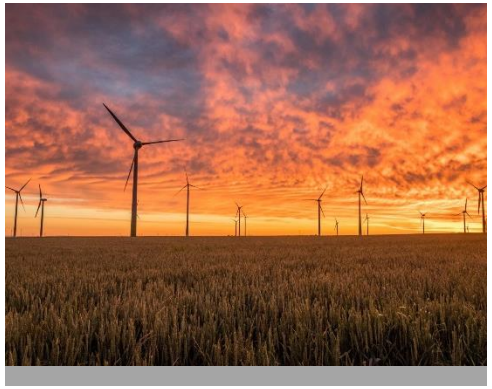
Langage de programmation JAVA

AMMOR Fatimazahra






Exemple Pratique : Gestion de location de vélos

Implémenter un système de gestion de location de vélos pour une entreprise. Il y aura une classe de base pour les vélos avec des classes dérivées qui représenteront les vélos classiques et les vélos électriques.

Besoins :

1- Pouvoir enregistrer les informations sur les vélos

2- Calculer automatiquement le prix de location des vélos pour une durée donnée par l'utilisateur.

JAVA

Les bases

1. Environnement et installation
2. Structure et syntaxe

3. Variables
4. Opérateurs
5. Méthodes

6. Conditions
7. Boucles
8. Tableaux, enums et collections

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. Classe immutable
9. Polymorphisme

Variables (1)

Réservation espace mémoire

Réutilisation de la valeur

JAVA fortement typé, obligation de déclarer

Types primitives des données : boolean, byte (-128 à 127), char, short, int, long, float, double

Déclaration des variables : spécifier le type de la variable, son nom et la valeur par défaut
Nom de la variable : camelCase

Initialisations des variables : permet de donner une valeur par défaut, ça permet d'éviter des erreurs en cours d'exécution.

Suppression des variables : tiens sa portée dans l'endroit où elle a été créé. Lorsqu'on sort de notre méthode ou notre classe où la variable a été déclaré, cette variable n'existe plus. JAVA libère la mémoire et détruit la variable (garbage Collector).

Création de constantes: mot-clé « final ». On ne peut pas changer la valeur (ex: valeur de PI)
Nom de la constante : CONSTANTE_CAS

Opérateurs (2)

Types d'opérateurs :

= : Opérateur d'affectation

➔ âge = 23

+ : Opérateur de concaténation

➔ « nom : » + nom

+ - * / % ( ) : Opérateurs arithmétique

➔ resultat = (2+5)/2;

+=, -=, *=, /=, %= : Opérateurs raccourci

➔ x+=2 (x = x+2)

==, !=, <, <=, >, >=, !(inverse), === (vérification du type aussi) : Opérateurs de comparaison

➔ 14 === « 14 » est « false » même s'ils sont égaux mais non pas le même type

➔ Ou !true (=false)

Package (3)

Pour toutes les classes, prédéfinies ou définies par le développeur, Java a besoin de les organiser. Pour cela, java mets les classes dans des packages.

```
public class NumberPicker {
public static void main(String[] args) {
Random r = new Random(); // DOES
NOT COMPILE
System.out.println(r.nextInt(10));
}
}
```

```
import java.util.Random; // import
tells us where to find Random
public class NumberPicker {
public static void main(String[] args) {
Random r = new Random();
System.out.println(r.nextInt(10)); // a
number 0-9
}
}
```

Si la définition du package est manquante
la classe appartiendra à un package "**par défaut**" et ne sera placée dans aucun dossier de package. Cependant, ce n'est pas une pratique recommandée.
Exemple : *package com.oracle.demos.personnes;*

Classes prédéfinies (4)

java.util

- **ArrayList** : Une implémentation dynamique de tableau qui peut changer de taille.
- **HashMap** : Une table de hachage qui stocke des paires clé-valeur.
- **HashSet** : Une collection qui n'autorise pas les doublons.
- **Scanner** : Permet de lire des données d'entrée, notamment à partir de la console.
- **Date** : Représente une date et une heure. Elle est souvent remplacée par LocalDate et LocalDateTime.
- **Calendar** : Utilisé pour manipuler des dates et des heures.
- **Collections** : Classe utilitaire avec des méthodes statiques pour manipuler ou créer des collections (par ex., sort, shuffle).

java.io

- **File** : Représente un fichier ou un répertoire dans le système de fichiers.
- **FileReader et FileWriter** : Pour lire et écrire dans des fichiers texte.
- **BufferedReader et BufferedWriter** : Améliorent la performance de la lecture et de l'écriture de texte.
- **PrintWriter** : Permet d’écrire des données formatées dans un fichier ou un flux.
- **ObjectInputStream et ObjectOutputStream** : Utilisés pour la sérialisation et la désérialisation d'objets.

Classes prédéfinies (4)

Java.lang

- **String** : La classe pour manipuler des chaînes de caractères (immuables).
- **StringBuilder et StringBuffer** : Utilisées pour la manipulation de chaînes modifiables.
- **Integer, Double, Float, Long, Short, Byte** : Les classes enveloppes pour les types primitifs correspondants.
- **Math** : Fournit des méthodes mathématiques comme sqrt(), abs(), pow(), random().
- **Exception** : La classe de base pour toutes les exceptions vérifiées.
- **RuntimeException** : La classe de base pour toutes les exceptions non vérifiées.
- **IOException** : Gérée lors des opérations d'entrée/sortie.
- **System** : fournit des outils pour interagir avec le système d'exploitation (accéder aux flux d'entrée/sortie, obtenir l'heure système, ou encore gérer des propriétés système).

L'importation du package java.lang est implicite et importé dans chaque programme Java (pas besoin de l'importer manuellement).

java.time

- **LocalDateTime** : Représente une date et une heure.
- **Duration** : Représente une durée de temps.
- **Period** : Représente une période exprimée en années, mois, et jours.

javax.swing

- **JFrame** : Représente une fenêtre.
- **JPanel** : Un conteneur pour organiser d'autres composants.
- **JButton** : Représente un bouton.

Méthodes (5)

Bloc de code qui effectue une tâche spécifique, réutilisable, et est associé à une classe ou à un objet.
Toujours liée à une classe ou un objet.
Voici les éléments clés d'une méthode en Java :

1. **Modificateur d'accès** : Définit la visibilité de la méthode (public, private, protected).
2. **Type de retour** : Le type de donnée que la méthode renvoie (peut être void si elle ne renvoie rien).
3. **Nom de la méthode** : Identifie la méthode, il doit suivre **les conventions de nommage** de Java.
4. **Paramètres** : Liste des paramètres (ou arguments) passés à la méthode, entre parenthèses.
5. **Corps de la méthode** : Le bloc de code qui s'exécute lorsque la méthode est appelée.
6. **Instruction de retour (optionnelle)** : Utilisée pour renvoyer une valeur (**return**), si le type de retour n'est pas void.

Pour appeler une méthode, vous devez utiliser le nom de la méthode et fournir les paramètres requis (s'il y en a). Si la méthode appartient à une instance d'une classe, vous devez d'abord créer une instance (objet) pour y accéder.

Types de Méthodes (5)

Méthodes d'instance : Ces méthodes nécessitent une instance de la classe pour être appelées. Elles peuvent accéder aux variables d'instance et autres méthodes de la classe.
Méthodes static : Ces méthodes appartiennent à la classe plutôt qu'à une instance de la classe. Elles peuvent être appelées sans créer d'instance. Cependant, elles ne peuvent pas accéder directement aux membres d'instance (non-static) de la classe.
Méthodes abstraites : Déclarées dans une classe abstraite ou une interface. Elles n'ont pas de corps et doivent être implémentées par les sous-classes.
Méthodes par défaut (à partir de Java 8) : Utilisées dans les interfaces, elles fournissent une implémentation par défaut pour éviter que toutes les classes implémentant l'interface aient à définir cette méthode.

Méthodes (5)

Surcharge de méthodes

La **surcharge de méthodes** en Java permet de définir plusieurs méthodes avec le même nom mais des signatures différentes dans une même classe. Cela signifie que ces méthodes doivent différer par :

1. **Le nombre de paramètres**
2. **Le type de paramètres**
3. **L'ordre des paramètres** (si les types diffèrent)

En revanche, le type de retour **ne suffit pas** à distinguer deux méthodes. Il doit y avoir une différence dans les paramètres.

La surcharge permet d'améliorer la lisibilité du code en utilisant un même nom de méthode pour des tâches similaires, mais avec des variations dans les paramètres d'entrée.

```
// Méthode additionner avec deux entiers
    public int additionner(int a, int b) {
        return a + b;
    }
```

```
 // Surcharge : additionner avec trois entiers
    public int additionner(int a, int b, int c) {
        return a + b + c;
    }
```

```
// Surcharge : additionner avec deux nombres flottants
    public double additionner(double a, double b) {
        return a + b;
    }
```

Méthodes (5)

Redéfinition de méthode

Dans le contexte de l'héritage, la **redéfinition de méthode** (ou **overriding**) signifie que la sous-classe peut fournir sa propre implémentation d'une méthode héritée de la superclasse. Cela permet à la sous-classe de modifier le comportement d'une méthode existante pour l'adapter à ses propres besoins.

Différence entre surcharge et redéfinition :

- **Surcharge (overloading)** : C'est lorsque plusieurs méthodes dans une même classe ont le **même nom** mais des **signatures différentes** (nombre ou type de paramètres).
- **Redéfinition (overriding)** : C'est lorsque la sous-classe fournit une nouvelle implémentation d'une méthode existante de la superclasse avec **la même signature** (même nom, même paramètres).

En Java, il est recommandé d'utiliser l'annotation **@Override** lorsque vous redéfinissez une méthode. Cela permet de vérifier que la méthode dans la sous-classe a bien la même signature que celle de la superclasse, et que vous ne créez pas par inadvertance une méthode différente.

Méthodes (5)

Caractéristique	Surcharge (Overloading)	Redéfinition (Overriding)
Classe	Dans la même classe	Dans une sous-classe
Nom de la méthode	Doit être identique	Doit être identique
Signature des paramètres	Doit être différente (nombre, types ou ordre des paramètres)	Doit être exactement la même
Type de retour	Peut être différent	Doit être identique (ou covariant pour les objets)
Mot-clé `@Override`	Pas nécessaire	Utilisé pour indiquer qu'une méthode est redéfinie
Utilisation de `super`	Pas nécessaire	Peut être utilisé pour appeler la méthode de la superclasse

Accessibilité (6)

Un agent de sécurité pour les classes
Public : Visible pour toute autre classe
protected : Visible pour les classes du même package ou pour les sous-classes
Package ou package-private (accès par défaut) : Visible uniquement pour les classes dans le même package
private : Visible uniquement au sein de la même classe

Lieu / Niveau	Dans la Classe	Classe Fille (même package)	Classe Fille (autre package)	Même package	Autre package
Public	OUI	OUI	OUI	OUI	OUI
Protected	OUI	OUI	OUI	OUI	NON
Private	OUI	NON	NON	NON	NON
Package	OUI	OUI	NON	OUI	NON

```
public class ParkTrip {
        public void skip1() {}
        default void skip2() {} // DOES NOT COMPILE
        void public skip3() {} // DOES NOT COMPILE
        void skip4() {}
}
```

Exemple Pratique : Gestion de location de vélos

1. Créez un projet avec deux packages :
   - com.gestion.velos
   - com.gestion.gestionvelos

2. Dans le package com.gestion.velos : classe *Velo* :
   - Les variables nom (String), anneeFabrication (int) et prixMarche (double) et prixLocation (double).
   - Un constructeur pour initialiser ces variables.
   - Une méthode afficherDetails() qui sera redéfinie par les classes dérivées.
   - Une méthode calculerPrixLocation(int duree) qui calcule le prix de location pour une durée spécifiée par le client.

3. Dans le package com.gestion.velos, créez de sous-classes VeloClassique et VeloElectrique :
   - VeloClassique aura un champ spécifique typeVelo (String).
   - VeloElectrique aura un champ spécifique vitesse (int).
   - Implémentez la méthode afficherDetails() dans chaque classe pour afficher les détails spécifiques à chaque vélo.
   - Utilisez l'héritage pour réutiliser les membres de la classe Velo.

Exemple Pratique : Gestion de location de vélos

4. Dans le package com.gestion.gestionvelos, créez une classe GestionVelos avec la méthode main(String[] args) :

- **Créez des instances de VeloClassique et VeloElectrique, initialisez-les avec des valeurs.**
- **Affichez leurs détails en utilisant la méthode afficherDetails().**
- **Demandez à l'utilisateur de spécifier la durée de location puis calculer et afficher le prix de location à payer.**

```
Int duree = 0;
// Créer un objet Scanner pour lire l'entrée utilisateur
Scanner scanner = new Scanner(System.in);
// Demander à l'utilisateur d'entrer une valeur pour le nombre de jours de location
System.out.println("Veuillez entrer la duree de location pour votre vélo : ");
duree = scanner.nextInt();  // Lire un entier
```

Classe abstraite Vs Interface (7)

Classe abstraite : Une classe abstraite est une classe qui ne peut pas être instanciée directement. Elle peut contenir des méthodes abstraites (sans corps) et des méthodes concrètes (avec une implémentation). Une classe qui hérite d'une classe abstraite doit soit implémenter toutes ses méthodes abstraites, soit être déclarée elle-même comme abstraite.

Interface : Une interface est un contrat qui définit un ensemble de méthodes que les classes doivent implémenter. Par défaut, les méthodes dans une interface sont abstraites (avant Java 8), mais à partir de Java 8, les interfaces peuvent aussi avoir des **méthodes par défaut** (avec une implémentation) et des **méthodes statiques**.

Classe abstraite Vs Interface (7)

Utilise le mot-clé *abstract* pour définir une classe abstraite et pour déclarer des méthodes abstraites.
Une classe ne peut hériter que d'une seule classe abstraite
Peut avoir des variables d'instance, des variables statiques, des constantes, et ces variables peuvent avoir n'importe quelle visibilité (private, protected, public)
Peut avoir des constructeurs, mais ils ne peuvent être appelés que par les sous-classes lors de la construction d'objets.

Utilise le mot-clé *interface* pour déclarer une interface.
Une classe peut implémenter plusieurs interfaces (héritage multiple via les interfaces). Cela permet à une classe d'adhérer à plusieurs contrats en même temps.
Toutes les variables déclarées dans une interface sont implicitement public, static, et final. Ce sont essentiellement des constantes.
Ne peut pas avoir de constructeurs, car elles ne sont pas censées être instanciées directement.

Quand utiliser? Classe abstraite Vs Interface (7)

Classe abstraite :

- Utilisez une **classe abstraite** lorsque vous souhaitez fournir une implémentation partielle ou lorsque vous voulez partager des comportements communs entre plusieurs classes tout en laissant certaines méthodes à implémenter.
- Une classe abstraite est appropriée lorsque les classes dérivées partagent un état commun (avec des variables d'instance).

Interface :

- Utilisez une **interface** lorsque vous voulez définir un contrat que plusieurs classes, même sans lien hiérarchique, doivent respecter. Cela permet à différentes classes d'implémenter un même ensemble de comportements, même si elles ne partagent pas de base commune.
- Les interfaces sont idéales lorsque vous avez besoin de **l'héritage multiple**.

Quand utiliser? Classe abstraite Vs Interface (7)

Caractéristique	Classe Abstraite	Interface
Héritage	Une seule classe abstraite peut être héritée	Une classe peut implémenter plusieurs interfaces
Constructeurs	Peut avoir des constructeurs	Pas de constructeurs (car non instanciable)
Méthodes	Peut contenir des méthodes abstraites et concrètes	Avant Java 8 : uniquement abstraites; depuis Java 8 : méthodes abstraites, par défaut et statiques
Variables	Peut avoir des variables d'instance	Seulement des constantes (static, final)
Modificateurs d'accès	Méthodes et variables peuvent avoir n'importe quel modificateur	Méthodes et variables sont `public` par défaut

Exemple Pratique : Gestion de location de vélos

1. Créez un projet avec deux packages :
   - com.gestion.velos
   - com.gestion.gestionvelos

2. Dans le package com.gestion.velos : la classe abstraite *Velo* :
   - Les variables nom (String), anneeFabrication (int) et prixMarche (double) et prixLocation (double).
   - Un constructeur pour initialiser ces variables.
   - Une méthode abstraite afficherDetails() qui sera redéfinie par les classes dérivées.
   - Une méthode calculerPrixLocation(int duree) qui calcule le prix de location pour une durée spécifiée par le client.

3. Dans le package com.gestion.velos, créez de sous-classes VeloClassique et VeloElectrique :
   - VeloClassique aura un champ spécifique typeVelo (String).
   - VeloElectrique aura un champ spécifique vitesse (int).
   - Implémentez la méthode afficherDetails() dans chaque classe pour afficher les détails spécifiques à chaque vélo.
   - Utilisez l'héritage pour réutiliser les membres de la classe Velo.

Exemple Pratique : Gestion de location de vélos

4. Dans le package com.gestion. gestionvelos, créez une classe GestionVelos avec la méthode main(String[] args) :

- **Créez des instances de VeloClassique et VeloElectrique, initialisez-les avec des valeurs.**
- **Affichez leurs détails en utilisant la méthode afficherDetails().**
- **Demandez à l'utilisateur de spécifier la durée de location puis calculer et afficher le prix de location à payer.**

```
Int duree = 0;
// Créer un objet Scanner pour lire l'entrée utilisateur
Scanner scanner = new Scanner(System.in);
// Demander à l'utilisateur d'entrer une valeur pour le nombre de jours de location
System.out.println("Veuillez entrer la duree de location pour votre vélo : ");
duree = scanner.nextInt();  // Lire un entier
```

```
package com.gestion.velos;

Public abstract class Velo {
    protected String nom;
    protected double prixMarche;
    protected int anneeFabrication;
    protected double prixLocationParJour;

    public Velo(String nom, double prixMarche, int
anneeFabrication, double prixLocationParJour) {
        this. nom = nom;
        this. prixMarche = prixMarche;
        this. anneeFabrication = anneeFabrication;
        this.prixLocationParJour = prixLocationParJour;
    }

    public abstract void afficherDetails();

    public double calculerCoutLocation(int jours) {
        return prixLocationParJour * jours;
    }
}
```

```
package com.gestion.velos;

public class VeloClassique extends Velo {
   private String typeVelo;

   public VeloClassique(String nom, double prixMarche,
int anneeFabrication, double prixLocationParJour, String
typeVelo) {
      super(nom, prixMarche, anneeFabrication,
            prixLocationParJour);
      this. typeVelo = typeVelo;
   }

   @Override
   public void afficherDetails() {
            super.afficherDetails();
            System.out.println(« Type velo : " + typeVelo);
}
}
```

```
package com.gestion.velos;

public class VeloElectrique extends Velo {
   private double vitesse;

   public VeloElectrique(String nom, double prixMarche, int
anneeFabrication, double prixLocationParJour, double vitesse) {
            super(nom, prixMarche, anneeFabrication,
            prixLocationParJour);
            this. vitesse = vitesse;
   }

   @Override
   public void afficherDetails() {
      super.afficherDetails();
      System.out.println(« Vitesse: " +     vitesse);
   }
}
```

```
package com.gestion.gestionvelos;

import com.gestion.velos.VeloClassique;
import com.gestion.vehicules.VeloElectrique;

public class GestionVelo {
    public static void main(String[] args) {
        // Création d’un velo classique
 VeloClassique velo1 = new
VeloClassique(« VTT",5000.00, 2020, 200.0);

        // Création d'un Velo Electrique
 VeloElectrique velo2 = new VeloElectrique("xxx",
15000.00, 2018, 1000.0);

       // Créer un objet Scanner pour lire l'entrée
                utilisateur,
      // déclarer une variable pour stocker la durée
        Scanner scanner = new Scanner(System.in);
        Int joursLocation = 0;
```

```
        // Afficher les détails du velo classique
        System.out.println("Détails du velo classique :");
        velo1.afficherDetails();
        // Demander à l'utilisateur d'entrer une valeur pour le
nombre de jours de location
        System.out.println("Veuillez entrer le nombre de jours
                de location du velo classique: ");
        joursLocation = scanner.nextInt();  // Lire un entier
        System.out.println("Coût location pour »+
joursLocation+ « jours : " +
velo1.calculerCoutLocation(joursLocation) + " dh\n");

        // Afficher les détails du velo electrique
        System.out.println("Détails du velo electrique :");
        velo2.afficherDetails();
        // Demander à l'utilisateur d'entrer une valeur pour le
                nombre de jours de location
        System.out.println("Veuillez entrer le nombre de jours
                de location du velo electrique : ");
        joursLocation = scanner.nextInt();  // Lire un entier
        System.out.println("Coût location pour »+
joursLocation+« jours : " +
velo2.calculerCoutLocation(joursLocation) + " dh");
    }
}
```

Exemple Pratique : Gestion d’une bibliothèque

Gestion de livres dans une bibliothèque : Gérer les informations des livres (livres classiques et magazines), les emprunts, et les retours.

Besoins :

1- Pouvoir enregistrer les livres disponibles et afficher les informations lorsqu’on en a besoin

2- Implémenter les méthodes pour vérifier la disponibilité des livres

Conditions (6)

Les **conditions en Java** permettent de contrôler l'exécution du code en fonction de certaines expressions booléennes. Java offre plusieurs structures pour gérer les conditions, comme les instructions if, else if, else, et l'instruction switch. Ces structures permettent de prendre des décisions au sein du programme en exécutant différents blocs de code en fonction de la valeur d'une condition donnée.

```
int age = 18;
if (age >= 18) {
    System.out.println("Vous
êtes majeur.");
}
```

```
int note = 75; if (note >= 90) {
System.out.println("Excellent"); }
else if (note >= 75) {
System.out.println("Bien"); } else
if (note >= 50) {
System.out.println("Passable"); }
else {
System.out.println("Insuffisant")
; }
```

```
int age = 20;
String resultat = (age >= 18) ? "Majeur" : "Mineur";
System.out.println(resultat);  // Affiche "Majeur"
```

Conditions (6)

Les **conditions en Java** permettent de contrôler l'exécution du code en fonction de certaines expressions booléennes. Java offre plusieurs structures pour gérer les conditions, comme les instructions if, else if, else, et l'instruction switch. Ces structures permettent de prendre des décisions au sein du programme en exécutant différents blocs de code en fonction de la valeur d'une condition donnée.

```
int jour = 3;

switch (jour) {
    case 1:
        System.out.println("Lundi");
        break;
    case 2:
        System.out.println("Mardi");
        break;
    case 3:
        System.out.println("Mercredi");
        break;
    case 4:
        System.out.println("Jeudi");
        break;
    case 5:
        System.out.println("Vendredi");
        break;
    case 6:
        System.out.println("Samedi");
        break;
    case 7:
        System.out.println("Dimanche");
        break;
    default:
        System.out.println("Jour invalide");
}
```

1- Classe Livre (classe de base) :

- Variables :
  - titre (chaîne de caractères) : le titre du livre.
  - auteur (chaîne de caractères) : l'auteur du livre.
  - anneePublication (entier) : l'année de publication.
  - disponible (booléen) : si le livre est disponible ou non.
- Méthodes :
  - emprunter() : Change l'état de disponibilité du livre à false.
  - retourner() : Change l'état de disponibilité du livre à true.
  - afficherDetails() : Affiche les informations du livre.

2. Classe LivreClassique (classe dérivée de Livre) :

- Variables supplémentaires :
  - genre (chaîne de caractères) : Le genre du livre (Science-fiction, Roman policier, etc.).
- Méthodes :
  - Redefinir la méthode afficherDetails() pour inclure le genre du livre.

3. Classe Magazine (classe dérivée de Livre) :

- Variables supplémentaires :
  - numero (entier) : Numéro de l'édition du magazine.
  - moisPublication (chaîne de caractères) : Mois de publication.
- Méthodes :
  - Redéfinir la méthode afficherDetails() pour afficher les informations spécifiques du magazine.

4. Classe GestionBibliotheque :

- Contient la méthode main qui :
  - Crée plusieurs instances de LivreRoman et Magazine.
  - Affiche les détails de chaque livre.
  - Gère les emprunts et les retours de livres.

```
package gestion.bibliotheque;

public class Livre {
    protected String titre;
    protected String auteur;
    protected int anneePublication;
    protected boolean disponible;

    public Livre(String titre, String auteur, int
anneePublication) {
        this.titre = titre;
        this.auteur = auteur;
        this.anneePublication = anneePublication;
        this.disponible = true;  // Le livre est
disponible par défaut
    }
```

```
    public void emprunter() {
        if (disponible) {
            disponible = false;
            System.out.println(titre + " a été emprunté.");
        } else {
            System.out.println(titre + " n'est pas disponible.");
        }
    }

    public void retourner() {
        disponible = true;
        System.out.println(titre + " a été retourné à la
bibliothèque.");
    }

    public void afficherDetails() {
        System.out.println("Titre : " + titre + ", Auteur : " + auteur
+ ", Année : " + anneePublication);
        System.out.println("Disponible : " + (disponible ? "Oui" :
"Non"));
    }
}
```

```
package gestion.bibliotheque;

public class LivreClassique extends Livre {
    private String genre;

    public LivreClassique(String titre, String auteur,
int anneePublication, String genre) {
        super(titre, auteur, anneePublication);
        this.genre = genre;
    }

    @Override
    public void afficherDetails() {
        super.afficherDetails();
        System.out.println("Genre : " + genre);
    }
}
```

```
package gestion.bibliotheque;

public class Magazine extends Livre {
    private int numero;
    private String moisPublication;

    public Magazine(String titre, String auteur, int
anneePublication, int numero, String moisPublication) {
        super(titre, auteur, anneePublication);
        this.numero = numero;
        this.moisPublication = moisPublication;
    }

    @Override
    public void afficherDetails() {
        super.afficherDetails();
        System.out.println("Numéro : " + numero + ", Mois de
publication : " + moisPublication);
    }
}
```

```
package gestion;

import gestion.bibliotheque.LivreRoman;
import gestion.bibliotheque.Magazine;

public class GestionBibliotheque {
    public static void main(String[] args) {
        // Création de livres
 LivreClassique roman1 = new LivreClassique("Le
Seigneur des Anneaux", "J.R.R. Tolkien", 1954,
"Fantasy");
 LivreClassique roman2 = new
LivreClassique("1984", "George Orwell", 1949,
"Science-fiction");

        // Création de magazines
        Magazine mag1 = new Magazine("National
Geographic", "Divers", 2023, 254, "Octobre");
        Magazine mag2 = new Magazine("Science &
Vie", "Divers", 2023, 892, "Septembre");
```

```
 // Afficher les détails des livres
        System.out.println("Détails du premier roman :");
        roman1.afficherDetails();
        System.out.println("\nDétails du deuxième roman :");
        roman2.afficherDetails();

        // Emprunter un livre
        roman1.emprunter();
        roman1.afficherDetails();  // Vérifier disponibilité après
emprunt

        // Retourner le livre
        roman1.retourner();
        roman1.afficherDetails();  // Vérifier disponibilité après
retour

        // Afficher les détails des magazines
        System.out.println("\nDétails du premier magazine :");
        mag1.afficherDetails();
        System.out.println("\nDétails du deuxième magazine :");
        mag2.afficherDetails();
    }
}
```

Exemple Pratique : Gestion de location de voitures

Société qui a plusieurs voitures et motos à louer

Besoins :

1- Pouvoir enregistrer les voitures disponibles et afficher les informations lorsqu'on en a besoin

2- Calculer automatiquement le prix de location de chaque voiture/moto en prenant en compte la durée souhaitée par le client

Exemple Pratique : Gestion de location de voitures

1.Créez un projet avec deux packages :
- com.location.vehicules
- com.location.main

2.Dans le package com.location.vehicules : classe *Vehicule* :
- Les variables sont privées : marque (String), modele (String), et prixParJour (double).
- Un constructeur pour initialiser ces variables.
- Une méthode afficherDetails() qui sera redéfinie par les classes dérivées.
- Une méthode qui calcule le prix de location en ayant comme paramètre la durée souhaitée par le client
- Des accesseurs (getters) pour accéder aux variables.

3.Dans le package com.location.vehicules, créez de sous-classes Voiture et Moto :
- Voiture aura un champ spécifique nombreDePortes (int).
- Moto aura un champ spécifique cylindree (int).
- Implémentez la méthode afficherDetails() dans chaque classe pour afficher les détails spécifiques à chaque véhicule.
- Utilisez l'héritage pour réutiliser les membres de la classe Vehicule.

Exemple Pratique : Gestion de location de voitures

4. Dans le package com.location.main, créez une classe Main avec la méthode main(String[] args) :

- **Créez des instances de Voiture et Moto, initialisez-les avec des valeurs.**
- **Affichez leurs détails en utilisant la méthode afficherDetails().**
- **Calculez et affichez le coût total de la location pour un nombre donné de jours en utilisant l'opérateur de multiplication sur prixParJour.**

```java
package com.location.vehicules;

public abstract class Vehicule {
    protected String marque;
    protected String modele;
    protected double prixParJour;

    public Vehicule(String marque, String modele, double prixParJour) {
        this.marque = marque;
        this.modele = modele;
        this.prixParJour = prixParJour;
    }

    public abstract void afficherDetails();

    public double getPrixParJour() {
        return prixParJour;
    }
}
```

```java
package com.location.vehicules;

public class Voiture extends Vehicule {
    private int nombreDePortes;

    public Voiture(String marque, String modele, double
prixParJour, int nombreDePortes) {
        super(marque, modele, prixParJour);
        this.nombreDePortes = nombreDePortes;
    }

    @Override
    public void afficherDetails() {
        System.out.println("Voiture: " + marque + " " + modele + ", "
+ nombreDePortes + " portes, " + prixParJour + "€/jour");
    }
}
```

```
package com.location.vehicules;

public class Moto extends Vehicule {
    private int cylindree;

    public Moto(String marque, String modele,
double prixParJour, int cylindree) {
        super(marque, modele, prixParJour);
        this.cylindree = cylindree;
    }

    @Override
    public void afficherDetails() {
        System.out.println("Moto: " + marque + "
" + modele + ", " + cylindree + "cc, " +
prixParJour + "€/jour");
    }
}
```

```
package com.location.main;

import com.location.vehicules.*;

public class Main {
    public static void main(String[] args) {
        Voiture voiture = new Voiture("Toyota", "Corolla", 50, 4);
        Moto moto = new Moto("Yamaha", "MT-07", 30, 700);

        voiture.afficherDetails();
        moto.afficherDetails();

         Scanner scanner = new Scanner(System.in);  // Créer un objet
                Scanner pour lire l'entrée utilisateur

        // Demander à l'utilisateur d'entrer une valeur pour le nombre de           jours de location
        System.out.print("Veuillez entrer le nombre de jours de location : ");
        int joursLocation = scanner.nextInt();  // Lire un entier

        double coutVoiture = voiture.getPrixParJour() * joursLocation;
        double coutMoto = moto.getPrixParJour() * joursLocation;

        System.out.println("Coût de location de la voiture pour " +      joursLocation + " jours : " +
coutVoiture + "€");
        System.out.println("Coût de location de la moto pour " +        joursLocation + " jours : " +
coutMoto + "€");
    }
}
```