vous avez les outils installés :

```
java -version      # Java 11 minimum
                    (17 ou 21 recommandé)
mvn -version       # Maven 3.6+
git --version      # Git 2.0+
```

2.3 Étape 1.2 : Création du projet Maven

Option A : Avec archetype Maven (recommandé)

```
mvn archetype:generate \
  -DgroupId=com.fittrack \
  -DartifactId=fittrack-pro \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DarchetypeVersion=1.4 \
  -DinteractiveMode=false

cd fittrack-pro
```

Option B : Manuellement via votre IDE

- IntelliJ IDEA : File → New → Project → Maven
- Eclipse : File → New → Maven Project

2.4 Étape 1.3 : Configuration du pom.xml

Ouvrez `pom.xml` et relisez et éventuellement adapter la configuration :

Les versions des dépendances peuvent être mises à jour selon vos besoins et vérifier sur Maven Repository.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.fittrack</groupId>
  <artifactId>fittrack-pro</artifactId>
  <version>0.1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>FitTrack Pro</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>
    <!-- JUnit 5 pour les tests -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.9.2</version>
      <scope>test</scope>
    </dependency>
```

```

        <!-- H2 Database (nous
l'utiliserons plus tard) -->
        <dependency>
            <groupId>com.h2database</
groupId>
            <artifactId>h2</artifactId>
            <version>2.1.214</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>

<groupId>org.apache.maven.plugins</
groupId>

                <artifactId>maven-
surefire-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
        </plugins>
    </build>
</project>

```

2.5 Étape 1.4 : Structure du projet

Le projet est organisé en packages selon la convention Maven pour permettre une bonne séparation des responsabilités.

Créez la structure de packages :

```

mkdir -p src/main/java/com/fittrack/
model
mkdir -p src/main/java/com/fittrack/
strategy
mkdir -p src/main/java/com/fittrack/
dao
mkdir -p src/main/java/com/fittrack/
datasource
mkdir -p src/main/resources
mkdir -p src/test/java/com/fittrack

```

Votre structure doit ressembler à :

```

fittrack-pro/
├── pom.xml
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/fittrack/
│   │   │       ├── model/
│   │   │       ├── strategy/
│   │   │       ├── dao/
│   │   │       └── datasource/
│   │   └── resources/
│   └── test/
│       ├── java/
│       └── com/fittrack/

```

2.6 Étape 1.5 : Initialisation Git

Pour suivre l'historique des modifications, initialisez un dépôt Git :

```

# Initialisation du dépôt
git init

# Création du .gitignore
cat > .gitignore << EOF
# Fichiers compilés
target/
*.class

# IDE
.idea/
*.iml
.vscode/
.settings/
.project
.classpath

# OS
.DS_Store
Thumbs.db
EOF

# Premier commit
git add .

```

```
git commit -m "Initial commit: Maven
project setup"

# Création de la branche de feature
git checkout -b feature/domain-model
```

Si elle n'existe pas déjà, créer une classe `App.java` dans `src/main/java/com/fittrack/` qui contient la méthode `main` et affiche un message de bienvenue.

2.7 Étape 1.6 : Connexion à GitHub

```
# Sur GitHub, créez un nouveau
repository "fittrack-pro"
# Puis connectez votre dépôt local :

git remote add origin https://github.
com/VOTRE_USERNAME/fittrack-pro.git
git push -u origin feature/domain-
model
```

2.8 ☒ Validation Partie 1

Vérifiez que :

- ☐ `mvn clean compile` fonctionne sans erreur
- ☐ Votre structure de packages est correcte
- ☐ Git est initialisé avec un commit initial
- ☐ Votre code est poussé sur GitHub

3 Partie 2 : Premier modèle objet simple

3.1 Objectifs

- Créer vos premières classes Java
- Comprendre les getters/setters
- Découvrir les tests unitaires de base

3.2 Étape 2.1 : Classe

ExerciceSimple

Dans `src/main/java/com/fittrack/model/`, créez `ExerciceSimple.java` :

```
package com.fittrack.model;

/**
 * Représente un exercice physique
 * simple avec un nom et une durée.
 */
public class ExerciceSimple {

    private String nom;
    private int duree; // en minutes

    /**
     * Constructeur
     * @param nom le nom de l'exercice
     * (ex: "Course à pied")
     * @param duree la durée en minutes
     */
    public ExerciceSimple(String nom,
int duree) {
        this.nom = nom;
        this.duree = duree;
    }

    // Getters
    public String getNom() {
        return nom;
    }

    public int getDuree() {
        return duree;
    }

    // Setters
    public void setNom(String nom) {
        this.nom = nom;
    }

    public void setDuree(int duree) {
        this.duree = duree;
    }

    /**
     * Affiche les informations de
```

```

l'exercice
    */
    @Override
    public String toString() {
        return nom + " - " + duree +
" min";
    }
}

```

3.3 Étape 2.2 : Premier test unitaire

Un test unitaire permet de vérifier qu'une unité de code (classe ou méthode) fonctionne comme prévu. Pour cela nous allons utiliser JUnit 5, un framework de test populaire en Java. Une classe de Test doit être placée dans le répertoire `src/test/java/` et suivre la même structure de packages que la classe testée. Elle doit également avoir le suffixe `Test` dans son nom. Elle possède des méthodes annotées avec `@Test` qui contiennent les assertions (`assertEquals`, `assertTrue`, etc.) pour vérifier le comportement attendu.

Dans `src/test/java/com/fittrack/`, créez `ExerciceSimpleTest.java` :

```

package com.fittrack;

import
com.fittrack.model.ExerciceSimple;
import org.junit.jupiter.api.Test;
import static
org.junit.jupiter.api.Assertions.*;

/**
 * Tests pour la classe ExerciceSimple
 */
public class ExerciceSimpleTest {

    @Test
    void testCreationExercice() {
        // Arrange (préparation)
        String nom = "Course";

```

```

        int duree = 30;

        // Act (action)
        ExerciceSimple exercice = new
ExerciceSimple(nom, duree);

        // Assert (vérification)
        assertEquals("Course",
exercice.getNom());
        assertEquals(30,
exercice.getDuree());
    }

    @Test
    void testModificationDuree() {
        ExerciceSimple exercice = new
ExerciceSimple("Vélo", 45);

        exercice.setDuree(60);

        assertEquals(60,
exercice.getDuree());
    }

    @Test
    void testToString() {
        ExerciceSimple exercice = new
ExerciceSimple("Natation", 40);

        String resultat =
exercice.toString();

        assertTrue(resultat.contains("Natation"));
        assertTrue(resultat.contains("40"));
    }
}

```

3.4 Étape 2.3 : Exécution des tests

```

# Compiler et tester
mvn clean test

# Vous devriez voir :

```

```
# Tests run: 3, Failures: 0, Errors:
0, Skipped: 0
```

Vous pouvez également exécuter les tests depuis votre IDE (clic droit sur la classe de test → Run).

3.5 Étape 2.4 : Classe **SeanceSimple**

Créez `SeanceSimple.java` dans le package `model` :

```
package com.fittrack.model;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

/**
 * Représente une séance
 * d'entraînement contenant plusieurs
 * exercices
 */
public class SeanceSimple {

    private String nom;
    private LocalDate date;
    private List<ExerciceSimple>
exercices;

    public SeanceSimple(String nom,
LocalDate date) {
        this.nom = nom;
        this.date = date;
        this.exercices = new
ArrayList<>();
    }

    /**
     * Ajoute un exercice à la séance
     */

    public
void ajouterExercice(ExerciceSimple
exercice) {
```

```
        exercices.add(exercice);
    }

    /**
     * Calcule la durée totale de
la séance
     * @return durée totale en minutes
     */
    public int getDureeTotale() {
        int total = 0;
        for (ExerciceSimple ex :
exercices) {
            total += ex.getDuree();
        }
        return total;
    }

    /**
     * Retourne le nombre d'exercices
     */
    public int getNombreExercices() {
        return exercices.size();
    }

    // Getters
    public String getNom() {
        return nom;
    }

    public LocalDate getDate() {
        return date;
    }

    public List<ExerciceSimple>
getExercices() {
        return new
ArrayList<>(exercices); // copie
défensive
    }

    @Override
    public String toString() {
        return String.format("Séance
```

```
'%s' du %s - %d exercices (%d min)",
                                nom,
date,                getNombreExercices(),
getDureeTotale());
    }
}
```

3.6 Étape 2.5 : Test de SeanceSimple

Créez SeanceSimpleTest.java :

```
package com.fittrack;

import
com.fittrack.model.ExerciceSimple;
import
com.fittrack.model.SeanceSimple;
import org.junit.jupiter.api.Test;
import
org.junit.jupiter.api.BeforeEach;
import java.time.LocalDate;

import                                static
org.junit.jupiter.api.Assertions.*;

public class SeanceSimpleTest {

    private SeanceSimple seance;

    @BeforeEach
    void setUp() {
        // Exécuté avant chaque test
        seance =
new SeanceSimple("Séance du matin",
LocalDate.now());
    }

    @Test
    void testSeanceVide() {
        assertEquals(0,
seance.getNombreExercices());
        assertEquals(0,
seance.getDureeTotale());
    }
}
```

```
@Test
void testAjouterUnExercice() {
    ExerciceSimple exercice = new
ExerciceSimple("Pompes", 10);

    seance.ajouterExercice(exercice);

        assertEquals(1,
seance.getNombreExercices());
        assertEquals(10,
seance.getDureeTotale());
    }

    @Test
                                void
testAjouterPlusieursExercices() {
        seance.ajouterExercice(new
ExerciceSimple("Course", 30));
        seance.ajouterExercice(new
ExerciceSimple("Vélo", 45));
        seance.ajouterExercice(new
ExerciceSimple("Natation", 20));

        assertEquals(3,
seance.getNombreExercices());
        assertEquals(95,
seance.getDureeTotale());
    }
}
```

3.7 Étape 2.6 : Commit Git

```
# Vérifier les modifications
git status

# Ajouter les fichiers
git add src/

# Commit avec un message descriptif
git commit -m "feat: add
ExerciceSimple and SeanceSimple
classes with tests"
```

```
# Pousser sur GitHub
git push origin feature/domain-model
```

3.8 ☒ Validation Partie 2

Vérifiez que :

- ☐ Tous les tests passent (`mvn test`)
- ☐ Vous comprenez la structure des tests (Arrange, Act, Assert)
- ☐ Vos commits Git sont explicites
- ☐ Le code compile sans warnings

4 Partie 3 : Introduction aux interfaces

4.1 Objectifs

- Comprendre le concept d'interface
- Implémenter le polymorphisme
- Utiliser les streams Java

4.2 Étape 3.1 : Théorie - Pourquoi les interfaces ?

Problème actuel : Si nous voulons ajouter d'autres types d'activités (étirements, méditation), nous devons modifier `SeanceSimple`.

Solution : Définir un **contrat** (interface) que toutes les activités doivent respecter.

4.3 Étape 3.2 : Interface **Activite**

Créez `Activite.java` dans le package `model` :

```
package com.fittrack.model;

/**
 * Interface définissant le contrat
 * pour toute activité physique
 */
public interface Activite {

    /**
```

```
        * @return le nom de l'activité
        */
        String getNom();

    /**
        * @return la durée en minutes
        */
        int getDuree();

    /**
        * @return une description
        textuelle de l'activité
        */
        String getDescription();
    }
}
```

4.4 Étape 3.3 : Classe abstraite

Exercice

Une classe abstraite permet de factoriser du code commun tout en forçant les sous-classes à implémenter certaines méthodes dont on ne connaît pas l'implémentation à l'avance.

Créez `Exercice.java` :

```
package com.fittrack.model;

/**
 * Classe abstraite représentant un
 * exercice physique.
 * Factorise le code commun à tous
 * les exercices.
 */
public abstract class Exercice
    implements Activite {

    protected String nom;
    protected String description;
    protected int duree;

    public Exercice(String nom, String
        description, int duree) {
        if (nom == null ||
```

```

nom.trim().isEmpty()) {
    throw new
IllegalArgumentException("Le nom ne
peut pas être vide");
}
if (duree <= 0) {
    throw new
IllegalArgumentException("La durée
doit être positive");
}

this.nom = nom;
this.description = description;
this.duree = duree;
}

@Override
public String getNom() {
    return nom;
}

@Override
public int getDuree() {
    return duree;
}

@Override
public String getDescription() {
    return description;
}

@Override
public String toString() {
    return String.format("%s (%d
min): %s", nom, duree, description);
}
}

```

4.5 Étape 3.4 : Classes concrètes

ExerciceCardio.java :

```

package com.fittrack.model;

```

```

/**
 * Exercice de type cardio (course,
vélo, natation...)
 */
public class ExerciceCardio extends
Exercice {

    public ExerciceCardio(String nom,
String description, int duree) {
        super(nom, description, duree);
    }

    /**
     * Méthode spécifique aux exercices
cardio
     */
    public String getType() {
        return "CARDIO";
    }
}

```

ExerciceForce.java :

```

package com.fittrack.model;

/**
 * Exercice de type force
(musculation, pompes, tractions...)
 */
public class ExerciceForce extends
Exercice {

    private int series;
    private int repetitions;

    public ExerciceForce(String nom,
String description, int duree) {
        super(nom, description, duree);
        this.series = 0;
        this.repetitions = 0;
    }

    public ExerciceForce(String nom,

```

```

String description, int duree,
                    int series,
int repetitions) {
    super(nom, description, duree);
    this.series = series;
    this.repetitions = repetitions;
}

public String getType() {
    return "FORCE";
}

public int getSeries() {
    return series;
}

public int getRepetitions() {
    return repetitions;
}

@Override
public String toString() {
    if (series > 0 && repetitions
> 0) {
        return String.format("%s
- %dx%d reps", super.toString(),
                    series,
repetitions);
    }
    return super.toString();
}
}

```

4.6 Étape 3.5 : Classe Seance refactorisée

Créez Seance.java (utilisant l'interface Activite):

```

package com.fittrack.model;

import java.time.LocalDate;
import java.util.ArrayList;

```

```

import java.util.List;

/**
 * Séance d'entraînement composée
d'activités
 */
public class Seance {

    private int id;
    private String nom;
    private LocalDate date;
    private List<Activite> activites;

    public Seance(String nom,
LocalDate date) {
        this.nom = nom;
        this.date = date;
        this.activites = new
ArrayList<>();
    }

    /**
     * Ajoute une activité à la séance
     */
    public void
ajouterActivite(Activite activite) {
        activites.add(activite);
    }

    /**
     * Calcule la durée totale avec
les streams Java 8+
     */
    public int getDureeTotale() {
        return activites.stream()
            .mapToInt(Activite::getDuree)
            .sum();
    }

    public int getNombreActivites() {
        return activites.size();
    }
}

```

```

// Getters/Setters
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getNom() {
    return nom;
}

public LocalDate getDate() {
    return date;
}

    public List<Activite>
getActivites() {
    return new
ArrayList<>(activites);
}

@Override
public String toString() {
    return String.format("Séance
'%s' (%s) - %d activités, %d min",
nom,
date,
getNombreActivites(),
getDureeTotale());
}
}

```

4.7 Étape 3.6 : Tests complets

Créez SeanceTest.java :

```

package com.fittrack;

import com.fittrack.model.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import

```

```

org.junit.jupiter.api.DisplayName;
import java.time.LocalDate;

import static
org.junit.jupiter.api.Assertions.*;

@DisplayName("Tests de la classe
Seance")
public class SeanceTest {

    private Seance seance;

    @BeforeEach
    void setUp() {
        seance = new Seance("Séance
Full Body", LocalDate.now());
    }

    @Test
    @DisplayName("Une séance vide a
une durée de 0")
    void testSeanceVide() {
        assertEquals(0,
seance.getDureeTotale());
        assertEquals(0,
seance.getNombreActivites());
    }

    @Test
    @DisplayName("On peut ajouter
différents types d'activités")
    void testAjouterDifférentsTypesActivites()
    {
        // Polymorphisme : Activite
peut être Cardio ou Force
        Activite cardio = new
ExerciceCardio("Course", "5km", 30);
        Activite force = new
ExerciceForce("Pompes", "4 séries",
15, 4, 20);

        seance.ajouterActivite(cardio);
        seance.ajouterActivite(force);
    }
}

```

```

        assertEquals(2,
seance.getNombreActivites());
        assertEquals(45,
seance.getDureeTotale());
    }

    @Test
    @DisplayName("La durée totale est
correctement calculée")
    void testCalculDureeTotale() {
        seance.ajouterActivite(new
ExerciceCardio("Vélo", "10km", 25));
        seance.ajouterActivite(new
ExerciceCardio("Course", "3km",
20));
        seance.ajouterActivite(new
ExerciceForce("Squats", "Jambes",
10));

        assertEquals(55,
seance.getDureeTotale());
    }
}

```

Test pour ExerciceForce :

```

package com.fittrack;

import
com.fittrack.model.ExerciceForce;
import org.junit.jupiter.api.Test;
import static
org.junit.jupiter.api.Assertions.*;

public class ExerciceForceTest {

    @Test
    void testCreationExerciceForce() {
        ExerciceForce exercice =
new ExerciceForce(
        "Pompes",
        "Pectoraux et triceps",

```

```

        10,
        4,
        15
    );

        assertEquals("Pompes",
exercice.getNom());
        assertEquals(10,
exercice.getDuree());
        assertEquals(4,
exercice.getSeries());
        assertEquals(15,
exercice.getRepetitions());
    }

    @Test
    void testNomVideDeclenche Erreur()
    {
        assertThrows(IllegalArgumentException.class,
() -> {
            new ExerciceForce("",
"Description", 10);
        });
    }

    @Test
    void testDureeNegativeDeclenche
Erreur() {
        assertThrows(IllegalArgumentException.class,
() -> {
            new ExerciceForce("Test",
"Description", -5);
        });
    }
}

```

4.8 Étape 3.7 : Commit

```

git add src/
git commit -m "feat: introduce
Activite interface and exercise
hierarchy

```

- Add Activite interface
- Create abstract Exercice class
- Implement ExerciceCardio and ExerciceForce
- Refactor Seance to use Activite interface
- Add comprehensive tests with validation"

```
git push origin feature/domain-model
```

4.9 ☒ Validation Partie 3

Vérifiez que :

- ☐ Vous comprenez la différence entre interface et classe abstraite
- ☐ Tous les tests passent
- ☐ Vous comprenez le polymorphisme (`Activite a = new ExerciceCardio(...)`)
- ☐ La validation des données fonctionne (exceptions)

5 Partie 4 : Pattern Strategy

5.1 Objectifs

- Implémenter le pattern Strategy
- Comprendre l'injection de dépendance
- Calculer les calories selon différentes intensités

5.2 Étape 4.1 : Théorie du Pattern Strategy

Problème : Le calcul des calories dépend de l'intensité de l'exercice, et cette intensité peut varier.

Solution : Extraire l'algorithme de calcul dans une stratégie interchangeable.

5.3 Étape 4.2 : Interface CalculStrategy

Dans le package strategy, créez `CalculStrategy.java` :

```
package com.fittrack.strategy;

/**
 * Stratégie de calcul des calories
 * Basée sur le MET (Metabolic Equivalent of Task)
 */
public interface CalculStrategy {

    /**
     * @return le coefficient MET pour
     * cette intensité
     */
    double getMet();

    /**
     * @return le nom de la stratégie
     */
    String getNom();
}
```

5.4 Étape 4.3 : Implémentations concrètes

`IntensiteFaible.java` :

```
package com.fittrack.strategy;

public class IntensiteFaible
implements CalculStrategy {

    @Override
    public double getMet() {
        return 3.5; // Marche lente,
        yoga doux
    }

    @Override
```

```

    public String getNom() {
        return "Intensité faible";
    }
}

```

IntensiteModeree.java :

```

package com.fittrack.strategy;

public class IntensiteModeree
implements CalculStrategy {

    @Override
    public double getMet() {
        return 6.0; // Jogging,
vélo modéré
    }

    @Override
    public String getNom() {
        return "Intensité modérée";
    }
}

```

IntensiteHaute.java :

```

package com.fittrack.strategy;

public class IntensiteHaute
implements CalculStrategy {

    @Override
    public double getMet() {
        return 8.5; // Course rapide,
HIIT
    }

    @Override
    public String getNom() {
        return "Intensité haute";
    }
}

```

IntensiteMaximale.java :

```

package com.fittrack.strategy;

public class IntensiteMaximale
implements CalculStrategy {

    @Override
    public double getMet() {
        return 12.0; // Sprint,
CrossFit intense
    }

    @Override
    public String getNom() {
        return "Intensité maximale";
    }
}

```

5.5 Étape 4.4 : Modification de l'interface Activite

Modifiez `Activite.java` pour ajouter le calcul de calories :

```

package com.fittrack.model;

import com.fittrack.strategy.CalculStrategy;

public interface Activite {

    String getNom();
    int getDuree();
    String getDescription();

    /**
     * Calcule les calories brûlées
     selon une stratégie donnée
     * @param strategy la stratégie
     de calcul
     * @return nombre de calories
     estimées
     */
    double getCalories(CalculStrategy

```

```
strategy);
}
```

5.6 Étape 4.5 : Mise à jour des classes d'exercice

Exercice.java (classe abstraite) :

```
package com.fittrack.model;

import com.fittrack.strategy.CalculStrategy;

public abstract class Exercice
implements Activite {

    protected String nom;
    protected String description;
    protected int duree;

    public Exercice(String nom, String
description, int duree) {
        if (nom == null ||
nom.trim().isEmpty()) {
            throw new
IllegalArgumentException("Le nom ne
peut pas être vide");
        }
        if (duree <= 0) {
            throw new
IllegalArgumentException("La durée
doit être positive");
        }

        this.nom = nom;
        this.description = description;
        this.duree = duree;
    }

    @Override
    public String getNom() {
        return nom;
    }
}
```

```
@Override
public int getDuree() {
    return duree;
}

@Override
public String getDescription() {
    return description;
}

/**
 * Méthode abstraite : chaque
type d'exercice calcule ses calories
différemment
 */
@Override
    public abstract
double getCalories(CalculStrategy
strategy);

@Override
    public String toString() {
        return String.format("%s (%d
min): %s", nom, duree, description);
    }
}
```

ExerciceCardio.java :

```
package com.fittrack.model;

import com.fittrack.strategy.CalculStrategy;

public class ExerciceCardio extends
Exercice {

    public ExerciceCardio(String nom,
String description, int duree) {
        super(nom, description, duree);
    }

    @Override
        public double
```

```

getCalories(CalculStrategy strategy)
{
    // Formule simple : durée
    × MET
    return duree *
strategy.getMet();
}

public String getType() {
    return "CARDIO";
}
}

```

ExerciceForce.java :

```

package com.fittrack.model;

import com.fittrack.strategy.CalculStrategy;

public class ExerciceForce extends
Exercice {

    private int series;
    private int repetitions;

    public ExerciceForce(String nom,
String description, int duree) {
        super(nom, description, duree);
        this.series = 0;
        this.repetitions = 0;
    }

    public ExerciceForce(String nom,
String description, int duree,
int series,
int repetitions) {
        super(nom, description, duree);
        this.series = series;
        this.repetitions = repetitions;
    }

    @Override
    public double

```

```

getCalories(CalculStrategy strategy)
{
    // Les exercices de force
    brûlent 20% de calories en plus
    return duree *
strategy.getMet() * 1.2;
}

public String getType() {
    return "FORCE";
}

public int getSeries() {
    return series;
}

public int getRepetitions() {
    return repetitions;
}

@Override
public String toString() {
    if (series > 0 && repetitions
> 0) {
        return String.format("%s
- %dx%d reps", super.toString(),
series,
repetitions);
    }
    return super.toString();
}
}

```

5.7 Étape 4.6 : Ajout du calcul de calories dans Seance

Modifiez Seance.java :

```

package com.fittrack.model;

import com.fittrack.strategy.CalculStrategy;
import java.time.LocalDate;
import java.util.ArrayList;

```

```

import java.util.List;

public class Seance {

    private int id;
    private String nom;
    private LocalDate date;
    private List<Activite> activites;

    public Seance(String nom,
LocalDate date) {
        this.nom = nom;
        this.date = date;
        this.activites = new
ArrayList<>();
    }

    public void
ajouterActivite(Activite activite) {
        activites.add(activite);
    }

    public int getDureeTotale() {
        return activites.stream()
            .mapToInt(Activite::getDur
                .sum();
    }

    /**
     * Calcule les calories totales
     selon une stratégie
     */
    public double
getCaloriesTotales(CalculStrategy
strategy) {
        return activites.stream()
            .mapToDouble(a
-> a.getCalories(strategy))
                .sum();
    }

    public int getNombreActivites() {
        return activites.size();
    }

```

```

    }

    // Getters/Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
    }

    public LocalDate getDate() {
        return date;
    }

    public List<Activite>
getActivites() {
        return new
ArrayList<>(activites);
    }

    @Override
    public String toString() {
        return String.format("Séance
'%s' (%s) - %d activités, %d min",
                                nom,
                                date,
                                getNombreActivites(),
                                getDureeTotale());
    }
}

```

5.8 Étape 4.7 : Tests du pattern Strategy

Créez CalculStrategyTest.java :

```

package com.fittrack;

import com.fittrack.model.*;

```

```

import com.fittrack.strategy.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;
import java.time.LocalDate;

import static org.junit.jupiter.api.Assertions.*;

@DisplayName("Tests du pattern Strategy pour le calcul des calories")
public class CalculStrategyTest {

    @Test
    @DisplayName("Cardio avec intensité modérée calcule correctement")
    void testCardioIntensiteModeree() {
        ExerciceCardio course = new ExerciceCardio("Course", "5km", 30);
        CalculStrategy strategie = new IntensiteModeree();

        double calories = course.getCalories(strategie);

        // 30 min × 6.0 MET = 180 calories
        assertEquals(180.0, calories, 0.01);
    }

    @Test
    @DisplayName("Force brûle 20% de calories en plus")
    void testForceAvecBonus() {
        ExerciceForce pompes = new ExerciceForce("Pompes", "Pectoraux", 10);

        CalculStrategy strategie = new IntensiteModeree();

```

```

        double calories = pompes.getCalories(strategie);

        // 10 min × 6.0 MET × 1.2 = 72 calories
        assertEquals(72.0, calories, 0.01);
    }

    @Test
    @DisplayName("Même exercice, stratégies différentes, résultats différents")
    void testDifferentesStrategies() {
        ExerciceCardio velo = new ExerciceCardio("Vélo", "Route", 45);

        double caloriesFaible = velo.getCalories(new IntensiteFaible());
        double caloriesModeree = velo.getCalories(new IntensiteModeree());
        double caloriesHaute = velo.getCalories(new IntensiteHaute());
        double caloriesMax = velo.getCalories(new IntensiteMaximale());

        // Vérifie que plus l'intensité augmente, plus les calories augmentent
        assertTrue(caloriesFaible < caloriesModeree);
        assertTrue(caloriesModeree < caloriesHaute);
        assertTrue(caloriesHaute < caloriesMax);
    }

    @Test

```

```

    @DisplayName("Calcul des calories
    totales d'une séance")
    void testCaloriesTotalesSeance() {
        Seance seance
        = new Seance("Morning Workout",
        LocalDate.now());
        seance.ajouterActivite(new
        ExerciceCardio("Course", "3km",
        20));
        seance.ajouterActivite(new
        ExerciceForce("Squats", "Jambes",
        15));
        seance.ajouterActivite(new
        ExerciceCardio("Vélo", "Cool down",
        10));

        CalculStrategy strategie =
        new IntensiteHaute();

        double total = seance.getCaloriesTotales(strategy);

        // Course: 20×8.5 = 170
        // Squats: 15×8.5×1.2 = 153
        // Vélo: 10×8.5 = 85
        // Total = 408
        assertEquals(408.0, total,
        0.01);
    }
}

```

5.9 Étape 4.8 : Commit

```

git add src/
git commit -m "feat: implement
Strategy pattern for calories
calculation

- Add CalculStrategy interface
- Implement 4 intensity levels
(Faible, Moderee, Haute, Maximale)
- Update Activite interface with
getCalories method
- Add calorie calculation in Exercice

```

```

subclasses
- Add getCaloriesTotales in Seance
- Comprehensive tests for strategy
pattern"

git push origin feature/domain-model

```

5.10 ☒ Validation Partie 4

Vérifiez que :

- ☐ Vous comprenez le pattern Strategy
- ☐ Vous pouvez changer de stratégie au run-time
- ☐ Les tests de calories passent
- ☐ Vous comprenez l'avantage (extensibilité)

6 Partie 5 : Patterns Factory et Builder (1h15)

6.1 Objectifs

- Simplifier la création d'objets avec Factory
- Construire des objets complexes avec Builder
- Améliorer la lisibilité du code

6.2 Étape 5.1 : Factory Pattern

Le pattern Factory permet de centraliser la création d'objets, facilitant ainsi la maintenance et l'extensibilité du code. Il est particulièrement utile lorsque la création d'objets est complexe ou dépend de conditions spécifiques. Il peut s'agir d'une simple méthode statique ou d'une classe dédiée. Il est aussi d'obtenir des sous-classes spécifiques sans exposer la logique de création au client.

Créez `ActiviteFactory.java` dans le package `model` :

```
package com.fittrack.model;
```

```

import java.util.Map;

/**
 * Factory pour créer des activités
 * de manière centralisée
 */
public class ActiviteFactory {

    /**
     * Crée une activité selon le
     * type spécifié
     *
     * @param type "cardio" ou "force"
     * @param params map contenant
     * nom, description, duree, series,
     * repetitions
     * @return une instance d'Activite
     */

    public static Activite createActivite(String type,
        Map<String, Object> params) {
        String nom = (String)
        params.get("nom");
        String description = (String)
        params.getOrDefault("description",
        "");
        int duree = (int)
        params.get("duree");

        switch (type.toLowerCase()) {
            case "cardio":
                return
                new ExerciceCardio(nom, description,
                duree);

            case "force":
                if
                (params.containsKey("series") &&
                params.containsKey("repetitions")) {
                    int series =
                    (int) params.get("series");
                    int repetitions
                    = (int) params.get("repetitions");

```

```

                return
                new ExerciceForce(nom, description,
                duree, series, repetitions);
            } else {
                return
                new ExerciceForce(nom, description,
                duree);
            }

            default:
                throw
                new IllegalArgumentException("Type
                inconnu: " + type);
        }
    }

    /**
     * Méthodes de convenance pour
     * créer rapidement des activités
     */

    public static Activite
    createCardio(String nom, String
    description, int duree) {
        return new ExerciceCardio(nom,
        description, duree);
    }

    public static Activite
    createForce(String nom, String
    description, int duree,
        int
        series, int repetitions) {
        return
        new ExerciceForce(nom, description,
        duree, series, repetitions);
    }
}

```

Vous pouvez maintenant créer des activités de manière centralisée :

```

Map<String, Object> params = new
HashMap<>();

```

```

params.put("nom", "Course");
params.put("description", "5km");
params.put("duree", 30);
Activite cardio = ActiviteFactory.createAc
(params);

```

6.3 Étape 5.2 : Builder Pattern pour Seance

Le pattern Builder est utile pour construire des objets complexes étape par étape. Il s'agit de séparer la construction d'un objet de sa représentation, permettant ainsi de créer différents types et représentations d'un objet en utilisant le même processus de construction.

Modifiez `Seance.java` pour ajouter un Builder interne :

```

package com.fittrack.model;

import com.fittrack.strategy.CalculStrategie;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

public class Seance {

    private int id;
    private String nom;
    private LocalDate date;
    private List<Activite> activites;

    // Constructeur privé pour forcer
    l'utilisation du Builder
    private Seance(Builder builder) {
        this.nom = builder.nom;
        this.date = builder.date;
        this.activites = new
        ArrayList<>(builder.activites);
    }

    // Constructeur public
    pour compatibilité avec les tests

```

```

existants

    public Seance(String nom,
LocalDate date) {
        this.nom = nom;
        this.date = date;
        this.activites = new
        ArrayList<>();
    }

    public void
ajouterActivite(Activite activite) {
        activites.add(activite);
    }

    public int getDureeTotale() {
        return activites.stream()
            .mapToInt(Activite::getDuree)
            .sum();
    }

    public double
getCaloriesTotales(CalculStrategy
strategy) {
        return activites.stream()
            .mapToDouble(a
-> a.getCalories(strategy))
            .sum();
    }

    public int getNombreActivites() {
        return activites.size();
    }

    // Getters/Setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNom() {

```

```

        return nom;
    }

    public LocalDate getDate() {
        return date;
    }

    public List<Activite>
getActivites() {
        return new
ArrayList<>(activites);
    }

    @Override
    public String toString() {
        return String.format("Séance
'%s' (%s) - %d activités, %d min",
                                nom,
date,        getNombreActivites(),
getDureeTotale());
    }

    /**
     * Builder pour construire une
Seance de manière fluide
     */
    public static class Builder {
        private String nom;
        private LocalDate date;
        private List<Activite>
activites = new ArrayList<>();

        public Builder setNom(String
nom) {
            this.nom = nom;
            return this;
        }

        public Builder
setDate(LocalDate date) {
            this.date = date;
            return this;
        }
    }

```

```

        public Builder
addActivite(Activite activite) {
            this.activites.add(activite);
            return this;
        }

        public
Builder addActivites(List<Activite>
activites) {
            this.activites.addAll(activites);
            return this;
        }

        public Seance build() {
            if (nom == null ||
nom.trim().isEmpty()) {
                throw new
IllegalStateException("Le nom de la
séance est requis");
            }
            if (date == null) {
                throw new
IllegalStateException("La date est
requis");
            }
            return new Seance(this);
        }
    }
}

```

Vous pouvez maintenant créer des séances de manière fluide et lisible :

```

Seance seance = new Seance.Builder()
    .setNom("Séance HIIT")
    .setDate(LocalDate.now())
    .addActivite(new
ExerciceCardio("Burpees", "Full
body", 3))
    .addActivite(new
ExerciceCardio("Mountain climbers",

```

```

        "Core", 3))
            .addActivite(new
ExerciceForce("Pompes", "Pectoraux",
2, 3, 15))
            .addActivite(new
ExerciceCardio("Jumping jacks",
"Cardio", 2))
            .build();

```

6.4 Étape 5.3 : Tests des patterns Factory et Builder

Créez FactoryBuilderTest.java :

```

package com.fittrack;

import com.fittrack.model.*;
import com.fittrack.strategy.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;
import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;

import static org.junit.jupiter.api.Assertions.*;

@DisplayName("Tests des patterns  
Factory et Builder")
public class FactoryBuilderTest {

    @Test
    @DisplayName("Factory crée un  
exercice cardio")
    void testFactoryCardio() {
        Map<String, Object> params =
new HashMap<>();
        params.put("nom", "Natation");
        params.put("description",
"Crawl");
        params.put("duree", 40);

```

```

        Activite activite = ActiviteFactory.createActivite
params);

        assertNotNull(activite);
        assertTrue(activite instanceof
ExerciceCardio);
        assertEquals("Natation",
activite.getNom());
        assertEquals(40,
activite.getDuree());
    }

    @Test
    @DisplayName("Factory crée un  
exercice de force avec séries")
    void testFactoryForceAvecSeries()
    {
        Map<String, Object> params =
new HashMap<>();
        params.put("nom", "Développé
couché");
        params.put("description",
"Pectoraux");
        params.put("duree", 20);
        params.put("series", 5);
        params.put("repetitions", 8);

        Activite activite = ActiviteFactory.createActivite
params);

        assertTrue(activite instanceof
ExerciceForce);
        ExerciceForce force =
(ExerciceForce) activite;
        assertEquals(5,
force.getSeries());
        assertEquals(8,
force.getRepetitions());
    }

    @Test
    @DisplayName("Factory lance une  
exception pour type inconnu")

```

```

    void testFactoryTypeInconnu() {
        Map<String, Object> params =
new HashMap<>();
        params.put("nom", "Test");
        params.put("duree", 10);

        assertThrows(IllegalArgumentException.class,
() -> {
            ActiviteFactory.createActivite("y",
params);
        });
    }

    @Test
    @DisplayName("Builder construit
une séance complète")
    void testBuilderSeanceComplete() {
        Seance seance = new
Seance.Builder()
            .setNom("Full Body Workout")
            .setDate(LocalDate.of(2026,
1, 25))
            .addActivite(ActiviteFactory.crea
"Tapis", 10))
            .addActivite(ActiviteFactory.crea
"Jambes", 15, 4, 12))
            .addActivite(ActiviteFactory.crea
"Endurance", 20))
            .build();

        assertEquals("Full Body
Workout", seance.getNom());
        assertEquals(3,
seance.getNombreActivites());
        assertEquals(45,
seance.getDureeTotale());
    }

    @Test
    @DisplayName("Builder vérifie que
le nom est requis")
    void testBuilderNomRequis() {
        assertThrows(IllegalStateException.class,
() -> {
            new Seance.Builder()
                .setDate(LocalDate.now())
                .build();
        });
    }

    @Test
    @DisplayName("Builder vérifie que
la date est requise")
    void testBuilderDateRequise() {
        assertThrows(IllegalStateException.class,
() -> {
            new Seance.Builder()
                .setNom("Test")
                .build();
        });
    }

    @Test
    @DisplayName("Exemple
d'utilisation fluide du Builder")
    void testBuilderFluent() {
        // Démonstration de la syntaxe
        fluide et lisible

        Seance seance = new
Seance.Builder()
            .setNom("Séance HIIT")
            .setDate(LocalDate.now())
            .addActivite(new
ExerciceCardio("Burpees", "Full
body", 3))
            .addActivite(new
ExerciceCardio("Mountain climbers",
"Core", 3))
            .addActivite(new
ExerciceForce("Pompes", "Pectoraux",
2, 3, 15))
            .addActivite(new
ExerciceCardio("Jumping jacks",
"Cardio", 2))
            .build();
    }

```

```

        CalculStrategy intensiteMax
= new IntensiteMaximale();

assertTrue(seance.getDureeTotale() >
0);
        assertTrue(seance.getCaloriesTotales
> 0);
    }
}

```

6.5 Étape 5.4 : Application de démonstration

Il est temps de montrer l'utilisation des patterns Factory et Builder dans une application principale.

Créez Main.java dans le package racine com.fittrack :

```

package com.fittrack;

import com.fittrack.model.*;
import com.fittrack.strategy.*;
import java.time.LocalDate;

/**
 * Application de démonstration des
 * fonctionnalités
 */
public class Main {

    public static void main(String[]
args) {

        System.out.println("===
FitTrack Pro - Démonstration ===\n");

        // Création d'une séance avec
le Builder

        Seance seance = new
Seance.Builder()
            .setNom("Séance du matin")
            .setDate(LocalDate.now())

```

```

            .addActivite(ActiviteFactory.createCardio(
                "Échauffement",
                "Jogging léger",
                10
            ))
            .addActivite(ActiviteFactory.createForce(
                "Squats",
                "Jambes et fessiers",
                15,
                4,
                12
            ))
            .addActivite(ActiviteFactory.createCardio(
                "Course",
                "5km allure modérée",
                30
            ))
            .addActivite(ActiviteFactory.createForce(
                "Pompes",
                "Pectoraux et triceps",
                10,
                3,
                20
            ))
            .build();

        System.out.println(seance);
        System.out.println();

        // Affichage des activités
        System.out.println("Détail
des activités :");
        for (Activite activite :
seance.getActivites()) {
            System.out.println(" -
" + activite);
        }
        System.out.println();

        // Calcul des calories selon
différentes intensités
        CalculStrategy[] strategies
= {

```

```

        new IntensiveFaible(),
        new IntensiveModeree(),
        new IntensiveHaute(),
        new IntensiveMaximale()
    };

    System.out.println("Estimation
des calories brûlées :");
    for (CalculStrategy strategy :
strategies) {
        double calories =
seance.getCaloriesTotales(strategy);
        System.out.printf(" -
%s : %.0f cal\n",
            strategy.getNom(),
calories);
    }

    System.out.println("\n=== Fin
de la démonstration ===");
}
}

```

6.6 Étape 5.5 : Exécution

La compilation et l'exécution de l'application principale peuvent être faites via Maven en ligne de commande ou depuis l'IDE. Il est essentiel de pouvoir compiler et exécuter indépendamment de l'IDE.

```

# Compiler
mvn clean compile

# Exécuter
mvn                                exec:java
-Dexec.mainClass="com.fittrack.Main"

# Ou si vous ajoutez le plugin exec
dans pom.xml :

```

Ajoutez dans pom.xml :

```

<build>
  <plugins>
    <plugin>

    <groupId>org.apache.maven.plugins</
groupId>
      <artifactId>maven-surefire-
plugin</artifactId>
      <version>2.22.2</version>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</
groupId>
      <artifactId>exec-maven-
plugin</artifactId>
      <version>3.1.0</version>
      <configuration>

      <mainClass>com.fittrack.Main</
mainClass>

      </configuration>
    </plugin>
  </plugins>
</build>

```

Pour aller plus loin, il est très utile de configurer des plugins dans Maven pour pouvoir produire des JAR exécutables de vos classes, un répertoire contenant toutes les dépendances, générer de la documentation, ...

6.7 Étape 5.6 : Commit

```

git add src/ pom.xml
git commit -m "feat: implement
Factory and Builder patterns

- Add ActiviteFactory for centralized
object creation
- Implement Builder pattern in Seance
class
- Add comprehensive tests for both
patterns

```

```
- Create Main demo application
- Add exec plugin to pom.xml"

git push origin feature/domain-model
```

6.8 ☒ Validation Partie 5

Vérifiez que :

- ☐ La Factory simplifie la création d'objets
- ☐ Le Builder rend le code plus lisible
- ☐ L'application Main s'exécute correctement
- ☐ Tous les tests passent

7 Partie 6 : Persistance JDBC - Base H2

7.1 Objectifs

- Configurer une base de données H2 en mémoire
- Implémenter le pattern DAO
- Persister et récupérer des données

7.2 Étape 6.1 : Configuration de la base de données

Pour travailler proprement avec une base de données, nous commencerons par créer une classe de gestion de la base de données.

Dans le package `datasource`, créez `DatabaseManager.java` :

```
package com.fittrack.datasource;

import org.h2.jdbcx.JdbcDataSource;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

/**
```

```
 * Gère la connexion à la base de
données H2 et l'initialisation des
tables
 */
public class DatabaseManager {

    private final DataSource
dataSource;

    public DatabaseManager() {
        JdbcDataSource ds = new
JdbcDataSource();
        // Base de données en mémoire
ds.setURL("jdbc:h2:mem:fittrackdb;DB_CLOSE_DELAY=
ds.setUser("sa");
ds.setPassword("");
this.dataSource = ds;
    }

    /**
     * Retourne la source de données
     */
    public DataSource getDataSource()
    {
        return dataSource;
    }

    /**
     * Crée les tables nécessaires
     */
    public void createTables() throws
SQLException {
        try (Connection conn =
dataSource.getConnection();
            Statement stmt =
conn.createStatement()) {

            // Table des séances
            stmt.execute(
                "CREATE TABLE IF NOT
EXISTS SEANCE (" +
                " id INT AUTO_INCREMENT
PRIMARY KEY," +
```

```

        "    nom VARCHAR(255)
NOT NULL," +
        "    date DATE NOT
NULL" +
        "    )"
    );

    // Table des exercices
    stmt.execute(
        "CREATE TABLE IF NOT
EXISTS EXERCICE (" +
        "    id INT AUTO_INCREMENT
PRIMARY KEY," +
        "    seance_id INT NOT
NULL," +
        "    type VARCHAR(50)
NOT NULL," +
        "    nom VARCHAR(255)
NOT NULL," +
        "    description
VARCHAR(500)," +
        "    duree INT NOT
NULL," +
        "    series INT DEFAULT
0," +
        "    repetitions INT
DEFAULT 0," +
        "    FOREIGN KEY
(seance_id) REFERENCES SEANCE(id) ON
DELETE CASCADE" +
        "    )"
    );

    System.out.println("✓
Tables créées avec succès");
}

/**
 * Supprime toutes les tables
(utile pour les tests)
 */
public void dropTables() throws

```

```

SQLException {
    try (Connection conn =
dataSource.getConnection();
        Statement stmt =
conn.createStatement()) {

        stmt.execute("DROP TABLE
IF EXISTS EXERCICE");
        stmt.execute("DROP TABLE
IF EXISTS SEANCE");

        System.out.println("✓
Tables supprimées");
    }
}

```

7.3 Étape 6.2 : Interface DAO

Ensuite pour chaque entité persistée (les classes `Seance` et `Activite`), nous allons créer une interface DAO définissant les opérations CRUD. Cela permet d'isoler la logique de persistance du reste de l'application. Et donc de faciliter les changements futurs (changement de SGBD, etc.) ou même simplement les tests unitaires (en mockant les DAO pour ne pas utiliser une base de données réelle).

Le design des DAO peut varier selon les besoins, mais pour cet exemple, nous allons créer un DAO simple pour la classe `Seance`. Il s'agit d'une interface avec les méthodes de base pour sauvegarder, récupérer, mettre à jour et supprimer des séances.

Dans une application réelle, vous auriez une DAO globale pour `Activite` ou des DAO spécifiques pour chaque type d'exercice, mais pour simplifier, nous allons gérer les exercices directement via le DAO de `Seance`.

Dans le package `dao`, créez `SeanceDAO.java` :

```

package com.fittrack.dao;

import com.fittrack.model.Seance;
import java.sql.SQLException;
import java.util.List;

/**
 * Interface définissant les
 * opérations CRUD pour les séances
 */
public interface SeanceDAO {

    /**
     * Sauvegarde une séance en base
     * de données
     * @param seance la séance à
     * sauvegarder
     * @return l'ID généré
     */
    int save(Seance seance) throws
    SQLException;

    /**
     * Récupère une séance par son ID
     * @param id l'identifiant
     * @return la séance ou null si
     * non trouvée
     */
    Seance findById(int id) throws
    SQLException;

    /**
     * Récupère toutes les séances
     * @return liste de toutes les
     * séances
     */
    List<Seance> findAll() throws
    SQLException;

    /**
     * Supprime une séance
     * @param id l'identifiant
     * @return true si supprimée

```

```

    */
    boolean delete(int id) throws
    SQLException;
}

```

7.4 Étape 6.3 : Implémentation H2SeanceDAO

L'interface DAO doit maintenant être implémentée pour interagir avec la base de données H2 via JDBC.

Créez H2SeanceDAO.java dans le package dao :

```

package com.fittrack.dao;

import com.fittrack.model.*;
import javax.sql.DataSource;
import java.sql.*;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

/**
 * Implémentation JDBC du DAO pour H2
 */
public class H2SeanceDAO implements
    SeanceDAO {

    private final DataSource
    dataSource;

    public H2SeanceDAO(DataSource
    dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public int save(Seance seance)
    throws SQLException {
        String sqlSeance = "INSERT INTO
        SEANCE (nom, date) VALUES (?, ?)";
        String sqlExercice = "INSERT
        INTO EXERCICE (seance_id, type,

```

```

nom, description, duree, series,
repetitions) " +
                                "VALUES
(?, ?, ?, ?, ?, ?, ?)";

        try (Connection conn =
dataSource.getConnection()) {

conn.setAutoCommit(false);    //
Transaction

        try {
            // 1. Insérer la séance
            int seanceId;

            try
(PreparedStatement pstmt =
conn.prepareStatement(sqlSeance,
Statement.RETURN_GENERATED_KEYS)) {
                pstmt.setString(1,
seance.getNom());
                pstmt.setDate(2,
Date.valueOf(seance.getDate()));
                pstmt.executeUpdate();

                // Récupérer
l'ID généré
                try (ResultSet rs
= pstmt.getGeneratedKeys()) {
                    if (rs.next()) {
                        seanceId
= rs.getInt(1);
seance.setId(seanceId);
                    } else {
                        throw new
SQLException("Échec de la création de
la séance, pas d'ID généré");
                    }
                }
            }

            // 2. Insérer
les exercices
        }
    }
}

```

```

(PreparedStatement pstmt =
conn.prepareStatement(sqlExercice))
{
    for (Activite
activite : seance.getActivites()) {
        pstmt.setInt(1,
seanceId);

        // Déterminer
le type
        String type;
        int series
= 0;
        int repetitions
= 0;

        if (activite
 instanceof ExerciceCardio) {
            type
= "CARDIO";
        } else if (activite
 instanceof ExerciceForce) {
            type
= "FORCE";
            ExerciceForce
force = (ExerciceForce) activite;
            series
= force.getSeries();
            repetitions
= force.getRepetitions();
        } else {
            type
= "UNKNOWN";
        }

        pstmt.setString(2,
type);
        pstmt.setString(3,
activite.getNom());
        pstmt.setString(4,
activite.getDescription());
        pstmt.setInt(5,
activite.getDuree());
    }
}

```

```

        pstmt.setInt(6,
series);
        pstmt.setInt(7,
repetitions);

pstmt.executeUpdate();
    }
}

conn.commit();
return seanceId;

} catch (SQLException e) {
    conn.rollback();
    throw e;
}
}

@Override
public Seance findById(int id)
throws SQLException {
    String sqlSeance = "SELECT *
FROM SEANCE WHERE id = ?";
    String sqlExercices = "SELECT
* FROM EXERCICE WHERE seance_id = ?";

    try (Connection conn =
dataSource.getConnection()) {

        // 1. Récupérer la séance
        Seance seance;
        try (PreparedStatement pstmt
= conn.prepareStatement(sqlSeance))
        {
            pstmt.setInt(1, id);

            try (ResultSet rs =
pstmt.executeQuery()) {
                if (!rs.next()) {
                    return null; //
Séance non trouvée
                }

```

```

                String nom
= rs.getString("nom");
                LocalDate date
= rs.getDate("date").toLocalDate();
                seance = new
Seance(nom, date);
                seance.setId(id);
            }
        }

        // 2. Récupérer les exercices
        try
(PreparedStatement pstmt
= conn.prepareStatement(sqlExercices))
        {
            pstmt.setInt(1, id);

            try (ResultSet rs =
pstmt.executeQuery()) {
                while (rs.next()) {
                    String type
= rs.getString("type");
                    String nom
= rs.getString("nom");
                    String description
= rs.getString("description");
                    int duree
= rs.getInt("duree");

                    Activite activite;
                    if
("CARDIO".equals(type)) {
                        activite =
new ExerciceCardio(nom, description,
duree);
                    } else if
("FORCE".equals(type)) {
                        int series
= rs.getInt("series");
                        int repetitions
= rs.getInt("repetitions");
                        activite

```

```

= new ExerciceForce(nom, description,
duree, series, repetitions);
        } else {
            continue; //
Type inconnu, on ignore
        }

seance.ajouterActivite(activite);
    }
}

return seance;
}

@Override
public List<Seance> findAll()
throws SQLException {
    List<Seance> seances = new
ArrayList<>();
    String sql = "SELECT id FROM SEANCE
ORDER BY date DESC";

    try (Connection conn =
dataSource.getConnection();
        Statement stmt =
conn.createStatement();
        ResultSet rs =
stmt.executeQuery(sql)) {

        while (rs.next()) {
            int id = rs.getInt("id");
            Seance seance
= findById(id);
            if (seance != null) {
                seances.add(seance);
            }
        }

        return seances;
    }
}

```

```

@Override
public boolean delete(int id)
throws SQLException {
    String sql = "DELETE FROM
SEANCE WHERE id = ?";

    try (Connection conn =
dataSource.getConnection();
        PreparedStatement pstmt
= conn.prepareStatement(sql)) {

        pstmt.setInt(1, id);
        int rowsAffected =
pstmt.executeUpdate();
        return rowsAffected > 0;
    }
}

```

7.5 Étape 6.4 : Tests d'intégration

Après avoir implémenté le DAO, il est crucial de vérifier que tout fonctionne correctement avec des tests d'intégration. Les tests d'intégration vont s'assurer que les opérations CRUD fonctionnent comme prévu avec la base de données H2. Ils s'agit de test qui vérifient l'interaction entre plusieurs composants (DAO et base de données dans ce cas).

Créez DatabaseIntegrationTest.java :

```

package com.fittrack;

import com.fittrack.dao.*;
import com.fittrack.datasource.DatabaseManager;
import com.fittrack.model.*;
import com.fittrack.strategy.*;
import org.junit.jupiter.api.*;

import java.sql.SQLException;
import java.time.LocalDate;
import java.util.List;

```

```

import org.junit.jupiter.api.Assertions.*;

static

@Test
@DisplayName("Tests d'intégration avec la base de données H2")
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class DatabaseIntegrationTest {

    private static DatabaseManager dbManager;
    private static SeanceDAO seanceDAO;

    @BeforeAll
    static void setupDatabase() throws SQLException {
        dbManager = new DatabaseManager();
        dbManager.createTables();
        seanceDAO = new H2SeanceDAO(dbManager);
    }

    @BeforeEach
    void cleanDatabase() throws SQLException {
        // Nettoyer entre chaque test
        dbManager.dropTables();
        dbManager.createTables();
    }

    @Test
    @Order(1)
    @DisplayName("Sauvegarder une séance simple")
    void testSaveSeance() throws SQLException {
        Seance seance = new Seance("Test Séance", LocalDate.now());
        seance.ajouterActivite(new ExerciceCardio("Course", "5km", 30));

        int id = seanceDAO.save(seance);

        assertTrue(id > 0);
        assertEquals(id, seance.getId());
    }

    @Test
    @Order(2)
    @DisplayName("Récupérer une séance par ID")
    void testFindById() throws SQLException {
        // Sauvegarder Seance original = new Seance("Morning Run", LocalDate.of(2026, 1, 25));
        original.ajouterActivite(new ExerciceCardio("Jogging", "Park", 40));
        original.ajouterActivite(new ExerciceForce("Pompes", "Pectoraux", 10, 3, 15));

        int id = seanceDAO.save(original);

        // Récupérer Seance loaded = seanceDAO.findById(id);

        assertNotNull(loaded);
        assertEquals("Morning Run", loaded.getNom());
        assertEquals(LocalDate.of(2026, 1, 25), loaded.getDate());
        assertEquals(2, loaded.getNombreActivites());
        assertEquals(50, loaded.getDureeTotale());
    }
}

```

```

    @Test
    @Order(3)
    @DisplayName("Les calories sont
correctement recalculées")
    void
testCaloriesApresChargement() throws
SQLException {
        Seance original
= new Seance("HIIT Session",
LocalDate.now());
        original.ajouterActivite(new
ExerciceCardio("Burpees", "Full
body", 5));
        original.ajouterActivite(new
ExerciceForce("Squats", "Jambes", 5,
4, 20));

        CalculStrategy strategy =
new IntensiteHaute();
        double caloriesOriginal = original.ge

        int id =
seanceDAO.save(original);
        Seance loaded =
seanceDAO.findById(id);

        double caloriesLoaded =
loaded.getCaloriesTotales(strategy);

        assertEquals(caloriesOriginal,
caloriesLoaded, 0.01);
    }

    @Test
    @Order(4)
    @DisplayName("Récupérer toutes
les séances")
    void testFindAll() throws
SQLException {
        // Sauvegarder plusieurs
séances

        seanceDAO.save(new

```

```

Seance.Builder()
        .setNom("Séance 1")
        .setDate(LocalDate.now())
        .addActivite(new
ExerciceCardio("Course", "Test",
20))
        .build();

        seanceDAO.save(new
Seance.Builder()
        .setNom("Séance 2")
        .setDate(LocalDate.now().minusDays(1))
        .addActivite(new
ExerciceForce("Pompes", "Test", 10))
        .build());

        seanceDAO.save(new
Seance.Builder()
        .setNom("Séance 3")
        .setDate(LocalDate.now().minusDays(2))
        .addActivite(new
ExerciceCardio("Vélo", "Test", 30))
        .build());

        List<Seance> seances =
seanceDAO.findAll();

        assertEquals(3,
seances.size());
    }

    @Test
    @Order(5)
    @DisplayName("Supprimer une
séance")
    void testDelete() throws
SQLException {
        Seance seance = new Seance("To
Delete", LocalDate.now());
        seance.ajouterActivite(new
ExerciceCardio("Test", "Test", 10));

        int id =

```

```

seanceDAO.save(seance);

        boolean deleted =
seanceDAO.delete(id);
        assertTrue(deleted);

        Seance loaded =
seanceDAO.findById(id);
        assertNull(loader);
    }

    @Test
    @Order(6)
    @DisplayName("Séance non trouvée
retourne null")
    void testFindByIdNonExistant()
throws SQLException {
        Seance seance =
seanceDAO.findById(9999);
        assertNull(seance);
    }
}

```

7.6 Étape 6.5 : Exécution des tests

```
mvn clean test
```

Vous devriez voir tous les tests passer, y compris les tests d'intégration.

Dans une application réelle, les test d'intégrations sont séparés des tests unitaires et exécutés dans un profil Maven spécifique avec un plugin comme `maven-failsafe-plugin` lors de l'étape d'intégration `mvn verify`.

7.7 Étape 6.6 : Application complète avec persistance

Modifiez `Main.java` pour inclure la persistance :

```

package com.fittrack;

import com.fittrack.dao.*;
import com.fittrack.datasource.DatabaseManager;
import com.fittrack.model.*;
import com.fittrack.strategy.*;

import java.sql.SQLException;
import java.time.LocalDate;
import java.util.List;

public class Main {

    public static void main(String[]
args) {
        try {
            System.out.println("===
FitTrack Pro - Application complète
===\n");

            // 1. Initialiser la base
de données
            DatabaseManager dbManager
= new DatabaseManager();
            dbManager.createTables();
            SeanceDAO seanceDAO = new H2SeanceDAO(dbManager.

            System.out.println("✓ Base
de données initialisée\n");

            // 2. Créer et sauvegarder
une séance
            Seance seance1 = new
Seance.Builder()
                .setNom("Séance
Full Body")
                .setDate(LocalDate.now())
                .addActivite(ActiviteFactory.createCardio("É
Tapis", 10))
                .addActivite(ActiviteFactory.createForce("Sq
uats", 15, 4, 12))
                .addActivite(ActiviteFactory.createCardio("C
ourir", 30))

```

```

        .addActivite(ActiviteFactory.c
"Pectoraux", 10, 3, 20))
        .build();

        int id1 =
seanceDAO.save(seance1);
        System.out.println("✓
Séance sauvegardée avec ID: " + id1);
        System.out.println(" "
+ seance1 + "\n");

// 3. Créer une deuxième
séance
        Seance seance2 = new
Seance.Builder()
        .setNom("Séance Cardio")
        .setDate(LocalDate.now().minus
        .addActivite(ActiviteFactory.c
"Route", 45))
        .addActivite(ActiviteFactory.c
"Crawl", 30))
        .build();

        int id2 =
seanceDAO.save(seance2);
        System.out.println("✓
Séance sauvegardée avec ID: " + id2);
        System.out.println(" "
+ seance2 + "\n");

// 4. Récupérer toutes
les séances
        System.out.println("📁
Toutes les séances enregistrées:");
        List<Seance> seances
= seanceDAO.findAll();
        for (Seance s : seances) {
            System.out.println("
" + s);

// Calculer les calories
CalculStrategy strategy
= new IntensiteModeree();

```

```

        double calories
= s.getCaloriesTotales(strategy);
        System.out.printf("
Calories (%s): %.0f cal\n",
strategy.getNom(), calories);
    }

    System.out.println("\n===
Application terminée avec succès
===");

    } catch (SQLException e) {
        System.err.println("✗ Erreur
base de données: " + e.getMessage());
        e.printStackTrace();
    }
}

```

7.8 Étape 6.7 : Commit final

```

git add src/ pom.xml
git commit -m "feat: implement JDBC
persistence with H2 database

- Add DatabaseManager for H2
configuration
- Implement SeanceDAO interface
- Create H2SeanceDAO with full CRUD
operations
- Add comprehensive integration tests
- Update Main with persistence demo
- Transaction support for data
integrity"

git push origin feature/domain-model

```

7.9 ☒ Validation Partie 6

Vérifiez que :

- ☐ La base H2 se crée correctement
- ☐ Les données persistent et se récupèrent
- ☐ Les tests d'intégration passent

- ☐ L'application Main fonctionne avec la DB

8 Partie 7 : Merge et documentation

8.1 Objectifs

- Fusionner votre branche feature
- Créer un README complet
- Tag de version

8.2 Étape 7.1 : Préparation du merge

```
# S'assurer que tout est commité
git status

# Retourner sur main
git checkout main

# Merger la feature
git merge feature/domain-model

# Pousser sur GitHub
git push origin main
```

8.3 Étape 7.2 : Création du README

Pour documenter votre projet, un bon README est essentiel. Il s'agit de fournir une vue d'ensemble claire, les instructions d'installation, les exemples d'utilisation, et toute autre information pertinente. On utilise pour cela le format Markdown.

Créez README.md à la racine :

```
# ☐ FitTrack Pro

Application Java de gestion et suivi
de séances d'entraînement sportif.

## 📁 Fonctionnalités
```

```
- ☐ Modélisation d'exercices (Cardio,
Force)
- ☐ Création de séances
d'entraînement
- ☐ Calcul automatique de durée et
calories
- ☐ Différentes intensités d'exercice
(Strategy Pattern)
- ☐ Persistance en base de données H2
- ☐ Tests unitaires et d'intégration
complets
```

☐ Technologies

```
- **Java 11+**
- **Maven 3.6+**
- **H2 Database** (en mémoire)
- **JUnit 5** (tests)
```

☐ Installation

```
```bash
Cloner le projet
git clone https://github.com/VOTRE_
USERNAME/fittrack-pro.git
cd fittrack-pro
```

```
Compiler
mvn clean compile
```

```
Lancer les tests
mvn test
```

```
Exécuter l'application
mvn exec:java
```
```

📁 Structure du projet

```
...
src/main/java/com/fittrack/
├─ model/          # Modèle de domaine
```

```
(Seance, Activite, Exercice...)
├─ strategy/      # Stratégies de
calcul de calories
├─ dao/           # Data Access Objects
├─ datasource/    # Configuration
base de données
└─ Main.java      # Application
principale
```

```

## ☐ Exemples d'utilisation

### Créer une séance

```
```java
Seance seance = new Seance.Builder()
    .setNom("Morning Workout")
    .setDate(LocalDate.now())
    .addActivite(ActiviteFactory.createCardio(),
"5km", 30))
    .addActivite(ActiviteFactory.createForce(),
"Pectoraux", 10, 3, 20))
    .build();
```
```

### Calculer les calories

```
```java
CalculStrategy strategy = new
IntensiteHaute();
double calories =
seance.getCaloriesTotales(strategy);
System.out.printf("Calories brûlées:
%.0f cal\n", calories);
```
```

### Persister en base de données

```
```java
DatabaseManager dbManager = new
DatabaseManager();
dbManager.createTables();
```

```
SeanceDAO dao = new H2SeanceDAO(dbManager.getDataSource());
int id = dao.save(seance);
```

```
Seance loaded = dao.findById(id);
```
```

## 🛠 Tests

```
```bash
# Tous les tests
mvn test

# Tests spécifiques
mvn test -Dtest=SeanceTest
mvn test -Dtest=DatabaseIntegrationTest
```
```

## ☐ Design Patterns implémentés

- **Strategy**: Calcul des calories selon différentes intensités
- **Factory**: Création centralisée d'activités
- **Builder**: Construction fluide de séances complexes
- **DAO**: Abstraction de la persistance

## 👤 Auteur

Votre Nom - M1 InfoMath/CNAM

## 📄 Licence

Projet pédagogique - 2026

## 8.4 Étape 7.3 : Tag de version

```
git add README.md
git commit -m "docs: add comprehensive README"
git push origin main
```

```
Créer un tag
git tag -a v1.0.0 -m "Version
1.0.0 - Application complète avec
persistance"
git push origin v1.0.0
```

## 8.5 ☒ Validation Partie 7

Vérifiez que :

- ☐ La branche feature est mergée
- ☐ Le README est complet et clair
- ☐ Un tag de version est créé
- ☐ Tout est poussé sur GitHub

## 9 Partie 8 : Extensions optionnelles (Bonus)

### 9.1 Extension 1 : Rapports et statistiques

Créez `SeanceStats.java` dans un nouveau package `stats` :

```
package com.fittrack.stats;

import com.fittrack.model.Seance;
import com.fittrack.strategy.CalculStrategy;
import java.util.List;

public class SeanceStats {

 public static double
 moyenneDuree(List<Seance> seances) {
 return seances.stream()
 .mapToInt(Seance::getDureeT)
 .average()
 .orElse(0.0);
 }

 public static double
 totalCalories(List<Seance> seances,
 CalculStrategy strategy) {
```

```
 return seances.stream()
 .mapToDouble(s
-> s.getCaloriesTotales(strategy))
 .sum();
 }

 public static int
 nombreTotalActivites(List<Seance>
seances) {
 return seances.stream()
 .mapToInt(Seance::getNombreActivites)
 .sum();
 }
}
```

### 9.2 Extension 2 : Export CSV

Créez `CsvExporter.java` :

```
package com.fittrack.export;

import com.fittrack.model.*;
import com.fittrack.strategy.CalculStrategy;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;

public class CsvExporter {

 public static void
 exportSeances(List<Seance> seances,
 String filename,
 CalculStrategy strategy) throws
 IOException {
 try (FileWriter writer = new
 FileWriter(filename)) {
 // En-tête
 writer.write("Nom,Date,Durée
(min),Calories,Nb Activités\n");

 // Données
 for (Seance s : seances) {
 writer.write(String.format("%s,%s,
```

```

%d,%.0f,%d\n",
 s.getNom(),
 s.getDate(),
 s.getDureeTotale(),
 s.getCaloriesTotales(strategy),
 s.getNombreActivites()
));
 }
}
}
}

```

### 9.3 Extension 3 : Interface console interactive

Créez ConsoleApp.java :

```

package com.fittrack;

import com.fittrack.dao.*;
import com.fittrack.datasource.DatabaseManager;
import com.fittrack.model.*;
import com.fittrack.strategy.*;

import java.sql.SQLException;
import java.time.LocalDate;
import java.util.List;
import java.util.Scanner;

public class ConsoleApp {

 private static SeanceDAO seanceDAO;

 private static Scanner scanner =
new Scanner(System.in);

 public static void main(String[]
args) throws SQLException {
 DatabaseManager dbManager =
new DatabaseManager();
 dbManager.createTables();
 seanceDAO = new H2SeanceDAO(dbManager

```

```

 boolean running = true;

 while (running) {
 afficherMenu();
 int choix =
scanner.nextInt();
 scanner.nextLine(); //
Consommer le retour ligne

 switch (choix) {
 case 1:
 creerSeance();
 break;
 case 2:
 afficherSeances();
 break;
 case 3:
 afficherStatistiques();
 break;
 case 0:
 running = false;
 break;
 default:
 System.out.println("Choix
 invalide");
 }
 }

 System.out.println("Au
 revoir !");
 }

 private static void afficherMenu()
 {
 System.out.println("\n===
 FitTrack Pro ===");
 System.out.println("1. Créer
 une séance");
 System.out.println("2. Afficher
 les séances");
 System.out.println("3.
 Statistiques");
 }
}

```

```

 System.out.println("0.
Quitter");
 System.out.print("Votre choix:
");
 }

 private static void creerSeance()
throws SQLException {
 System.out.print("Nom de la
séance: ");
 String nom =
scanner.nextLine();

 Seance seance = new Seance(nom,
LocalDate.now());

 boolean ajouterActivites
= true;
 while (ajouterActivites) {
 System.out.print("Type
(1=Cardio, 2=Force, 0=Terminer): ");
 int type = scanner.nextInt();
 scanner.nextLine();

 if (type == 0) {
 ajouterActivites
= false;
 continue;
 }

 System.out.print("Nom de
l'exercice: ");
 String nomEx =
scanner.nextLine();

 System.out.print("Description: ");
 String desc =
scanner.nextLine();

 System.out.print("Durée
(min): ");
 int duree =
scanner.nextInt();

```

```

 scanner.nextLine();

 if (type == 1) {
 seance.ajouterActivite(new
ExerciceCardio(nomEx, desc, duree));
 } else {
 System.out.print("Séries:
");
 int series =
scanner.nextInt();
 System.out.print("Répétitions: ");
 int reps =
scanner.nextInt();
 scanner.nextLine();

 seance.ajouterActivite(new
ExerciceForce(nomEx, desc, duree,
series, reps));
 }

 int id =
seanceDAO.save(seance);
 System.out.println("✓ Séance
créée avec l'ID: " + id);
 }

 private static
void afficherSeances() throws
SQLException {
 List<Seance> seances =
seanceDAO.findAll();

 if (seances.isEmpty()) {
 System.out.println("Aucune
séance enregistrée");
 return;
 }

 System.out.println("\n=== Vos
séances ===");
 for (Seance s : seances) {

```

```

 System.out.println(s);
 System.out.printf("
Calories (intensité modérée): %.0f
cal\n",
 s.getCaloriesTotales(new
IntensiteModeree()));
 }
}

 private static
void afficherStatistiques() throws
SQLException {
 List<Seance> seances =
seanceDAO.findAll();

 if (seances.isEmpty()) {
 System.out.println("Aucune
séance pour calculer les
statistiques");
 return;
 }

 System.out.println("\n===
Statistiques ===");
 System.out.println("Nombre de
séances: " + seances.size());

 int totalDuree =
seances.stream()
 .mapToInt(Seance::getDureeTotale)
 .sum();
 System.out.println("Durée
totale: " + totalDuree + " min");

 double totalCal =
seances.stream()
 .mapToDouble(s
-> s.getCaloriesTotales(new
IntensiteModeree()))
 .sum();
 System.out.printf("Calories
totales: %.0f cal\n", totalCal);

```

```

 }
}

```

## 10 Conclusion

Félicitations ! Vous avez développé une appli-  
cation Java complète en utilisant :

- ☑ POO avancée (interfaces, héritage, poly-  
morphisme)
- ☑ Design patterns (Strategy, Factory, Builder,  
DAO)
- ☑ Maven pour la gestion de projet
- ☑ Git pour le versionnage
- ☑ JUnit 5 pour les tests
- ☑ JDBC avec H2 pour la persistance

### 10.1 Prochaines étapes possibles

- Migrer vers une vraie base de données (Post-  
greSQL)
- Utiliser JPA au dessus de JDBC
- Implémenter une API REST (Quarkus) ...