

JAVA OBJECT-ORIENTED PROGRAMMING (OOP)



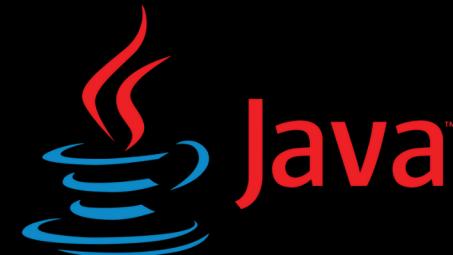
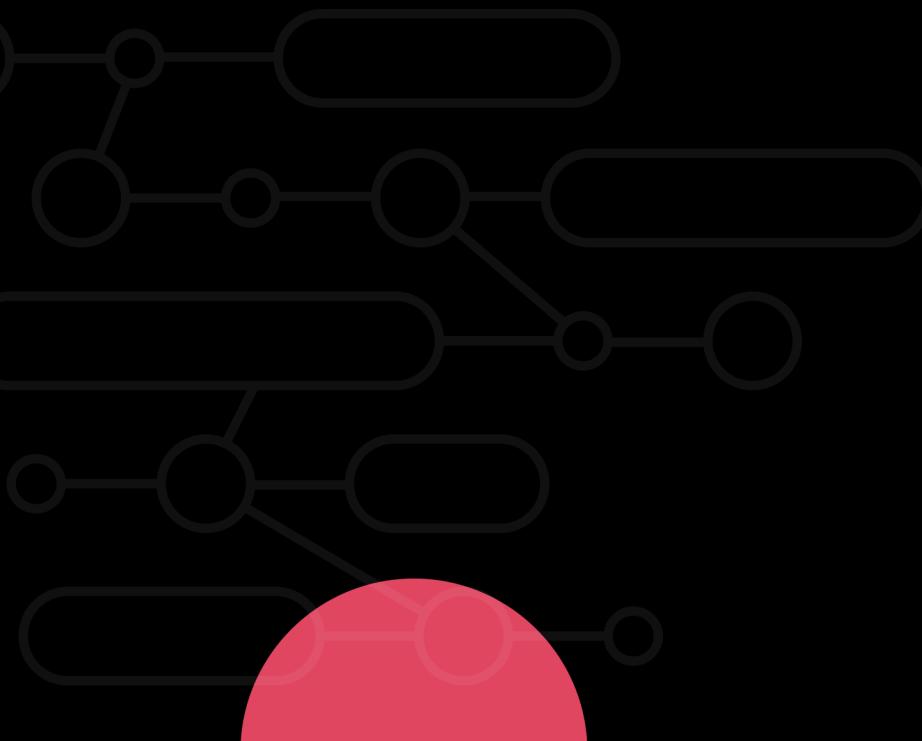


PROGRAMMATION ORIENTÉE OBJET (OOP) EN JAVA

La Programmation Orientée Objet (OOP) est un paradigme de programmation qui repose sur le concept d'objets. Ces objets sont des instances de classes, et la programmation orientée objet favorise l'organisation du code de manière modulaire et la réutilisation du code.

les principaux concepts de la POO en Java :

1. *Classes*
2. *Objets*
3. *Encapsulation*
4. *Héritage*
5. *Polymorphisme*
6. *Abstraction*



CLASSES & OBJECT





CLASSES ET OBJETS



Une classe est un modèle ou un plan pour créer des objets. Les objets sont des instances de classes. Une classe définit les caractéristiques (attributs) et les comportements (méthodes) que les objets créés à partir de cette classe auront.

Définition d'une Classe :

Une classe en Java est déclarée en utilisant le mot-clé **class**

Exemple

```
// Définition d'une classe
public class Personne {
    // Attributs (variables d'instance)
    String nom;
    int age;

    // Méthode (comportement)
    public void direBonjour() {
        System.out.println("Bonjour, je m'appelle " + nom + " et j'ai "
    }
}
```



CLASSES ET OBJETS



Création d'Objets (Instanciation) :

Une fois qu'une classe est définie, des objets peuvent être créés à partir de cette classe. Cela s'appelle l'instanciation.

```
public class Main {  
    public static void main(String[] args) {  
        // Création d'un objet de la classe Personne  
        Personne personnel = new Personne();  
  
        // Initialisation des attributs  
        personnel.nom = "Alice";  
        personnel.age = 25;  
  
        // Appel de la méthode  
        personnel.direBonjour();  
    }  
}
```

Exemple

JAVA CLASS ATTRIBUTES





JAVA CLASS ATTRIBUTES



Les attributs de classe en Java, également appelés variables de classe ou champs de classe, représentent les données ou les états que chaque objet de la classe peut avoir. Ces attributs définissent les caractéristiques des objets créés à partir de la classe.

Déclaration d'Attributs de Classe :

Les attributs de classe sont définis à l'intérieur de la classe mais à l'extérieur des méthodes. Ils peuvent être accompagnés de modificateurs d'accès (public, private, protected, ou aucun) pour définir leur visibilité.

Exemple

```
public class MaClasse {  
    // Attributs de classe  
    public int attributPublic;  
    private String attributPrive;  
    protected double attributProtege;  
    static int attributStatique;  
  
    // ... autres membres de la classe ...  
}
```



JAVA CLASS ATTRIBUTES

Déclaration d'Attributs de Classe :

- **attributPublic**: Un attribut accessible publiquement depuis n'importe quelle classe.
- **attributPrive**: Un attribut accessible uniquement à l'intérieur de la classe.
- **attributProtege**: Un attribut accessible à l'intérieur de la classe et par ses sous-classes.
- **attributStatique**: Un attribut statique partagé par toutes les instances de la classe.



CLASSES ET OBJETS



Accès aux **Attributs de Classe**:

Les attributs de classe peuvent être accessibles et modifiés à l'aide de méthodes publiques appelées accesseurs (**getters**) et mutateurs (**setters**). Cela permet de contrôler l'accès aux données de la classe.

```
public class MaClasse {  
    private int attributPrive;  
  
    // Accesseur (Getter)  
    public int getAttributPrive() {  
        return attributPrive;  
    }  
  
    // Mutateur (Setter)  
    public void setAttributPrive(int valeur) {  
        attributPrive = valeur;  
    }  
}
```

Exemple



CLASSES ET OBJETS



Utilisation des **Attributs de Classe**:

Une fois que vous avez créé une instance de la classe, vous pouvez accéder et manipuler ses attributs à l'aide de l'opérateur dot (.).

Exemple

Dans cet exemple,
setAttributPrive est utilisé pour modifier la valeur de **attributPrive**, et **getAttributPrive** est utilisé pour obtenir sa valeur.

```
public class UtilisationAttributs {  
    public static void main(String[] args) {  
        MaClasse objet = new MaClasse();  
  
        // Utilisation des accesseurs et mutateurs  
        objet.setAttributPrive(42);  
        int valeur = objet.getAttributPrive();  
  
        System.out.println("Valeur de l'attribut privé : " + valeur);  
    }  
}
```



CONSTRUCTEUR





CONSTRUCTEUR



Un constructeur en Java est une méthode spéciale qui est appelée lorsqu'un objet est créé à l'aide de l'opérateur **new**. Il est utilisé pour initialiser les propriétés d'un objet et peut avoir des paramètres permettant de spécifier des valeurs initiales.

1. Nom du Constructeur :

- Le nom d'un constructeur doit être identique au nom de la classe dans laquelle il est déclaré.

2. Pas de Type de Retour :

- Un constructeur n'a pas de type de retour, même pas **void**.

3. Utilisation de this :

- Le mot-clé **this** est utilisé pour faire référence à l'instance actuelle de la classe. Il est souvent utilisé pour différencier les variables d'instance des paramètres du constructeur.



CONSTRUCTEUR



4. **Types de Constructeurs :**

a. **Constructeur par Défaut (Default Constructor) :**

- Un constructeur sans paramètres est appelé constructeur par défaut.
- Si aucun constructeur n'est défini dans la classe, Java fournit automatiquement un constructeur par défaut.

```
public class MaClasse {  
    // Constructeur par défaut généré automatiquement  
    public MaClasse() {  
        // Code d'initialisation par défaut  
    }  
}
```



CONSTRUCTEUR



4. Types de Constructeurs :

b. Constructeur Paramétré :

- Un constructeur avec des paramètres permet d'initialiser les propriétés de l'objet avec des valeurs spécifiques.

```
public class MaClasse {  
    private int valeur;  
  
    // Constructeur copieur  
    public MaClasse(MaClasse autreObjet) {  
        this.valeur = autreObjet.valeur;  
    }  
}
```

MODIFICATEURS D'ACCÈS



MODIFICATEURS D'ACCÈS



Les modificateurs (modifiers) en Java sont des mots-clés qui peuvent être ajoutés aux définitions de classes, de méthodes, de variables et d'autres éléments du code Java pour spécifier des comportements particuliers, des niveaux d'accès ou des caractéristiques spécifiques. Les modificateurs peuvent être divisés en plusieurs catégories, notamment les modificateurs d'accès, les modificateurs non d'accès, et les modificateurs spécifiques à certaines déclarations.

public : La déclaration est accessible de partout.

```
public class MyClass {  
    }
```

private : La déclaration est accessible uniquement au sein de la classe déclarante.

```
private int myVar;
```



MODIFICATEURS D'ACCÈS



protected : La déclaration est accessible dans le même paquet et par des sous-classes, même si elles sont dans un autre paquet.

```
protected void myMethod() {  
}
```

La (pas de modificateur) (Default) : La déclaration est accessible uniquement dans le même paquet (par défaut).

```
class PackagePrivateClass {  
}
```



ENCAPSULATION



ENCAPSULATION

L'encapsulation est l'un des quatre concepts fondamentaux de la programmation orientée objet (POO), les autres étant l'héritage, le polymorphisme, et l'abstraction. L'encapsulation consiste à regrouper les données (variables) et les méthodes (fonctions) qui les manipulent au sein d'une même unité, appelée classe. L'objectif principal de l'encapsulation est de restreindre l'accès aux détails internes d'un objet et de protéger l'intégrité des données.

Principes de l'Encapsulation en Java :

Variables d'Instance Privées : Les variables d'instance (attributs) d'une classe doivent être déclarées comme privées (`private`) pour les rendre inaccessibles depuis l'extérieur de la classe.

```
public class ExempleEncapsulation {  
    private int age; // Variable d'instance privée  
  
    // Méthode pour accéder à la variable d'instance (getter)  
    public int getAge() {  
        return age;  
    }  
  
    // Méthode pour modifier la variable d'instance (setter)  
    public void setAge(int nouvelAge) {  
        if (nouvelAge > 0) {  
            age = nouvelAge;  
        } else {  
            System.out.println("L'âge doit être positif.");  
        }  
    }  
}
```



ENCAPSULATION



Principes de l'Encapsulation en Java :

Méthodes d'Accès (Getters et Setters) : Pour permettre l'accès aux variables privées, on utilise des méthodes d'accès, souvent appelées getters (pour obtenir la valeur) et setters (pour modifier la valeur).

```
ExempleEncapsulation objet = new ExempleEncapsulation();
objet.setAge(25); // Utilisation du setter pour définir l'âge
int ageActuel = objet.getAge(); // Utilisation du getter pour obtenir :
```

Encapsulation des Complexités : En encapsulant les données et en fournissant des méthodes d'accès, une classe peut masquer la complexité de sa mise en œuvre interne. Cela facilite la maintenance du code et permet de modifier l'implémentation sans affecter le code client.



ENCAPSULATION



Méthodes Getter et Setter:

Les méthodes getters et setters sont utilisées pour accéder et modifier les attributs privés d'une classe de manière contrôlée. Elles agissent comme une interface publique à travers laquelle les interactions avec les données encapsulées peuvent être effectuées.

Méthodes Getter :

```
public TypeRetour getNomAttribut() {  
    return nomAttribut;  
}
```

Exemple

```
public class Personne {  
    private String nom;  
  
    public String getNom() {  
        return nom;  
    }
```



ENCAPSULATION



Méthodes Getter et Setter:

Méthodes Setter

```
public void setNomAttribut(TypeDonnee nouvelleValeur) {  
    // Logique de validation optionnelle  
    nomAttribut = nouvelleValeur;  
}
```

Exemple

```
public class Personne {  
  
    private int age;  
  
    public void setAge(int nouvelAge) {  
        if (nouvelAge >= 0) {  
            age = nouvelAge;  
        }  
    }  
}
```

HÉRITAGE





HÉRITAGE

L'héritage permet à une classe appelée "classe fille" ou "sous-classe" d'hériter des propriétés et comportements d'une autre classe appelée "classe mère" ou "superclasse". La classe fille peut alors étendre ou spécialiser les fonctionnalités de la classe mère.

Exemple

Classe Mère (Superclasse) :

```
public class Animal {  
    protected String nom;  
  
    public Animal(String nom) {  
        this.nom = nom;  
    }  
  
    public void manger() {  
        System.out.println(nom + " mange.");  
    }  
}
```

Classe Fille (Sous-classe)

```
public class Chien extends Animal {  
    public Chien(String nom) {  
        super(nom); // Appel au constructeur de la classe mère  
    }  
  
    public void aboyer() {  
        System.out.println(nom + " aboie.");  
    }  
}
```



HÉRITAGE



Utilisation de l'Héritage :

1. La classe Chien hérite de la classe Animal.
2. Le constructeur de la classe fille utilise `super(nom)` pour appeler le constructeur de la classe mère.
3. La classe Chien hérite de la méthode `manger()` de la classe Animal.
4. La classe Chien ajoute sa propre méthode `aboyer()`.

Avantages de l'Héritage :

- **Réutilisation du Code :**

Économise le temps de développement en réutilisant les fonctionnalités existantes.

- **Hiérarchie Logique :**

Crée une hiérarchie logique entre les classes, facilitant la compréhension du code.

- **Maintenance Facilitée :**

Les modifications apportées à la classe mère sont automatiquement reflétées dans les classes filles.

- **Polymorphisme :**

Permet l'utilisation de polynômes pour traiter des objets de différentes sous-classes de manière uniforme.



POLYMORPHISME





POLYMERPHISME

Le polymorphisme est un concept fondamental de la Programmation Orientée Objet (POO) qui permet à un même nom (méthode ou opérateur) d'avoir différents comportements en fonction du contexte. Il se décline en deux formes principales : le surcharge (overloading) et le remplacement (overriding).

Surcharge (Overloading)

Dans cet exemple, la classe `Calculateur` surcharge la méthode **additionner** avec des signatures différentes (types `int` et `double`).

```
public class Calculateur {  
    public int additionner(int a, int b) {  
        return a + b;  
    }  
  
    public double additionner(double a, double b) {  
        return a + b;  
    }  
  
    public String concatener(String a, String b) {  
        return a + b;  
    }  
}
```



POLYMORPHISME



Redefinition (Overriding)

Dans cet exemple, la classe Chien remplace la méthode faireDuBruit de la classe Animal pour définir un comportement spécifique.

```
public class Animal {  
    public void faireDuBruit() {  
        System.out.println("L'animal fait un bruit.");  
    }  
  
    public class Chien extends Animal {  
        @Override  
        public void faireDuBruit() {  
            System.out.println("Le chien aboie.");  
        }  
    }  
}
```



POLYMORPHISME



Avantages du Polymorphisme :

- **Flexibilité :**
 - Permet de traiter des objets de différentes classes de manière uniforme .
- **Extensibilité :**
 - Facilite l'ajout de nouvelles fonctionnalités sans modifier le code existant .
- **Réutilisation du Code :**
 - Encourage la réutilisation du code en permettant des méthodes avec le même nom .
- **Compréhension du Code :**
 - Améliore la lisibilité du code en permettant des noms de méthodes cohérents .

ABSTRACTION





ABSTRACTION

L'abstraction est l'un des piliers fondamentaux de la Programmation Orientée Objet (POO) en Java. Elle consiste à représenter les caractéristiques essentielles d'un objet tout en masquant les détails complexes de son implémentation. L'abstraction permet de créer des modèles conceptuels qui simplifient la compréhension et la manipulation d'objets.

Exemple d'Abstraction :

Considérons une classe abstraite `Forme` représentant différents types de formes géométriques. Cette classe peut avoir une méthode abstraite `calculerAire()` qui sera implémentée par les sous-classes spécifiques.

```
// Classe abstraite représentant une forme
public abstract class Forme {
    // Méthode abstraite pour calculer l'aire
    public abstract double calculerAire();
}

// Sous-classe représentant un cercle
public class Cercle extends Forme {
    private double rayon;

    // Constructeur et implémentation de la méthode abstraite
    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    @Override
    public double calculerAire() {
        return Math.PI * rayon * rayon;
    }
}

// Sous-classe représentant un rectangle
public class Rectangle extends Forme {
    private double longueur;
    private double largeur;

    // Constructeur et implémentation de la méthode abstraite
    public Rectangle(double longueur, double largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
    }

    @Override
    public double calculerAire() {
        return longueur * largeur;
    }
}
```



ABSTRACTION



Exemple d'Abstraction :

Dans cet exemple, la classe abstraite Forme définit une abstraction générale avec une méthode abstraite ***calculerAire()***. Les sous-classes **Cercle** et **Rectangle** concrétisent cette abstraction en fournissant des implémentations spécifiques de la méthode ***calculerAire()*** pour leurs formes respectives.

JAVA PACKAGES ET API (INTERFACES DE PROGRAMMATION APPLICATIVE)





JAVA PACKAGES :

un package est une collection de classes et d'interfaces liées qui fournissent une organisation logique des classes. Les packages aident à éviter les conflits de noms et permettent une gestion plus claire du code.

Exemple

Dans cet exemple, la classe **Outil** est placée dans le package `com.monentreprise.util`. Pour l'utiliser dans une autre classe (**Main**), nous devons importer le package.

```
// Déclaration d'un package
package com.monentreprise.util;

// Classe dans le package
public class Outil {
    public static void afficherMessage(String message) {
        System.out.println(message);
    }
}

// Utilisation de la classe dans un autre fichier
import com.monentreprise.util.Outil;

public class Main {
    public static void main(String[] args) {
        Outil.afficherMessage("Bonjour, monde!");
    }
}
```



JAVA API (INTERFACES DE PROGRAMMATION APPLICATIVE) :

L'API Java est une collection de packages pré-construits contenant des classes et des interfaces qui fournissent des fonctionnalités prêtes à l'emploi.

Exemple

```
import java.util.Date;
import java.text.SimpleDateFormat;

public class MaClasse {
    public static void main(String[] args) {
        // Utilisation de l'API pour manipuler les dates
        Date dateActuelle = new Date();
        SimpleDateFormat format = new SimpleDateFormat( pattern: "dd-MM-yyyy HH:mm:ss");
        String dateFormatee = format.format(dateActuelle);

        System.out.println("Date actuelle formatée : " + dateFormatee);
    }
}
```

Dans cet exemple, nous utilisons les classes Date et SimpleDateFormat fournies par l'API Java pour manipuler et formater des dates. L'import import java.util.Date; est nécessaire pour utiliser la classe Date depuis le package java.util.

INNER CLASSES





JAVA INNER CLASSES

une classe interne (inner class) est une classe définie à l'intérieur d'une autre classe. Elle a accès aux membres de la classe externe, y compris les membres privés. Il existe différents types de classes internes, dont les classes membres (member inner classes), les classes locales (local classes), les classes anonymes (anonymous classes), et les classes statiques (static nested classes)

Exemple

```
public class Outer {  
    private int outerVar = 10;  
  
    // Classe Membre  
    public class Inner {  
        public void display() {  
            System.out.println("Valeur externe : " + outerVar);  
        }  
    }  
  
    public static void main(String[] args) {  
        Outer outerObj = new Outer();  
        Outer.Inner innerObj = outerObj.new Inner();  
        innerObj.display();  
    }  
}
```

INTERFACES





JAVA INTERFACES



Une interface en Java est une collection de méthodes abstraites. Une classe peut implémenter une ou plusieurs interfaces. Les interfaces fournissent un moyen de définir un contrat pour les classes qui les implémentent.

Caractéristiques des Interfaces en Java :

Méthodes abstraites : Les interfaces peuvent contenir des méthodes abstraites qui doivent être implémentées par les classes qui les utilisent.

```
interface MonInterface {  
    void methode1();  
    void methode2();  
}
```



JAVA INTERFACES



Caractéristiques des Interfaces en Java :

Constantes : Les interfaces peuvent définir des constantes, qui sont automatiquement publiques, statiques et finales.

```
interface MonInterface {  
    int VALEUR_CONSTANTE = 42;  
}
```



JAVA INTERFACES



Caractéristiques des Interfaces en Java :

Implémentation : Les classes implémentent une interface à l'aide du mot-clé implements. Toutes les méthodes déclarées dans l'interface doivent être implémentées par la classe.

```
class MaClasse implements MonInterface {  
    public void methode1() {  
        // Implémentation de la méthode 1  
    }  
  
    public void methode2() {  
        // Implémentation de la méthode 2  
    }  
}
```



JAVA INTERFACES



Caractéristiques des Interfaces en Java :

Héritage multiple : En Java, une classe peut implémenter plusieurs interfaces, permettant ainsi l'héritage multiple.

```
interface InterfaceA {  
    void methodeA();  
}  
  
interface InterfaceB {  
    void methodeB();  
}  
  
class MaClasse implements InterfaceA, InterfaceB {  
    public void methodeA() {  
        // Implémentation de methodeA  
    }  
  
    public void methodeB() {  
        // Implémentation de methodeB  
    }  
}
```



JAVA INTERFACES



Utilisation des Interfaces

Les interfaces sont souvent utilisées pour créer des contrats communs entre des classes différentes. Elles peuvent également être utilisées pour obtenir une certaine forme de polymorphisme

```
public class ProgrammePrincipal {  
    public static void main(String[] args) {  
        // Création d'une instance de la classe implémentant l'interface  
        MonInterface instance = new MaClasse();  
  
        // Appel des méthodes définies dans l'interface  
        instance.methode1();  
        instance.methode2();  
  
        // Accès à une constante de l'interface  
        System.out.println("Valeur constante : " + MonInterface.VALEUR_  
    }  
}
```



JAVA INTERFACES



Interface Fonctionnelle (Functional Interface)

Une interface fonctionnelle est une interface qui ne contient qu'une seule méthode abstraite. À partir de Java 8, une nouvelle annotation `@FunctionalInterface` est introduite pour déclarer explicitement une interface comme fonctionnelle.

```
// Interface fonctionnelle
@FunctionalInterface
interface Operation {
    int calculer(int a, int b);
}
```

Les interfaces en Java fournissent une abstraction puissante permettant de définir des contrats communs entre les classes. Elles sont largement utilisées dans la programmation orientée objet pour favoriser la modularité et la réutilisation du code.

```
// Utilisation de l'interface fonctionnelle
public class Calculatrice {
    public static void main(String[] args) {
        Operation addition = (a, b) -> a + b;
        System.out.println("Addition : " + addition.calculer(5, 3));
    }
}
```

ENUMS





JAVA ENUMS

Les énumérations (enums) en Java sont un moyen de définir un type de données qui représente un ensemble fixe de constantes nommées. Les énumérations offrent une manière propre et concise de représenter des ensembles de valeurs qui ne changent pas fréquemment.

Déclaration d'une Enum en Java :

```
// Définition d'une énumération
enum JourSemaine {
    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

```
public class Main {
    public static void main(String[] args) {
        // Utilisation de l'enum dans une variable
        JourSemaine jour = JourSemaine.MERCREDI;

        // Utilisation dans une instruction switch
        switch (jour) {
            case LUNDI:
                System.out.println("C'est le début de la semaine !");
                break;
            case MERCREDI:
                System.out.println("Milieu de la semaine.");
                break;
            // ... d'autres cas ...
            default:
                System.out.println("C'est le week-end !");
        }
    }
}
```

Utilisation d'une Enum en Java

JAVA USER INPUT





JAVA USER INPUT

Pour obtenir une entrée utilisateur, vous pouvez utiliser la classe Scanner de la bibliothèque java.util. La classe Scanner permet de lire des données à partir de différentes sources, comme la console, les fichiers, etc

Exemple d'Entrée Utilisateur en Java

```
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        // Création d'une instance de Scanner liée à la console
        Scanner scanner = new Scanner(System.in);

        // Demande à l'utilisateur d'entrer un nombre
        System.out.print("Entrez un nombre entier : ");

        // Lecture de l'entrée utilisateur comme un entier
        int nombre = scanner.nextInt();

        // Demande à l'utilisateur d'entrer une chaîne de caractères
        System.out.print("Entrez une chaîne de caractères : ");

        // Lecture de l'entrée utilisateur comme une chaîne de caractères
        String texte = scanner.next();

        // Affichage des données saisies
        System.out.println("Vous avez saisi le nombre : " + nombre);
        System.out.println("Vous avez saisi la chaîne : " + texte);

        // Fermeture du scanner pour libérer les ressources
        scanner.close();
    }
}
```



JAVA USER INPUT



Explication du Code :

1. Importez la classe Scanner de java.util.
2. Créez une instance de Scanner liée à System.in (l'entrée standard, généralement la console).
3. Utilisez les méthodes de Scanner pour lire différentes données (par exemple, nextInt() pour lire un entier, next() pour lire une chaîne de caractères).
4. Affichez ou utilisez les données lues.
5. N'oubliez pas de fermer le scanner à la fin pour éviter les fuites de ressources.



MERCI