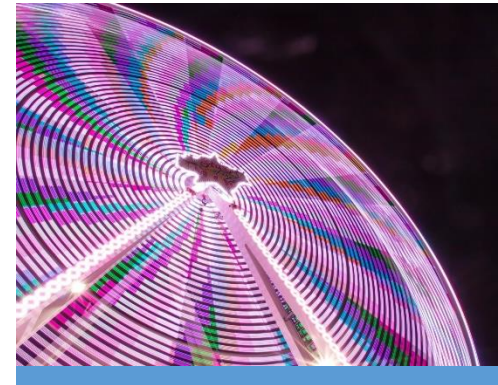


Langage de programmation JAVA

AMMOR Fatimazahra



Exemple Pratique : Gestion des employés

Objectif = Créer une application simple pour gérer différents types d'employés dans une entreprise : temps plein, temps partiel, stagiaire, etc.

Utilisation des tableaux pour stocker les informations des employés
Utilisation du polymorphisme pour gérer différents types d'employés et calculer leur salaire en fonction de leurs spécificités.

JAVA

```
graph TD; JAVA --> Les_bases[Les bases]; JAVA --> L_Orienté_Obj[\"L'Orienté Objet en JAVA\"]; Les_bases --> Bases_List[\"1. Environnement et installation<br/>2. Structure et syntaxe<br/>3. Variables<br/>4. Opérateurs<br/>5. Méthodes<br/>6. Conditions<br/>7. Tableaux<br/>8. Boucles<br/>9. Listes<br/>10. Enums\"]; L_Orienté_Obj --> Oo_List[\"1. Classes vs Objets<br/>2. Constructeurs<br/>3. Héritage<br/>4. Classes prédéfinies<br/>5. Accessibilité<br/>6. Package<br/>7. Classe abstraite vs Interface<br/>8. Polymorphisme\"];
```

Les bases

1. Environnement et installation
2. Structure et syntaxe
3. Variables
4. Opérateurs
5. Méthodes
6. Conditions
7. Tableaux
8. Boucles
9. Listes
10. Enums


L'Orienté Objet en JAVA

1. Classes vs Objets
2. Constructeurs
3. Héritage
4. Classes prédéfinies
5. Accessibilité
6. Package
7. Classe abstraite vs Interface
8. Polymorphisme

Les tableaux

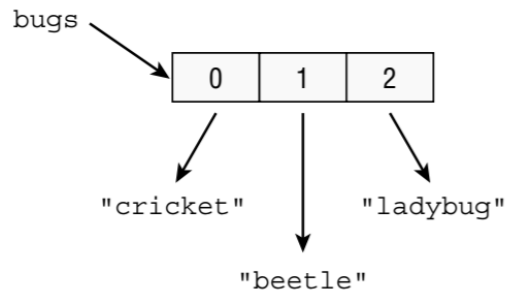
```
int[] moreNumbers = new int[] {42, 55, 99};
```

```
int[] moreNumbers = {42, 55, 99};
```



```
String letters; ⇔ char[] letters;
```

```
String[] bugs = { "cricket", "beetle", "ladybug" };  
String[] alias = bugs;  
System.out.println(bugs.equals(alias)); // true
```



```
int[] tableau = new int[5]; // La taille du tableau est fixée à 5
```

```
int[] numAnimals;  
int [] numAnimals2;  
int []numAnimals3;  
int numAnimals4[];  
int numAnimals5 [];
```

```
Arrays.sort(numbers);  
➔ Trie croissant  
Arrays.binarySearch(n  
umbers, 9);  
➔ Cherche l'index du  
nombre 9
```

```
int[] ids, types;
```

```
int[][] differentSizes = {{1, 4}, {3}, {9,8,7}};
```

```
int ids[], types;
```


?

```
import java.util.*; // import whole package including Arrays  
import java.util.Arrays; // import just Arrays
```

Les boucles

```
int counter = 0;
while (counter < 10) {
    double price = counter * 10;
    System.out.println(price);
    counter++;
}
```

```
int lizard = 0;
do {
    lizard++;
} while(false);
System.out.println(lizard);
```



```
int pen = 2;
int pigs = 5;
while(pen < 10)
    pigs++;
```

```
public void printNames(String[] names) {
    for(var name : names)
        System.out.println(name);
}
```

```
for( ; ; )
    System.out.println("Hello World");
```

```
public void printNames(List<String> names) {
    for(var name : names)
        System.out.println(name);
}
```

```
for(int i=0; i < 10; i++)
    System.out.println("Value is: "+i);
System.out.println(i); // DOES
NOT COMPILE
```

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
    System.out.print(y + " ");
}
System.out.print(x + " ");
```

Polymorphisme

- Permet à des objets de différents types de **répondre de manière uniforme** à des appels de méthodes.
- Imaginons que nous avons un programme qui gère différents types d'animaux. Nous avons une classe `Animal`, qui est une classe générique, et deux sous-classes spécifiques : `Chien` et `Chat`. Chaque animal peut produire un son (`faireDuBruit`), mais le type de son dépend de l'animal spécifique.
- Grâce au polymorphisme, on peut écrire du code qui traite **tout type d'animal de manière uniforme**, même si chaque animal réagit différemment lorsqu'on appelle la méthode `faireDuBruit()`.

```
public class TestPolymorphisme {  
    public static void main(String[] args) {  
        // Déclarons des variables de type parent Animal  
        Animal monChien = new Chien();  
        Animal monChat = new Chat();  
  
        // Grâce au polymorphisme, la méthode appropriée est appelée à l'exécution  
        monChien.faireDuBruit(); // Affiche : "Le chien aboie"  
        monChat.faireDuBruit(); // Affiche : "Le chat miaule"  
  
        // On peut également traiter plusieurs types d'animaux de manière générique  
        faireFaireDuBruit(monChien);  
        faireFaireDuBruit(monChat);  
    }  
  
    // Méthode générique qui accepte tout type d'Animal  
    public static void faireFaireDuBruit(Animal animal) {  
        animal.faireDuBruit(); // Le bon comportement est appelé automatiquement  
    }  
}
```

Polymorphisme

- Imaginons maintenant qu'o souhaite ajouter une nouvelle sous-classe nommée Oiseau, qui a aussi sa propre manière de faire du bruit ("L'oiseau chante").
- On n'as pas besoin de modifier le code principal pour supporter ce nouveau type d'animal. On peut simplement ajouter une nouvelle classe Oiseau et l'utiliser immédiatement avec le même code.

```
class Oiseau extends Animal {  
    @Override  
    public void faireDuBruit() {  
        System.out.println("L'oiseau chante");  
    }  
}  
  
// Dans ton programme principal :  
Animal monOiseau = new Oiseau();  
monOiseau.faireDuBruit(); // Affiche : "L'oiseau chante"
```


Classe de base Employe :

- Est une classe abstraite Employe qui représente un employé général :
 - nom (String)
 - prenom (String)
 - salaireBase (double)
- Elle doit avoir un constructeur pour initialiser ces attributs.
- On ajoute une méthode abstraite calculerSalaire() qui sera implémentée au sein de chaque sous-classe pour calculer le salaire final de chaque employé.

Les sous-classes

Créez trois sous-classes dérivées de « Employe » :

- **EmployeTempsPlein** : Représente un employé à temps plein. Son salaire est égal à son salaire de base.
- **EmployeTempsPartiel** : Représente un employé à temps partiel. Son salaire est égal à la moitié d'un salaire de base.
- **Stagiaire** : Représente un stagiaire. Son salaire est égal à une somme fixe (par exemple, 2000dh) sans tenir compte du salaire de base.

La classe GestionEmployes

- Contient un tableau d'employés (Employe[]).
- Implémentez une méthode ajouterEmploye(Employe e) pour ajouter un employé au tableau.
- Implémentez une méthode afficherSalaires() qui parcourt le tableau, et qui calcule en même temps le salaire de chaque employé.

La classe Main

- C'est la classe principale, nous permettra de créer une instance de GestionEmployes et ajoutez plusieurs employés de différents types (temps plein, temps partiel, stagiaire).
- Affichez ensuite les salaires calculés pour tous les employés.

```
// Classe abstraite Employe
abstract class Employe {
    protected String nom;
    protected String prenom;
    protected double salaireBase;

    public Employe(String nom, String prenom, double
salaireBase) {
        this.nom = nom;
        this.prenom = prenom;
        this.salaireBase = salaireBase;
    }

    public abstract double calculerSalaire();

    @Override
    public String toString() {
        return prenom + " " + nom;
    }
}
```

```
// Classe EmployeTempsPlein
class EmployeTempsPlein extends Employe {
    public EmployeTempsPlein(String nom, String prenom,
        double salaireBase) {
        super(nom, prenom, salaireBase);
    }
    @Override
    public double calculerSalaire() {
        return salaireBase;
    }
}
```

```
// Classe EmployeTempsPartiel
class EmployeTempsPartiel extends Employe {
    public EmployeTempsPartiel(String nom, String prenom,
        double salaireBase) {
        super(nom, prenom, salaireBase);
    }
    @Override
    public double calculerSalaire() {
        return salaireBase/2;
    }
}
```

```
// Classe Stagiaire
class Stagiaire extends Employe {
    private final double salaireFixe = 2000;

    public Stagiaire(String nom, String prenom) {
        super(nom, prenom, 0);
    }

    @Override
    public double calculerSalaire() {
        return salaireFixe;
    }
}
```

```
// Classe GestionEmployes
class GestionEmployes {
    private Employe[] employees;
    private int index;
    public GestionEmployes(int taille) {
        employees = new Employe[taille];
        index = 0;
    }
    public void ajouterEmploye(Employe e) {
        if (index < employees.length) {
            employees[index++] = e;
        } else {
            System.out.println("Tableau d'employés plein !");
        }
    }
    public void afficherSalaires() {
        for (Employe e : employees) {
            if (e != null) {
                System.out.println(e.toString() + " : " +
e.calculerSalaire() + « dh");
            }
        }
    }
}
```

```
// Classe principale
public class Main {
    public static void main(String[] args) {
        GestionEmployes gestion = new GestionEmployes(3);

        Employe e1 = new EmployeTempsPlein("Alamane", « Karim", 12000);
        Employe e2 = new EmployeTempsPartiel("Marismane", « Ayman", 6000);
        Employe e3 = new Stagiaire("naouras", « Inass");

        gestion.ajouterEmploye(e1);
        gestion.ajouterEmploye(e2);
        gestion.ajouterEmploye(e3);

        gestion.afficherSalaires();
    }
}
```

Exemple Pratique : Gestion des employés

Objectif = Créer une variation de l'application

Gérer un autre type d'employé : freelance qui est payé par jour.

L'administrateur nous donne le prix par jour de chaque employé freelance.

Utilisation des listes pour stocker les informations des employés

Utilisation du polymorphisme pour gérer différents types d'employés et calculer leur salaire en fonction de leurs spécificités.

Pour le type freelance, faire le calcul : $\text{salaireParJour} * 20$;
(travaille 20 jours par mois)

Les listes

Cela rendra le programme plus flexible, car une ArrayList peut changer de taille dynamiquement et est plus facile à manipuler que des tableaux statiques.

```
ArrayList<Integer> liste = new ArrayList<>(); // Taille dynamique
liste.add(1);
liste.add(2);
liste.add(int i, E e) // insérer l'element e dans l'index i
liste.get(index) // renvoie l'élément qui se trouve à l'index spécifié
liste.set(index, e) // remplace l'élément qui se trouve dans index par l'element e
liste.remove(index) // supprime l'élément qui se trouve dans index
liste.size() // renvoie le nombre d'éléments dans la liste
liste.isEmpty() // renvoie true si la liste est vide
liste.clear() // supprime les elements de la liste
liste.contains(e) // renvoie true si la liste contient l'élément e
...etc.
```

```
public void printNames(List<String> names) {
    for(String name : names)
        System.out.println(name);
}
```

Créez quatre classes dérivées de « **Employe** » qui implémentent la méthode calculerSalaire():

- **EmployeTempsPlein** : Représente un employé à temps plein.

Son salaire est égal à son salaire de base.

- **EmployeTempsPartiel** : Représente un employé à temps partiel. Son salaire est égal à la moitié d'un salaire de base.

- **Stagiaire** : Représente un stagiaire. Son salaire est égal à une somme fixe (par exemple, 2000dh) sans tenir compte du salaire de base.

- **Freelance** : représente un travailleur indépendant. Son salaire est égal à son salaire par jour * 20 (car le freelance travaille 20jours/mois).

La classe GestionEmployes

- Contient une liste d'employés
- Implémentez une méthode ajouterEmploye(Employe e) pour ajouter un employé à la liste.
- Implémentez une méthode afficherSalaires() qui parcourt la liste, affiche les employés et calcule le salaire de chaque employé.

```
// Classe Freelance
class Freelance extends Employe {
    private double salaireParJour = 0;

    public Freelance(String nom, String prenom,
                     double salaireJour) {
        super(nom, prenom, 0);
        this.salaireParJour = salaireJour
    }

    @Override
    public double calculerSalaire() {
        return salaireJour*20;
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
// Classe GestionEmployes
class GestionEmployes {
    // Utilisation d'une ArrayList au lieu d'un tableau
    private List<Employe> employes;
    public GestionEmployes() {
        // Initialiser l'ArrayList
        employes = new ArrayList<>();
    }
    // Ajouter un employé à l'ArrayList
    public void ajouterEmploye(Employe e) {
        employes.add(e);
    }
    // Afficher les salaires des employés
    public void afficherSalaires() {
        for (Employe e : employes) {
            System.out.println(e + " : " + e.calculerSalaire() + " dh");
        }
    }
}
```



```
// Classe principale
public class Main {
    public static void main(String[] args) {
        GestionEmployes gestion = new GestionEmployes();

        // Ajouter des employés de différents types
        Employe e1 = new EmployeTempsPlein("Alamane", « Karim", 12000);
        Employe e2 = new EmployeTempsPartiel("Marismane", « Ayman", 6000);
        Employe e3 = new Stagiaire("naouras", « Inass");
        Employe e4 = new Freelance(« fadi", « Amine » ,3000);

        // Ajout à l'ArrayList
        gestion.ajouterEmploye(e1);
        gestion.ajouterEmploye(e2);
        gestion.ajouterEmploye(e3);
        gestion.ajouterEmploye(e4);

        // Afficher les salaires de tous les employés
        gestion.afficherSalaires();
    }
}
```

Exemple Pratique : Gestion des employés

Objectif = Créer une autre variation de l'application
Gérer les types des employés grâce aux enums plutôt que d'utiliser le polymorphisme

Utilisation des listes toujours pour stocker les informations des employés
Utilisation des enums pour gérer différents types d'employés et calculer leur salaire en fonction de leurs spécificités.

Les enums

Un **enum** (abréviation de "enumeration") est un type spécial de classe utilisé pour représenter un groupe fixe de constantes, c'est-à-dire des valeurs qui ne changent pas. Les enums sont particulièrement utiles lorsque vous avez un ensemble pré-défini de valeurs possibles pour une variable.

Les enums peuvent avoir des méthodes, des constructeurs et des attributs comme les classes ordinaires.

```
public enum TypeEmploye {  
    TEMPS_PLEIN, TEMPS_PARTIEL, STAGIAIRE  
}
```

Déclaration d'un enum

Si dans un fichier séparé, le fichier doit avoir le même nom : TypeEmploye.java

```
Employe.TypeEmploye type = Employe.TypeEmploye.TEMPS_PLEIN;
```

Pour y accéder depuis n'importe quelle classe

Classe de base Employe :

- Est une classe qui représente un employé général :
 - nom (String)
 - prenom (String)
 - salaireBase (double)
 - salaireJour (double)
 - typeEmploye (enum)
- Elle doit avoir un constructeur pour initialiser ces attributs.
- Une méthode pour affichage.
- Une méthode calculerSalaire() qui fera le calcul du salaire final de chaque employé.

La classe GestionEmployes

- Contient une liste d'employés.
- Implémentez une méthode ajouterEmploye(Employe e) pour ajouter un employé à la liste.
- Implémentez une méthode afficherSalaires() qui parcourt la liste, et m'afficher le salaire de chaque employé.

```
enum TypeEmploye {  
    TEMPS_PLEIN,  
    TEMPS_PARTIEL,  
    STAGIAIRE,  
    FREELANCE  
}
```

```
private String nom;  
    private String prenom;  
    private double salaireBase;  
    private double salaireJour;  
    private TypeEmploye t;
```

```
// Enum pour représenter le type d'employé
enum TypeEmploye {
    TEMPS_PLEIN,
    TEMPS_PARTIEL,
    STAGIAIRE,
    FREELANCE
}

// Classe Employe
class Employe {
    private String nom;
    private String prenom;
    private double salaireBase;
    private double salaireJour;
    private TypeEmploye t;

    public Employe(String nom, String prenom, double
        salaireBase, double salaireJour,
        TypeEmploye t) {
        this.nom = nom;
        this.prenom = prenom;
        this.salaireBase = salaireBase;
        this.t = t;
    }
}
```

```
// Méthode pour calculer le salaire en fonction du type
// d'employé
public double calculerSalaire() {
    switch (t) {
        case TEMPS_PLEIN:
            return salaireBase; // Employé temps plein : salaire
                                // de base

        case TEMPS_PARTIEL:
            return salaireBase/2; // Employé temps partiel

        case STAGIAIRE:
            return 2000; // Stagiaire : salaire fixe de 2000dh

        case Freelance:
            return salaireJour*20; // Employé freelance

        default:
            return 0;
    }
}

@Override
public String toString() {
    return prenom + " " + nom + " (" + t + ")";
}
}
```

```
// Classe GestionEmployes
class GestionEmployes {
    private ArrayList<Employe> employes;

    public GestionEmployes() {
        employes = new ArrayList<>();
    }

    public void ajouterEmploye(Employe e) {
        employes.add(e);
    }

    public void afficherSalaires() {
        for (Employe e : employes) {
            System.out.println(e + " : " +
                               e.calculerSalaire() + " dhs");
        }
    }
}
```

```
// Classe principale
public class Main {
    public static void main(String[] args) {
        GestionEmployes gestion = new GestionEmployes();

        // Ajouter des employés de différents types
        Employe e1 = new Employe("Alamane", « Karim",
                                   12000, 0, TypeEmploye.TEMPS_PLEIN);
        Employe e2 = new Employe("Marismane", « Ayman",
                                   6000, 0, TypeEmploye.TEMPS_PARTIEL);
        Employe e3 = new Employe("naouras", "Inass",0, 0,
                                   TypeEmploye.STAGIAIRE);
        Employe e4 = new Freelance(« fadi", « Amine« ,0, 0,
                                   TypeEmploye.Freelance);

        gestion.ajouterEmploye(e1);
        gestion.ajouterEmploye(e2);
        gestion.ajouterEmploye(e3);
        gestion.ajouterEmploye(e4);

        // Afficher les salaires de tous les employés
        gestion.afficherSalaires();
    }
}
```

Exemple Pratique : Gestion d'une école

**Objectif = Créer une application qui gère le personnel d'une école
Gérer les types des employés grâce au polymorphisme et aux listes.**

Nous avons des étudiants, des enseignants, et du personnel administratif.

- **Personne** (classe de base) : Cette classe représente les attributs et comportements communs à toutes les personnes de l'école (nom, âge, etc.), elle contient une méthode abstraite pour afficher le rôle de chaque personnes de l'école (redéfini au sein de chaque sous classe et m'affiche le rôle de chaque personne).
- **Étudiant, Enseignant, PersonnelAdministratif** (sous-classes de **Personne**) : Ces classes héritent de **Personne** et ajoutent des comportements spécifiques.
- **École** : Cette classe contient une **ArrayList** de personnes et permet d'ajouter des personnes à l'école et de les lister.


```
import java.util.ArrayList;
```

```
// Classe de base Personne
```

```
abstract class Personne {
```

```
    private String nom;
```

```
    private int age;
```

```
    public Personne(String nom, int age) {
```

```
        this.nom = nom;
```

```
        this.age = age;
```

```
    }
```

```
    public String getNom() {
```

```
        return nom;
```

```
    }
```

```
    public int getAge() {
```

```
        return age;
```

```
    }
```

```
    // Méthode abstraite pour être implémentée par les sous-classes
```

```
    public abstract void afficherRole();
```

```
}
```

```
// Classe Étudiant qui hérite de Personne
```

```
class Etudiant extends Personne {
```

```
    private String filiere;
```

```
    public Etudiant(String nom, int age, String filiere) {
```

```
        super(nom, age);
```

```
        this.filiere = filiere;
```

```
    }
```

```
    @Override
```

```
    public void afficherRole() {
```

```
        System.out.println("Je suis un étudiant en " + filiere);
```

```
    }
```

```
}
```

```
// Classe Enseignant qui hérite de Personne
class Enseignant extends Personne {
    private String matiere;

    public Enseignant(String nom, int age, String matiere) {
        super(nom, age);
        this.matiere = matiere;

        @Override
        public void afficherRole() {
            System.out.println("Je suis un enseignant de " + matiere);
        }
    }
}
```

```
// Classe PersonnelAdministratif qui hérite de Personne
class PersonnelAdministratif extends Personne {
    private String poste;

    public PersonnelAdministratif(String nom, int age, String
        poste) {
        super(nom, age);
        this.poste = poste;
    }

    @Override
    public void afficherRole() {
        System.out.println("Je suis un membre du personnel
        administratif : " + poste);
    }
}
```

```
// Classe École pour gérer la liste des personnes
class Ecole {
    private ArrayList<Personne> personnes = new ArrayList<>();

    // Ajoute une personne à l'école
    public void ajouterPersonne(Personne personne) {
        personnes.add(personne);
    }

    // Affiche toutes les personnes de l'école
    public void listerPersonnes() {
        for (Personne personne : personnes) {
            System.out.println(personne.getNom() + ", " +
personne.getAge() + " ans.");
            personne.afficherRole(); // Polymorphisme : la méthode
spécifique est appelée
        }
    }
}
```

```
// Classe principale pour exécuter le programme
public class Main {
    public static void main(String[] args) {
        // Création d'une instance d'école
        Ecole ecole = new Ecole();

        // Ajout de personnes à l'école
        ecole.ajouterPersonne(new Etudiant("« Ali", 20,
"Informatique"));
        ecole.ajouterPersonne(new Enseignant("M.
Naouras", 45, "Mathématiques"));
        ecole.ajouterPersonne(new
PersonnelAdministratif("Mme Ghilbane", 35,
"Secrétaire"));

        // Lister toutes les personnes
        ecole.listerPersonnes();
    }
}
```

Classe immuable

Classe immuable (ou *immutable class*) = Classe dont les instances ne peuvent pas être modifiées après leur création.

- ➔ Tous les champs de l'objet sont final et, une fois que l'objet est créé, son état ne peut pas être changé.
- ➔ Plusieurs avantages, comme la sécurité des threads, la simplicité et la prévisibilité dans l'utilisation des objets.

Voici quelques caractéristiques d'une classe immuable :

- 1.Champs privés et finaux** : Les attributs de la classe sont déclarés comme private et final, ce qui empêche leur modification après la construction de l'objet.
- 2.Pas de méthodes "setter"** : Il n'y a pas de méthodes permettant de modifier les valeurs des attributs de l'objet.
- 3.Constructeur** : L'état de l'objet est défini via un constructeur, qui initialise tous les champs.
- 4.Méthodes de copie** : Si l'objet contient des références à d'autres objets (comme des collections), ces objets doivent être également copiés (deep copy) pour éviter que des modifications sur ces objets n'affectent l'objet immuable.

```

import java.util.Objects;
public final class Player {
    private final String name;
    public Player(String name) { this.name = name; }
    public String getName() { return name; }
    @Override
    public String toString() {
        return name;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Player)) return false;
        Player player = (Player) o;
        return Objects.equals(name, player.name);
    }
    @Override
    public int hashCode() {
        return Objects.hash(name);
    }
}

```

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public final class Team {
    private final String teamName;
    private final List<Player> players;
    public Team(String teamName, List<Player> players) {
        this.teamName = teamName;
        // Création d'une copie profonde de la liste de joueurs
        this.players = new ArrayList<>();
        for (Player player : players) {
            this.players.add(new Player(player.getName())); // Deep
                                                                copy de chaque joueur
        }
    }
    public String getTeamName() { return teamName; }
    public List<Player> getPlayers() {
        // Retourne une nouvelle liste pour éviter la modification
        // de l'originale
        return new ArrayList<>(players);
    }
}

```

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        Player player1 = new Player("Alice");
        Player player2 = new Player("Bob");
        List<Player> playerList = Arrays.asList(player1, player2);
        Team team = new Team("Warriors", playerList);
        // Affichage de l'équipe et des joueurs
        System.out.println("Team: " + team.getTeamName());
        System.out.println("Players: " + team.getPlayers());
        // Tentative de modification
        playerList.get(0).getName(); // Ne modifie pas l'objet original dans Team
        // (Aucune méthode pour changer le nom, car Player est immuable)
        // Test de l'indépendance des objets
        Player player3 = new Player("Charlie");
        playerList.set(0, player3); // Modification de la liste d'origine
        // Vérifions si l'équipe a été affectée
        System.out.println("After modification of original list:");
        System.out.println("Players in team: " + team.getPlayers());
    }
}
```