



l'Introduction à la Programmation Orientée Objet (OOP)

Année Universitaire : 2024 / 2025

Professeur : ELMRABET ABDELALI



Résumé des Chapitres

- | | |
|---------------------------------|----------------------------------|
| 1. Introduction | 9. Tableaux |
| 2. Variables | 10. Boucles For |
| 3. Types de Données | 11. Boucles While |
| 4. Opérateurs | 12. Boucles Do While |
| 5. Chaînes de Caractères | 13. Liste de Tableaux |
| 6. Entrées Utilisateur | 14. HashMap |
| 7. Instructions Conditionnelles | 15. Programmation Orientée Objet |
| 8. Cas Switch | |





Introduction

Introduction à Java

Java est un langage de programmation orienté objet, multiplateforme, robuste et sécurisé.

Il a été développé par Sun Microsystems (désormais Oracle) et est largement utilisé dans le développement logiciel.

- Langage compilé et interprété (bytecode + JVM)
- Fortement typé et orienté objet
- Très utilisé dans les systèmes d'entreprise, les applications web, mobiles et embarquées





Introduction

Programmation Orientée Objet (POO) en Java

La POO est un paradigme qui organise le code autour des **objets** et des **classes**. Java repose entièrement sur ce modèle.

- Concepts clés : **encapsulation**, **héritage**, **polymorphisme**, **abstraction**
- Java permet une **bonne structuration** du code, **réutilisabilité** et **maintenabilité**
- Chaque objet est une **instance** d'une **classe** représentant une **entité métier**





Introduction

Java pour les applications mobiles – Android Studio

Java est l'un des langages principaux utilisés pour développer des applications Android via **Android Studio**.

- Android SDK offre des APIs Java pour accéder au matériel et services Android
- Les composants principaux : **Activities, Fragments, Services, Intents**
- Android Studio propose un environnement complet avec un éditeur graphique et un émulateur





Introduction

Java pour les applications web – Spring Boot

Spring Boot est un framework Java qui facilite le développement rapide d'applications web.

- Permet de créer des APIs REST rapidement
- Intègre des outils comme **Spring Data**, **Spring Security**, **Thymeleaf**
- Compatible avec des bases de données comme MySQL, PostgreSQL ou H2





Introduction

Java EE (Jakarta EE) pour les applications d'entreprise

Java EE (désormais Jakarta EE) est un ensemble de spécifications pour développer des applications Java d'entreprise robustes.

- Inclut servlets, JSP, EJB, JPA, CDI, JSF
- Idéal pour les grandes applications à forte charge et besoin de scalabilité
- S'intègre bien avec des serveurs comme GlassFish, WildFly, Payara





Introduction

Pourquoi choisir Java ?

Java reste un choix solide pour plusieurs raisons dans le développement moderne :

- **Portabilité** grâce à la JVM : "Write Once, Run Anywhere"
- **Communauté active** et abondance de bibliothèques
- **Écosystème mature** pour le web, la mobilité et les systèmes embarqués





Introduction

Environnements pour coder en Java

Java peut être programmé dans différents environnements selon le type de projet et le niveau du développeur.

- **Eclipse** : riche en outils, adapté aux projets complexes en entreprise
- **IntelliJ IDEA** : performant, intelligent, très utilisé par les professionnels
- **NetBeans** : simple, soutenu par Oracle, bon pour les débutants
- **Android Studio** : spécialisé pour le développement mobile Android





Introduction

Solution choisie : Replit

Pour bien commencer, nous allons utiliser [Replit](#) :

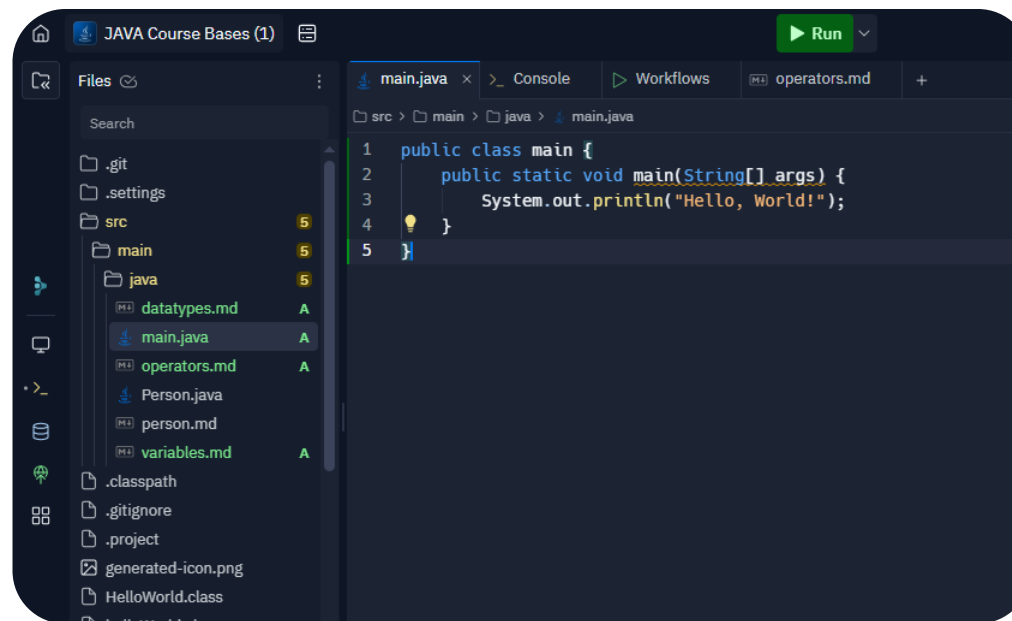
- Plateforme en ligne, gratuite et accessible sans installation
- Permet de coder en Java depuis un navigateur
- Idéal pour apprendre les bases rapidement, en toute simplicité





Introduction

Solution choisie : Replit





Introduction - Exemple

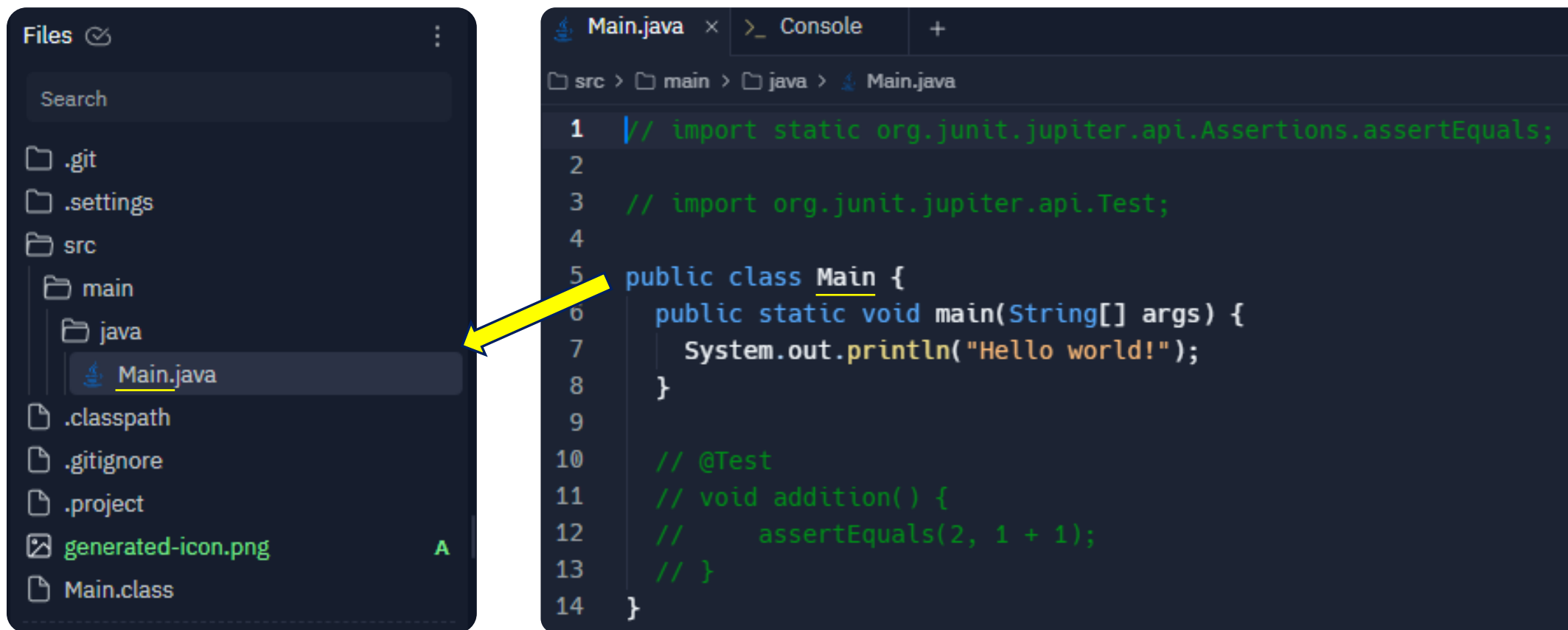


```
Main.java x Console +
src > main > java > Main.java

1 // import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 // import org.junit.jupiter.api.Test;
4
5 public class Main {
6     public static void main(String[] args) {
7         System.out.println("Hello world!");
8     }
9
10    // @Test
11    // void addition() {
12    //     assertEquals(2, 1 + 1);
13    // }
14 }
```



Introduction - Exemple



The screenshot displays an IDE interface. On the left, the 'Files' panel shows a project structure with folders like '.git', '.settings', 'src', 'main', and 'java'. The file 'Main.java' is highlighted in the 'java' folder. A yellow arrow points from this file to the code editor on the right. The code editor shows the content of 'Main.java' with the following code:

```
1 // import static org.junit.jupiter.api.Assertions.assertEquals;
2
3 // import org.junit.jupiter.api.Test;
4
5 public class Main {
6     public static void main(String[] args) {
7         System.out.println("Hello world!");
8     }
9
10    // @Test
11    // void addition() {
12    //     assertEquals(2, 1 + 1);
13    // }
14 }
```



Introduction - Exemple

```
4
5 public class Main {
6     public static void main(String[] args) {
7         System.out.println("Hello world!");
8     }
9
```

```
3
4
5 ✓ public class Main {
6 ✓     public static void main(String[] arguments) {
7         System.out.println("Hello world!");
8     }
9
```



Variables - Définition

Définition des Variables en Java

En Java, une **variable** est un conteneur qui stocke des données. Chaque variable a un certain **type** qui détermine les valeurs qu'elle peut contenir et les opérations qui peuvent être réalisées sur elle. Elle possède également une **portée** qui détermine où elle peut être utilisée dans le programme.



Variables - Exemple

Déclaration et Initialisation des Variables

- **Déclaration** : Associe un nom de variable avec un type.
Exemple : `int age;`
- **Initialisation** : Attribue une valeur à une variable.
Exemple : `age = 25;`
- **Déclaration et Initialisation Simultanées** : Fait dans la même ligne.
Exemple : `int age = 25;`



Variables - Exemple

Bonnes Pratiques de Nommage

- **Commencer par une lettre, `_`, ou `$`** : Les noms de variables ne doivent pas commencer par un chiffre.
Bon exemple : `int myAge;`
Mauvais exemple : `int 6age;`
- **Pas d'Espaces** : Les espaces dans les noms de variables provoquent des erreurs.
Bon exemple : `int myAge;`
Mauvais exemple : `int mon age;`
- **Nom Représentatif** : Utiliser des noms qui reflètent l'utilité de la variable.
Exemple : `int compteur;` au lieu de `int c;`



Variables - Exemple

Utilisation de static avec les Variables

- **Déclaration sans static** : La variable appartient à une instance spécifique de la classe.

Exemple :

```
class Main { int age = 25; System.out.println("I am " + age + " years old."); }
```

- **Déclaration avec static** : Partagée parmi toutes les instances de la classe.

Exemple :

```
class Main { static int age = 25; public static void main(String[] args) { System.out.println("I am " + age + " years old."); } }
```



Variables - Exemple

Points à Éviter

- **Double Déclaration dans le même Scope** : Ne jamais déclarer deux variables avec le même nom dans le même bloc ou méthode.

Exemple :

```
int age = 25; int age = 30; // Erreur
```

- **Utiliser des Variables Non Initialisées** : Provoque des erreurs au moment de l'exécution.

Exemple :

```
int age; System.out.println("I am " + age + " years old."); // Erreur
```



Variables - Exemple



```
Main.java x >_ Console +
src > main > java > Main.java
1 class Main {
2     static int userAge = 30; // Variable statique avec un nom descriptif et un bon choix de nom
3
4     public static void main(String[] args) {
5         System.out.println("I am " + userAge + " years old.");
6     }
7 }
```



Variables - Exemple

Java Language Keywords

Here is a list of keywords in the Java programming language. You cannot use any of the following as identifiers in your programs. The keywords `const` and `goto` are reserved, even though they are not currently used. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

| | | | | |
|------------------------|-----------------------|-------------------------|-------------------------|---------------------------|
| <code>abstract</code> | <code>continue</code> | <code>for</code> | <code>new</code> | <code>switch</code> |
| <code>assert***</code> | <code>default</code> | <code>goto*</code> | <code>package</code> | <code>synchronized</code> |
| <code>boolean</code> | <code>do</code> | <code>if</code> | <code>private</code> | <code>this</code> |
| <code>break</code> | <code>double</code> | <code>implements</code> | <code>protected</code> | <code>throw</code> |
| <code>byte</code> | <code>else</code> | <code>import</code> | <code>public</code> | <code>throws</code> |
| <code>case</code> | <code>enum****</code> | <code>instanceof</code> | <code>return</code> | <code>transient</code> |
| <code>catch</code> | <code>extends</code> | <code>int</code> | <code>short</code> | <code>try</code> |
| <code>char</code> | <code>final</code> | <code>interface</code> | <code>static</code> | <code>void</code> |
| <code>class</code> | <code>finally</code> | <code>long</code> | <code>strictfp**</code> | <code>volatile</code> |
| <code>const*</code> | <code>float</code> | <code>native</code> | <code>super</code> | <code>while</code> |

* not used

** added in 1.2

*** added in 1.4

**** added in 5.0



Types de Données - Définition

Types de Données en Java

En Java, un **type de donnée** définit la taille et le type de valeurs que peut contenir une variable. Les types de données sont classés en deux catégories principales : les types primitifs et les types de référence.



Types de Données - Exemple

Types Primitifs

Les **types primitifs** contiennent directement les valeurs. Ils incluent :

- **int** : Pour les entiers. Exemple : `int myNumber = 10;`
- **double** : Pour les valeurs décimales. Exemple : `double myDecimal = 10.5;`
- **boolean** : Pour les valeurs vrai/faux. Exemple : `boolean isJavaFun = true;`
- **char** : Pour les caractères. Exemple : `char initial = 'J';`



Types de Données - Exemple

Types de Référence

Les **types de référence** stockent des références à des objets. Ils incluent :

- **Objets** : Instanciés à partir de classes. **Exemple** : `Person person = new Person();`
- **Tableaux** : Collections ordonnées d'éléments. **Exemple** : `int[] numbers = {1, 2, 3};`
- **Chaînes de caractères** : Séquences de caractères immuables. **Exemple** : `String text = "Hello";`

Ces types de données sont essentiels pour structurer et manipuler les informations dans une application Java.



Types de Données - Exemple

Types Primitifs

Types Entiers

- **byte** : Entier de 8 bits. Représentation des valeurs de -128 à 127.
 - *byte aSingleByte = 100;*
- **short** : Entier de 16 bits. Représentation des valeurs de -32 768 à 32 767.
 - *short aSmallNumber = 20000;*
- **int** : Entier de 32 bits. Représentation des valeurs de -2 147 483 648 à 2 147 483 647.
 - *int anInteger = 2147483647;*
- **long** : Entier de 64 bits. Représentation des valeurs de -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807.
 - *long aLargeNumber = 9223372036854775807L;*



Types de Données - Exemple

Types Primitifs

Types Décimaux

- **float** : Virgule flottante de 32 bits. Représente des valeurs de `~1.4E-45~` à `~3.4028235E38~`.
 - `float aFloat = 3.4028F;`
- **double** : Virgule flottante de 64 bits. Représente des valeurs de `~4.9E-324~` à `~1.7976931348623157E308~`.
 - `double aDouble = 1.79769313;`



Types de Données - Exemple

Types Primitifs

Booléens

- **boolean** : Peut être soit `true` soit `false`.
 - `boolean isWeekend = false;`

Caractères

- **char** : Caractère Unicode de 16 bits.
 - `char copyrightSymbol = '\u00A9';`



Types de Données - Exemple

Conversion de Types

Conversion Implicite

Java permet parfois une conversion implicite de type plus petit vers un type plus grand (exemple : entier vers décimal) :

```
int number1 = 5;
```

```
double number2 = number1; // Conversion implicite
```



Types de Données - Exemple

Conversion de Types

Conversion Explicite

Pour convertir un type plus grand vers un type plus petit (comme un nombre décimal vers un entier), une conversion explicite est nécessaire :

Sans Cast Explicite (Erreur)

```
double number1 = 5.8;
```

```
int number2 = number1; // Génère une erreur
```

Avec Cast Explicite

```
double number1 = 5.8;
```

```
int number2 = (int) number1; // Conversion explicite correcte
```



Types de Données - Exemple



```
Main.java x >_ Console +
src > main > java > Main.java
1 class Main {
2     public static void main(String[] args) {
3         double number1 = 5.8;
4         int number2 = (int) number1; // Explicit casting from double to int
5
6         System.out.println(number2);
7     }
8 }
```



Opérateurs - Définition

Opérateurs en Java

Les opérateurs en Java sont utilisés pour effectuer diverses opérations sur les variables et les valeurs. Voici un aperçu des principaux opérateurs avec des exemples :



Opérateurs - Exemple

Opérateurs Arithmétiques

Effectuent des opérations mathématiques basiques :

- **Addition (+)** : Ajoute deux nombres. Exemple : $(12 + 6)$ // Affiche 18
- **Soustraction (-)** : Soustrait un nombre d'un autre. Exemple : $(12 - 6)$ // Affiche 6
- **Multiplication (*)** : Multiplie deux nombres. Exemple : $(12 * 6)$ // Affiche 72
- **Division (/)** : Divise un nombre par un autre. Exemple : $(12 / 6)$ // Affiche 2
- **Modulo (%)** : Reste de la division entre deux nombres. Exemple : $(12 \% 6)$ // Affiche 0



Opérateurs - Exemple

Opérateurs de Comparaison

Comparaient deux valeurs :

- **Égal à (==)** : Vérifie l'égalité. Exemple : `(12 == 15) // Affiche false`
- **Différent de (!=)** : Vérifie la différence. Exemple : `(12 != 15) // Affiche true`
- **Plus grand que (>) et Plus petit que (<)** : Comparaison d'ordre. Exemple : `(12 > 15) // Affiche false`



Opérateurs - Exemple

Opérateurs Logiques

Utilisés pour combiner des expressions booléennes :

- **ET logique (&&)** : Renvoie vrai si les deux opérandes sont vrais. Exemple : `(age >= 18 && age <= 40)`
- **OU logique (||)** : Renvoie vrai si l'un des opérandes est vrai. Exemple : `(isStudent || isLibraryMember)`
- **NON logique (!)** : Inverse l'état logique. Exemple : `!isStudent`



Opérateurs - Exemple

Opérateurs d'Affectation

Utilisés pour assigner des valeurs :

- **Affectation simple (=)** : Assigne une valeur. **Exemple** : `a = 1;`
- **Incrémentation (++)** et **Décrémentation (--)** : Augmente ou diminue une valeur.
 - Post-incrémentation : **Exemple** : `number++`
 - Pré-incrémentation : **Exemple** : `++number`

Ces opérateurs sont fondamentaux en Java pour manipuler les données et contrôler le flux des opérations au sein d'un programme.



Opérateurs - Exemple



```
1 public class Main{
2
3     public static void main(String[] args) {
4         int a = 12;
5         int b = 6;
6         boolean isJavaFun = true;
7
8         // Opérateurs arithmétiques
9         System.out.println("Addition: " + (a + b)); // 18
10        System.out.println("Soustraction: " + (a - b)); // 6
11        System.out.println("Multiplication: " + (a * b)); // 72
12        System.out.println("Division: " + (a / b)); // 2
13        System.out.println("Modulo: " + (a % b)); // 0
14
15        // Opérateurs de comparaison
16        System.out.println("Is Equal: " + (a == b)); // false
17        System.out.println("Is Not Equal: " + (a != b)); // true
18    }
19 }
```



Opérateurs - Exemple



```
19 // Opérateurs logiques
20 System.out.println("And Condition: " + ((a > b) && isJavaFun)); // true
21 System.out.println("Or Condition: " + ((a < b) || isJavaFun)); // true
22
23 // Incrémentation et Décrémentation
24 int count = 5;
25 System.out.println("Count: " + (++count)); // 6
26
27 int score = 20;
28 System.out.println("Score: " + (--score)); // 19
29 }
30 }
```

```
30 }
```

```
30 }
```

```
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
```



Chaînes de Caractères - Définition

Chaînes de Caractères en Java

Les chaînes de caractères en Java, ou `String`, sont des objets qui représentent des séquences immuables de caractères. Elles sont utilisées pour stocker et manipuler du texte dans les programmes Java. Voici quelques concepts clés sur leur utilisation :



Chaînes de Caractères - Exemple

Déclaration et Initialisation

- **Caractère unique** : Un seul caractère peut être stocké dans une variable `char`.
Exemple : `char percentSign = '%';`
- **Chaîne de caractères** : Utilisée pour stocker une séquence de caractères.
Exemple : `String name = "John Doe";`



Chaînes de Caractères - Exemple

Création d'un Nouvel Objet String

- **Nouvel objet String** : Crée explicitement un nouvel objet string, ce qui est moins fréquent pour des raisons de performance.

Exemple : `String name = new String("John Doe");`



Chaînes de Caractères - Exemple

Comparaison de Chaînes

- **Comparaison avec `==`** : Vérifie si deux références pointent vers le même objet.
Exemple : `string1 == string2;`
- **Comparaison de valeur avec `.equals()`** : Vérifie si le contenu de deux chaînes est le même.
Exemple : `string1.equals(string2);`



Chaînes de Caractères - Exemple

Manipulations de Chaînes

- **Vérification de la Longueur** : La méthode ``.length()`` renvoie le nombre de caractères dans une chaîne.
Exemple : `name.length();` // Retourne 11
- **Vérification si Vide** : La méthode ``.isEmpty()`` retourne ``true`` si la chaîne est vide.
Exemple : `name.isEmpty();` // Retourne false

Transformation de Chaînes

- **Conversion Maj/Min** : Transforme la chaîne en majuscules ou minuscules.
Exemple : `name.toUpperCase();` // Retourne "YASSINE HARIT"



Chaînes de Caractères - Exemple

Remplacement et Contenu

- **Remplacer une Sous-chaîne** : Utilise ``.replace()`` pour substituer une partie de la chaîne.
Exemple : `string.replace("fun", "powerful");`
- **Vérification de la Présence d'un Élément** : La méthode ``.contains()`` vérifie si une sous-chaîne est présente.
Exemple : `string.contains("ocean");` // Retourne true



Chaînes de Caractères - Exemple

Formatage et Comparaison Ignorant Casse

- **Formatage avec Printf** : Permet d'insérer des variables dans une chaîne.

Exemple :

Java

```
1 String formattedString = String.format("Bonjour! Je m'appelle %s.", name);
```

- **Comparaison Ignorant Maj/Min** : Utilise ``.equalsIgnoreCase()`` pour comparer indépendamment de la casse.

Exemple : `string1.equalsIgnoreCase(string2);`



Entrées Utilisateur - Définition

Entrées Utilisateur en Java

En Java, la capture d'entrées utilisateur est fondamentale pour créer des applications interactives. Cela permet aux programmes de recevoir des informations directement de l'utilisateur pour les traiter, les analyser ou émettre des résultats personnalisés. La classe `Scanner`, disponible dans le package `java.util`, est couramment utilisée pour lire les entrées depuis la console.



Entrées Utilisateur - Exemple

Utilisation de Scanner en Java

La classe **Scanner** de la bibliothèque `java.util` est essentielle pour lire les entrées utilisateur, comme celles tapées au clavier. Elle facilite la capture de différents types de données textuelles, numériques et booléennes.

- **Importance de la bibliothèque** : Elle permet une interaction directe avec les utilisateurs, rendant les applications plus dynamiques et réactives.
- **Déclaration avant utilisation** : Avant de lire les entrées, il est nécessaire d'importer la classe **Scanner** et de créer une instance :

Exemple : Import de la classe et création de l'instance de **Scanner**.

```
1 import java.util.Scanner;
```

```
Scanner scanner = new Scanner(System.in);
```



Entrées Utilisateur - Exemple

Différence entre `print()` et `println()`

Les méthodes `print()` et `println()` sont utilisées pour afficher des messages à l'utilisateur, mais elles fonctionnent légèrement différemment :

- `print()` : Affiche le texte sur la même ligne sans ajouter de saut de ligne.
 - **Exemple** : Afficher un message puis ajouter un autre message sur la même ligne.
- `println()` : Affiche le texte et ajoute un saut de ligne à la fin.
 - **Exemple** : Afficher un message puis passer à la ligne suivante.



Entrées Utilisateur - Exemple

Problème de `nextLine()` après `nextInt()`

Lors de l'utilisation de `nextInt()`, le retour à la ligne est souvent laissé dans le flux d'entrée, ce qui peut provoquer des problèmes lors de la lecture suivante avec `nextLine()`.

- **Problème courant :** La méthode `nextLine()` lit les restes de la ligne après un `nextInt()`.

Exemple : Utiliser `nextInt()` suivi de `nextLine()` peut entraîner une lecture incorrecte.



Entrées Utilisateur - Exemple

Solution avec `parseInt()` ou Gestion de Ligne

Pour éviter les problèmes liés à `nextLine()`, utilisez `parseInt()` ou consommez le retour à la ligne :

- **Solution avec `parseInt()`** : Convertir une entrée de chaîne en entier.
Exemple : Utiliser `Integer.parseInt` pour convertir une chaîne en entier.
- **Consommation du retour de ligne** : Utilisez simplement `nextLine()` après `nextInt()` pour retirer le saut de ligne.

Exemple : Lire un entier, puis retirer le saut de ligne résiduel.



Entrées Utilisateur - Exemple



```
1  import java.util.Scanner;
2
3  class Main {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6
7          System.out.print("Quel est votre nom ? "); // Ask for the user's name (in the same line) !!!
8          String name = scanner.nextLine();
9
10         System.out.println("Votre nom est " + name); // Print the user's name
11
12         scanner.close();
13     }
14 }
```



Instructions Conditionnelles - Définition

Instructions Conditionnelles en Java

Les instructions conditionnelles permettent aux programmes de prendre des décisions en fonction des conditions spécifiées. En Java, cela inclut l'utilisation de `if`, `else if`, `else`, et `switch` pour contrôler le flux du programme.



Instructions Conditionnelles - Exemple

Utilisation de **if**, **else if**, et **else**

L'instruction **if** vérifie si une condition est vraie, et exécute le bloc de code correspondant. **else if** et **else** fournissent des alternatives lorsque la première condition **if** n'est pas satisfaisante.

- **Exemple** : Vérification de conditions pour des calculs simples entre deux nombres.
- **if** : Vérifiez si une condition initiale est vraie.
- **else if** : Fournit une autre condition à vérifier si la précédente est fausse.
- **else** : Exécute un bloc par défaut si aucune condition précédente n'est vraie.



Instructions Conditionnelles - Exemple

Exemple avec Décision Basée sur Entrée Utilisateur

Pour une application simple de calcul, l'utilisateur peut choisir l'opération à effectuer après avoir saisi deux nombres :

- **Choix d'une opération** : L'utilisateur saisit soit **add**, **sub**, **mul**, **div** pour effectuer des opérations telles que l'addition ou la division.
- **Traitement des cas** : Chaque opération est traitée dans son propre bloc de code, avec des vérifications comme l'erreur de division par zéro.



Instructions Conditionnelles - Exemple



```
18     if (operation.equals("add")) {
19         System.out.printf("%f + %f = %f\n", nombre1, nombre2, nombre1 + nombre2);
20     } else if (operation.equals("sub")) {
21         System.out.printf("%f - %f = %f\n", nombre1, nombre2, nombre1 - nombre2);
22     } else if (operation.equals("mul")) {
23         System.out.printf("%f * %f = %f\n", nombre1, nombre2, nombre1 * nombre2);
24     } else if (operation.equals("div")) {
25         if (nombre2 == 0) {
26             System.out.println("Impossible de diviser par zéro.");
27         } else {
28             System.out.printf("%f / %f = %f\n", nombre1, nombre2, nombre1 / nombre2);
29         }
30     } else {
31         System.out.printf("%s n'est pas une opération supportée.\n", operation);
32     }
33 }
```



Cas Switch - Définition

Utilisation du **switch** en Java

Le **switch** est une structure de contrôle en Java qui permet d'exécuter différents blocs de code en fonction de la valeur d'une expression. C'est une alternative efficace aux multiples **else if** lorsqu'il s'agit de comparer la même variable ou expression à plusieurs valeurs possibles.



Cas Switch - Définition

Définition

- **switch** : Évalue l'expression donnée et exécute le bloc de code associé au **case** correspondant.
- **case** : Spécifie une valeur possible que l'expression peut prendre. Si l'expression correspond à un **case**, le bloc de code lié à ce **case** s'exécute.
- **break** : Utilisé pour sortir du bloc **switch** après l'exécution d'un **case**. Sans **break**, le programme continue vers le prochain **case** (phénomène de l'exécution en cascade).
- **default** : Représente le bloc de code à exécuter si aucune correspondance n'est trouvée avec les autres **case**.



Cas Switch - Exemple



```
switch (operation) {  
    case "add":  
        System.out.printf("%f + %f = %f\n", nombre1, nombre2, nombre1 + nombre2);  
        break;  
    case "sub":  
        System.out.printf("%f - %f = %f\n", nombre1, nombre2, nombre1 - nombre2);  
        break;  
    case "mul":  
        System.out.printf("%f * %f = %f\n", nombre1, nombre2, nombre1 * nombre2);  
        break;  
    case "div":  
        if (nombre2 == 0) {  
            System.out.println("Impossible de diviser par zéro.");  
        } else {  
            System.out.printf("%f / %f = %f\n", nombre1, nombre2, nombre1 / nombre2);  
        }  
        break;  
    default:  
        System.out.printf("%s n'est pas une opération supportée.\n", operation);  
}
```

```
System.out.printf("%s n'est pas une opération supportée.\n", operation);  
default:
```



Tableaux - Définition

Tableaux en Java

Les tableaux en Java sont des structures de données qui permettent de stocker des collections d'éléments du même type. Chaque élément peut être accédé rapidement via son index.

Définition

- **Tableau** : Collection d'éléments du même type, accessibles par un index.
- **Index** : Position d'un élément dans le tableau, commençant généralement à zéro.
- **Taille** : Le nombre total d'éléments qu'un tableau peut contenir, défini lors de sa création.



Tableaux - Exemple

Exemple et Utilisation

Exemple : Voici comment créer et manipuler des tableaux simples de caractères.

- **Création et Accès aux Éléments :** Initialiser un tableau et accéder à ses éléments.

Exemple : Définir un tableau de voyelles et afficher un élément spécifique.

Java

```
1 char[] vowels = {'a', 'e', 'i', 'o', 'u'};  
2 System.out.println("Troisième voyelle : " + vowels[2]); // Affiche 'i'
```



Tableaux - Exemple

Exemple et Utilisation

- **Afficher tous les éléments** : Utiliser `Arrays.toString()` pour afficher le contenu entier d'un tableau.

Exemple : Afficher toutes les voyelles.

Java

```
1 System.out.println(Arrays.toString(vowels)); // Affiche [a, e, i, o, u]
```



Tableaux - Exemple

Exemple et Utilisation

- **Changer la valeur d'un élément :** Modifier un élément existant dans le tableau.

Exemple : Changer 'i' en 'x'.

Java

```
1 vowels[2] = 'x';  
2 System.out.println(Arrays.toString(vowels)); // Affiche [a, e, x, o, u]
```



Tableaux - Exemple

Exemple et Utilisation

- Calculer la longueur : Utiliser `~.length~` pour obtenir la taille du tableau.

Exemple : Imprimer la longueur du tableau.

Java

```
1 System.out.println("Longueur du tableau : " + vowels.length); // Affiche 5
```



Tableaux - Exemple



```
4 class Main {  
5     public static void main(String[] args) {  
6         // Direct initialization of the char array with vowels  
7         char vowels[] = {'a', 'e', 'i', 'o', 'u'};  
8  
9         // Change the third element (index 2) from 'i' to 'x'  
10        vowels[2] = 'x';  
11  
12        // Print the array using Arrays.toString()  
13        System.out.println(Arrays.toString(vowels));  
14    }  
15 }
```

```
12 }  
14 }  
15 System.out.println(Arrays.toString(vowels));
```



Boucles For - Définition

Boucles for en Java

Les boucles **for** en Java sont utilisées pour exécuter une séquence d'instructions de manière répétitive pendant un certain nombre de fois, facilitant ainsi l'itération sur les tableaux et les collections de données.

Définition

- **Boucle for** : Structure de contrôle qui permet l'exécution répétée d'un bloc de code. Elle est composée de trois parties principales : l'initialisation, la condition, et l'incrémentation.
- **Utilisation typique** : Parcourir des tableaux, générer des séquences standard, et itérer avec précision.



Boucles For - Exemple

Exemple et Utilisation

Exemple : Différentes manières d'utiliser les boucles **for** pour travailler avec des données.

- **Boucle simple de 1 à 10 :** Itération classique pour afficher des nombres séquentiels.

Exemple : Imprimer les nombres de 1 à 10.

Java

```
1 for (int number = 1; number <= 10; number++) {  
2     System.out.println(number); // Affiche chaque nombre de 1 à 10  
3 }
```



Boucles For - Exemple

Exemple et Utilisation

- **Boucle avec index** : Parcourir et accéder aux éléments d'un tableau à l'aide de leur index.

Exemple : Accéder et imprimer chaque élément d'un tableau.

Java

```
1 int[] numbers = {1, 2, 3, 4, 5};
2 for (int index = 0; index < numbers.length; index++) {
3     System.out.println(numbers[index]); // Affiche chaque élément du tableau
4 }
```



Boucles For - Exemple

Exemple et Utilisation

- **Calculer la somme d'un tableau** : Accumulation de valeurs à l'intérieur d'une boucle.

Exemple : Calculer la somme des valeurs d'un tableau.

Java

```
1 int sum = 0;
2 for (int number : numbers) {
3     sum += number;
4 }
5 System.out.println(sum); // Affiche la somme totale
```



Boucles For - Exemple

Exemple et Utilisation

- **Imprimer une table de multiplication** : Utiliser une boucle pour des calculs répétitifs.

Exemple : Afficher le tableau de multiplication du chiffre 5.

Java

```
1 int number = 5;
2 for (int multiplier = 1; multiplier <= 10; multiplier++) {
3     System.out.printf("%d x %d = %d\n", number, multiplier, number * multiplier);
4 }
```

- **Double boucle** : Utiliser des boucles imbriquées pour des tâches plus complexes, comme générer des tables de multiplication complètes.



Boucles While - Définition

Boucles while en Java

Les boucles **while** sont utilisées en Java pour répéter un bloc de code tant qu'une condition donnée reste vraie. Avant chaque itération, la condition est évaluée : si elle est vraie, le bloc de code s'exécute; si elle est fausse, la boucle s'arrête.

- **Structure de la boucle while** : Elle commence par le mot-clé **while** suivi d'une condition entre parenthèses, et un bloc de code à exécuter tant que la condition est vraie.
- **Utilisation typique** : Les boucles **while** sont pratiques lorsque le nombre d'itérations n'est pas connu à l'avance et dépend d'une certaine condition externe.



Boucles While - Exemple



```
1  class Main {  
2      public static void main(String[] args) {  
3          int number = 5;  
4          int multiplier = 1;  
  
5          while (multiplier <= 10) { // determiner nombre d'itération maximal  
6              System.out.printf("%d x %d = %d \n", number, multiplier, number * multiplier);  
7              multiplier++; // ajout dans chaque itération  
8          }  
9      }  
10 }  
11 }
```



Boucles Do While - Définition

Boucles do-while en Java

La boucle **do-while** en Java est similaire à une boucle **while** mais avec une différence clé : le bloc de code de la boucle **do-while** s'exécute au moins une fois avant que la condition soit évaluée.

- **Structure de la boucle do-while** : Commence par le mot-clé **do**, suivi par le bloc de code à exécuter. La condition à vérifier vient après le bloc, après le mot-clé **while**.
- **Utilisation typique** : Les boucles **do-while** sont utiles lorsque vous souhaitez garantir que le bloc de code s'exécute au moins une fois, par exemple pour afficher un menu avant de vérifier l'entrée utilisateur.

Ces deux types de boucles fournissent des structures de contrôle permettant de gérer des séquences de tâches répétitives avec souplesse en Java.



Boucles Do While - Exemple



```
1 class Main {  
2     public static void main(String[] args) {  
3         int number = 5;  
4         int multiplier = 1;  
5  
6         do {  
7             System.out.printf("%d x %d = %d%n", number, multiplier, number * multiplier);  
8             multiplier++;  
9         } while (multiplier <= 10);  
10    }  
11 }
```




Liste de Tableaux - Définition

Listes de Tableaux en Java

Les listes de tableaux, ou **ArrayList**, en Java offrent une structure dynamique qui permet de stocker des éléments dans une liste. Contrairement aux tableaux traditionnels, les **ArrayList** peuvent changer de taille dynamiquement et offrent de nombreuses méthodes pour manipuler les données.



Liste de Tableaux - Définition

Définition

- **ArrayList** : Structure de données dynamique qui peut contenir des éléments du même type, avec la capacité de redimensionnement automatique.
- **Indexation** : Accès aux éléments par leur position, similaire à un tableau.
- **Méthodes fondamentales** : Ajout, suppression, accès, remplacement et tri des éléments.



Liste de Tableaux - Exemple

Exemple et Utilisation

Exemple : Manières d'interagir avec des **ArrayList** pour accomplir des tâches variées en Java.

- **Ajout et Affichage des éléments :** Utiliser les méthodes **add** pour insérer des éléments et **toString** pour les afficher.

Exemple : Ajouter des nombres à une liste et les afficher.

Java

```
1 ArrayList<Integer> numbers = new ArrayList<>();
2 numbers.add(1);
3 numbers.add(2);
4 numbers.add(3);
5 System.out.println(numbers.toString()); // Affiche [1, 2, 3]
```



Liste de Tableaux - Exemple

Exemple et Utilisation

- **Accéder par index** : Obtenir un élément spécifique en utilisant son index avec **get**.

Exemple : Afficher le troisième élément.

Java

```
1 System.out.println(numbers.get(2)); // Affiche 3
```



Liste de Tableaux - Exemple

Exemple et Utilisation

- **Enlever des éléments** : Supprimer par index ou par valeur avec `remove`.

Exemple : Enlever un élément spécifique.

Java

```
1 numbers.remove(Integer.valueOf(2)); // Supprime 2 de la liste
2 System.out.println(numbers.toString()); // Affiche [1, 3]
```



Liste de Tableaux - Exemple

Exemple et Utilisation

- **Vérifier la taille et la présence d'éléments** : Utiliser `size` pour la longueur de la liste et `contains` pour vérifier la présence.

Exemple : Nombre total et vérification de présence.

Java

```
1 System.out.println(numbers.size()); // Affiche 2
2 System.out.println(numbers.contains(1)); // Affiche true
```



Liste de Tableaux - Exemple

Exemple et Utilisation

- **Utilisation de boucles** : Modifier ou interagir avec les éléments en utilisant une boucle.

Exemple : Multiplier chaque élément par 2.

Java

```
1 numbers.forEach(number -> {  
2     numbers.set(numbers.indexOf(number), number * 2);  
3 });  
4 System.out.println(numbers.toString()); // Affiche [2, 6]
```



Liste de Tableaux - Exemple



```
1 import java.util.ArrayList;
2 import java.util.Comparator;
3
4 class Main {
5     public static void main(String[] args) {
6         ArrayList<Integer> numbers = new ArrayList<Integer>();
7
8         numbers.add(5);
9         numbers.add(3);
10        numbers.add(1);
11
12        System.out.println("before: " + numbers.toString());
13
14        numbers.forEach(number -> {
15            numbers.set(numbers.indexOf(number), number * 2);
16            System.out.println(number * 2);
17        });
18
19        System.out.println("after: " + numbers.toString());
20    }
21 }
```




Liste de Tableaux - Exemple

Différence entre les Tableaux et ArrayList en Java

Les **Tableaux** et **ArrayList** sont deux structures de données fondamentales en Java, chacune ayant ses propres caractéristiques et avantages.



Liste de Tableaux - Exemple

Tableaux

- **Taille Fixe** : La taille d'un tableau est définie lors de sa création et ne peut pas être modifiée.
- **Type Primitif ou Objet** : Peut contenir des types primitifs comme `int`, `char`, ainsi que des objets.
- **Performance** : Plus rapides pour des accès directs et sont généralement plus performants pour des manipulations de données fixes.
- **Syntaxe** : Utilise des crochets `[]` pour la déclaration et l'accès aux éléments.

```
1 int[] numbers = {1, 2, 3, 4, 5};  
2 System.out.println(numbers[2]); // Affiche 3, accès direct par index
```



Liste de Tableaux - Exemple

ArrayList

- **Taille Dynamique** : Peut croître ou diminuer en taille dynamiquement et automatiquement.
- **Objets Seuls** : Ne peut contenir que des objets, donc les types primitifs doivent être utilisés via les wrappers (`Integer`, `Character`, etc.).
- **Facilité d'Utilisation** : Offre des méthodes pratiques pour manipuler les éléments (ajout, suppression, tri).
- **Syntaxe** : Utilise la classe `ArrayList` et nécessite l'importation de `java.util.ArrayList`.

```
1 import java.util.ArrayList;
2 ArrayList<Integer> numbers = new ArrayList<>();
3 numbers.add(1);
4 numbers.add(2);
5 numbers.add(3);
6 System.out.println(numbers.get(2)); // Affiche 3, accès par méthode
```



Liste de Tableaux - Exemple

Choix entre Tableau et ArrayList

- **Tableau** : Idéal pour les données dont la taille ne change jamais ou pour les besoins de performance critique.
- **ArrayList** : Préférable pour les structures de données dynamiques où la flexibilité et la facilité de gestion sont requises.

Ces structures facilitent le stockage et la manipulation des données en Java, mais il est crucial de choisir la bonne selon les besoins spécifiques de votre application.



HashMap - Définition

Utilisation de HashMap en Java

HashMap est une structure de données en Java qui stocke des paires clé-valeur. Elle est idéale pour une recherche rapide des valeurs associées à une clé donnée, et pour stocker des données qui ne nécessitent pas d'ordre particulier.



HashMap - Définition

Définition

- **HashMap** : Une collection qui associe des clés uniques à des valeurs. Chaque clé peut être mappée au plus à une seule valeur.
- **Fonctionnalités principales** : Ajout, suppression, accès rapide via les clés, et vérification de l'existence d'une clé ou valeur.



HashMap - Exemple

Exemple et Utilisation

- **Création et Ajout** : Initialiser une **HashMap** et y ajouter des paires clé-valeur.

Exemple : Stocker des notes d'examen par matière.

Java

```
1 HashMap<String, Integer> notesExamens = new HashMap<>();  
2 notesExamens.put("Mathématiques", 75);  
3 notesExamens.put("Sociologie", 85);  
4 System.out.println(notesExamens.toString()); // Affiche {Mathématiques=75, Sociologie=85}
```

- **Accès à une Valeur** : Récupérer la valeur associée à une clé spécifique à l'aide de la méthode **get**.



HashMap - Exemple

Exemple et Utilisation

- **Accès à une Valeur** : Récupérer la valeur associée à une clé spécifique à l'aide de la méthode **get**.

Exemple : Obtenir la note d'anglais.

Java

```
1 System.out.println(notesExamens.get("Anglais")); // Affiche 95
```

- **Modification et Suppression** : Modifier une valeur existante avec **replace** et supprimer une entrée avec **remove**.



HashMap - Exemple

Exemple et Utilisation

Exemple : Mettre à jour une note et enlever une matière.

Java

```
1 notesExamens.replace("Mathématiques", 80);  
2 notesExamens.remove("Sociologie");
```

- **Insertion Conditionnelle :** Utiliser `putIfAbsent` pour ajouter une clé uniquement si elle n'est pas déjà présente.



HashMap - Exemple

Exemple et Utilisation

Exemple: Ajouter une note par défaut pour une matière absente.

Java

```
1 notesExamens.putIfAbsent("Philosophie", 70);
```

- **Vérifier l'Existence :** Vérifier si une clé ou une valeur existe avec `containsKey` et `containsValue`.



HashMap - Exemple

Exemple et Utilisation

Exemple : Vérifier si une note spécifique est présente.

Java

```
1 System.out.println(notesExamens.containsKey("Mathématiques")); // Affiche true
2 System.out.println(notesExamens.containsValue(100)); // Affiche true
```

HashMap est une structure de données flexible et efficace pour travailler avec des données associatives dans Java, offrant des performances optimales pour des recherches rapides et des mises à jour fréquentes.



HashMap - Exemple



```
1 import java.util.HashMap;
2
3 class Main {
4     public static void main(String[] args) {
5         HashMap<String, Integer> notesExamens = new HashMap<String, Integer>();
6
7         notesExamens.put("Mathématiques", 75);
8         notesExamens.put("Sociologie", 85);
9         notesExamens.put("Anglais", 95);
10        notesExamens.put("Programmation Informatique", 100);
11
12        notesExamens.forEach((sujet, note) -> {
13            notesExamens.replace(sujet, note + 10); // ajoute 10 points à chaque note
14        });
15
16        System.out.println(notesExamens.toString());
17    }
18 }
```