



Cours Java Standard



Devenir un pro Java



Date : ../../20..





Les bases de Java

✓✓ C'est quoi Java ?

Java est un langage de programmation orienté objet créé par James Gosling et Patrick Naughton en 1995, employés de la société Sun Microsystems qui a été ensuite rachetée en 2009 par la société Oracle qui détient et maintient désormais Java.

- 97 % des bureaux d'entreprise exécutent Java
- 89 % des bureaux (ou ordinateurs) des Etats-Unis exécutent Java
- 9 millions de développeurs Java dans le monde
- 3 milliards de téléphones mobiles exécutent Java
- 100 % des lecteurs Blu-ray livrés avec Java
- 5 milliards de cartes Java utilisées
- 125 millions de périphériques TV exécutent Java

Java est utilisé pour :

- Applications mobiles (en particulier les applications Android)
- Applications de bureau
- Des applications Web
- Serveurs Web et serveurs d'applications
- Jeux
- Et bien plus encore!

✓✓ Pourquoi utiliser Java ?

- Java fonctionne sur différentes plates-formes (Windows, Mac, Linux etc.)
- C'est l'un des langages de programmation les plus populaires au monde
- Il a une forte demande sur le marché du travail actuel
- Il est facile à apprendre et simple à utiliser
- Il est open-source et gratuit
- Il est sécurisé, rapide et puissant
- Il a un énorme support communautaire (des dizaines de millions de développeurs)
- Java est un langage orienté objet qui donne une structure claire aux programmes et permet de réutiliser le code, réduisant ainsi les coûts de développement
- Comme Java est proche de C++ et C# , il est facile pour les programmeurs de passer à Java

✓✓ Installation Java

Pour vérifier si Java est installé sur un PC Windows, tapez ce qui suit dans l'invite de commande (cmd) :

```
C:\Users\Your Name>java -version
```

Si Java est installé, vous verrez quelque chose comme ceci (selon la version)

```
java version "11.0.1" 2018-10-16 LTS

Java(TM) SE Runtime Environment 18.9 (build 11.0.1+13-LTS)

Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.1+13-LTS, mixed
mode)
```

Si Java n'est pas installé sur votre ordinateur, vous pouvez le télécharger gratuitement sur

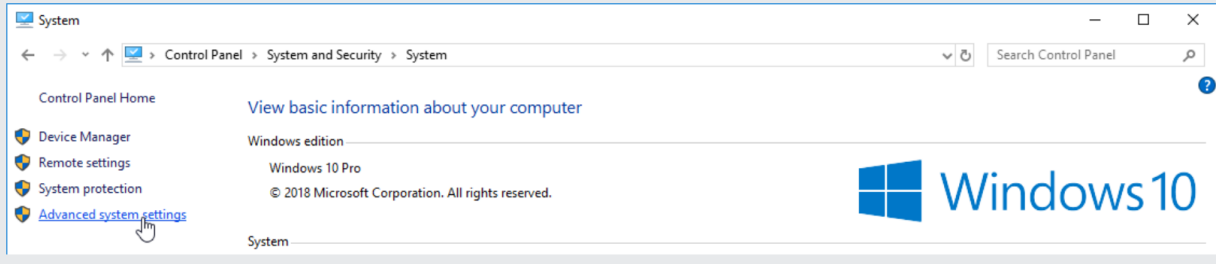
<https://www.oracle.com/java/technologies/downloads/#java20>.

Le kit de développement Java (JDK) est un ensemble d'outils essentiels pour le développement d'applications Java.

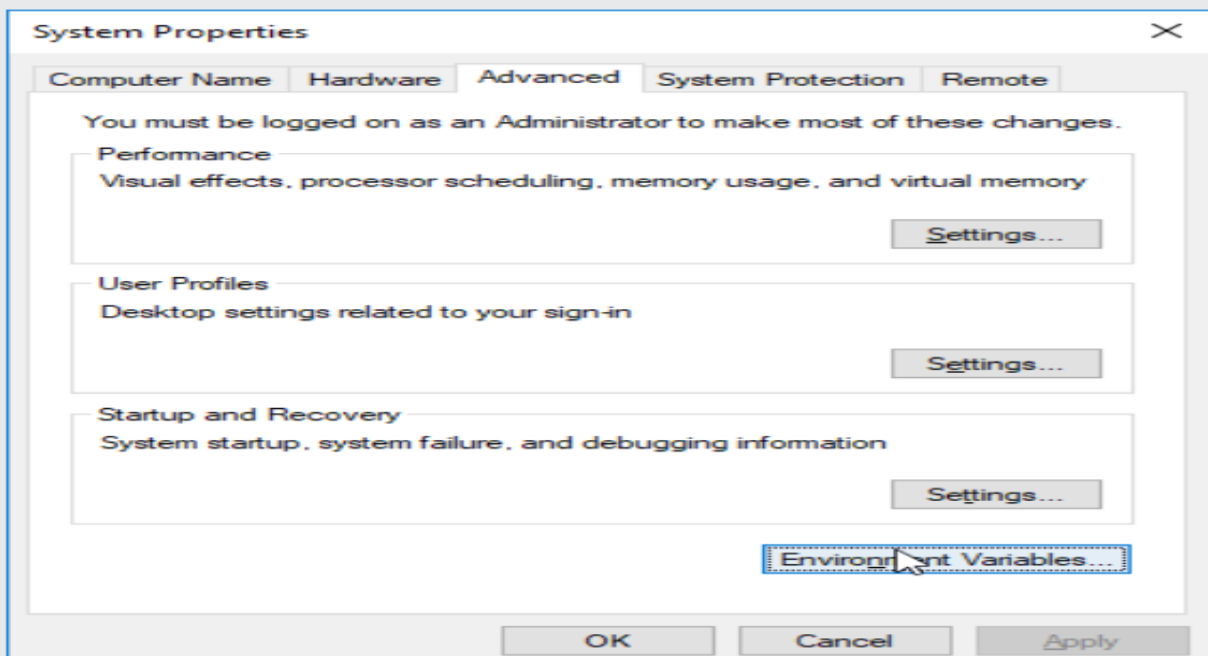
Il permet de fournir un environnement complet pour écrire, compiler, déboguer et exécuter des programmes Java

✓✓ Mettre Java dans les variables d'environnement

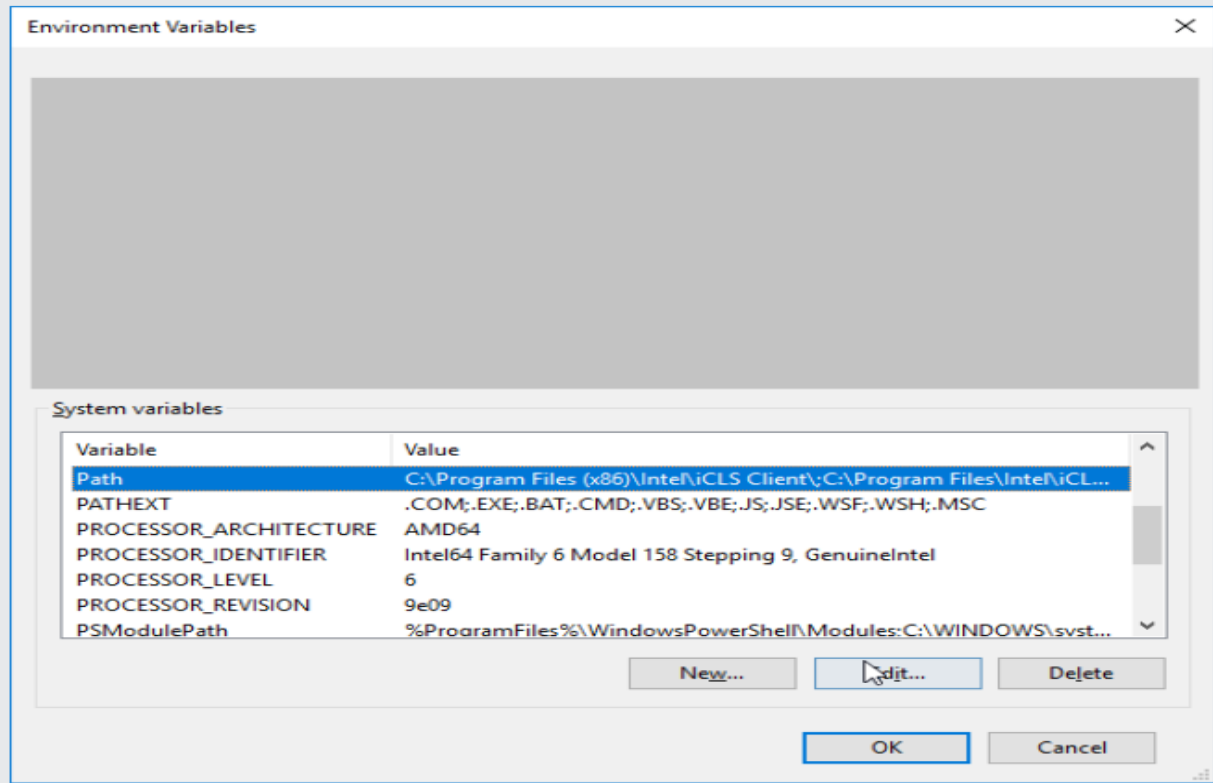
Étape 1



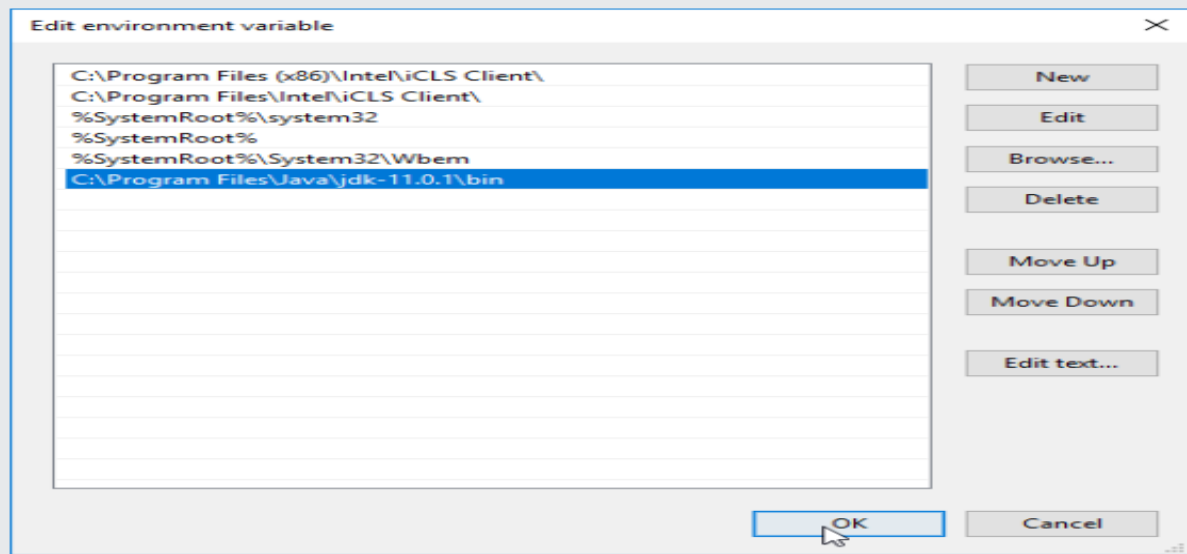
Étape 2



Étape 3



Étape 4



Fermer le cmd avant de retaper java -version

✓✓ Installation de IntelliJ version ultime

1. Install JetBrains IntelliJ IDEA version 2021.3 (Ultimate Edition) =>
<https://www.jetbrains.com/idea/download/other.html>
2. Pour windows : Copiez le fichier .jar (fineagent.jar) dans le Disque C:
3. Allez dans le repertoire ou s'est installer JetBrains (Exemple : C:\Program Files\JetBrains\IntelliJ IDEA 2021.3\bin)

ouvrez le fichier *.exe.vmoptions (Exemple : idea64.exe.vmoptions) avec Notepad++ ou bloc note puis ajouter à la fin ce

qui suit selon votre système d'exploitation puis sauvegarder :

windows:

-javaagent:c:/fineagent.jar

mac:

-javaagent:/Users/neo/fineagent.jar

linux:

-javaagent:/home/neo/fineagent.jar

4. Ouvrez JetBrains et copiez ce code dans Activation Code et Active

5AYV1D1RE5-eyJsaWNlbnNISWQioiI1QVIWMUQxUkU1IiwibGljZW5zZWVOYW
1lIjoiaHR0cHM6Ly93d3cuamI3ZWljaGVuZ3podS5jb20iLCJhc3NpZ25lZU5hbW
UiOiIiLCJhc3NpZ25lZUVtYWlsIjoiIiwibGljZW5zZVJlc3RyaWN0aW9uIjoiIiw
iY2tDb25jdXJyZW50VXNlIjpmYWxzZSwicHJvZHVjdHMiOlt7ImNvZGUoIjJSSIs
ImZhbmGxiYWNRGF0ZSI6IjIwOTktMTItMzEiLCJwYXVlVXBUbyI6IjIwOTktMTIt

3WjAfMR0wGwYDVQQDDBRwcm9kMnktZnJvbS0yMDIwMTAxOTCCASIwDQYJ
KoZIhvcNAQEBBQADggEPADCCAQoCggEBAMK3eyr0+Oys/TwcQO+qYaXWGB
mXcEP4fR0bHHzZd/4WNGltXhecM80OWthA38BQRYAJBCKz/dSkO2Kj1H2y+7
KB5cIaOiJEyTESfTSgzQdot6laRBU8oxy9mmagI46M8zEEmpIPybY4YJj4HOwZi
BsMQEMxoTgMDLpuHX6uASoVhSj6koB55lOj4wEgsQBeAMWTAXmTI88ixE179
J8aBUkBGpL8w/tZzl9BJXZNF15gPfkS58rw8cdPzXLS0Yym37A2/KKFhfHzJc5K
hbaxqYzmkAfTMqPsIqCQ1lQUAqfiPn2gN2I1Z3/cQuEW27M55fXVr2DduQe5D
WzYJs85L50CAwEAAaOBmTCBljAJBgNVHRMEAjAAMB0GA1UdDgQWBQBk2hEi
lvWFQcCTR+gxI0z0wIQC/zBIBgNVHSMEQTA/gBSjnrZIZ0ISNkG9beC5tKBSi5f
xs6EcpBowGDEWMBQGA1UEAwwNSmV0UHJvZmlsZSBDQYIJANJssYOyg3nhM
BMGA1UdJQMMMAoGCCsGAQUFBwMBMAsGA1UdDwQEAwIFoDANBgkqhkiG9
w0BAQsFAAOCAQEAsCQBjO5wttco/Z5cj/o4GBrku8UtBBBVFq4xsBanshTHm4
deVxcTvta4aScV0TPKcaLqGqWx8A9v8XXO8dBbCuyXvWZteZ/C2Covg1xXiM9
9lz7VxqpjVmLdKanZn5u0gQSiYJdcfF+TdbmEIeSONN/kLXNq2hXdJQK2zk2J25
UZqu5EibRtTbdOzw6ZcfwJ8uOntXfsmAhnNICP3Wf/4wR/mwB0Ka4S+JA3IbF5
MUUmUZ/fjUaFarnin70us+Vxf/sZUi7u67wilvwVV0NAqDpthHUV0NRc4q+yOr2D
t/uCHdy4XRXLJfAv/z9/xBwNZZALNz3EtQL6IeIWWJByl3g==

✓✓ Premier bout de code

Créons notre premier fichier Java, appelé Main.java.

Le fichier doit contenir un message « Hello World », qui est écrit avec le code suivant

📁 Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println("Hello World");  
  
    }  
  
}
```

📁 Notes :

- ❖ Le nom de la classe doit être identique au nom du fichier.
- ❖ Chaque ligne de code exécutée en Java doit se trouver dans une classe
- ❖ Le nom de classe doit toujours commencer par une première lettre majuscule.
- ❖ Java est sensible à la casse : « MyClass » et « myclass » sont différentes.
- ❖ **System** est une classe Java intégrée qui contient des éléments utiles, tels que **out**, qui est l'abréviation de « sortie ». La méthode **println()**, abréviation de « print line », est utilisée pour imprimer une valeur à l'écran (ou dans un fichier). Il existe également une méthode **print()** similaire à **println()**. La seule différence est qu'il n'insère pas de nouvelle ligne à la fin de la sortie :

✓✓ Les commentaires en Java

Les commentaires peuvent être utilisés pour expliquer le code Java et le rendre plus lisible. Il peut également être utilisé pour empêcher l'exécution lors du test de code alternatif.

Commentaires sur une seule ligne

Les commentaires sur une seule ligne commencent par deux barres obliques (`//`). Tout texte entre `//` et la fin de la ligne est ignoré par Java (ne sera pas exécuté).

Cet exemple utilise un commentaire sur une seule ligne avant une ligne de code :

Exemple :

```
public static void main(String[] args) {  
    //Ceci est un commentaire  
  
    System.out.println("Hello World");  
  
    System.out.println("Hello World"); // This is a comment  
}
```

Commentaires multiligne

Les commentaires sur plusieurs lignes commencent `/*` et se terminent par `*/`. Tout texte entre `/*` et `*/` sera ignoré par Java.

Cet exemple utilise un commentaire multiligne (un bloc de commentaires) pour expliquer le code :

 Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        /*  
  
        * Le code ci-dessous affiche les mots Hello World  
  
        * à l'écran, et c'est incroyable  
  
        */  
  
        System.out.println("Hello World");  
  
    }  
  
}
```

✓✓ Les variables et constantes en Java

Les variables

Les variables sont des conteneurs permettant de stocker des valeurs de données.

Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        float note1 = 5.99f;  
  
        int age; // Déclaration  
  
        age = 15; // Affectation de valeur  
  
    }  
}
```

Les règles générales pour nommer les variables sont :

- ❖ Les noms peuvent contenir des lettres, des chiffres, des traits de soulignement et des signes dollar.
- ❖ Les noms doivent commencer par une lettre
- ❖ Les noms doivent commencer par une lettre minuscule et ne peuvent pas contenir d'espace.
- ❖ Les noms peuvent aussi commencer par \$ et _ (Déconseillé)
- ❖ Les noms sont sensibles à la casse ("myVar" et "myvar" sont des variables différentes)
- ❖ Les mots réservés (comme les mots-clés Java, tels que **int** ou **boolean**) ne peuvent pas être utilisés comme noms

Les constantes

Une constante est une variable dont la valeur est immuable et en lecture seule, c'est à dire, on ne pourra pas modifier sa valeur dans le programme;

Pour déclarer une constante en java on utilise le mot clé **final**.

Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        final int nbMaxEnfant = 15;  
  
        nbMaxEnfant = 20; // générera une erreur: Impossible  
                           d'attribuer une valeur à une variable finale  
  
    }  
  
}
```

✓✓ Les type de données en java

En Java, il existe différents **types** de variables. Les types de données sont divisés en deux groupes :

- ❖ **Les types de données primitifs** - tels que les entiers (**byte**, **short**, **int**, **long**), les nombres à virgule flottante (**float**, **double**), les caractères **char** et les booléens **boolean**.
- ❖ **Les types de données non primitifs** - tels que **String**, **Arrays**, **Class**, **Interface**... (vous en apprendrez davantage à ce sujet dans un chapitre ultérieur)

📁 **Note 1 :** Un type de données primitif spécifie la taille et le type des valeurs de variable et ne comporte aucune méthode supplémentaire. Il existe huit types de données primitifs en Java :

Type	Taille	Description
byte	1 octet	Stocke des nombres entiers de -128 à 127
short	2 octets	Stocke des nombres entiers de -32 768 à 32 767
int	4 octets	Stocke des nombres entiers de -2 147 483 648 à 2 147 483 647
long	8 octets	Stocke des nombres entiers de -9 223,372,036 854,775 808 à 9 223,372,036 854 775 807. Notez que vous devez terminer la valeur par un « L » :
float	4 octets	Stocke les nombres fractionnaires. Suffisant pour stocker 6 à 7 chiffres décimaux. Notez que vous devez terminer la valeur par un « f »
double	8 octets	Stocke les nombres fractionnaires. Suffisant pour stocker 15 chiffres décimaux. Notez que vous devez terminer la valeur par un « d »
boolean	1 octet	Stocke des valeurs vraies ou fausses
char	2 octets	Stocke un seul caractère / lettre ou des valeurs ASCII

 Note 2 :

- ❖ Même s'il existe de nombreux types numériques en Java, les plus utilisés pour les nombres sont **int** (pour les nombres entiers) et **double** (pour les nombres à virgule flottante).
- ❖ La **précision** d'une valeur à virgule flottante indique le nombre de chiffres que la valeur peut avoir après la virgule décimale. La précision de **float** n'est que de six ou sept chiffres décimaux, tandis que les variables **double** ont une précision d'environ 15 chiffres. Il est donc plus sûr de l'utiliser **double** pour la plupart des calculs.
- ❖ Un nombre à virgule flottante peut également être un nombre scientifique avec un « **e** » pour indiquer la puissance de 10.
- ❖ Une chaîne en Java est en fait un type de données **non primitif**, car elle fait référence à un objet. L'objet String possède des méthodes utilisées pour effectuer certaines opérations sur les chaînes.
- ❖ Les types de données non primitifs sont appelés **types de référence** car ils font référence à des objets. La principale différence entre les types de données **primitifs** et **non primitifs** est :
 - Les types primitifs sont prédéfinis (déjà définis) en Java. Les types non primitifs sont créés par le programmeur et ne sont pas définis par Java (sauf pour **String**).
 - Les types non primitifs peuvent être utilisés pour appeler des méthodes afin d'effectuer certaines opérations, contrairement aux types primitifs.
 - Un type primitif a toujours une valeur, tandis que les types non primitifs peuvent l'être **null**.
 - Un type primitif commence par une lettre minuscule, tandis que les types non primitifs commencent par une lettre majuscule.

 Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        byte nbFemme = 10;  
  
        short nbEnfant = 5000;  
  
        int age = 30;  
  
        long matricule = 150000000000L;  
  
        float note = 5.99f;  
  
        double taille = 19.99d;  
  
        char maLettre = 'D';  
  
        char myVar1 = 65, myVar2 = 66, myVar3 = 67; //code ASCII  
  
        boolean diabetique = false;  
  
        String nom = "Modou";  
  
  
        int x = 5, y = 6, z = 50;  
  
        float f1 = 35e3f;  
  
        double d1 = 12E4d;  
  
    }  
  
}
```

✓✓ Le transtypage ou casting

La conversion de type consiste à attribuer une valeur d'un type de données primitif à un autre type.

En Java, il existe deux types de casting :

- ❖ **Élargissement de la diffusion** (automatiquement) - Si on prend une variable de plus petite taille on le met dans une variable de plus grande taille, le transtypage se fait automatiquement on a pas besoin de caster manuellement.

Voici l'ordre des types de la plus petite taille à la plus grande taille.

byte-> short-> char-> int-> long-> float->double

- ❖ **Narrowing Casting** (manuellement) - Si on prend une variable de plus grande taille on le met dans une variable de plus petite taille, le transtypage doit se faire manuellement.

Voici l'ordre des types de la plus grande taille à la plus petite taille.

double-> float-> long-> int-> char-> short->byte

📁 **Exemple 1 :** Le transtypage s'effectue automatiquement (mettre tasse de thé dans une baril 😄)

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int myInt = 9; //plus petite taille  
  
        double myDouble = myInt; // myDouble reste double car elle à  
        un type plus grande que celle de myInt  
  
        System.out.println(myInt);        // Outputs 9  
  
        System.out.println(myDouble);    // Outputs 9.0  
  
    }  
}
```

📁 **Exemple 2 :** Le transtypage s'effectue manuellement (mettre un baril dans une tasse de thé 😄)

```
public class Main {  
  
    public static void main(String[] args) {  
  
        double myDouble = 9.78d;  
  
        int myInt = (int) myDouble; // Manual casting: double to int  
  
        System.out.println(myDouble);    // Outputs 9.78  
  
        System.out.println(myInt);        // Outputs 9  
  
    }  
}
```

✓✓ Les opérateurs en Java

Les opérateurs sont utilisés pour effectuer des opérations sur des variables et des valeurs.

Java divise les opérateurs dans les groupes suivants :

- ❖ Opérateurs arithmétiques
- ❖ Opérateurs d'affectation
- ❖ Opérateurs de comparaison
- ❖ Opérateurs logiques

Les opérateurs arithmétiques

Les opérateurs arithmétiques sont utilisés pour effectuer des opérations mathématiques courantes.

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$++x$
--	Decrement	Decreases the value of a variable by 1	$--x$

Les opérateurs d'affectation

Les opérateurs d'affectation sont utilisés pour attribuer des valeurs aux variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3

Les opérateurs logiques

Vous pouvez également tester des valeurs **true** ou **false** avec des opérateurs logiques. Les opérateurs logiques sont utilisés pour déterminer la logique entre des variables ou des valeurs :

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
	Logical or	Returns true if one of the statements is true	x < 5 x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Les opérateurs de comparaison

Les opérateurs de comparaison sont utilisés pour comparer deux valeurs (ou variables). C'est important en programmation, car cela nous aide à trouver des réponses et à prendre des décisions.

La valeur de retour d'une comparaison est soit **true** ou **false**. Ces valeurs sont appelées *valeurs booléennes*.

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

✓✓ Les échappement

Étant donné que les chaînes doivent être écrites entre guillemets, Java comprendra mal cette chaîne et générera une erreur :

📁 Exemple :

```
public class Main {  
    public static void main(String[] args) {  
        String txt = "We are the so-called \"Vikings\" from the  
        north.";  
    }  
}
```

La solution pour éviter ce problème est d'utiliser le **caractère d'échappement antislash** .

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

📁 Exemple :

```
public class Main {  
    public static void main(String[] args) {  
        String txt = "We are the so-called \"Vikings\" from the  
        north.";  
    }  
}
```

 Note :

Les autres séquences d'échappement courantes valides en Java sont :

Code	Result
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace

✓✓ Les package en java

Un package en Java est utilisé pour regrouper les classes associées. Considérez-le comme **un dossier dans un répertoire de fichiers** . Nous utilisons des packages pour éviter les conflits de noms et pour écrire un code mieux maintenable. Les packages sont divisés en deux catégories :

- ❖ Packages intégrés (packages de l'API Java)
- ❖ Packages définis par l'utilisateur (créez vos propres packages)

Packages intégrés

L'API Java est une bibliothèque de classes pré-écrite, dont l'utilisation est gratuite, incluses dans l'environnement de développement Java.

La bibliothèque contient des composants pour la gestion des entrées, la programmation de bases de données et bien plus encore.

La bibliothèque est divisée en **packages** et **classes** . Cela signifie que vous pouvez soit importer une seule classe (avec ses méthodes et attributs), soit un package entier contenant toutes les classes appartenant au package spécifié.

Pour utiliser une classe ou un package de la bibliothèque, vous devez utiliser le mot-clé **import** :

```
import package.name.Class;    // Import a single class
import package.name.*;       // Import the whole package
```

Exemple :

Dans l'exemple ci-dessus, **java.util** est un package, tandis que **Scanner** est une classe du package **java.util**.

Pour utiliser la classe **Scanner**, créez un objet de la classe et utilisez l'une des méthodes disponibles trouvées dans la documentation de la classe **Scanner**. Dans notre exemple, nous utiliserons la méthode **nextLine()** qui permet de lire une ligne complète :

```
import java.util.Scanner;

class MyClass {

    public static void main(String[] args) {

        Scanner myObj = new Scanner(System.in);

        System.out.println("Entrer votre nom");

        String userName = myObj.nextLine();

        System.out.println("Salut : " + userName);

    }

}
```

 **Note :** Pour importer un package entier, terminez la phrase par un astérisque (*). L'exemple suivant importera TOUTES les classes du package **java.util** :

```
import java.util.*;
```

Créer son propre Package

Pour créer votre propre package, vous devez comprendre que Java utilise un répertoire de système de fichiers pour les stocker. Tout comme les dossiers sur votre ordinateur :

```
└─ root
  └─ mypack
    └─ MyPackageClass.java
```

Pour créer un package, utilisez le mot-clé **package**

```
package mypack;

class MyPackageClass {

    public static void main(String[] args) {

        System.out.println("This is my package!");

    }

}
```

✓✓ Entrée utilisateur Java (Scanner)

La classe **Scanner** est utilisée pour obtenir les entrées de l'utilisateur et elle se trouve dans le package **java.util**.

Pour utiliser la classe **Scanner**, créez un objet de la classe et utilisez l'une des méthodes disponibles trouvées dans la documentation de la classe **Scanner**. Dans notre exemple, nous utiliserons la méthode **nextLine()** utilisée pour lire les chaînes :

```
import java.util.Scanner; // Import the Scanner class

public class Main {

    public static void main(String[] args) {

        Scanner myObj = new Scanner(System.in);

        System.out.println("Enter username");

        String userName = myObj.nextLine(); // Read user input

        System.out.println("Bonjour : " + userName); // Output user
        input

    }

}
```

Types d'entrée

Dans l'exemple ci-dessus, nous avons utilisé la méthode **nextLine()** pour lire les chaînes. Pour lire d'autres types, consultez le tableau ci-dessous :

Méthodes	Description
nextBoolean()	Lit une valeur booléenne de l'utilisateur
nextByte()	Lit une valeur d'octet de l'utilisateur
nextDouble()	Lit une double valeur de l'utilisateur
nextFloat()	Lit une valeur flottante de l'utilisateur
nextInt()	Lit une valeur entier de l'utilisateur
nextLine()	Lit une valeur de chaîne de l'utilisateur
nextLong()	Lit une longue valeur de l'utilisateur
nextShort()	Lit une valeur courte de l'utilisateur

 **Exemple :** Nous utilisons ici différentes méthodes pour lire des données de différents types :

```
import java.util.Scanner;

class Main {

    public static void main(String[] args) {

        Scanner myObj = new Scanner(System.in);

        System.out.println("Enter name, age and salary:");

        // String input

        String name = myObj.nextLine();
```

```
// Numerical input

int age = myObj.nextInt();

double salary = myObj.nextDouble();


// Output input by user

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Salary: " + salary);

}

}
```

✓✓ Les structures conditionnelles

Java contient les instructions conditionnelles suivantes :

- ❖ Utiliser **if** pour spécifier un bloc de code à exécuter, si une condition spécifiée est vraie.
- ❖ Utiliser **else** pour spécifier un bloc de code à exécuter, si la même condition est fausse.
- ❖ Utiliser **else if** pour spécifier une nouvelle condition à tester, si la première condition est fausse.
- ❖ Utiliser **switch** pour spécifier de nombreux blocs de code alternatifs à exécuter

Syntaxe if :

```
if (condition) {  
    // block of code to be executed if the condition is true  
}
```

Exemple if :

```
public static void main(String[] args) {  
  
    int x = 20;  
  
    int y = 18;  
  
    if (x > y) {  
  
        System.out.println("x is greater than y");  
  
    }  
  
}
```

Syntaxe if...else:

```
if (condition) {
    // block of code to be executed if the condition is true
} else {
    // block of code to be executed if the condition is false
}
```

Exemple if...else :

```
public static void main(String[] args) {

    int time = 20;

    if (time < 18) { System.out.println("Good day.");

    } else {System.out.println("Good evening.");}

}
```

Syntaxe else...if:

```
if (condition1) {
    // block of code to be executed if condition1 is true
} else if (condition2) {
    // block of code to be executed if the condition1 is false and condition2 is true
} else {
    // block of code to be executed if the condition1 is false and condition2 is false
}
```

Exemple else...if :

```
public static void main(String[] args) {

    int time = 20;

    if (time < 18) { System.out.println("Good day.");

    } else if(time < 1) {System.out.println("Good evening.");}

    else {System.out.println("Good .");}

}
```

Les opérateurs ternaires

Il existe également un raccourci **if else**, appelé **opérateur ternaire** car il se compose de trois opérandes.

Il peut être utilisé pour remplacer plusieurs lignes de code par une seule ligne, et est le plus souvent utilisé pour remplacer de simples instructions **if else** :

Syntaxe :

```
variable = (condition) ? expressionTrue : expressionFalse;
```

 **Exemple :** Reprenons l'exemple précédent avec les ternaires

```
public static void main(String[] args) {  
  
    int time = 20;  
  
    String result = (time < 18) ? "Good day." : "Good evening.";  
  
    System.out.println(result);  
  
}
```

switch case

Au lieu d'écrire **plusieurs** déclarations **if..else**, vous pouvez utiliser la déclaration **switch**.

L'instruction **switch** sélectionne l'un des nombreux blocs de code à exécuter. On utilise switch case si on connaît les valeurs à tester à l'avance.

Syntaxe :

```
switch(expression) {  
    case x:  
        // code block  
        break;  
    case y:  
        // code block  
        break;  
    default:  
        // code block  
}
```

Voilà comment cela fonctionne:

- L'expression **switch** est évaluée une fois.
- La valeur de l'expression est comparée aux valeurs de chaque **case**.
- S'il y a une correspondance, le bloc de code associé est exécuté.
- Les mots-clés **break** et **default** sont facultatifs et seront décrits plus loin dans ce chapitre.

Exemple :

```
public static void main(String[] args) {  
  
    int day = 4;  
  
    switch (day) {  
  
        case 1:  
  
            System.out.println("Monday");  
  
            break;  
  
    }  
}
```

```
        case 2:

            System.out.println("Tuesday");

            break;

        case 3:

            System.out.println("Wednesday");

            break;

        case 4:

            System.out.println("Thursday");

            break;

        case 5:

            System.out.println("Friday");

            break;

        case 6:

            System.out.println("Saturday");

            break;

        case 7:

            System.out.println("Sunday");

            break;

        default:

            System.out.println("Jour introuvable !");

    }

    // Outputs "Thursday" (day 4)

}
```

✓✓ Les principales méthodes de chaîne

Une chaîne en Java est en fait un objet contenant des méthodes permettant d'effectuer certaines opérations sur des chaînes.

 **length()** : Permet de retrouver la longueur d'une chaîne.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        String txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
  
        System.out.println("La longueur de la chaîne txt est: " +  
            txt.length());  
  
    }  
  
}
```

 **toUpperCase()** et **toLowerCase()** : Permettent respectivement de transformer une chaîne en majuscule et en minuscule.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        String txt = "Hello World";  
  
        System.out.println(txt.toUpperCase()); // "HELLO WORLD"  
  
        System.out.println(txt.toLowerCase()); // "hello world"  
  
    }  
  
}
```

 **indexOf()** : renvoie l'index (la position) de la première occurrence d'un texte spécifié dans une chaîne (y compris les espaces). Elle renvoie -1 si l'élément n'est pas trouvé:

```
public class Main {

    public static void main(String[] args) {

        String txt = "Please locate where 'locate' occurs!";

        System.out.println(txt.indexOf("locate")); // Outputs 7

    }

}
```

 **lastIndexOf()** : renvoie l'index (la position) de la dernière occurrence d'un texte spécifié dans une chaîne (y compris les espaces). Elle renvoie -1 si l'élément n'est pas trouvé :

```
public static void main(String[] args) {

    String myStr = "Bonjour la planète Terre, vous êtes une
grande planète.";

    System.out.println(myStr.lastIndexOf("planète")); //47

}
```

 **concat()** : Permet de concaténer deux chaînes.

```
public class Main {

    public static void main(String[] args) {

        String prenom = "Modou ", nom = "Sarr";

        System.out.println("Bonjour " + prenom.concat(nom));

    }

}
```

 **charAt()** : Permet de renvoyer le caractère à l'index spécifié dans une chaîne. L'index du premier caractère est 0. Elle lève une exception si l'index passé en argument n'existe pas. Il faut noter aussi que si on recherche dans une chaîne qui a plusieurs mots "Modou Sarr", la recherche commence ici par Sarr.

```
public static void main(String[] args) {

    String myStr = "Hello";

    System.out.println(myStr.charAt(0)); //H

}
```

 **compareTo()** : Compare deux chaînes. Elle renvoie 0 si les chaînes sont égales, sinon, elle renvoie le nombre de caractères de différence.

```
public static void main(String[] args) {

    String myStr1 = "Hello";

    String myStr2 = "Hello";

    System.out.println(myStr1.compareTo(myStr2)); // Retourne 0

    car ils sont égaux

}
```

 **compareToIgnoreCase()** : Compare deux chaînes en ignorant les différences entre les minuscules et les majuscules. Elle renvoie 0 si les chaînes ne sont pas égales, sinon, elle renvoie le nombre de caractères de différence.

```
public static void main(String[] args) {

    String myStr1 = "HELLO, myStr2 = "hello";

    System.out.println(myStr1.compareToIgnoreCase(myStr2)); //0

}
```

 **equals()** : Compare deux chaînes et renvoie vrai si les chaînes sont égales sinon, elle renvoie faux. Il y a aussi equalsIgnoreCase() qui fait la même chose en ignorant la casse.

```
public static void main(String[] args) {

    String myStr1 = "Hello";

    String myStr2 = "Hello";

    String myStr3 = "Another String";

    System.out.println(myStr1.equals(myStr2)); // true

    System.out.println(myStr1.equals(myStr3)); // false

}
```

 **contains() et contentEquals()** : Vérifie respectivement si une chaîne contient une séquence de caractères ou si une chaîne contient exactement la chaîne passée en paramètre:

```
public static void main(String[] args) {

    String myStr = "Hello";

    System.out.println(myStr.contains("Hel")); // true

    System.out.println(myStr.contains("e")); // true

    System.out.println(myStr.contains("Hi")); // false

}
```

 **copyValueOf(tab, indexStart, indexEnd)** : Renvoie un nouveau tableau de chaîne de caractère et y copie les caractères du tableau passé en paramètre.

```
public static void main(String[] args) {

    char[] tabCaracteres = { 'H', 'e', 'l', 'l', 'o' };

    String myStr = "";
```

```

    myStr = myStr.copyOfValueOf(tabCaracteres, 0, 5);

    System.out.println("Returned String: " + myStr); //Hello
}

```

 **startsWith() et endsWith:** Vérifie respectivement si une chaîne commence ou termine par le(s) caractère(s) spécifié(s).

```

public static void main(String[] args) {

    String myStr = "Hello";

    System.out.println(myStr.startsWith("Hel")); // true

    System.out.println(myStr.startsWith("llo")); // false

    System.out.println(myStr.endsWith("o")); // true

}

```

 **hashCode():** Renvoie le code de hachage d'une chaîne :

```

public static void main(String[] args) {

    String myStr = "Hello";

    System.out.println(myStr.hashCode()); // 69609650

}

```

 **isEmpty():** Vérifie si une chaîne est vide ou non :

```

public static void main(String[] args) {

    String myStr1 = "Hello";

    String myStr2 = "";

    System.out.println(myStr1.isEmpty()); //false

    System.out.println(myStr2.isEmpty()); //true

}

```

 **replace() ou replaceAll():** Remplacent toutes les occurrences d'un caractère ou séquence de caractère par un autre dans une chaîne de caractère. Pour replaceAll(), les paramètres doivent être des chaînes de caractère et non pas un caractère.

```
public static void main(String[] args) {

    String myStr = "Hello";

    String txt = "bonjour";

    System.out.println(myStr.replace('l', 'p')); //hepoo

    System.out.println(txt.replace("jour", "soir")); //bonsoir

}
```

 **replaceFirst():** Remplace la première occurrence trouvée. Les paramètres doivent être des chaînes de caractère et non pas un caractère.

```
public static void main(String[] args) {

    String txt = "bonjour bonjour";

    System.out.println(txt.replaceFirst("jour", "soir"));

    //bonsoir bonjour

}
```

 **substring():** Permet d'extraire une séquence de caractère dans une chaîne de caractère.

```
public static void main(String[] args) {

    String txt = "bonjour bonjour";

    System.out.println(txt.substring(0, 3)); //bon

}
```

 **toCharArray():** Permet de convertir une chaîne de caractère en un tableau de caractère.

```
public static void main(String[] args) {  
  
    String txt = "bonjour";  
  
    System.out.println(txt.toCharArray()[0]); //b  
  
}
```

 **trim():** Supprimez les espaces des deux côtés d'une chaîne :

```
public static void main(String[] args) {  
  
    String myStr = "      Hello World!      ";  
  
    System.out.println(myStr);  
  
    System.out.println(myStr.trim());  
  
}
```

 **codePointAt(position):** Renvoie le code ASCII d'une lettre

```
public static void main(String[] args) {  
  
    String myStr = "Allo";  
  
    int result = myStr.codePointAt(0);  
  
    System.out.println(result); //le code ascii de A est 65  
  
}
```

✓✓ Les principales méthodes mathématique

La classe Java Math dispose de nombreuses méthodes qui vous permettent d'effectuer des tâches mathématiques sur des nombres.

 **Math.max(x,y)** : Permet de trouver la valeur la plus élevée de x et y

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println(Math.max(5, 10)); //Affiche 10  
  
    }  
  
}
```

 **Math.min(x,y)** : Permet de trouver la valeur la plus basse de x et y

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println(Math.min(5, 10)); //Affiche 5  
  
    }  
  
}
```

 **Math.sqrt(x)** : Renvoie la racine carrée de x

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println(Math.sqrt(64)); //Affiche 8.0 } }
```

 **Math.abs(x)** : Renvoie la valeur absolue (positive) de x

```
public static void main(String[] args) {  
  
    System.out.println(Math.abs(-4.7)); //Affiche 4.7  
  
}
```

 **Math.random(x)** : Renvoie Un nombre aléatoire compris entre 0,0 (inclus) et 1,0 (exclusif).

```
System.out.println(Math.random());
```

 Note:

Pour avoir plus de contrôle sur le nombre aléatoire, par exemple, si vous souhaitez uniquement un nombre aléatoire compris entre 0 et 100, vous pouvez utiliser la formule suivante :

```
int randomNum = (int) (Math.random() * 101); // 0 to 100
```

✓✓ Les boucles en java

Les boucles peuvent exécuter un bloc de code tant qu'une condition spécifiée est atteinte. Les boucles sont pratiques car elles permettent de gagner du temps, de réduire les erreurs et de rendre le code plus lisible.

La boucle while

La boucle **while** parcourt un bloc de code tant qu'une condition spécifiée est vrai:

Syntaxe :

```
while (condition) {  
    // code block to be executed  
}
```

Exemple :

```
public static void main(String[] args) {  
  
    int i = 0;  
  
    while (i < 5) {  
  
        System.out.println(i); //De 0 à 4  
  
        i++;  
  
    }  
  
}
```

La boucle for

Lorsque vous savez exactement combien de fois vous souhaitez parcourir un bloc de code, utilisez la boucle **for** au lieu d'une boucle **while** :

Syntaxe :

```
for (statement 1; statement 2; statement 3) {
    // code block to be executed
}
```

- ❖ L'instruction 1 est exécutée (une fois) avant l'exécution du bloc de code.
- ❖ L'instruction 2 définit la condition d'exécution du bloc de code.
- ❖ L'instruction 3 est exécutée (à chaque fois) après l'exécution du bloc de code.

Exemple 1 :

```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        System.out.println(i);
    }
}
```

Exemple 2 : Afficher les valeurs paires entre 0 et 10 :

```
public static void main(String[] args) {
    for (int i = 0; i <= 10; i = i + 2) {
        System.out.println(i);
    }
}
```

La boucle for-each

Il existe également une boucle " for-each ", qui est utilisée exclusivement pour parcourir les éléments d'un tableau :

Syntaxe :

```
for (type variableName : arrayName) {  
    // code block to be executed  
}
```

Exemple :

```
public static void main(String[] args) {  
    String[] cars = { "Volvo", "BMW", "Ford", "Mazda" };  
    for (String car : cars) {  
        System.out.println(car);  
    }  
}
```

Boucle imbriquées

Il est également possible de placer une boucle à l'intérieur d'une autre boucle. C'est ce qu'on appelle une boucle imbriquée .

La « boucle interne » sera exécutée une fois pour chaque itération de la « boucle externe » :

Exemple :

```
public static void main(String[] args) {  
  
    // Outer loop  
  
    for (int i = 1; i <= 2; i++) {  
  
        System.out.println("Outer: " + i); // Executes 2 times  
  
        // Inner loop  
  
        for (int j = 1; j <= 3; j++) {  
  
            System.out.println(" Inner: " + j); // Executes 6  
            times (2 * 3)  
  
        }  
  
    }  
  
}
```

✓✓ Les instructions break et continue

l'instruction break

Nous avons déjà vu l'instruction **break** utilisée dans un chapitre précédent. Il a été utilisé pour « sortir » d'une déclaration **switch**.

L'instruction **break** peut également être utilisée pour sortir d'une boucle .

Cet exemple arrête la boucle lorsque i est égal à 4 :

```
public static void main(String[] args) {  
  
    for (int i = 0; i < 10; i++) {  
  
        if (i == 4) { break; }  
  
        System.out.println(i);  
  
    }  
  
}
```

l'instruction continue

L'instruction **continue** interrompt une itération (dans la boucle), si une condition spécifiée se produit, et continue avec l'itération suivante dans la boucle. Cet exemple ignore la valeur 4 :

```
public static void main(String[] args) {  
  
    for (int i = 0; i < 10; i++) {  
  
        if (i == 4) { continue; //saute la valeur 4 }  
  
        System.out.println(i);  
  
    }  
  
}
```

✓✓ Les tableaux simples en java

Les tableaux sont utilisés pour stocker plusieurs valeurs dans une seule variable, au lieu de déclarer des variables distinctes pour chaque valeur.

📁 Pour **déclarer** un tableau, définissez le type de variable entre crochets :

```
String[] cars;
```

Nous avons maintenant déclaré une variable contenant un tableau de chaînes.

📁 Pour y **insérer** des valeurs, vous pouvez placer les valeurs dans une liste séparées par des virgules, entre accolades :

```
String[] cars = { "Volvo", "BMW", "Ford", "Mazda" };
```

📁 Pour **créer** un tableau d'entiers, vous pouvez écrire :

```
int[] myNum = { 10, 20, 30, 40 };
```

📁 Vous pouvez **accéder** à un élément du tableau en vous référant au numéro d'index. Cette instruction accède à la valeur du premier élément de cars :

```
public static void main(String[] args) {  
  
    String[] cars = { "Volvo", "BMW", "Ford", "Mazda" };  
  
    System.out.println(cars[0]); // Outputs Volvo  
  
}
```

📁 Pour **modifier** la valeur d'un élément spécifique, reportez-vous au numéro d'index :

```
public static void main(String[] args) {  
  
    String[] cars = { "Volvo", "BMW", "Ford", "Mazda" };  
  
    cars[0] = "Opel";  
  
    System.out.println(cars[0]); // Outputs Opel  
  
}
```

📁 Pour **connaître** le nombre d'éléments d'un tableau, utilisez la propriété **length**

```
public static void main(String[] args) {  
  
    String[] cars = { "Volvo", "BMW", "Ford", "Mazda" };  
  
    System.out.println(cars.length); // Outputs 4  
  
}
```

Les tableaux multidimensionnels

Un tableau multidimensionnel est un tableau de tableaux.

Les tableaux multidimensionnels sont utiles lorsque vous souhaitez stocker des données sous forme de tableau, comme un tableau avec des lignes et des colonnes (Chaque objet `{}` représente une ligne).

Pour créer un tableau à deux dimensions, ajoutez chaque tableau dans son propre ensemble d' **accolades**.

```
int[][] myNumbers = { { 1, 2, 3, 4 }, { 5, 6, 7 } };
```

Pour accéder aux éléments du tableau **myNumbers** , spécifiez deux index : un pour le tableau et un pour l'élément à l'intérieur de ce tableau. N'oubliez pas que chaque accolade est un tableau et l'index commence par zéro.

Cet exemple accède au troisième élément (2) du deuxième tableau (1) de myNumbers :

```
public static void main(String[] args) {  
  
    int[][] myNumbers = { { 1, 2, 3, 4 }, { 5, 6, 7 } };  
  
    System.out.println(myNumbers[1][2]); // Outputs 7  
  
}
```

Exemple 1 : Exemple avec une boucle

```
public static void main(String[] args) {

    int[][] myNumbers = { { 1, 2, 3, 4 }, { 5, 6, 7 } };

    System.out.println(myNumbers.length); //2

    for (int i = 0; i < myNumbers.length; ++i) { //sere a blouclé
        2 fois

        //premier tour i = 0, deuxième tour, i = 1

        for (int j = 0; j < myNumbers[i].length; ++j) {

            System.out.println(myNumbers[i][j]);

        }

    }

}
```

Exemple 2 : Exemple avec une boucle

```
public static void main(String[] args) {

    // Exemple tableau multidimensionnel

    String[][] tab1 = {{"Lundi", "Mardi", "Mercredi"}, {"Jeudi",
        "Vendredi", "Samedi", "Dimanche"}};

    for (int i = 0; i < tab1.length; i++) { //2

        // tab1[i].length = 3 => Lundi, Mardi et Mercredi

        for (int j = 0; j < tab1[i].length; j++) {
```

```

        System.out.println(tab1[i][j]);
    }

}

// Similaire à

String[] tab2 = {"Lundi", "Mardi", "Mercredi", "Jeudi",
                "Vendredi", "Samedi", "Dimanche"};

for (int i = 0; i < tab2.length; i++) {

    System.out.println(tab2[i]);

}

}

```

Exemple :

Exercice d'application

Écrire un programme qui permet de convertir Celsius en Fahrenheit ou l'inverse ;

- Le programme demande quelle conversion nous souhaitons effectuer, Celsius vers Fahrenheit ou l'inverse ;
- On n'autorise que les modes de conversion définis dans le programme (un simple contrôle sur la saisie fera l'affaire) ;
- Enfin, on demande à la fin à l'utilisateur s'il veut faire une nouvelle conversion, ce qui signifie que l'on doit pouvoir revenir au début du programme !

À toutes fins utiles, voici la formule de conversion pour passer des degrés Celsius en degrés Fahrenheit : $F = \frac{9}{5} \times C + 32$;
pour l'opération inverse, c'est comme ceci : $C = \frac{(F-32) \times 5}{9}$.



 Correction :

```
import java.util.Scanner;

class Hello {

    public static void main(String[] args) {

        //Notre objet Scanner

        Scanner sc = new Scanner(System.in);

        //initialisation des variables

        double aConvertir, convertit=0;

        char reponse=' ', mode = ' ';

        System.out.println("CONVERTISSEUR DEGRÉS CELSIUS ET DEGRÉS
        FAHRENHEIT");

        System.out.println("-----");

        do { //tant que réponse = 0 //boucle principale

            // Récupération du mode

            do { //tant que réponse n'est pas O ou N

                mode = ' ';

                System.out.println("Choisissez le mode de conversion : ");

                System.out.println("1 - Convertisseur Celsius - Fahrenheit");
```

```
System.out.println("2 - Convertisseur Fahrenheit - Celsius
");

mode = sc.nextLine().charAt(0);

if(mode != '1' && mode != '2')

    System.out.println("Mode inconnu, veuillez réitérer votre
choix.");

}while (mode != '1' && mode != '2');

//saisie de la température à convertir

System.out.println("Température à convertir :");

aConvertir = sc.nextDouble();

//Pensez à vider la ligne lue

sc.nextLine();

//Selon le mode, on calcule différemment et on affiche le
résultat

if(mode == '1'){

    convertit = ((9.0/5.0) * aConvertir) + 32.0;

    System.out.print(aConvertir + " °C correspond à : ");

    System.out.println(arrondi(convertit, 2) + " °F.");

}
```

```
else{

    convertit = ((aConvertir - 32) * 5) / 9;

    System.out.print(aConvertir + " °F correspond à : ");

    System.out.println(arrondi(convertit, 2) + " °C.");

}

//On invite l'utilisateur à recommencer ou à quitter

do{

    System.out.println("Souhaitez-vous convertir une autre

    température ? (O/N)");

    reponse = sc.nextLine().charAt(0);

    }while(reponse != 'O' && reponse != 'N');

    }while(reponse == 'O');

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

//Fin de programme

}

public static double arrondi(double A, int B) {

    return (double) ( (int) (A * Math.pow(10, B) + .5)) /

    Math.pow(10, B);

}

}
```

✓✓ Les dates en Java

Java n'a pas de classe **Date** intégrée, mais nous pouvons importer le package **java.time** pour fonctionner avec l'API de date et d'heure. Ce package comprend de nombreuses classes de date et d'heure. Par exemple:

Classes	Description
LocalDate	Représente une date (année, mois, jour (yyyy-mm-dd))
LocalTime	Représente un temps (heure, minute, seconde et nanosecondes (HH-MM-SS-NS))
LocalDateTime	Représente à la fois une date et une heure (yyyy-mm-dd-hh-mm-ss-ns)
DateTimeFormatter	Formateur pour l'affichage et l'analyse des objets de date d'heure

📁 **Exemple 1:** Afficher la date actuelle

```
import java.time.LocalDate; // import the LocalDate class

public class Main {

    public static void main(String[] args) {

        LocalDate myObj = LocalDate.now(); // Create a date object

        System.out.println(myObj); // Display the current date

    }

}
```

Exemple 2: Afficher la l'heure actuelle

```
import java.time.LocalDateTime; // import the LocalDateTime class

public class Main {

    public static void main(String[] args) {

        LocalDateTime myObj = LocalDateTime.now();

        System.out.println(myObj);

    }

}
```

Exemple 3: Afficher la date et l'heure actuelle.

```
import java.time.LocalDateTime; // import the LocalDateTime class

public class Main {

    public static void main(String[] args) {

        LocalDateTime myObj = LocalDateTime.now();

        System.out.println(myObj);

    }

}
```

📁 **Exemple 4:** Formatage de la date et de l'heure

Le "T" dans l'exemple ci-dessus est utilisé pour séparer la date de l'heure. Vous pouvez utiliser la classe **DateTimeFormatter** avec la méthode **ofPattern()** dans le même package pour formater ou analyser des objets date-heure. L'exemple suivant supprimera à la fois le « T » et les nanosecondes de la date-heure :

```
import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

public class Hello {

    public static void main(String[] args) {

        // Avant formatage

        LocalDateTime date = LocalDateTime.now();

        System.out.println("Avant formatage : " + date);

        // Formatage...

        System.out.println("Formatage...");

        DateTimeFormatter formatter =

        DateTimeFormatter.ofPattern("dd/mm/yyyy HH:mm:ss");

        String dateFormater = formatter.format(date);

        // Apres formatage

        System.out.println("Apres formatage : " + dateFormater);}}
```

La méthode **ofPattern()** accepte toutes sortes de valeurs, si vous souhaitez afficher la date et l'heure dans un format différent. Par exemple:

Valeur	Exemple
<i>yyyy-MM-dd</i>	"1988-09-29"
<i>dd/MM/yyyy</i>	"29/09/1988"
<i>dd-MMM-yyyy</i>	"29-Sep-1988"
<i>E, MMM dd yyyy</i>	"Thu, Sep 29 1988"

✓✓ Les tableaux dynamique en Java - ArrayList

La classe **ArrayList** est un tableau redimensionnable , qui peut être trouvé dans le package **java.util**.

La différence entre un tableau statique et un tableau **ArrayList** en Java est que dans un tableau statique la taille du tableau ne peut pas être modifiée (si vous souhaitez ajouter ou supprimer des éléments dans un tableau statique, vous devez en créer un nouveau). Tandis que des éléments peuvent être ajoutés et supprimés à tout moment dans un **ArrayList** . La syntaxe est également légèrement différente :

```
ArrayList<String> cars = new ArrayList<String>();
```

Ajouter un élément ~ add()

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");

        cars.add("BMW");

        cars.add("Ford");

        cars.add("Mazda");

        System.out.println(cars); // [Volvo, BMW, Ford, Mazda]

    }

}
```

Accéder à un élément ~ get()

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");

        cars.add("BMW");

        System.out.println(cars.get(0)); // Volvo

    }

}
```

Modifier un élément ~ set()

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");

        cars.add("BMW");

        cars.set(0, "Opel");

        System.out.println(cars.get(0)); // Opel

    }

}
```

Supprimer un élément ~ remove()

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo"); cars.add("BMW");

        cars.set(0, "Opel");

        cars.remove(0);

        System.out.println(cars.get(0)); // BMW

    }

}
```

Vider le tableau ~ clear()

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo"); cars.add("BMW");

        cars.clear();

        System.out.println(cars); // []

    }

}
```

Taille du tableau ~ size()

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");

        cars.add("BMW");

        System.out.println(cars.size()); // 2

    }

}
```

Parcourir un ArrayList

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo"); cars.add("BMW");

        cars.add("Ford");

        cars.add("Mazda");

        for (int i = 0; i < cars.size(); i++) {

            System.out.println(cars.get(i));

        }

    }

}
```

Trier un ArrayList

```
import java.util.ArrayList;

import java.util.Collections; // Import the Collections class

public class Main {

    public static void main(String[] args) {

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo");

        cars.add("BMW");

        cars.add("Ford");

        cars.add("Mazda");

        Collections.sort(cars); // Sort cars

        for (String i : cars) {

            System.out.println(i);

        }

    }

}
```

✓✓ Les tableaux dynamique en Java - LinkedList

Dans le chapitre précédent, vous avez découvert la classe **ArrayList**. La classe **LinkedList** est presque identique à la **ArrayList**:

📁 Exemple :

```
// Import the LinkedList class
import java.util.LinkedList;

public class Main {

    public static void main(String[] args) {

        LinkedList<String> cars = new LinkedList<String>();

        cars.add("Volvo");

        cars.add("BMW");

        cars.add("Ford");

        cars.add("Mazda");

        System.out.println(cars); // [Volvo, BMW, Ford, Mazda]

    }

}
```

Différence entre ArrayList vs LinkedList

La classe **LinkedList** est une collection qui peut contenir de nombreux objets du même type, tout comme la classe **ArrayList** .

La classe **LinkedList** possède toutes les mêmes méthodes que la classe **ArrayList** car elles implémentent toutes deux l'interface **List**. Cela signifie que vous pouvez ajouter des éléments, modifier des éléments, supprimer des éléments et effacer la liste de la même manière.

Cependant, même si la classe **ArrayList** et la classe **LinkedList** peuvent être utilisées de la même manière, elles sont construites de manière très différente.

Comment fonctionne la classe ArrayList

La classe **ArrayList** contient un tableau régulier. Lorsqu'un élément est ajouté, il est placé dans le tableau. Si le tableau n'est pas assez grand, un nouveau tableau plus grand est créé pour remplacer l'ancien et l'ancien est supprimé.

Comment fonctionne la classe LinkedList

La classe **LinkedList** stocke ses éléments dans des « conteneurs ». La liste a un lien vers le premier conteneur et chaque conteneur a un lien vers le conteneur suivant de la liste. Pour ajouter un élément à la liste, l'élément est placé dans un nouveau conteneur et ce conteneur est lié à l'un des autres conteneurs de la liste.

Utilisez un **ArrayList** pour stocker et accéder aux données, ainsi que **LinkedList** pour manipuler les données.

Les Méthodes LinkedList

Dans de nombreux cas, la classe **ArrayList** est plus efficace car il est courant d'avoir besoin d'accéder à des éléments aléatoires dans la liste, mais la classe **LinkedList** propose plusieurs méthodes pour effectuer certaines opérations plus efficacement :

Méthode	Description
addFirst()	Ajoute un élément au début de la liste.
addLast()	Ajoute un élément à la fin de la liste
removeFirst()	Supprimer un élément du début de la liste.
removeLast()	Supprimer un élément de la fin de la liste
getFirst()	Récupère le premier élément
getLast()	Récupère le dernier élément

✓✓ Les HashMap

Dans le chapitre **ArrayList**, vous avez appris que les tableaux stockent les éléments sous forme de collection ordonnée et que vous devez y accéder avec un numéro d'index (type **int**). Cependant **HashMap**, stockez les éléments dans des paires " **clé / valeur** ", et vous pouvez y accéder par un index d'un autre type (par exemple un **String**).

Un objet est utilisé comme clé (index) pour un autre objet (valeur). Il peut stocker différents types : **String** clés et **Integer** valeurs, ou le même type, comme : **String** clés et **String** valeurs.

Les HashMap sont l'équivalent des tableau associatif en PHP ou les objet en JavaScript

📁 Exemple :

```
import java.util.HashMap; // import the HashMap class

public class Main {

    public static void main(String[] args) {

        HashMap<String, String> capitalCities = new HashMap<String,
        String>();

    }

}
```

Ajouter un élément ~ put()

```
// Import the HashMap class
import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        HashMap<String, String> tab = new HashMap<String, String>();

        // Add keys and values (Country, City)

        tab.put("prenom", "Modou");

        tab.put("nom", "Sarr");

        tab.put("pays", "Sénégal");

        tab.put("age", "12");

        System.out.println(tab);

        // {prenom=Modou, nom=Sarr, age=12, pays=Sénégal}

    }

}
```

Accéder à la valeur ~ get()

```
tab.get("prenom")
```

Supprimer un élément ~ remove()

```
tab.remove("prenom");
```

Vider le tableau

```
tab.clear();
```

Taille du tableau

```
tab.size();
```

Parcourez une HashMap

Parcourez les éléments d'un **HashMap** avec une boucle **for-each** .

Remarque : utilisez la méthode **keySet()** si vous souhaitez uniquement les clés et utilisez la méthode **values()** si vous souhaitez uniquement les valeurs :

```
// Print keys
for (String i : tab.keySet()) {
    System.out.println(i);
}

// Print values
for (String i : tab.values()) {
    System.out.println(i);
}

// Print keys and values
for (String i : tab.keySet()) {
    System.out.println("key: " + i + " value: " + tab.get(i));
}
```

📁 **Note :** Les clés et les valeurs d'un HashMap sont en réalité des objets. Dans les exemples ci-dessus, nous avons utilisé des objets de type « String ». N'oubliez pas qu'une chaîne en Java est un objet (pas un type primitif). Pour utiliser d'autres types, tels que int, vous devez spécifier une classe wrapper équivalente : **Integer**. Pour les autres types primitifs, utilisez : **Boolean** pour boolean, **Character** pour char, **Double** pour double, etc :

📁 **Exemple :**

```
// Import the HashMap class
import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

        // Create a HashMap object called people

        HashMap<String, Integer> people = new HashMap<String, Integer>();

        // Add keys and values (Name, Age)

        people.put("John", 32);

        people.put("Steve", 30);

        people.put("Angie", 33);

        for (String i : people.keySet()) {

            System.out.println("key: " + i + " value: " + people.get(i));

        }

    }

}
```

✓✓ Les HashSet

Un HashSet est une collection d'éléments où chaque élément est unique et se trouve dans le package **java.util** :

📁 Exemple :

```
import java.util.HashSet; // Import the HashSet class

public class Main {

    public static void main(String[] args) {

        HashSet<String> cars = new HashSet<String>();

    }

}
```

📖 Ajouter un élément ~ add()

```
// Import the HashSet class

import java.util.HashSet;

public class Main {

    public static void main(String[] args) {

        HashSet<String> cars = new HashSet<String>();

        cars.add("Volvo"); cars.add("BMW");

        cars.add("Ford"); cars.add("BMW");

        cars.add("Mazda");

        System.out.println(cars);

    }

}
```

 Vérifier si un élément existe ~ contains()

```
cars.contains("Mazda");
```

 Supprimer un élément ~ remove()

```
cars.remove("Volvo");
```

 Vider le tableau ~ clear()

```
cars.clear();
```

 Taille du tableau ~ size()

```
cars.size();
```

✓✓ Les itérateurs

Un **Iterator** est un objet qui peut être utilisé pour parcourir des collections, comme **ArrayList** et **HashSet**... On l'appelle « itérateur » car « itérer » est le terme technique pour désigner une boucle.

Pour utiliser un itérateur, vous devez l'importer depuis le package **java.util**.

Obtenir un itérateur

```
// Import the ArrayList class and the Iterator class

import java.util.ArrayList;

import java.util.Iterator;

public class Main {

    public static void main(String[] args) {

        // Make a collection

        ArrayList<String> cars = new ArrayList<String>();

        cars.add("Volvo"); cars.add("BMW");

        cars.add("Ford"); cars.add("Mazda");

        // Get the iterator

        Iterator<String> it = cars.iterator();

        // Print the first item

        System.out.println(it.next());

    }

}
```

Pour parcourir une collection, utilisez les méthodes **hasNext()** et **next()** de **Iterator** :

```
while (it.hasNext()) { System.out.println(it.next()); }
```

✓✓ Les classes wrapper en java

Les classes wrapper fournissent un moyen d'utiliser des types de données primitifs (**int**, **boolean**, etc.) comme objets.

Le tableau ci-dessous montre le type primitif et la classe wrapper équivalente :

Type de données primitives	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Parfois, vous devez utiliser des classes wrapper, par exemple lorsque vous travaillez avec des objets Collection, tels que **ArrayList**, où les types primitifs ne peuvent pas être utilisés (la liste ne peut stocker que des objets):

```
ArrayList<int> tab1 = new ArrayList<int>(); // Invalid  
  
ArrayList<Integer> tab2 = new ArrayList<Integer>(); // Valid
```

Création d'objets wrapper

Pour créer un objet wrapper, utilisez la classe wrapper au lieu du type primitif. Pour obtenir la valeur, vous pouvez simplement imprimer l'objet :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Integer myInt = 5;  
  
        Double myDouble = 5.99;  
  
        Character myChar = 'A';  
  
        System.out.println(myInt);  
  
        System.out.println(myDouble);  
  
        System.out.println(myChar);  
  
    }  
}
```

Puisque vous travaillez maintenant avec des objets, vous pouvez utiliser certaines méthodes pour obtenir des informations sur l'objet spécifique.

Par exemple, les méthodes suivantes sont utilisées pour obtenir la valeur associée à l'objet wrapper correspondant : **intValue()**, **byteValue()**, **shortValue()**, **longValue()**, **floatValue()**, **doubleValue()**, **charValue()**, **booleanValue()**.

Cet exemple produira le même résultat que l'exemple ci-dessus :

Exemple :

```
public class Main {
```

```
public static void main(String[] args) {  
  
    Integer myInt = 5;  
  
    Double myDouble = 5.99;  
  
    Character myChar = 'A';  
  
    System.out.println(myInt.intValue());  
  
    System.out.println(myDouble.doubleValue());  
  
    System.out.println(myChar.charValue());  
  
}  
}
```

Une autre méthode utile est la méthode **toString()** utilisée pour convertir les objets wrapper en chaînes.

Dans l'exemple suivant, nous convertissons un **Integer** en **String** et utilisons la méthode **length()** de la classe **String** pour afficher la longueur de la « chaîne » :

Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Integer myInt = 100;  
  
        String myString = myInt.toString();  
  
        System.out.println(myString.length());  
  
    }  
  
}
```

✓✓ Les exceptions en java

Lors de l'exécution du code Java, différentes erreurs peuvent survenir : des erreurs de codage faites par le programmeur, des erreurs dues à une mauvaise saisie ou d'autres choses imprévisibles.

Lorsqu'une erreur se produit, Java s'arrête normalement et génère un message d'erreur. Le terme technique pour cela est le suivant : Java lancera une **exception** (lancera une erreur).

L'instruction **try** vous permet de définir un bloc de code à tester pour les erreurs pendant son exécution.

L'instruction **catch** permet de définir un bloc de code à exécuter, si une erreur survient dans le bloc **try**.

Les mots-clés **try** et **catch** viennent par paires :

```
try {  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```

📁 **Exemple :** Cela générera une erreur, car `myNumbers[10]` n'existe pas.

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int[] myNumbers = {1, 2, 3};  
  
        System.out.println(myNumbers[10]); // error!  
  
    }  
  
}
```

Si une erreur se produit, nous pouvons utiliser **try...catch** pour détecter l'erreur et exécuter du code pour la gérer afin de rendre l'erreur beaucoup plus claire et sécurisée :

 Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        try {  
  
            int[] myNumbers = {1, 2, 3};  
  
            System.out.println(myNumbers[10]);  
  
        } catch (Exception e) {  
  
            System.out.println("Quelque chose s'est mal passé.");  
  
        }  
  
    }  
  
}
```

L'instruction **finally** vous permet d'exécuter du code, après **try...catch**, quel que soit le résultat :

Exemple :

```
public class Main {  
  
    public static void main(String[] args) {  
  
        try {  
  
            int[] myNumbers = {1, 2, 3};  
  
            System.out.println(myNumbers[10]);  
  
        } catch (Exception e) {  
  
            System.out.println("Quelque chose s'est mal passé.");  
  
        } finally {  
  
            System.out.println("Le «Try Catch» est terminé.");  
  
        }  
  
    }  
  
}
```

Le mot-clé throw

L'instruction **throw** vous permet de créer une erreur personnalisée.

L'instruction **throw** est utilisée avec un **type d'exception**. Il existe de nombreux types d'exceptions disponibles en Java : **ArithmeticException**, **FileNotFoundException**, **ArrayIndexOutOfBoundsException**, **SecurityException**, etc :

 **Exemple :** Lancez une exception si l'âge est inférieur à 18 ans (imprimez "Accès refusé"). Si l'âge est de 18 ans ou plus, inscrivez « Accès accordé » :

```
public class Main {  
  
    static void checkAge(int age) {  
  
        if (age < 18) {  
  
            throw new ArithmeticException("Accès refusé - Vous devez avoir  
            au moins 18 ans.");  
  
        }  
  
        else {  
  
            System.out.println("Accès accordé - vous êtes assez vieux!");  
  
        }  
  
    }  
  
  
    public static void main(String[] args) {  
  
        checkAge(15); // Fixer l'âge à 15 ans (ce qui est inférieur à 18  
        ...)  
  
    }  
  
}
```

📁 **Note :** Il faut noter que toutes les instructions qui suivent l'instruction **throw** ne sont pas exécutées car ce dernier lève une exception et arrête le programme.

📁 **Exemple :**

```
public class Hello {

    int age = 12;

    public void setAge1(int age)
    {
        if (age < 18) {
            throw new IllegalArgumentException("L'âge doit être > à 18");
        }

        this.age = age;

        System.out.println("Age modifié !");
    }

    public void setAge2(int age)
    {
        try {
            setAge1(age);
        } catch (IllegalArgumentException e) {
            e.getMessage();

            e.printStackTrace();
        }
    }
}
```

```
}  
  
finally {  
    System.out.println("Fin du programme !");  
}  
}  
  
public static void main(String[] args) {  
    Hello h = new Hello();  
    h.setAge2(12);  
}  
}
```

✓✓ Expressions Java Lambda

Les expressions Lambda ont été ajoutées dans Java 8.

Une expression lambda est un court bloc de code qui prend des paramètres et renvoie une valeur. Les expressions Lambda sont similaires aux méthodes, mais elles n'ont pas besoin de nom et elles peuvent être implémentées directement dans le corps d'une méthode.

📁 L'expression lambda la plus simple contient un seul paramètre et une expression :

```
parameter -> expression
```

📁 Pour utiliser plusieurs paramètres, placez-les entre parenthèses :

```
(parameter1, parameter2) -> expression
```

📁 Les expressions sont limitées. Ils doivent renvoyer immédiatement une valeur et ne peuvent pas contenir de variables, d'affectations ou d'instructions telles que **if** ou **for**. Afin d'effectuer des opérations plus complexes, un bloc de code peut être utilisé avec des accolades. Si l'expression lambda doit renvoyer une valeur, alors le bloc de code doit avoir une instruction **return**.

```
(parameter1, parameter2) -> { code block }
```

Les expressions Lambda sont généralement transmises en tant que paramètres à une fonction :

Exemple :

```
import java.util.ArrayList;

public class Main {

    public static void main(String[] args) {

        ArrayList<Integer> numbers = new ArrayList<Integer>();

        numbers.add(5);

        numbers.add(9);

        numbers.add(8);

        numbers.add(1);

        numbers.forEach( (n) -> { System.out.println(n); } );

    }

}
```

Les expressions Lambda peuvent être stockées dans des variables si le type de la variable est une interface qui n'a qu'une seule méthode. L'expression lambda doit avoir le même nombre de paramètres et le même type de retour que cette méthode. Java intègre un grand nombre de ces types d'interfaces, telles que l'interface **Consumer** (trouvée dans le package **java.util**) utilisée par les listes.

Exemple :

```
import java.util.ArrayList;

import java.util.function.Consumer;

public class Main {

    public static void main(String[] args) {

        ArrayList<Integer> numbers = new ArrayList<Integer>();

        numbers.add(5);

        numbers.add(9);

        numbers.add(8);

        numbers.add(1);

        Consumer<Integer> method = (n) -> { System.out.println(n); };

        numbers.forEach( method );

    }

}
```



La POO en Java

✓✓ La programmation orienté objet

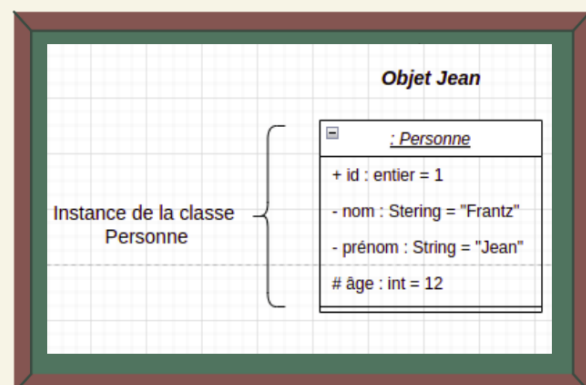
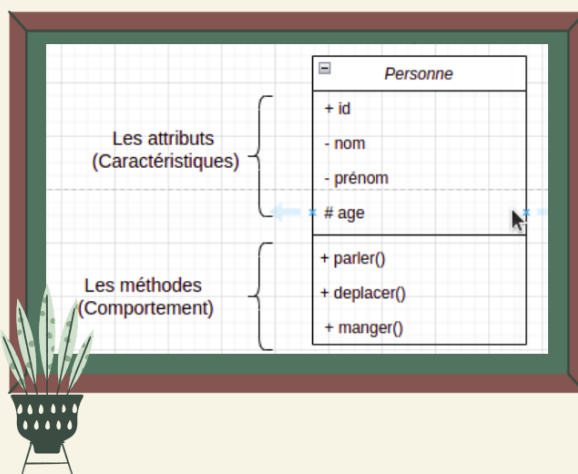
La programmation procédurale consiste à écrire des procédures ou des méthodes qui effectuent des opérations sur les données, tandis que la programmation orientée objet consiste à créer des objets contenant à la fois des données et des méthodes.

La programmation orientée objet présente plusieurs avantages par rapport à la programmation procédurale :

- ❖ La POO est plus rapide et plus facile à exécuter
- ❖ La POO fournit une structure claire pour les programmes
- ❖ La POO aide à garder le code Java SEC « Don't Repeat Yourself » (DRY) et rend le code plus facile à maintenir, modifier et déboguer.
- ❖ La POO permet de créer des applications entièrement réutilisables avec moins de code et un temps de développement plus court

La connaissance des notions de Classe, objet, entité, instance, attribut et méthode est obligatoire quand on travaille sur la POO.

Entité, Objet, Instance et Classe



Les classes

Tout en Java est associé à des classes et des objets, ainsi qu'à ses attributs et méthodes. Par exemple : dans la vraie vie, une voiture est un objet. La voiture a **des attributs**, tels que le poids et la couleur, et **des méthodes**, telles que la conduite et le freinage.

Une classe est comme un constructeur d'objets ou un « plan » pour créer des objets. Pour créer une classe, on utilise le mot-clé **class**:

 **Exemple :** Créer une classe personne. Une personne est caractérisée par son nom, prénom, adresse et âge.

```
public class Personne {  
  
    String nom;  
  
    String prenom;  
  
    String adresse;  
  
    int age;  
  
}
```

Les Objets

En Java, un objet est créé à partir d'une classe. Nous avons déjà créé la classe nommée **Personne**, nous pouvons donc maintenant l'utiliser pour créer des objets.

Pour créer un objet de **Personne**, précisez le nom de la classe, suivi du nom de l'objet, et utilisez le mot-clé **new**:

Exemple :

```
public class Personne {  
  
    // Attributs (Variable dans une classe)  
  
    String nom = "Sarr";  
  
    String prenom = "Modou";  
  
    String adresse = "Pikine";  
  
    int age = 12;  
  
  
    public static void main(String[] args) {  
  
        Personne modou = new Personne(); //objet modou  
  
        Personne issa = new Personne(); // objet issa  
  
  
        // Changement des infos de issa  
  
        issa.prenom = "Issa";  
  
        issa.nom = "Pouye";  
  
  
        System.out.println(modou.nom); //Sarr  
  
        System.out.println(issa.nom); //Pouye  
  
    }  
}
```

Les méthodes

Les méthodes sont déclarées au sein d'une classe. Une méthode, c'est une action que peut effectuer un objet.

Exemple :

```
public class Personne {

    // Attributs

    String nom = " Sarr";

    String prenom = "Modou";

    String adresse = "Pikine";

    int age = 12;

    // Méthode manger

    public void manger() {

        System.out.println("Je suis entrain de manger");

    }

    // Méthode getAge

    public String getAge() {

        return this.prenom + this.nom + " a " + this.age + " ans";

    }

    public static void main(String[] args) {

        Personne modou = new Personne(); //objet modou

        modou.manger();

        System.out.println( modou.getAge());

    }

}
```

Constructeur

Dans les exemples précédents, nous avons fixé les attributs, ce qui fait que à chaque fois qu'on crée un nouvel objet, il a les mêmes valeurs (Problématique). Comment faire de telle sorte que si on crée un objet Issa aura le nom Issa et non pas Modou par exemple, pour bien faire cela, la création d'un constructeur sera nécessaire.

Un constructeur en Java est une **méthode spéciale** utilisée pour initialiser des objets. Le constructeur est appelé lorsqu'un objet d'une classe est créé. Il peut être utilisé pour définir les valeurs initiales des attributs d'objet. Le nom du constructeur doit **correspondre au nom de la classe** et ne peut pas avoir de **type de retour** (comme **void**).

Exemple :

```
public class Personne {  
  
    // Attributs  
  
    String nom = " Sarr";  
  
    String prenom = "Modou";  
  
    String adresse = "Pikine";  
  
    int age = 12;  
  
    // Constructeur  
  
    public Personne(String prenom, String nom, String adresse, int  
        age)  
  
    {  
  
        this.prenom = prenom;  

```

```
        this.nom = nom;

        this.adresse = adresse;

        this.age = age;
    }

    public static void main(String[] args) {

        //objet modou

        Personne modou = new Personne("Modou", "Sarr", "Pikine", 12);

        //objet issa

        Personne issa = new Personne("Issa", "Pouye", "Oukam", 13);

        System.out.println(modou.prenom); //Modou

        System.out.println(issa.prenom);  //Issa

    }
}
```

Les modificateurs d'accès

Vous connaissez désormais bien le mot-clé **public** qui apparaît dans presque tous nos exemples. Le mot-clé **public** est un **modificateur d'accès**, ce qui signifie qu'il est utilisé pour définir le niveau d'accès aux classes, attributs, méthodes et constructeurs.

Nous divisons les modificateurs en deux groupes :

- ❖ **Modificateurs d'accès** - contrôle le niveau d'accès
- ❖ **Modificateurs de non-access** - ne contrôlent pas le niveau d'accès, mais fournissent d'autres fonctionnalités

Les modificateurs d'accès

❖ Pour les classes :

Modificateur	Description
public	La classe est accessible par toute autre classe
default	La classe n'est accessible que par des classes du même package. Ceci est utilisé lorsque vous ne spécifiez pas de modificateur.

❖ Pour les attributs, méthode et constructeurs

Modificateur	Description
public	Le code est accessible pour toutes les classes
private	Le code n'est accessible que dans la classe déclarée
protected	Le code est accessible dans le même package et sous-classes.
default	Le code n'est accessible que dans le même package. Ceci est utilisé lorsque vous ne spécifiez pas de modificateur.

Les modificateurs de non accès

❖ Pour les classes

Modificateur	Description
final	La classe ne peut pas être héritée par d'autres classes
abstract	La classe ne peut pas être utilisée pour créer des objets (pour accéder à une classe abstraite, elle doit être héritée d'une autre classe.

❖ Pour les attributs et les méthodes

Modificateur	Description
final	Les attributs et les méthodes ne peuvent pas être remplacés / modifiés
abstract	Ne peut être utilisé que dans une classe abstraite et ne peut être utilisé que sur les méthodes. La méthode n'a pas de corps, par exemple Résumé Void run (); Le corps est fourni par la sous-classe (hérite de).
static	Les attributs et les méthodes appartiennent à la classe, plutôt qu'un objet. On pourra utiliser alors l'attribut ou la méthode sans créer un objet mais plutôt le nom de classe.
transient	Les attributs et les méthodes sont ignorés lors de la sérialisation de l'objet les contenant
volatile	La valeur d'un attribut n'est pas en cache-thread-localement et est toujours lue à partir de la "mémoire principale"

Exemple (Final) :

Permet de mettre en attribut en constante.

```
public class Main {  
  
    final int x = 10;  
  
    final double PI = 3.14;  
  
    public static void main(String[] args) {  
  
        Main myObj = new Main();  
  
        myObj.x = 50; // error: cannot assign a value to a final variable  
  
        myObj.PI = 25; //error: cannot assign a value to a final variable  
  
        System.out.println(myObj.x);  
  
    }  
}
```

📁 Exemple (static) :

Une méthode **static** signifie qu'elle est accessible sans créer d'objet de la classe, contrairement à **public**:

```
public class Main {  
  
    // Static method  
  
    static void myStaticMethod() {  
  
        System.out.println("Les méthodes statiques peuvent être appelées  
        sans créer d'objets");  
  
    }  
  
    // Public method  
  
    public void myPublicMethod() {  
  
        System.out.println("Les méthodes publiques doivent être appelées  
        en créant des objets");  
  
    }  
  
    // Main method  
  
    public static void main(String[] args) {  
  
        myStaticMethod(); // Appelez la méthode statique  
  
        // myPublicMethod(); Cela publierait une erreur  
  
        Main myObj = new Main(); // Créer un objet de Main  
  
        myObj.myPublicMethod(); // Appelez la méthode public  
  
    }  
}
```

✓✓ Les classes internes en Java

En Java, il est également possible d'imbriquer des classes (une classe dans une classe). Le but des classes imbriquées est de regrouper les classes qui vont ensemble, ce qui rend votre code plus lisible et maintenable.

Pour accéder à la classe interne, créez un objet de la classe externe, puis créez un objet de la classe interne :

```
// Dans un fichier ExterneClass.java

class ExterneClass {

    int x = 10;

    class InterneClass {

        int y = 5;

    }

}

// Dans un fichier Main.java

public class Main { //classe interne aussi

    public static void main(String[] args) {

        ExterneClass externeClasse = new ExterneClass();

        ExterneClass.InterneClass interneClasse = externeClasse.new

        InterneClass();

        System.out.println(interneClasse.y + externeClasse.x); //5 +

        10 = 15

    }

}
```

Classe interne privée

Contrairement à une classe « normale », une classe interne peut être **private** ou **protected**. Si vous ne souhaitez pas que les objets extérieurs accèdent à la classe interne, déclarez la classe comme **private** :

Exemple :

```
// Dans un fichier ExterneClass.java

class ExterneClass {

    int x = 10;

    private class InterneClass {

        int y = 5;

    }

}

// Dans un fichier Main.java

public class Main { //classe interne aussi

    public static void main(String[] args) {

        ExterneClass externeClasse = new ExterneClass();

        ExterneClass.InterneClass interneClasse = externeClasse.new

        InterneClass(); //erreur

        System.out.println(interneClasse.y + externeClasse.x);

        //erreur

    }

}
```

Classe interne statique

Une classe interne peut également être **static**, ce qui signifie que vous pouvez y accéder sans créer d'objet de la classe externe :

Exemple :

```
// Dans un fichier ExterneClass.java

class ExterneClass {

    int x = 10;

    static class InterneClass {

        int y = 5;

    }

}

// Dans un fichier Main.java

public class Main {

    public static void main(String[] args) {

        ExterneClass.InterneClass myInner = new

        ExterneClass.InterneClass(); // On passe pas par la classe ext

        System.out.println(myInner.y);

    }

}
```

 **Remarque :** tout comme les attributs et les méthodes static, une classe interne static n'a pas accès aux membres de la classe externe.

Accéder à la classe externe depuis la classe interne

L'un des avantages des classes internes est qu'elles peuvent accéder aux attributs et méthodes de la classe externe :

Exemple :

```
// Dans OuterClass.java

class OuterClass {

    int x = 10;

    class InnerClass {

        public int myInnerMethod() {

            return x;

        }

    }

}

// Dans Main.java

public class Main {

    public static void main(String[] args) {

        OuterClass myOuter = new OuterClass();

        OuterClass.InnerClass myInner = myOuter.new InnerClass();

        System.out.println(myInner.myInnerMethod()); // Outputs 10

    }

}
```

Exercice Les Classes Internes & classes locales

Exercice:

Créer une classe A caractérisée par nom et poids.

Dans A, créer une méthode test() qui affiche "Bonjour tout le monde".

Créer une classe interne B dans A caractérisé par une taille. B hérite de C (classe à créer) et implémente l'interface D (Interface à créer). Cette dernière déclare une méthode aire().

Créer le constructeur de C qui affiche "Je suis le constructeur de C".

Créer un bloc d'initialisation d'instance juste avant le constructeur de B (constructeur sans paramètre).

Surcharger le constructeur de B.

Créer une méthode testerB() dans B qui appelle la méthode test() de sa classe externe.

Redéfinissez la méthode aire() de D puis créer à l'intérieur de aire() une classe locale nommée ClassInternTest. Créer son constructeur puis la méthode testClasseInterne() qui affiche "Test classe Interne réussi !".

Appeler la méthode testClasseInterne() dans la classe ClassInternTest.

Appeler la méthode aire() de B dans le constructeur de A.

✓✓ Les 4 piliers de la poo



Abstraction

L'abstraction des données est le processus consistant à masquer certains détails et à afficher uniquement les informations essentielles à l'utilisateur.

L'abstraction peut être réalisée soit avec **des classes abstraites**, soit avec **des interfaces**.

Le mot-clé **abstract** est un modificateur de non-accès, utilisé pour les classes et les méthodes :

- ❖ Une **Classe abstraite** : est une classe restreinte qui ne peut pas être utilisée pour créer des objets (pour y accéder, elle doit être héritée d'une autre classe). Une classe abstraite peut avoir à la fois des méthodes abstraites et non abstraites.

- ❖ Une **Méthode abstraite** : ne peut être utilisée que dans une classe abstraite et n'a pas de corps. Le corps est fourni par la sous-classe (héritée de).

📁 Exemple 1 :

```
// abstract class

abstract class Personne {

    public String fname = "John";

    public int age = 24;

    public abstract void study(); // abstract method (déclaration)
}

// Student hérite Personne

class Student extends Personne {

    public int graduationYear = 2018;

    public void study() { //Implémentation

        System.out.println("Étudier toute la journée");

    }

}

class Second {

    public static void main(String[] args) {

        // Objet modou

        Student modou = new Student();
    }
}
```

```
        System.out.println("Name: " + modou.fname);

        System.out.println("Age: " + modou.age);

        System.out.println("Graduation Year: " + modou.graduationYear);

        modou.study();
    }
}
```

Exemple 2 :

```
// Abstract class

abstract class Animal {

    // Abstract method (does not have a body)

    public abstract void animalSound();

    // Regular method

    public void sleep() {

        System.out.println("kroth kroth");

    }

}
```

```
// Mouton hérite Animal

class Mouton extends Animal {

    public void animalSound() {

        System.out.println("Mbéééééééééé");

    }

}

class Main {

    public static void main(String[] args) {

        Mouton mouton = new Mouton();

        mouton.animalSound();

        mouton.sleep();

    }

}
```

Encapsulation

Le sens de **l'encapsulation** est de s'assurer que les données "sensibles" sont cachées aux utilisateurs. Pour y parvenir, vous devez :

- ❖ Déclarer les variables/attributs de classe comme **private**
- ❖ Fournir des méthodes publiques **get** et **set** pour accéder et mettre à jour la valeur d'une variable **private**

Vous avez appris dans le chapitre précédent que les variables **private** ne sont accessibles qu'au sein de la même classe (une classe extérieure n'y a pas accès). Cependant, il est possible d'y accéder si nous fournissons des méthodes publiques **get** et **set** .

La méthode **get** renvoie la valeur de la variable et la méthode **set** définit la valeur.

La syntaxe des deux est qu'ils commencent par soit **get** ou **set**, suivi du nom de la variable, avec la première lettre en majuscule :

Exemple :

```
public class Personne {  
  
    private String name; // private = restricted access  
  
    // Getter  
  
    public String getName() { return name; }  
  
    // Setter  
  
    public void setName(String newName) {this.name = newName;}  
  
}
```

```

public class Main {

    public static void main(String[] args) {

        Personne modou = new Personne();

        modou.name = "John"; // error

        System.out.println(modou.name); // error

        System.out.println(modou.getName()); // ok

    }

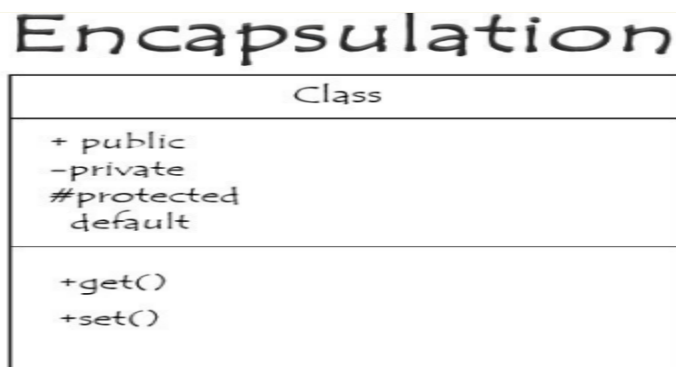
}

```

Pourquoi l'encapsulation ?

- ❖ Meilleur contrôle des attributs et des méthodes de classe
- ❖ Les attributs de classe peuvent être rendus en lecture seule (si vous utilisez uniquement la méthode **get**) ou en écriture seule (si vous utilisez uniquement la méthode **set**)
- ❖ Flexible : le programmeur peut modifier une partie du code sans affecter les autres parties
- ❖ Sécurité accrue des données (On peut faire des contrôles dans les méthodes **get** et **set**).

Exemple :



Héritage

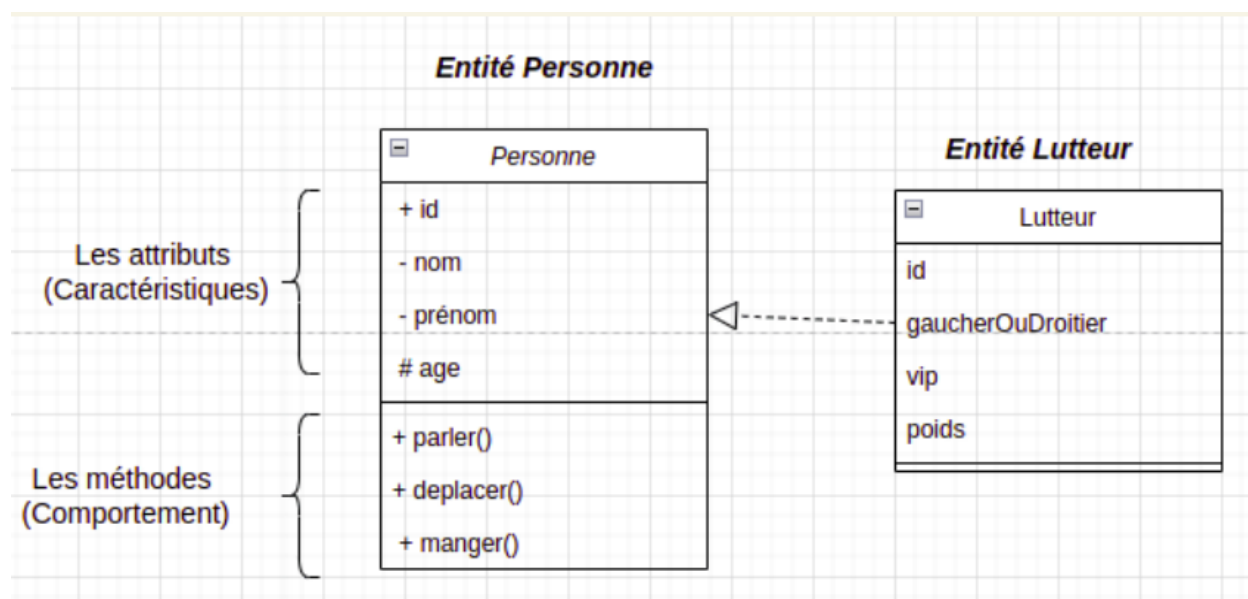
En Java, il est possible d'hériter d'attributs et de méthodes d'une classe à une autre. Nous regroupons la « notion de patrimoine » en deux catégories :

- ❖ **sous-classe** (enfant) - la classe qui hérite d'une autre classe
- ❖ **superclasse** (parent) - la classe héritée de

Pour hériter d'une classe, utilisez le mot-clé **extends**.

Si vous voulez créer une classe qui est légèrement différente d'une classe existante, vous n'avez pas besoin de dupliquer votre code. À la place, vous pouvez étendre la classe existante dans une sous-classe qui hérite de ses attributs et méthodes et écrire dans cette sous-classe les fonctionnalités supplémentaires.

Exemple 1 :



 Exemple 2 :

```
class Vehicle {

    protected String brand = "Ford";           // Vehicle attribute

    public void honk() {                        // Vehicle method

        System.out.println("Tuut, tuut!");

    }

}

class Car extends Vehicle {

    private String modelName = "Mustang";      // Car attribute

    public static void main(String[] args) {

        // Create a myCar object

        Car myCar = new Car();

        // Appelle de méthode du parent

        myCar.honk();

        // Appelle attribut du parent

        System.out.println(myCar.brand + " " + myCar.modelName);

    } }
```

Polymorphisme

Le polymorphisme signifie « plusieurs formes », et cela se produit lorsque nous avons de nombreuses classes liées les unes aux autres par héritage.

Comme nous l'avons précisé dans le chapitre précédent ; **L'héritage** nous permet d'hériter des attributs et des méthodes d'une autre classe. **Le polymorphisme** permet à chaque classe qui hérite la super classe d'utiliser ces méthodes pour effectuer leur propre implémentation. Cela nous permet d'effectuer une seule action de différentes manières.

Par exemple, pensez à une superclasse appelée **Animal** qui possède une méthode appelée **animalSound()**. Les sous-classes d'animaux pourraient être des cochons, des chats, des chiens, des oiseaux - Et ils ont également leur propre implémentation d'un son d'animal (le cochon grogne et le chat miaule, etc.) :

Exemple :

```
class Animal {  
  
    public void animalSound() {  
  
        System.out.println("L'animal fait un son");  
  
    }  
}  
  
class Pig extends Animal {  
  
    public void animalSound() {  
  
        System.out.println("Le cochon dit: wee wee");  
  
    }  
}
```

```
class Dog extends Animal {  
  
    public void animalSound() {  
  
        System.out.println("Le chien dit: bow wow");  
  
    }  
  
}  
  
class Main {  
  
    public static void main(String[] args) {  
  
        Animal myAnimal = new Animal();  
  
        Animal myPig = new Pig();  
  
        Animal myDog = new Dog();  
  
        myAnimal.animalSound();  
  
        myPig.animalSound();  
  
        myDog.animalSound();  
  
    }  
  
}
```

✓✓ Surcharge et Redéfinition de méthode

Surcharge de méthode

La surcharge de méthode est un concept qui permet à une classe d'avoir plusieurs méthodes portant le même nom, si leurs listes d'arguments sont différentes. Il est possible en Java de surcharger le constructeur permettant ainsi à une classe d'avoir plus d'un constructeur ayant des listes d'arguments différentes.

```
public class Personne {  
  
    private String nom;  
  
    public Personne(String nom) {  
  
        this.nom = nom;  
  
    }  
  
    public Personne() {  
  
        System.out.println("Salut");  
  
    }  
  
}
```

Redéfinition de méthode

Redéfinition signifie avoir deux méthodes avec le même nom et les mêmes paramètres, l'une des méthodes est dans la classe mère et l'autre dans la sous classe. La redéfinition permet à une sous classe de fournir une implémentation spécifique d'une méthode déjà fournie à sa classe mère.

✓✓ Exercice 1

TP 1

Ecrire un programme qui reproduit l'effet d'allumage d'une lampe. Une lampe est allumée par un interrupteur spécifique caractérisé par sa position de type boolean et la lampe à allumée.

- Créer un constructeur Lampe qui reçoit l'état de la lampe.
- Créer la méthode presserInterrupteur() qui change l'état de la lampe.
- Créer la méthode getPosition() qui return "On" si la position de l'interrupteur est true et Off au cas échéant.
- Créer la méthode changerPositionInterrupteur() qui change la position de l'interrupteur.

NB: L'état de l'interrupteur doit toujours être égal à l'état de la lampe.



✓✓ Exercice 2

Application Capital Games

Énoncé :

Créer un programme nommé CapitalGames qui demande à l'utilisateur la capitale d'un pays spécifique. Si l'utilisateur donne une bonne réponse, le programme affiche "Bonne réponse" et incrémente son score, sinon il affiche "Mauvaise réponse !" et lui indique la bonne réponse.

A la fin du jeu, le programme doit être capable d'afficher le timeElapsed c'est-à-dire le temps qu'il a fallu à l'utilisateur pour répondre toutes les questions. Le programme doit aussi afficher le score du jeu.

Données :

```
{ "Senegal", "Dakar" },
{ "France", "Paris" },
{ "Nigeria", "Abuja" },
{ "Gabon", "Libreville" },
{ "Allemagne", "Berlin" },
{ "Italie", "Rome" },
{ "Monako", "Monako" },
{ "Liberia", "Monrovia" },
{ "Perou", "Lima" }
```

NB : timeElapsed = Le temps final du jeu - Le temps initial du jeu

✓✓ Les interfaces en java

Une autre façon de réaliser l'abstraction en Java consiste à utiliser les interfaces.

Une interface est une "classe complètement abstraite" utilisée pour regrouper les méthodes associées avec des corps vides. Toutes les méthodes d'une interface sont static, c'est pourquoi le mot clé static est optionnel.

📁 Exemple :

```
// interface  
  
interface Animal {  
  
    public void animalSound();  
  
    public void run();  
  
}
```

Pour accéder aux méthodes d'interface, l'interface doit être "implémentée" (un peu comme héritée) par une autre classe avec le mot-clé **implements** (au lieu de **extends**). Le corps de la méthode d'interface est fourni par la sous classe (classe fille) qui implémente l'interface :

📁 Exemple :

```
// Interface  
  
interface Animal {  
  
    // Déclaration  
  
    public void animalSound();  
  
    public void sleep();  
  
}
```

```
// Mouton "implémente" l'interface Animal

class Mouton implements Animal {

    // Implémentation animalSound()

    public void animalSound() {

        System.out.println("Mbéééééééééé");

    }

    // Implémentation sleep()

    public void sleep() {

        System.out.println("kroth kroth");

    }

}

class Main {

    public static void main(String[] args) {

        Mouton mouton = new Mouton();

        mouton.animalSound();

        mouton.sleep();

    }

}
```

 Remarque

- ❖ Comme **les classes abstraites** , les interfaces **ne peuvent pas** être utilisées pour créer des objets (dans l'exemple ci-dessus, il n'est pas possible de créer un objet "Animal" dans la classe Main)
- ❖ Les méthodes d'interface n'ont pas de corps - le corps est fourni par la sous classe qui l'implémente.
- ❖ Lors de l'implémentation d'une interface, vous devez redéfinir toutes ses méthodes
- ❖ Les méthodes d'interface sont par défaut **abstract** et **public**
- ❖ Les attributs de l'interface sont par défaut **public**, **static** et **final**
- ❖ Une interface ne peut pas contenir de constructeur (car elle ne peut pas être utilisée pour créer des objets)

 Pourquoi et quand utiliser les interfaces ?

- 1) Pour assurer la sécurité - masquez certains détails et affichez uniquement les détails importants d'un objet (interface).
- 2) Java ne prend pas en charge "l'héritage multiple" (une classe ne peut hériter que d'une seule superclasse). Cependant, cela peut être réalisé avec des interfaces, car la classe peut **implémenter** plusieurs interfaces.

 **Note :** Pour implémenter plusieurs interfaces, séparez-les par une virgule (voir l'exemple ci-dessous).

 Example :

```
interface FirstInterface {

    public void myMethod(); // interface method

}

interface SecondInterface {

    public void myOtherMethod(); // interface method

}

class DemoClass implements FirstInterface, SecondInterface {

    public void myMethod() {

        System.out.println("Some text..");

    }

    public void myOtherMethod() {

        System.out.println("Some other text...");

    }

}

class Main {

    public static void main(String[] args) {

        DemoClass myObj = new DemoClass();

        myObj.myMethod();

        myObj.myOtherMethod();

    }

}
```

✓✓ Les énumération en java

Une **énumération** est une "classe" spéciale qui représente un groupe de **constantes** (variables immuables, comme les variables **final**).

Pour créer une énumération , utilisez le mot-clé **enum** (au lieu de **class** ou **interface**) et séparez les constantes par une virgule. Notez qu'ils doivent être en lettres majuscules. Vous pouvez accéder aux constantes **enum** avec la syntaxe **point** :

📁 Exemple :

```
// Dans Level.java

enum Level {

    LOW,

    MEDIUM,

    HIGH

}

// Dans Main.java

public class Main {

    public static void main(String[] args) {

        Level myVar = Level.MEDIUM;

        System.out.println(myVar) ;

    }

}
```

Enumération interne

Vous pouvez également avoir un **enum** interne à une classe :

Exemple :

```
public class Main {  
  
    enum Level {  
  
        LOW,  
  
        MEDIUM,  
  
        HIGH  
    }  
  
  
    public static void main(String[] args) {  
  
        Level myVar = Level.MEDIUM;  
  
        System.out.println(myVar) ;  
  
    }  
}
```

Énumération dans une instruction Switch

Les énumérations sont souvent utilisées dans les instructions **switch** pour vérifier les valeurs correspondantes :

Exemple :

```
// Dans Level.java

enum Level { LOW, MEDIUM, HIGH }

// Dans Main.java

public class Main {

    public static void main(String[] args) {

        Level myVar = Level.MEDIUM;

        switch(myVar) {

            case LOW:

                System.out.println("Low level");

                break;

            case MEDIUM:

                System.out.println("Medium level");

                break;

            case HIGH:

                System.out.println("High level");

                break;

        }

    }

}
```

Parcourez une énumération

Le type **enum** a une méthode **values()** qui renvoie un tableau de toutes les constantes enum. Cette méthode est utile lorsque vous souhaitez parcourir les constantes d'une énumération :

```
for (Level myVar : Level.values()) {  
  
    System.out.println(myVar) ;  
  
}
```

Différence entre les énumérations et les classes

- ❖ Une énumération peut, tout comme une **class**, avoir des attributs et des méthodes. La seule différence est que les constantes d'énumération sont **public**, **static** et **final** (inchangables - ne peuvent pas être remplacées).
- ❖ Une énumération ne peut pas être utilisée pour créer des objets et ne peut pas étendre à d'autres classes (mais elle peut implémenter des interfaces).

Pourquoi et quand utiliser les énumérations ?

Utilisez des énumérations lorsque vous avez des valeurs dont vous savez qu'elles ne changeront pas, comme les jours du mois, les jours, les couleurs, le jeu de cartes, etc.

✓✓ Les fichiers en Java

La gestion des fichiers est une partie importante de toute application.

Java dispose de plusieurs méthodes pour créer, lire, mettre à jour et supprimer des fichiers. La classe **File** du package **java.io** nous permet de travailler avec des fichiers.

Pour utiliser la classe **File**, créez un objet de la classe et spécifiez le nom du fichier ou du répertoire :

📁 Exemple :

```
import java.io.File; // Import the File class

public class Main {

    public static void main(String[] args) {

        File myObj = new File("filename.txt"); // Spécifiez le nom de
fichier

    }

}
```

La classe **File** dispose de nombreuses méthodes utiles pour créer et obtenir des informations sur les fichiers.

 Exemple :

Méthode	Type	Description
canRead()	Boolean	Tests si le fichier est lisible ou non
canWrite()	Boolean	Tests si le fichier est écrite ou non
createNewFile()	Boolean	Crée un fichier vide
delete()	Boolean	Supprime un fichier
exists()	Boolean	Teste si le fichier existe
getName()	String	Renvoie le nom du fichier
getAbsolutePath()	String	Renvoie le chemin d'accès absolu du fichier
length()	Long	Renvoie la taille du fichier en octets
list()	String[]	Renvoie un tableau des fichiers dans le répertoire
mkdir()	Boolean	Crée un répertoire

Créer un fichier

Pour créer un fichier en Java, vous pouvez utiliser la méthode **createNewFile()**. Cette méthode renvoie une valeur booléenne : **true** si le fichier a été créé avec succès et **false** si le fichier existe déjà. Notez que la méthode est enfermée dans un bloc **try...catch**. Ceci est nécessaire car il renvoie un message **IOException** si une erreur se produit (si le fichier ne peut pas être créé pour une raison quelconque) :

Exemple :

```
import java.io.File;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        try {

            File myObj = new File("filename.txt");

            if (myObj.createNewFile()) {

                System.out.println("Fichier créer avec succès " +

                    myObj.getName());

            } else { System.out.println("Fichier existe déjà.");}

        } catch (IOException e) {

            System.out.println("Erreur X.");

            e.printStackTrace();

        }

    }

}
```

 Note :

Pour créer un fichier dans un répertoire spécifique (nécessite une autorisation), spécifiez le chemin du fichier et utilisez des doubles barres obliques inverses pour échapper le \ caractère " " (pour Windows). Sur Mac et Linux, vous pouvez simplement écrire le chemin, comme :
/Users/name/filename.txt

```
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

Écrire dans un fichier

Dans l'exemple suivant, nous utilisons la classe **FileWriter** avec sa méthode **write()** pour écrire du texte dans le fichier que nous avons créé dans l'exemple ci-dessus. Notez que lorsque vous avez fini d'écrire dans le fichier, vous devez le fermer avec la méthode **close()** :

Exemple :

```
import java.io.FileWriter;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        try {

            FileWriter myWriter = new FileWriter("filename.txt");

            myWriter.write("Les fichiers en Java peuvent être délicats,

            mais c'est assez amusant!");

            myWriter.close();

            System.out.println("Écrit avec succès au fichier.");

        } catch (IOException e) {

            System.out.println("Une erreur s'est produite.");

            e.printStackTrace();

        }

    }

}
```

Lire dans un fichier

Dans le chapitre précédent, vous avez appris à créer et à écrire dans un fichier. Dans l'exemple suivant, nous utilisons la classe **Scanner** pour lire le contenu du fichier texte que nous avons créé dans le chapitre précédent :

Exemple :

```
import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        try {

            File myObj = new File("filename.txt");

            Scanner myReader = new Scanner(myObj);

            while (myReader.hasNextLine()) {

                String data = myReader.nextLine();

                System.out.println(data);

            }

            myReader.close();

        } catch (FileNotFoundException e) {

            System.out.println("Une erreur s'est produite.");

            e.printStackTrace();

        }

    }

}
```

Obtenir des informations sur le fichier

Pour obtenir plus d'informations sur un fichier, utilisez l'une des méthodes **File** suivantes :

Exemple :

```
import java.io.File; // Import the File class

public class Main {

    public static void main(String[] args) {

        File myObj = new File("filename.txt");

        if (myObj.exists()) {

            System.out.println("File name: " + myObj.getName());

            System.out.println("Absolute path: " +

myObj.getAbsolutePath());

            System.out.println("Writeable: " + myObj.canWrite());

            System.out.println("Readable " + myObj.canRead());

            System.out.println("File size in bytes " + myObj.length());

        } else {

            System.out.println("The file does not exist.");

        }

    }

}
```

Remarque : Il existe de nombreuses classes disponibles dans l'API Java qui peuvent être utilisées pour lire et écrire des fichiers en Java : , **FileReader**, **BufferedReader**, **Files**, **Scanner**, **FileInputStream**, **FileWriter**, **BufferedWriter**, **FileOutputStream** etc. Celle à utiliser dépend de la version de Java avec laquelle vous travaillez et si vous devez lire des octets ou des caractères, et la taille du fichier/des lignes, etc.

Supprimer un fichier

Pour supprimer un fichier en Java, utilisez la méthode **delete()** :

Exemple :

```
import java.io.File; // Import the File class

public class Main {

    public static void main(String[] args) {

        File myObj = new File("filename.txt");

        if (myObj.delete()) {

            System.out.println("Deleted the file: " + myObj.getName());

        } else {

            System.out.println("Failed to delete the file.");

        }

    }

}
```

Supprimer un dossier

Vous pouvez également supprimer un dossier. Cependant, il doit être vide :

Exemple :

```
import java.io.File;

public class Main {

    public static void main(String[] args) {

        File myObj = new File("C:\\Users\\MyName\\Test");

        if (myObj.delete()) {

            System.out.println("Deleted the folder: " + myObj.getName());

        } else {

            System.out.println("Failed to delete the folder.");

        }

    }

}
```

✓✓ Des Astuces

Compter le nombre de mots dans une chaîne

Vous pouvez facilement compter le nombre de mots dans une chaîne avec l'exemple suivant :

Exemple :

```
import java.io.File;

public class Main {

    public static void main(String[] args) {

        String words = "One Two Three Four";

        int countWords = words.split("\\s").length;

        System.out.println(countWords); //4

    }

}
```

Ajouter deux nombres avec la saisie de l'utilisateur

Exemple :

```
import java.util.Scanner; // Import the Scanner class

class Main {

    public static void main(String[] args) {

        int x, y, sum;

        Scanner myObj = new Scanner(System.in); // Create a Scanner object

        System.out.println("Type a number:");

        x = myObj.nextInt(); // Read user input

        System.out.println("Type another number:");

        y = myObj.nextInt(); // Read user input

        sum = x + y; // Calculate the sum of x + y

        System.out.println("Sum is: " + sum); // Print the sum

    }

}
```

Inverser une chaîne

Vous pouvez facilement inverser une chaîne de caractères avec l'exemple suivant :

 Exemple :

```
import java.util.Scanner; // Import the Scanner class

class Main {

    public static void main(String[] args) {

        String originalStr = "Hello";

        String reversedStr = "";

        for (int i = 0; i < originalStr.length(); i++) {

            reversedStr = originalStr.charAt(i) + reversedStr;

        }

        System.out.println("Reversed string: " + reversedStr);

    }

}
```

 **djibysec8@gmail.com**

< **DSS NAG Fin !** />