JavaCore
Master Guide

Écrit par : Guillaume “JirAWS” JACQUELET
Date de publication : 2024
Version : 1.8.6

YouTube : https://www.youtube.com/c/JirAWS
Site web : https://www.jiraws.com
Mail : contact@jiraws.com

Sommaire 📖
[Summary]

Introduction 🌟
[Introduction]

Bienvenue dans le **JavaCore Master Guide**, le guide complet des mots-clés, concepts et principes de programmation à connaître pour devenir développeur Java.

La promesse de ce document est de fournir une définition claire, rédigée personnellement par l'auteur (JirAWS), pour chacun des termes jugés incontournables du langage Java.

Le monde de la programmation Java est vaste, et être capable de donner un sens à chaque terme de ce langage est primordial pour comprendre son envergure et ainsi mieux structurer ses idées.

L'Univers de l'informatique étant majoritairement anglophone, vous trouverez dans ce document des traductions anglais <-> français, permettant de mieux saisir la signification derrière les anglicismes fréquemment rencontrés sur Internet et dans les documentations souvent anglophones.

Bien qu'il soit de bonne pratique d'écrire son code (commentaires, noms de variables, méthodes, classes, etc.) en anglais, ce document vise à faciliter l'apprentissage de la programmation et s'écarte donc volontairement de cette pratique pour réduire les obstacles. Les exemples de code fournis contiennent délibérément du "franglais" (français/anglais) pour en faciliter la compréhension.

Ce document sert également de support de connaissances tout au long du programme JavaCore (Essential, Plus+ et Élite), une formation Java de A à Z disponible sur le site (https://www.jiraws.com).

Note de l'Auteur ✍️
[Author's Note]

Nous, développeurs, avons tous des termes, des concepts et des principes de programmation que nous ne connaissons que vaguement, sans être capables de les expliquer, ne serait-ce qu'en surface.

C'est en cela que j'ai tenu à ce que ce document soit le plus complet possible et accessible à tous. Les définitions ont chacune été rédigées par mes soins, avec mes mots et ma vision pédagogique des choses pour associer des phrases simples aux différents éléments de la programmation en Java.

À l'ère de l'IA, écrire un tel document sans assistance serait un challenge de taille, mais c'est bel et bien la direction qui a été choisie. Les définitions et les exemples de code ont entièrement été rédigés manuellement, et l'usage de l'IA (ChatGPT) n'a été employé que pour assurer la qualité finale du document avec notamment des vérifications de contenu ou encore des corrections orthographiques.

La programmation est un amalgame de logiques à comprendre et de connaissances à apprendre, et ce document a pour but de jouer ce rôle de recueil de connaissances minimales. C'est pourquoi ce document a été rédigé pour jouer un rôle important dans le programme **JavaCore**, ma formation Java de A à Z pour débutants.

Je vous conseille de consommer le contenu de ce guide sans modération car l'entièreté des termes présents et de leurs définitions aura une utilité, tôt ou tard, dans votre carrière.

Si vous connaissez l'ensemble des définitions présentes dans ce guide, vous pouvez alors considérer avoir une très bonne connaissance théorique du monde de la programmation en Java.

Comme d'habitude, vous avez la parole. Si vous estimez qu'un terme devrait, selon vous, avoir sa place dans ce document, n'hésitez pas à m'en faire part (par mail: contact@jiraws.com, ou via Discord).

Bon apprentissage !

Guillaume (JirAWS)

Mises à Jour 🔄
[Updates]

Ce document sera amené à évoluer dans le temps, avec des ajouts de nouvelles définitions et d'exemples de code, et si nécessaire, des corrections voire même des améliorations du contenu déjà présent.

Les différentes mises à jour auront toutes pour objectif de faire en sorte que ce guide conserve sa promesse initiale : proposer une explication claire pour chaque mot-clé à connaître dans le monde du langage Java.

Pour suivre les évolutions, une annexe de l'historique des versions est disponible en fin de document.

Soutien 💪
[Support]

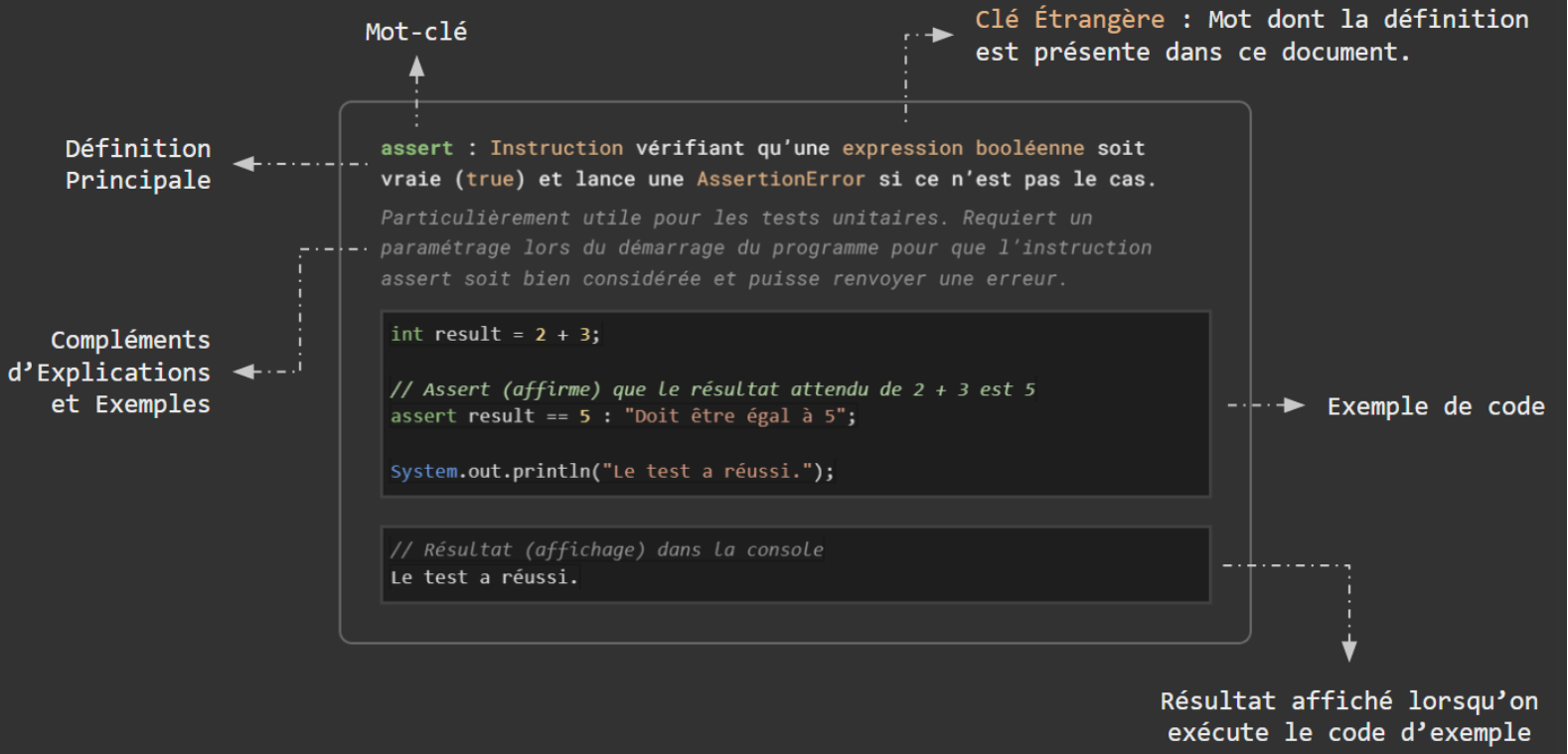
Pour celles et ceux qui souhaiteraient soutenir le développement de ce document, ou tout simplement me remercier pour les efforts fournis, une plateforme de dons est disponible :

https://www.buymeacoffee.com/JirAWS

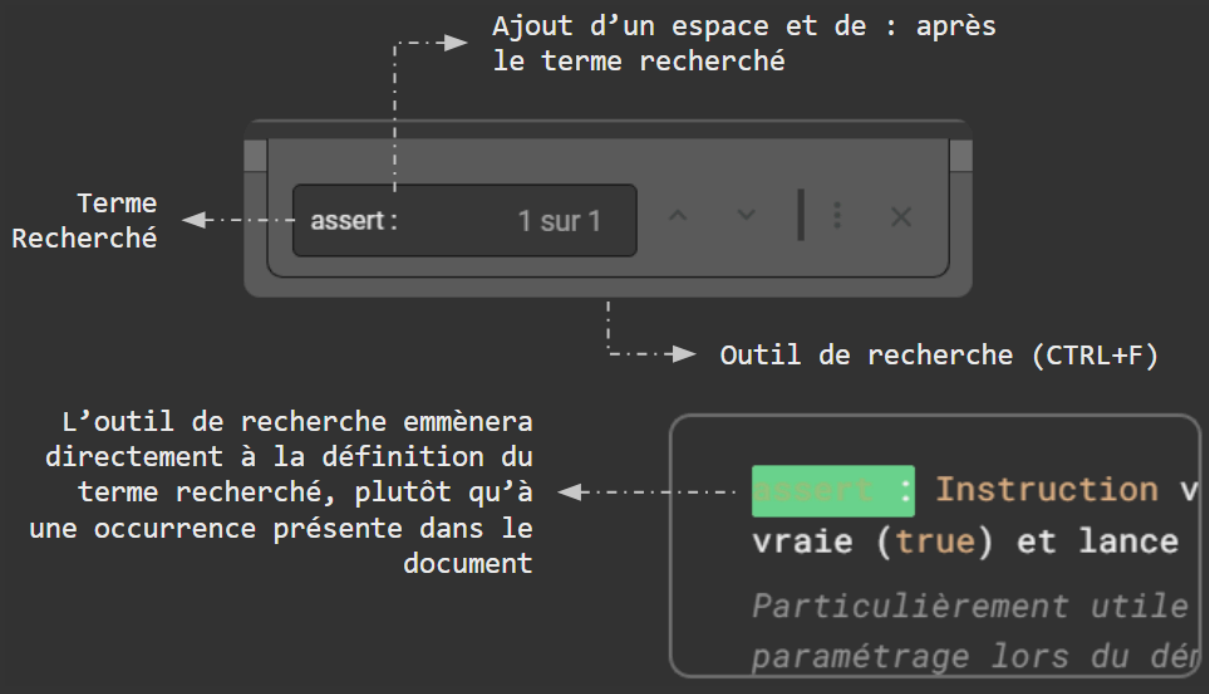
Note : On reste dans le thème de Java, avec pour symbole le café.

Mode d'Emploi 🔧
[How To]

Voici une brève illustration explicative de la structure du contenu.



Conseil de recherche avec l'outil CTRL+F pour parcourir le document.



Sommaire Alphabétique 🔍
[Alphabetical Summary]

1. Définitions Générales (General Definitions)

2. Les Quatre Piliers de la POO (The Four Pillars of OOP)

3. Environnement Java (Java Environment)

4. Mots-clés Java (Java Keywords)

5. Classes et Interfaces Java (Java Classes and Interfaces)

6. Méthodes Java (Java Methods)

7. Principes de Conception (Design Principles)

8. Formats de Fichiers (File Formats)

Bases du Langage Java ☕
[Java Language Basics]

1. Environnement Java (Java Environment)

JVM (Java Virtual Machine) : Moteur d'exécution qui permet de faire fonctionner des applications Java. C'est grâce à elle que l'on peut exécuter des programmes écrits en langage Java.

JRE (Java Runtime Environment) : Environnement d'exécution Java qui comprend la JVM et les bibliothèques (bibliothèque) nécessaires pour exécuter des applications Java. Il contient tous les éléments nécessaires pour faire fonctionner des applications Java, mais il ne contient pas les outils de développement du JDK.

JDK (Java Development Kit) : Kit de développement pour Java. Il inclut tout le nécessaire pour développer des applications Java, y compris un compilateur (compilation), des outils de débogage, ainsi que la JVM pour tester les applications développées. *C'est l'élément que vous devez installer sur votre ordinateur pour pouvoir écrire des programmes en Java sur votre IDE (Eclipse, IntelliJ, etc.).*

Garbage Collection (Collecte des Ordures) : Processus interne du langage Java qui vient débarrasser la mémoire vive des éléments qui ne sont plus utilisés. *Une variable déclarée dans un bloc de code sera, après exécution, détectée comme inutilisée par le Garbage Collector et supprimée de la mémoire pour libérer la place.*

WORE (Write Once, Run Everywhere) : Promesse centrale du langage Java visant à rendre un programme compatible multi-plateforme, c'est à dire exécutable sur "tous" les systèmes disposant d'une JVM, sans devoir adapter le code à chaque fois. *En pratique, l'idéal du "Write Once, Run Everywhere" est complexe à réaliser, car de nombreux facteurs tels que les différences d'environnements d'exécution et les spécificités des plateformes entrent en jeu. Une maîtrise approfondie de Java et de son écosystème est nécessaire pour naviguer ces défis et atteindre une véritable portabilité du code.*

2. Syntaxe de base (Basic Syntax)

Syntaxe : Ensemble de règles qui définissent les combinaisons valides de symboles pour écrire un programme.

Par exemple, la valeur d'une chaîne de caractères (`String`) en Java est écrite entre des " " (guillemets). L'utilisation des ' ' (apostrophes) entraîne une erreur de compilation car ces derniers sont réservés pour les caractères (`char` / `Character`).

```
String texte = "Un texte";
char caractere = 'c';
```

Variable : Espace mémoire accessible par un nom, permettant de stocker une valeur de type primitif ou référence. *Le terme "variable" vient directement du fait que sa valeur puisse varier dans le temps.*

```
int age = 25;

age = 26; // On peut changer la valeur affectée à la variable
```

Constante : Une variable dont la valeur ne peut pas être changée après son initialisation. *Comme une variable, une constante possède un nom et un type. En Java, il faut ajouter le modificateur `final` à une variable pour qu'elle soit considérée comme une constante.*

```
final int age = 25;

age = 26; // Erreur, une constante ne peut pas changer de valeur
```

Type : "Étiquette" indiquant la nature de la donnée que contiendra une variable, le retour d'une méthode ou même un élément générique (dans le cas de la généricité). *C'est une information qui est notamment considérée par un IDE, permettant ainsi aux développeurs d'écrire du code plus sûr et d' éviter les erreurs.*

Déclaration : Étape de création d'une variable, nécessitant la combinaison d'un type et d'un nom. *C'est une étape nécessaire pour réserver un espace mémoire dédié à la valeur qui sera affectée durant son initialisation.*

```
// Déclaration d'une variable nommée "age" de type "int" (entier)
int age;
```

Initialisation : Première affectation de valeur à une variable.

```
// Déclaration, puis initialisation, d'une variable nommée "age"
int age;
age = 25;

// Déclaration et initialisation d'une variable nommée "annee"
int annee = 1995;
```

Référence : Type de données composite qui pointe vers un objet dans la mémoire. *En lien direct avec la notion d'objet en Java, des structures regroupant les données et améliorant leur manipulation.*

null : Valeur spéciale d'une variable référence indiquant que cette dernière est "vide", ou plutôt, qu'elle n'est affectée à aucune référence mémoire (et donc, aucun objet).

```
// Une variable référence est créée, mais ne pointe vers aucun objet
Object object = null;
```

var : Type spécial d'une variable à utiliser dans sa déclaration pour invoquer l'inférence de type.

```
var age = 25;
```

Inférence : Processus par lequel la compilation détermine automatiquement le type d'une variable à partir de sa valeur.

```
/**
 *  le langage Java déduira automatiquement que la variable "age"
 *  est affectée à une valeur numérique entière (int)
 */
var age = 25;
```

Portée de Variable : Rayon d'action d'une variable déterminant dans quelle(s) partie(s) du code cette dernière peut être accédée et manipulée. Directement lié à la notion de bloc de code.

Une variable déclarée dans le bloc de code d'une condition if ne sera accessible que dans ce dernier, ou dans ceux se trouvant à l'intérieur (dans une autre condition imbriquée par exemple).

```
int age = 25;

if (age >= 18) {

	System.out.println("Vous êtes majeur en France.");

	// La variable age est également accessible dans ce bloc de code
	if (age >= 65) {
		System.out.println("Vous êtes également senior en France");
	}

	// Variable déclarée dans le bloc de code du if
	int uneVariableDansLeBlocIf = 1337;

}

/**
 * Erreur: La variable a été déclarée dans un sous-bloc de code.
 * Le mécanisme de la Portée de Variable empêche donc son utilisation.
 */
uneVariableDansLeBlocIf = 2000;
```

```
// Résultat (affichage) dans la console
Vous êtes majeur en France.
```

3. Types Primitifs (Primitive Types)

Primitif : Type de donnée le plus basique en programmation, permettant de représenter des informations primaires.

En Java, on retrouve 8 types primitifs couvrant l'ensemble des besoins de représentation des données:

- *Valeurs Numériques Entières : byte, short, int et long.*
- *Valeurs Flottantes (à virgule) : float et double.*
- *Valeurs Caractères : char.*
- *Valeurs Booléennes : boolean.*

byte : Type de données signé sur 8 bits pour les toutes petites valeurs numériques entières. *Intervalle de valeurs : -128 à +127*

```
byte distanceSalleDeSport = 2;
```

short : Type de données signé sur 16 bits pour les petites valeurs numériques entières. *Intervalle de valeurs : -32.768 à +32.767*

```
short rayonPlaneteTerre = 6371;
```

int : Type de données signé sur 32 bits pour les valeurs numériques entières. *Intervalle de valeurs : -2.147.483.648 à +2.147.483.647*

```
int distanceTerreLune = 384400;
```

long : Type de données signé sur 64 bits pour les grandes valeurs numériques entières. *Intervalle de valeurs : -2^63 à +2^63-1*

```
// On utilise 'L' ou 'l' à la fin de la valeur pour indiquer "long"
long distanceSoleilNeptune = 4500000000L;
```

char : Type de données non-signé sur 16 bits pour les caractères.

Intervalle de valeurs : 0 à 65.535. Les 65536 premiers caractères Unicode (de U+0000 à U+FFFF) font partie de la Plage du Plan Multilingue de Base (BMP pour Basic Multilingual Plane).

```
char aMajuscule = 'A'; // Valeur entière Unicode associée : 65
char bMajuscule = 'B'; // Valeur entière Unicode associée : 66
char aMinuscule = 'a'; // Valeur entière Unicode associée : 97
```

float : Type de données sur 32 bits pour les nombres à virgule flottante simple précision. *Intervalle de valeurs : 1.4E-45 à 3.4028234E38*

```
// On utilise 'F' ou 'f' à la fin de la valeur pour indiquer "float"
float tailleMoyenneHumain = 1.75F;
```

double : Type de données sur 64 bits pour les nombres à virgule flottante double précision. *Intervalle de valeurs : 4.9E-324 à 1.79769313486231157E308*

```
double approximationPi = 3.141592653589793;
```

boolean : Type de données sur 1 bit pour les valeurs true (vrai) ou false (faux). *Basé sur les transistors des processeurs de nos machines, de petits “interrupteurs” laissant, ou non, circuler l’électricité : ouvert (vrai / true / 1), ou fermé (faux / false / 0).*

true : Valeur booléenne vraie. *Équivalent de 1 en binaire.*

```
boolean like = true;
```

false : Valeur booléenne fausse. *Équivalent de 0 en binaire.*

```
boolean dislike = false;
```

4. Fonctions (Functions)

Fonction : Sous-programme composé d'un nom et d'un bloc de code pouvant avoir des paramètres (paramètre) et retourner (renvoyer) un résultat suite à son exécution.

Lorsque l'on demande à son (gentil) collègue de nous ramener un café, on fait appel à sa fonction "faireUnCafe()" contenant les actions (instruction) à réaliser, étape par étape, pour faire un café et par la suite nous le donner (renvoyer).

```
TypeDeRetour nomDuProgramme(TypeDuParamètre nomDuParamètre) {

	instruction1;
	instruction2;
	instruction3;

	return valeurRetournée;

}
```

Bloc de code : Ensemble d'instructions (instruction) délimitées par des accolades { } en Java.

*Des éléments comme une **méthode**, une **condition** et une **boucle** possèdent tous un bloc de code associé et ne sont en fait que des manières différentes de démarrer leur exécution.*

*La notion de bloc de code est liée à celle de la **portée de variable**.*

```
// Un bloc de code ne contenant qu'une seule instruction
{
	System.out.println("Bloc de code affichant ce texte");
}

// Un bloc de code peut en contenir d'autres
{

	{
		System.out.println("Bloc de code affichant ce texte");
	}

}
```

Paramètre : Variable spécifiée dans la déclaration d'une fonction agissant comme un "espace réservé" pour une valeur (argument) qui sera transmise au moment de l'appel de cette dernière.

C'est ce qui permet de fournir des informations (valeurs) à une fonction exécutée, malgré les contraintes de la portée de variable.

```
// Une fonction sans paramètre ()
void afficherNomFormation() {
	System.out.println("JavaCore");
}

// Une fonction avec un seul paramètre (int a)
int multiplierParDeux(int a) {
	return a * 2;
}

// Une fonction avec deux paramètres (int a, int b)
int addition(int a, int b) {
	return a + b;
}
```

Argument : Valeur passée dans le paramètre d'une fonction au moment de son appel (exécution). *On peut voir un paramètre comme l'étape de déclaration d'une variable, et un argument comme une valeur d'initialisation affectée lors de l'appel de la fonction.*

```
void fonction1(int a, int b) {
	/**
	 * a et b sont ici des paramètres de la fonction1
	 * soit des variables en attente d'initialisation
	 */
}

void programme() {

	/**
	 * 2 et 4 sont des valeurs (arguments) passées dans les paramètres
	 * a et b de la fonction1 lors de son appel
	 */
	fonction1(2, 4); // Instruction qui appelle (exécute) la fonction1

}
```

return : Instruction qui termine l'exécution d'une fonction et permettant de retourner une valeur si nécessaire.

Une fonction sans cette instruction s'arrêtera d'elle-même quand elle aura terminé d'exécuter ses instructions.

```
// Une fonction qui renvoie le résultat de l'addition de deux paramètres
int addition(int a, int b) {
	return a + b;
}

// Une fonction "appelle" la fonction "addition" et récupère le résultat
void programme() {

	int resultat = addition(2, 4);

	System.out.println("Résultat : " + resultat);

}
```

```
// Résultat (affichage) dans la console
Résultat : 6
```

void (vide) : Type de retour spécial indiquant qu'une fonction ne renvoie rien (aucun résultat). *On peut alors dire que cette fonction est une procédure.*

```
// Une fonction qui ne fait qu'afficher le texte passé en argument
void affichageDeTexte(String texte) {
	System.out.println(texte);
}
```

Procédure : Une fonction qui ne retourne pas de résultat, et qui utilise donc le type de retour void. *C'est un terme très peu utilisé en Java, bien qu'une fonction ne retournant aucun résultat puisse quand même être considérée comme une procédure.*

5. Opérateurs (Operators)

Opérateur : Symbole spécial utilisé pour effectuer des opérations spécifiques sur une ou plusieurs opérandes. *Il existe différents opérateurs, répondant chacun à des besoins spécifiques.*

Opérateurs Arithmétiques : Ensemble d'opérateurs (opérateur) permettant de réaliser les opérations mathématiques de base.

- *+ (Addition): Additionne deux valeurs.*
- *- (Soustraction): Soustrait une valeur d'une autre.*
- ** (Multiplication): Multiplie deux valeurs.*
- */ (Division): Divise une valeur par une autre.*
- *% (Modulo): Renvoie le reste de la division.*

```
int a = 2 + 3; // 5
int b = 9 - 5; // 4
int c = a * b; // 20 car 5 x 4
int d = c / 2; // 10 car 20 / 2
int e = d % 2; // 0 car 10 = 2 x 5 et il reste 0.
```

Opérateurs Incrémentation / Décrémentation : Paire d'opérateurs (opérateur) permettant l'addition et la soustraction de 1.

- *++ (Incrémentation): Augmente la valeur de 1.*
- *-- (Décrémentation): Diminue la valeur de 1.*

```
int a = 5;
System.out.println("Valeur initiale de 'a' : " + a);

a++; // a = a + 1;
System.out.println("Valeur de 'a' après incrémentation (++) : " + a);

a--; // a = a - 1;
System.out.println("Valeur de 'a' après décrémentation (--) : " + a);
```

```
// Résultat (affichage) dans la console
Valeur initiale de 'a' : 5
Valeur de 'a' après incrémentation (++) : 6
Valeur de 'a' après décrémentation (--) : 5
```

Opérateur Affectation / Assignation : Ensemble d'opérateurs (opérateur) ayant pour rôle d'attribuer une valeur à une variable.

Pour simplifier l'écriture de code, des opérateurs d'affectation (assignation) combinés ont été ajoutés au langage Java.

- *= (Affectation): Affecte une valeur à une variable.*
- *+= (Addition puis affectation): Additionne puis affecte.*
- *-= (Soustraction puis affectation): Soustrait puis affecte.*
- **= (Multiplication puis affectation): Multiplie puis affecte.*
- */= (Division puis affectation): Divise puis affecte.*
- *%= (Modulo puis affectation): Applique le modulo puis affecte.*

```java
int a = 5;
System.out.println("Valeur initiale de 'a' : " + a);

a += 6; // a = a + 6;
System.out.println("Valeur de 'a' après addition (+) : " + a);

a -= 5; // a = a - 5;
System.out.println("Valeur de 'a' après soustraction (-) : " + a);

a *= 4; // a = a * 4;
System.out.println("Valeur de 'a' après multiplication (*): " + a);

a /= 3; // a = a / 3;
System.out.println("Valeur de 'a' après division (/) : " + a);

a %= 2; // a = a % 2;
System.out.println("Valeur de 'a' après modulo (%) : " + a);
```

```
// Résultat (affichage) dans la console
Valeur initiale de 'a' : 5
Valeur de 'a' après addition (+) : 11
Valeur de 'a' après soustraction (-) : 6
Valeur de 'a' après multiplication (*): 24
Valeur de 'a' après division (/) : 8
Valeur de 'a' après modulo (%) : 0
```

Expression Booléenne : Combinaison d'opérandes dont le résultat est une valeur de type boolean. *C'est ce qu'un développeur utilise quotidiennement en programmation, car la logique informatique contemporaine est basée sur ces dernières.*

Opérateurs de Comparaison : Ensemble d'opérateurs (opérateur) comparant deux valeurs et renvoyant un résultat de type boolean.

- *== (Égal à): Vérifie si deux valeurs sont égales.*
- *!= (Différent de): Vérifie si deux valeurs sont différentes.*
- *> (Plus grand que): Vérifie si une valeur est plus grande.*
- *< (Plus petit que): Vérifie si une valeur est plus petite.*
- *>= (Plus grand ou égal à): Vérifie si une valeur est plus grande ou égale à une autre.*
- *<= (Plus petit ou égal à): Vérifie si une valeur est plus petite ou égale à une autre.*

```
boolean isEqual = 5 == 5; // true

boolean isDifferent = 5 != 10; // true

boolean isGreater = 10 > 5; // true

boolean isLower = 5 < 10; // true

boolean isGreaterOrEqual = 10 >= 10; // true

boolean isLowerOrEqual = 20 <= 19; // false
```

Opérateurs Logiques : Triplet d'opérateurs (opérateur) permettant de construire des expressions logiques, basées sur les tables de vérités AND (ET), OR (OU) et NOT (NON).

- *&& (ET): Renvoie true si les deux opérandes sont vraies.*
- *|| (OU): Renvoie true si au moins une des opérandes est vraie.*
- *! (NON): Inverse la valeur booléenne.*

```
int a = 5;
int b = 10;

boolean conditionAnd1 = (a > 5) && (b == 10); // false
boolean conditionAnd2 = (a > 1) && (b == 10); // true

boolean conditionOr1  = (a > 5) || (b == 10); // true
boolean conditionOr2  = (a > 1) || (b == 10); // true

boolean conditionNot1 = !(a == 5); // false
boolean conditionNot2 = !(b > 20); // true
```

6. Affirmations (Assertions)

assert : Instruction vérifiant qu'une expression booléenne soit vraie (true) et lance une AssertionError si ce n'est pas le cas.

Particulièrement utile pour les tests unitaires. Requiert l'activation des assertions lors du démarrage du programme pour que l'instruction assert soit bien considérée et puisse renvoyer une erreur.

```
int result = 2 + 3;

// Assert (affirme) que le résultat attendu de 2 + 3 est 5
assert result == 5 : "Doit être égal à 5";

System.out.println("Le test a réussi.");
```

```
// Résultat (affichage) dans la console
Le test a réussi.
```

AssertionError : Erreur qui se produit lorsque le résultat d'une "vérification d'affirmation" (assert) est faux (false).

Une assertion est une affirmation que le développeur suppose être vraie lors de l'exécution d'un code. Une assertion échoue lorsqu'une condition supposée vraie se révèle fausse dans le programme, lançant alors une AssertionError signalant au développeur le comportement anormal rencontré.

```
int result = 10 + 10;

// Assert (affirme) que le résultat attendu de 10 + 10 est 5
assert result == 5 : "Doit être égal à 5";

System.out.println("Le test a réussi.");
```

```
// Résultat (affichage) dans la console
Exception in thread "main" java.lang.AssertionError: Doit être égal à 5
    at TestAssertion.main(TestAssertion.java:6)
```

7. Structures Conditionnelles (Conditional Structures)

Condition : Bloc de code exécuté si, et seulement si, l'expression booléenne évaluée renvoie vrai (true). *Plus globalement, une condition est une vérification qui renvoie un résultat binaire, c'est-à-dire soit vrai (true), soit faux (false).*

if : Mot-clé utilisé pour déclarer une condition et dont le résultat déterminera l'exécution de son bloc de code associé.

Requiert une valeur de type boolean qui peut provenir d'une variable, d'une expression booléenne ou d'un retour de méthode.

```
int age = 25;

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

```
// Résultat (affichage) dans la console
Vous êtes majeur en France.
```

else : Mot-clé optionnel qui suit le bloc de code d'un if ou d'un else if et qui est exécuté si la ou les condition(s) précédente(s) sont fausses (false). *Peut se lire « sinon » et sert de comportement par défaut lorsque les conditions précédentes n'ont pas été remplies :*

- *Si (if) le restaurant est ouvert, on y va ce soir.*
- *Sinon (else), on se fait à manger.*

```
int age = 15;

if (age >= 18) {
	System.out.println("Vous êtes majeur en France.");
}
else {
	System.out.println("Vous êtes mineur en France.");
}
```

```
// Résultat (affichage) dans la console
Vous êtes mineur en France.
```

else if : Mot-clé optionnel qui permet d'écrire une condition alternative à un if et suivant le bloc de code de ce dernier.

- *Si (if) le restaurant est ouvert, on y va ce soir.*
- *Sinon si (else if) ils peuvent livrer, on commande.*
- *Sinon (else), on se fait à manger.*

```
int age = 17;

if (age >= 18) {
	System.out.println("Vous êtes majeur en France.");
}
else if (age == 17) {
	System.out.println("Vous êtes presque majeur en France.");
}
else {
	System.out.println("Vous êtes mineur en France.");
}
```

```
// Résultat (affichage) dans la console
Vous êtes presque majeur en France.
```

switch : Condition multiple sur une valeur, à la syntaxe (écriture) différente mais conservant la logique d'une condition standard (if). *Équivalent à un if, else if, …, else if, else mais en plus optimisé.*

case : Définit le code à exécuter si la valeur évaluée dans un switch correspond (est égale). *Équivalent au else if d'une condition.*

default : Définit le code à exécuter si aucun case ne correspond à la valeur évaluée dans un switch. *Équivalent au else d'une condition.*

```
char note = 'A';

switch (note) {
    case 'A':
        System.out.println("Excellent !");
        break;
    case 'B':
        System.out.println("Bien !");
        break;
    case 'C':
        System.out.println("Correct.");
        break;
    case 'D':
        System.out.println("Passable.");
        break;
    case 'E':
        System.out.println("Mauvais.");
        break;
    default:
        System.out.println("Note non reconnue.");
}
```

```
// Résultat (affichage) dans la console
Excellent !
```

yield : Instruction permettant de retourner une valeur depuis une expression switch. *Équivalent du return pour une méthode, permettant ainsi de récupérer un résultat dans une variable.*

```
char direction = 'N';

String cardinalPoint = switch (direction) {
    case 'N':
        yield "Nord";
    case 'S':
        yield "Sud";
    case 'E':
        yield "Est";
    case 'O':
        yield "Ouest";
    default:
        yield "Direction inconnue";
};

System.out.println("La direction est : " + cardinalPoint);
```

```
// Résultat (affichage) dans la console
La direction est : Nord
```

8. Boucles et Itérations (Loops and Iterations)

Boucle : Bloc de code exécuté autant de fois (itération) que l'expression booléenne évaluée sera vraie (true).

Mécanisme de programmation qui permet une forme de "raisonnement" (entre gros guillemets) de nos machines grâce à des vérifications en continu offrant la possibilité de réagir à des changements de données.

Itération : Résultat de l'action d'itérer, à savoir le fait de répéter un comportement ou une logique.

Une boucle itère tant que sa condition ne renvoie pas faux (false), autrement dit, elle répète l'exécution de son bloc de code tant que la condition renvoie vrai (true).

while : Boucle pouvant être lue “tant que” et qui continue de s’exécuter aussi longtemps que sa condition renverra vrai (true).

C’est une forme de boucle très utile lorsque l’évolution de la condition possède une part d’inconnu. Il faudra alors que son bloc de code, ou autre code extérieur à cette dernière, modifie les facteurs de sa condition pour qu’elle renvoie faux (false), tôt ou tard.

```
int age = 10;

// Tant que la valeur de age est strictement inférieure à 18
while (age < 18) {

    System.out.println("Vous n'êtes pas encore majeur (" + age + ")");

    /**
     * Incrémentation de age à chaque exécution du bloc de code
     * permettant ainsi d’atteindre petit à petit la valeur 18.
     */
    age++;
}

System.out.println("Vous êtes finalement majeur ! (" + age + ")");
```

```
// Résultat (affichage) dans la console
Vous n'êtes pas encore majeur (10)
Vous n'êtes pas encore majeur (11)
Vous n'êtes pas encore majeur (12)
Vous n'êtes pas encore majeur (13)
Vous n'êtes pas encore majeur (14)
Vous n'êtes pas encore majeur (15)
Vous n'êtes pas encore majeur (16)
Vous n'êtes pas encore majeur (17)
Vous êtes finalement majeur ! (18)
```

do : Extension d'une **boucle while** permettant d'exécuter une fois le **bloc de code** associé à cette dernière avant de commencer son travail d'**itération** en considérant sa **condition**.

Particulièrement utile lorsqu'on est certain d'effectuer une action au moins une fois, et potentiellement plusieurs fois selon la condition de la boucle.

*Exemple : Demander l'âge d'un utilisateur (do), et tant que (while) la valeur n'est pas conforme (entre 0 et 130 ans par exemple), la boucle redemandera à l'utilisateur d'entrer une valeur jusqu'à ce qu'elle soit valide (et donc que la condition renvoie faux (**false**)).*

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

do {

    System.out.print("Entrez votre age (0 - 130 ans) : ");

    // Cette instruction attend que l'utilisateur écrive dans la console

    /**
     * 1ère exécution (do)    : on suppose que l'utilisateur entre 200
     * 2ème exécution (while) : on suppose que l'utilisateur entre 25
    */
    age = scanner.nextInt();

    if (age < 0 || age > 130) {
         System.out.print("Age invalide : " + age + " ans.");
    }

} while (age < 0 || age > 130);

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

```
// Résultat (affichage) dans la console
Entrez votre age (0 - 130 ans) : 200
Age invalide : 200 ans.
Entrez votre age (0 - 130 ans) : 25
Age valide : 25 ans.
```

for : Boucle pouvant être lue “pour chaque”, avec une valeur de départ (initialisation), une condition d’arrêt et une opération de post-exécution permettant d’atteindre la condition d’arrêt.

C’est la forme de boucle la plus utilisée, offrant un contrôle précis sur le déroulement des itérations (itération), du début à la fin.

```
// Pour chaque valeur de i jusqu'à ce que i soit supérieur à 10.
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

```
// Résultat (affichage) dans la console
1
2
3
4
5
6
7
8
9
10
```

for-each : Boucle permettant d’itérer (parcourir) les éléments d’une liste (tableau ou collection) un par un.

Elle utilise le même mot-clé que la boucle for.

Elle simplifie l’itération sur une suite d’éléments, en évitant d’avoir à gérer les index manuellement comme avec une boucle for.

```
// Déclaration et instanciation d’une liste ["Java", "C", "Python"]
List<String> nameList = List.of("Java", "C", "Python");

for (String name : nameList) {
    System.out.println(name);
}
```

```
// Résultat (affichage) dans la console
"Java"
"C"
"Python"
```

break : Instruction permettant d'interrompre volontairement l'exécution d'une boucle ou d'un switch.

Les itérations qui auraient dû être exécutées sont alors annulées.

Très utile lorsqu'on a trouvé le résultat recherché et que poursuivre l'exécution de la boucle est alors devenu inutile.

```
// Boucle qui recherche le premier nombre divisible par 5
for (int i = 1; i <= 10; i++) {

    if (i % 5 == 0) {

        System.out.println("Premier nombre divisible par 5 : " + i);
        System.out.println("Interruption de la boucle avec break");

        break; // Sort de la boucle (annule les prochaines itérations)
    }
    else {
       System.out.println("Non divisible par 5 : " + i);
    }

}

System.out.println("Message qui s'affiche après la boucle");
```

```
// Résultat (affichage) dans la console
Non divisible par 5 : 1
Non divisible par 5 : 2
Non divisible par 5 : 3
Non divisible par 5 : 4
Premier nombre divisible par 5 : 5
Interruption de la boucle avec break
Message qui s'affiche après la boucle
```

continue : Instruction permettant de sauter (ignorer) l'exécution du bloc de code d'une boucle pour passer à l'itération suivante.

Très utile lorsqu'on veut ignorer un ou plusieurs cas spécifiques.

```java
// Boucle affichant des nombres excepté ceux qui sont divisibles par 5
for (int i = 1; i <= 10; i++) {

    if (i % 5 == 0) {

        System.out.println("Ignoré car divisible par 5 : " + i);

        continue; // Saute à la prochaine itération de la boucle
    }

    System.out.println(i);

}
```

```
// Résultat (affichage) dans la console
1
2
3
4
Ignoré car divisible par 5 : 5
6
7
8
9
Ignoré car divisible par 5 : 10
```

9. Imports et Paquets (Imports and Packages)

import : Permet d'intégrer le code d'un autre fichier du projet pour pouvoir l'utiliser. C'est une sorte de "copier-coller" du contenu du ou des fichiers cibles, ne tenant que sur une seule ligne.

Pour utiliser les éléments fournis par le JDK, il faut importer leurs fichiers sources avant de pouvoir utiliser leurs fonctionnalités.

Pour utiliser l'interface List<E> ou Stream<E>, vous devez importer au préalable leur code source grâce à import.

```
import java.util.List;
import java.util.stream.Stream;
```

package : Peut être vu comme un "dossier" en langage Java, contenant les différents fichiers Java (classe, interface, enum, etc.) d'une partie d'un projet. Un projet contient plusieurs packages, qui doivent former une arborescence logique et pertinente.

Un package nommé "voiture" ne devrait contenir que le code en lien direct avec la notion de voiture relative au projet.

Le mot-clé package et sa valeur (nom du package) sont positionnés à la première ligne d'un fichier Java pour indiquer son appartenance.

```
package com.jiraws.supercarproject.voiture;
```

Programmation Orientée Objet - POO 📦
[Object Oriented Programming - OOP]

La Programmation Orientée Objet (POO) est un paradigme de programmation dans lequel on va chercher à créer et à manipuler des “objets” pour les faire interagir entre eux. Ce paradigme vise à structurer le code de manière à faciliter le développement, la maintenance et la réutilisation des logiciels.

Le langage Java est principalement connu pour son appartenance à la famille des langages orientés objet, alors que ce n’est pourtant pas le seul paradigme sur lequel repose ce dernier. En effet, Java intègre également des aspects de la Programmation Impérative, qui inclut les fondamentaux communs à de nombreux langages, tels que les notions de variable, fonction, condition et boucle.

La notion d’objet n’est en fait qu’une approche structurante de la Programmation Impérative. Elle repose sur les mêmes principes fondamentaux, mais tend à les “positionner différemment” dans un programme afin d'en améliorer la qualité et la gestion.

La Programmation Orientée Objet permet donc de modéliser des concepts du monde réel sous la forme d’objets programmatiques, rendant le code plus intuitif et aligné sur la façon dont les humains (et donc, les développeurs) perçoivent le monde.

C’est le paradigme de programmation qui, depuis plus de 10 ans, est le plus prédominant dans le domaine du développement informatique.

On le retrouve dans les langages les plus populaires, dont Java, mais également C++, Python, JavaScript, PHP, C# ou encore Kotlin.

10. Les Quatre Piliers de la POO (The Four Pillars of OOP)

La Programmation Orientée Objet (POO) repose sur quatre grands principes: Encapsulation, Abstraction, Héritage et Polymorphisme. Ils nécessitent une certaine maîtrise de la programmation pour être compris et appliqués dans les règles de l'art.

Vous trouverez ci-dessous une définition succincte pour chacun d'entre eux, que votre expérience viendra consolider avec le temps.

Encapsulation : Mécanisme de limitation d'accès (en lecture et en écriture) aux membres (membre) d'une classe en utilisant la visibilité des éléments pour éviter les erreurs de manipulation.

Ce concept est essentiel à la POO, car il contribue à la sécurité, à la modularité et à la maintenabilité du code. Dans des cas d'utilisation plus poussés de l'Encapsulation, cette dernière peut également être un moyen d'abstraction, en cachant certains détails et en exposant seulement ce qui est nécessaire.

En pratique, bien que le principe d'Encapsulation soit relativement simple à mettre en place, il reste souvent incompris par les débutants. Il fait partie, avec le Polymorphisme, des principes de la POO les plus compliqués à concevoir pour les développeurs.

Note : Pas d'inquiétude à avoir si ce principe vous pose problème. Il est primordial pour respecter les fondamentaux de la POO, mais n'aura que peu d'impact sur votre apprentissage général de la programmation.

Abstraction : Conception visant à repousser l'implémentation des détails dans les classes (classe) qui seront réellement utilisées.

Dans une hiérarchie de classes, souvent représentée sous forme de pyramide, on essaiera toujours de faire en sorte que les niveaux supérieurs n'aient pour rôle que de "structurer" en utilisant des éléments abstraits (abstract) tels que les classes abstraites ou les interfaces. Les niveaux inférieurs, eux, devront implémenter le code (les détails) de cette structure qui sera véritablement utilisé dans un programme (on parle alors de classe "concrète").

Un bon exemple est la classe reine Object : son influence est omniprésente en Java car toutes les classes héritent d'une manière ou d'une autre de cette dernière, mais son utilisation directe est rare.

Héritage : Capacité d'un élément (classe ou interface) à transmettre ses membres (membre) à d'autres éléments, permettant ainsi la réutilisation et l'extension du code.

Si on souhaite représenter des voitures et des motos dans notre programme, il y a fort à parier que leurs classes auront des points communs, comme le fait d'avoir une marque, un nom de modèle, une couleur, ou encore un nombre de roues.

Sans l'héritage, ces classes auront des lignes de code explicitement similaires, à tel point que ces similarités pourront être copiées-collées d'une classe à l'autre. En cas de modification, il faudra alors répercuter les changements sur les deux classes, ou plus, si d'autres ont été ajoutées entre-temps.

Avec l'héritage, une classe générale regroupant les similarités (et qu'on pourra appeler "Véhicule") sera héritée par les classes plus spécifiques (Voiture et Moto). Ainsi, les éléments (membres) de la classe "mère" Véhicule seront automatiquement transférés dans les classes "filles" Voiture et Moto. Toutes modifications (ajouts / suppressions) seront également et automatiquement transférées aux classes filles.

Polymorphisme : Concept permettant à des éléments (objet) d'une même nature (classe) de prendre plusieurs formes et d'exécuter une même action (méthode) de manière différente.

Les voitures thermiques (à essence) et les voitures électriques sont deux types de voitures qui partagent énormément de similarités et ont un but commun, mais qu'on ne considère pas comme étant de même nature à cause de leurs différences fondamentales de motorisation.

Elles possèdent pourtant les mêmes fonctionnalités comme démarrer le moteur et accélérer, mais leur réalisation technique est totalement différente. Un moteur thermique injecte de l'essence pour provoquer une explosion, générant ainsi de la puissance transmise aux roues, tandis qu'un moteur électrique utilise le courant alternatif de l'électricité et l'électromagnétisme pour générer de la puissance.

En cela, on peut dire que les deux "véhicules" ont les mêmes comportements (méthode) de base, mais des exécutions (implémentation) différentes, ce qui relève du polymorphisme : un type d'objet similaire, répondant aux mêmes besoins, mais différemment.

11. Objets (Objects)

Objet : Instance composée d'attribut(s) et/ou de méthode(s), générée à partir d'une classe. *On construit un objet à partir d'un modèle d'objet (un plan de conception), à savoir une classe en Java.*

Instance : Exemplaire d'objet créé à partir d'une classe, aussi appelé une instance de cette classe. *Une instance, dans le contexte de la technologie et de l'informatique, se réfère à une unique occurrence d'une classe.*

this : Mot-clé permettant de faire référence à l'instance (objet) courante de la classe. Il permet de cibler un membre de l'instance.

Si on considère votre corps comme une instance (un exemplaire unique) de la classe "Humain", alors "this.prenom" réfèrera à votre prénom.

Une autre personne pourra également utiliser "this.prenom" pour faire référence à son propre prénom.

Deux personnes possédant le même prénom restent deux instances (exemplaires) bien distinctes. L'instruction "this.prenom" est similaire, mais elle ne fait pas référence à la même instance.

```
public class Voiture {

    private String marque;
    private String modele;

    /**
     * Le constructeur de la classe Voiture va créer une instance
     * (donc un exemplaire unique de cette classe, un objet)
     * et va cibler les deux attributs de cette instance grâce à "this"
     *
     * L'utilisation du mot-clé "this" permet de distinguer les éléments
     * * "this.marque" réfère à la variable d'instance
     * * "marque" réfère au paramètre du constructeur
     */
    public Voiture(String marque, String modele) {
        this.marque = marque;
        this.modele = modele;
    }

}
```

super : Mot-clé permettant de faire référence à un membre du parent (classe mère) direct d'un enfant (classe fille).

Si une classe mère déclare un attribut x alors sa classe fille pourra utiliser l'instruction "super.x" pour y accéder, même si cette dernière possède également un attribut du même nom.

Le mot-clé super est souvent utilisé dans les constructeurs pour faire appel au constructeur de la classe mère et ainsi éviter de réécrire deux fois le même code.

```
public class Parent {

    /**
     * Constructeur de la classe Parent
     */
    public Parent() {
        System.out.println("Constructeur de la classe Parent");
    }

}

public class Enfant extends Parent {

    /**
     * Constructeur de la classe Enfant
     */
    public Enfant() {

        super(); // Appel du constructeur de la classe Parent
        System.out.println("Constructeur de la classe Enfant");

    }

    // Entrée de programme exécutable
    public static void main(String[] args) {
        Enfant enfant = new Enfant();
    }

}
```

```
// Résultat (affichage) dans la console
Constructeur de la classe Parent
Constructeur de la classe Enfant
```

instanceof : Opérateur spécial permettant de vérifier si un objet est une instance d'une classe spécifique, d'une classe mère (héritage) ou d'une classe implémentant une interface.

Renvoie vrai (true) si l'instance est du type mentionné.

Particulièrement utile lorsqu'on fait usage de l'inférence de type.

Permet également de vérifier qu'un objet appartient à un type, avant de réaliser une conversion (Casting) qui pourrait entraîner une Exception dans le cas contraire.

```
var voiture = ... ; // On suppose qu'un objet est affecté

if (voiture instanceof Voiture) {

   System.out.println("Instance de la classe Voiture détectée !");

   // Conversion (casting) de l'objet en type Voiture
   Voiture voitureConvertie = (Voiture) voiture;

   System.out.println("La conversion a été effectuée avec succès.");

}
else {
   System.out.println("Il ne s'agit pas d'une Voiture.");
}
```

```
// Résultat (affichage) dans la console
Instance de la classe Voiture détectée !
La conversion a été effectuée avec succès.
```

12. Constructeurs (Constructors)

Instanciation : Processus de création d'un objet grâce à un constructeur. *Lors de l'instanciation d'un objet, Java réserve un emplacement dans la mémoire vive pour y stocker ses informations.*

Constructeur : Méthode spécifique, portant le nom de sa classe, sans type de retour et appelée pour la création d'une instance.

Si aucun constructeur n'est implémenté dans une classe, Java en fournit automatiquement un, sans paramètre et affectant des valeurs par défauts aux attributs de l'instance créée.

```
/**
 * Ci-dessous, le code d'une classe nommée Voiture sans aucun attribut
 * et possédant un constructeur permettant d'instancier des objets
 */
public class Voiture {

    /**
     * Si ce constructeur n'avait pas été implémenté, Java en aurait
     * automatiquement fourni un similaire, par défaut.
     */
    public Voiture() { }

}
```

new : Opérateur utilisé avec le constructeur d'une classe pour déclencher le processus d'instanciation, créant ainsi un objet.

```
/**
 * Cette instruction va demander à Java de créer un espace dans la
 * mémoire afin de stocker l'objet créé. La référence de cet
 * espace mémoire sera ensuite affectée à la variable "obj".
 */
Object obj = new Object();
```

Object : Classe Reine de la Programmation Orientée Objet en Java.

Toutes les classes héritent (héritage) de la classe Object en Java, même si ce n'est pas directement visible dans le code.

Elle fournit des méthodes (méthode) de base comme equals(), hashcode() ou encore toString(), qui seront détaillées dans la suite de ce guide.

```
/**
 * Même si l'héritage n'est pas visible avec le mot-clé extends Object
 * la classe Voiture hérite bel et bien de la classe Reine Object.
 */
public class Voiture {

}
```

Object.equals(Object obj) : Détermine l'égalité entre deux objets.

Renvoie vrai (true) si les objets sont égaux, faux (false) sinon.

Par défaut, cette méthode compare les références mémoires des deux objets pour vérifier qu'il s'agit bien du même élément (dans le même espace mémoire de la JVM).

Elle est souvent redéfinie (Override) dans les classes (classe) pour comparer les valeurs de leurs membres (membre) et ainsi faire une comparaison fonctionnelle plutôt que technique.

```
Voiture voiture1 = new Voiture("Ferrari", "F480");
Voiture voiture2 = new Voiture("Lamborghini", "Urus");

boolean egalite = voiture1.equals(voiture2); // false
```

Object.hashCode() : Retourne le code hash (Hachage) de l'objet.

Utilisé par les éléments de type Collection<E> basés sur le hash, comme HashSet<E>, permettant un accès aux éléments plus efficace.

```
Voiture voiture = new Voiture("Ferrari", "F480");

System.out.println(voiture.hashCode());
```

```
// Résultat (affichage) dans la console
1791741888
```

Object.toString() : Retourne une représentation de l'objet sous la forme d'une chaîne de caractères (String).

L'implémentation par défaut renvoie la forme: [nom_classe]@[hashCode].

Elle est souvent redéfinie (Override) par les classes (classe) pour avoir une représentation plus lisible des éléments dans un programme.

```
Voiture voiture = new Voiture("Ferrari", "F480");

System.out.println(voiture.toString());
```

```
// Résultat (affichage) dans la console
com.jiraws.supercarproject.voiture.Voiture@6acbcfc0
```

Object.getClass() : Retourne la classe de l'objet.

Permet de connaître précisément le type de l'objet concerné, en comparaison de l'opérateur instanceof qui peut renvoyer vrai (true) s'il y a un lien d'héritage entre l'objet et la classe indiquée.

```
public class Voiture extends Vehicule {

    public static void main(String[] args){

        Voiture voiture = new Voiture();

        // La classe Voiture hérite de Vehicule, la condition est valide
        if (voiture instanceof Vehicule) {
            System.out.println("Instance de Véhicule détectée !");
            System.out.println(voiture.getClass());
        }
        else {
            System.out.println("Il ne s'agit pas d'un Véhicule.");
        }

    }

}
```

```
// Résultat (affichage) dans la console
Instance de Véhicule détectée !
class com.jiraws.supercarproject.voiture.Voiture
```

13. Classes (Classes)

class (Classe) : Plan de conception permettant de définir la structure (membre) qu'auront les objets (objet) qu'elle créera.

La représentation d'une voiture en objet (POO) peut être la suivante:

- *Attributs (données) : Elle possède un nom de modèle, une marque, une couleur, ou encore un nombre de roues prédéterminé.*
- *Méthodes (comportements) : Elle a la capacité d'avancer, de freiner ou encore d'activer la climatisation.*

```
public class Voiture {

    /**
     * Attributs (Variables d'instance)
     */

    public String modèle;
    public String marque;
    public Couleur couleur;
    public int nombreRoues;

    /**
     * Méthodes (Comportements)
     */

    public void avancer() {
        // Code pour faire avancer la voiture
    }

    public void freiner() {
        // Code pour faire freiner la voiture
    }

    public void activerClimatisation() {
        // Code pour activer la climatisation
    }

    public void desactiverClimatisation() {
        // Code pour activer la climatisation
    }

}
```

Définition : Réfère à la spécification complète d'un élément.

- *La définition d'une classe inclut sa déclaration ainsi que l'implémentation de son corps : attribut(s), méthode(s), etc.*
- *La définition d'une méthode inclut son nom, son bloc de code, ses paramètres (paramètre), ainsi que son type de retour.*
- *La définition d'une variable inclut sa déclaration (type, nom) et éventuellement son initialisation.*

Membre : Terme référant à une composante de classe. Il peut cibler une variable (attribut), une fonction (méthode), une classe imbriquée ou encore une interface implémentée.

L'attribut "marque" de la classe Voiture, ou encore sa méthode lui permettant de démarrer le moteur, font partie de ses membres.

Attribut : Variable appartenant à une classe (static) ou à une instance (objet) créée à partir de cette dernière.

C'est la dénomination spécifique d'une variable en raison de son appartenance à une classe ou à un objet. On peut voir ce terme comme un synonyme de "variable", mais son usage dépend du contexte (là où se trouve la variable, dans une classe, ou pas).

Méthode : Fonction définie dans une classe ou dans une interface.

Aussi appelée "comportement" d'un objet. Comme pour le terme attribut, le terme "méthode" est un synonyme de "fonction" qui dépend du contexte (là où se trouve la fonction).

Override (Redéfinition) : Réécriture du corps d'une méthode pour remplacer celui reçu via l'héritage d'une classe ou d'une interface.

```
public class Voiture {

    private String marque;
    private String modele;

    // Réécriture d'un code plus adapté de la méthode Object.toString()
    @Override
    public String toString() {
        return "Modèle " + this.modele + " de la marque " + this.marque;
    }

}
```

Overload (Surcharge) : Implémentation multiple d'une méthode possédant le même nom mais ayant un type de retour et/ou des paramètres (paramètre) différents.

Le nom des méthodes est important car il permet de structurer logiquement le code de notre application.

La surcharge est un bon exemple mettant en évidence l'importance de bien nommer les éléments:

- *Cas 1 (Bonne pratique) : On surcharge correctement une méthode pour additionner des nombres, et ainsi, on facilite le travail des autres développeurs lorsqu'ils liront ce code.*

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

// Méthode surchargée pour additionner trois entiers
public int add(int a, int b, int c) {
    return a + b + c;
}
```

- *Cas 2 (Mauvaise pratique) : On ne surcharge pas la méthode, en préférant plutôt la création d'une autre distincte, alors qu'elles ont un objectif et un sens très similaires.*

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

// Méthode pour additionner trois entiers
public int additionDeTroisNombres(int a, int b, int c) {
    return a + b + c;
}
```

Classe Utilitaire : Une classe déclarant un ensemble de méthodes (méthode) statiques (static) offrant des fonctionnalités communes ou réutilisables, sans nécessiter l'instanciation d'objets.

On peut voir une classe utilitaire comme une bibliothèque.

14. Héritage (Inheritance)

extends (Extension) : Mot-clé permettant de lier par l'héritage une classe fille à sa classe mère.

Les membres (membre) de la classe mère sont alors transmis à la structure de la classe fille.

En Java, une classe ne peut hériter que d'une seule autre classe.

```
public class Voiture extends Vehicule { ... }
```

abstract (Abstraite) : Mot-clé permettant de rendre une classe ou une méthode abstraite.

- *Une classe abstraite est une classe qui perd la capacité de créer (instanciation) des objets grâce à un constructeur.*

  Elle conserve néanmoins son rôle de modèle d'objet.

```
public abstract class Vehicule { ... }
```

- *Une méthode abstraite est une méthode déclarée, mais sans corps de méthode (bloc de code) qui devra être redéfini (Override) par les classes filles.*

  Une classe contenant une méthode abstraite doit également être déclarée comme abstraite.

```
public abstract class Vehicule {

    /**
     * Chaque type de véhicule (voiture, moto, avion, bateau) démarre
     * d'une façon bien spécifique.
     *
     * Il est donc cohérent de rendre la méthode demarrer() abstraite
     * pour que les futures classes filles (Voiture, Moto, Avion, etc.)
     * soient contraintes de redéfinir leurs codes en fonction de
     * leurs spécificités de fonctionnement.
     */
    public abstract void demarrer();

}
```

sealed (Scellée) : Permet de rendre une classe ou une interface “scellée”, c’est à dire qu’elle ne peut être héritée (héritage) que par les éléments définis à l’avance avec le mot-clé permits.

C’est une approche visant à mieux contrôler l’héritage dans un projet, offrant ainsi aux développeurs plus de sécurité quant à l’utilisation d’une classe ou d’une interface.

permits : Permet de spécifier explicitement les classes (classe) et les interfaces (interface) qui peuvent étendre un type sealed.

Les classes qui hériteront devront être soit final soit non-sealed.

```
public sealed class Vehicule permits Voiture, Camion { ... }

/**
 * La classe Voiture n’aura (à priori) pas de classe fille dans le
 * projet, on la rend donc non-héritable grâce au modificateur final
 */
final class Voiture extends Vehicule { ... }
```

non-sealed : Permet à une classe ou à une interface héritant (héritage) un type sealed, de devenir un type héritable à nouveau.

Offre une flexibilité d'héritage au sein d'une hiérarchie scellée, permettant à certaines branches de cette hiérarchie de rester ouvertes à l'extension. Son utilisation doit être faite avec parcimonie, car elle va à l’encontre du principe de sceller des éléments.

```
public sealed class Vehicule permits Camion { ... }

/**
 * La classe Camion possèdera au moins une classe fille dans le
 * projet, on la rend donc à nouveau héritable grâce à non-sealed
 */
non-sealed class Camion extends Vehicule { ... }

/**
 * La classe CamionElectrique hérite sans contrainte de la classe Camion
 */
public class CamionElectrique extends Camion { ... }
```

15. Interface (Interface)

interface (Interface) : Une “sorte” de classe ne pouvant contenir aucun attribut, ne pouvant pas créer d’objet, et ne déclarant que des méthodes abstraites (abstract) qui seront redéfinies (Override) par les classes qui les implémenteront (implements).

On considère plus généralement une interface comme un “contrat” d’utilisation : cette dernière déclare des méthodes abstraites, forçant ainsi les classes qui les implémenteront à les redéfinir avec le code correspondant à leurs spécificités.

Une interface va permettre de regrouper les méthodes propres à un comportement, mais qui n’a pas de lien direct avec la représentation qu’on se fait d’un objet (voir l’exemple ci-dessous).

```
/**
 * Une voiture et une maison n’ont rien à voir conceptuellement
 * mais peuvent partager une fonctionnalité commune d’air climatisée.
 *
 * Les deux classes Voiture et Maison partageront alors une interface
 * nommée “Climatiseur”, déclarant les méthodes nécessaires à ce besoin.
 */
public interface Climatiseur {

    public void activerClimatisation();
    public void desactiverClimatisation();
    public void reglerTemperature(int temperature);

   /**
    * Une méthode pourra proposer un code par défaut, assurant ainsi
    * que ce dernier soit utilisé automatiquement si la classe
    * implémentant l’interface ne l’a pas redéfinit.
    */
    public default void soufflerAir() {
         // Code de l’implémentation par défaut
    }

}
```

Une interface peut en étendre (héritage) une autre, comme une classe peut le faire avec une autre classe.

```
public interface InterfaceA extends InterfaceB { ... }
```

implements : Mot-clé permettant à une classe d'implémenter une (ou plusieurs) interface(s).

Une classe déclarant l'implémentation d'une interface est alors contrainte de redéfinir (Override) ses méthodes déclarées. Si elle ne le fait pas (entièrement), une erreur de compilation aura lieu, empêchant ainsi le programme de pouvoir démarrer.

Une classe abstraite (abstract) déclarant l'implémentation d'une interface pourra s'abstenir de redéfinir (Override) les méthodes importées, en laissant ce travail à ses classes filles.

Contrairement à l'héritage de classe qui est unique en Java, une classe pourra implémenter plusieurs interfaces si nécessaire.

```
public interface Climatiseur {

    public void activerClimatisation();
    public void desactiverClimatisation();
    public void reglerTemperature(int temperature);

}

public class Voiture implements Climatiseur {

    @Override
    public void activerClimatisation() {
          // Implémentation du code spécifique...
    }

    @Override
    public void desactiverClimatisation() {
          // Implémentation du code spécifique...
    }

    @Override
    public void reglerTemperature(int temperature) {
          // Implémentation du code spécifique...
    }

}
```

16. Énumérations (Enumerations)

enum (Énumération) : Type de classe spécifiquement conçu pour représenter une suite d'éléments fixes (constante).

Visuellement différente d'une classe standard, une énumération offre néanmoins les mêmes fonctionnalités à l'exception de l'héritage.

Une enum ne peut pas utiliser l'héritage, car en interne elle est implicitement déclarée comme final et étend la classe java.lang.Enum même si ce n'est pas visible dans le code (comme pour l'héritage universel de la classe Object).

Deux bons exemples d'application des énumérations sont les jours de la semaine (de lundi à dimanche) et les directions cardinales (nord, sud, est, ouest). Ce sont des "suites" (listes) d'éléments fixes, qui ne vont à priori pas changer dans votre application une fois implémentés.

```
public enum PointCardinal {

    /**
     * Une suite de constantes ("NORD", "EST", "SUD", "OUEST")
     * avec un libellé ("N", "E", "S", "O") pour chacune d'entre elles.
     */
    NORD("N"),
    EST("E"),
    SUD("S"),
    OUEST("O");

    // Attribut pour le libellé court ("N", "E", "S", "O")
    private final String libelleCourt;

    // Constructeur privé pour l'énumération
    PointCardinal(String libelle) {
        this.libelleCourt = libelle;
    }

    // Méthode publique (Getter) pour accéder au libellé court
    public String getLibelleCourt() {
        return libelleCourt;
    }

}
```

17. Modificateurs (Modifiers)

Modificateur : Catégorie de mot-clés qui changent la signification d'une définition de classe, d'interface, de méthode ou de variable.

*Parmi les modificateurs, on retrouve **abstract** et **final** que nous avons croisé précédemment dans ce guide, ou encore ceux permettant de modifier la **visibilité**: **public**, **private**, **protected**.*

*Le mot-clé **static** est un bon exemple de modificateur, car son simple ajout change totalement l'appartenance d'un **attribut** (ou d'une **méthode**) à une classe ou à une **instance** de cette dernière.*

```
public class Voiture {

    public static int nombreInstanceVoiture; // Attribut de classe

    public int kilometrage; // Attribut d'instance

}
```

Visibilité : Mécanisme déterminant l'accessibilité d'un élément (classe, interface, attribut ou méthode) au sein d'un programme.

Elle joue un rôle direct dans le principe d'Encapsulation, car elle permet de limiter l'accès en lecture et en écriture des éléments.

*On change la visibilité d'un élément grâce à l'utilisation d'un **modificateur** de visibilité comme **public**, **private** ou **protected**.*

public (public) : L'élément est visible dans tout le projet.

*Déconseillé par défaut pour certains éléments : un **attribut** doit rester privé, et une **méthode** ne doit être publique que s'il y a un réel intérêt à ce que son accès par l'extérieur soit possible.*

```
public class Voiture {

    // Attribut public qui sera accessible de partout dans le projet
    public int kilometrage;

}
```

private (privé) : L'élément n'est visible que par les autres membres (membre) de sa classe ou de son interface.

Équivalent du "Secret Défense" : seuls les membres d'une même unité ne peuvent connaître cet élément privé. Pour le reste du projet, l'élément privé n'existe pas, il n'est ni visible, ni accessible.

Dans le cas d'une interface, la méthode déclarée privée devra posséder un corps (bloc de code) car elle ne sera pas transmise à la classe implémentant l'interface à cause de sa visibilité privée. Elle servira alors de méthode interne, utilisable par les autres méthodes.

```
public class Voiture {

    /**
     * L'attribut (variable d'instance) "kilometrage" ne sera visible,
     * et donc accessible et modifiable, que par les méthodes de
     * la classe Voiture et sera inaccessible pour le reste.
     */
    private int kilometrage;

}
```

protected (protégé) : L'élément est visible par les membres (membre) de sa classe et de ses classes filles (héritage), ainsi que par les autres éléments présents dans le même package.

On peut considérer cette visibilité comme un entre-deux de public et private. Il est cependant non-applicable dans une interface.

```
package com.jiraws.supercarproject.voiture;

public class Voiture {

    /**
     * L'attribut (variable d'instance) "kilometrage" sera visible,
     * et donc accessible et modifiable, par les méthodes de
     * la classe Voiture, par les classes filles de la classe Voiture
     * et par les éléments du package com.jiraws.supercarproject.voiture
     */
    protected int kilometrage;

}
```

default (package-private) : L'élément est visible par les membres (membre) de sa classe ou de son interface et par les éléments présents dans le même package.

C'est une visibilité automatiquement appliquée à un élément lorsqu'aucun modificateur n'est défini par le développeur.

À l'instar de la visibilité protected, la visibilité par défaut n'inclut pas l'héritage, sauf si la classe fille concernée se trouve dans le même package que la classe déclarant l'élément.

```
public class Voiture {

    /**
     * L'attribut (variable d'instance) "kilometrage" sera visible,
     * et donc accessible et modifiable, par les méthodes de sa classe
     * et par les éléments du package com.jiraws.supercarproject.voiture
     */
    int kilometrage;

}
```

Getter (Accesseur) : Méthode permettant d'accéder (de lire) la valeur d'un attribut privé (private) d'une classe.

Une méthode "getter" ne diffère en rien techniquement d'une autre méthode standard pour le langage Java, il ne s'agit que d'une bonne pratique respectant le grand principe d'Encapsulation en POO.

Bien qu'une méthode Getter soit souvent du code boilerplate, elle offre néanmoins la possibilité d'ajouter une implémentation spécifique à l'accès d'un attribut, comme par exemple en ne renvoyant qu'une copie de la valeur de ce dernier, pour éviter une modification involontaire en transmettant une référence.

```
public class Voiture {

    // Invisible de l'extérieur, mais accessible grâce à son Getter
    private String marque;

    public String getMarque() {
        return this.marque;
    }

}
```

Setter (Mutateur) : Méthode permettant de modifier (d'écrire) la valeur d'un attribut privé (private) d'une classe.

Une méthode "setter" ne diffère en rien techniquement d'une autre méthode standard pour le langage Java, il ne s'agit que d'une bonne pratique respectant le grand principe d'Encapsulation en POO.

Même si une méthode Setter est souvent du code boilerplate, elle offre néanmoins la possibilité d'ajouter une implémentation spécifique à la modification de la valeur d'un attribut, comme par exemple en empêchant certains changements qui pourrait fausser une information.

```
public class Voiture {

    private int kilometrage;

    /**
     * Un exemple de méthode Setter protégeant son attribut
     * d'une mauvaise modification, ou du moins, d'une modification
     * qui ne devrait pas avoir lieu.
     */
    public void setKilometrage(int nouveauKilometrage) {

        // Le kilométrage d'une voiture ne peut qu'augmenter
        if(this.kilometrage < nouveauKilometrage) {

            // La modification est ignorée au profit d'une alerte
            System.out.println("Réduction du kilométrage interdite.");

        }
        else {
            this.kilometrage = nouveauKilometrage;
        }

    }

}
```

static : Modificateur impliquant qu'un membre appartient à une classe plutôt qu'à une instance.

Un attribut ou une méthode ayant le modificateur static seront partagés par toutes les instances de leur classe, sans pour autant leur appartenir directement.

On fait ainsi la distinction entre:

- *Un attribut de classe et un attribut d'instance*
- *Une méthode de classe et une méthode d'instance*

```
public class Voiture {

    /**
     * Variable de classe partagée par chaque future instance
     */
    private static long nombreInstanceVoiture = 0;

    /**
     * Variables d'instance qui seront uniques et propres à chaque
     * future instance créée
     */
    private String marque;
    private int kilometrage;

}
```

native : Modificateur impliquant qu'une méthode est implémentée en code natif en dehors du programme Java.

C'est-à-dire du code écrit dans un autre langage de programmation comme C ou C++. Souvent utilisé pour accéder à des bibliothèques (bibliothèque) système ou à des fonctions spécifiques au matériel qui ne sont pas disponibles en Java.

Note: Vous ne croiserez que (très) rarement (voire jamais) ce modificateur, mais il est quand même bon de connaître le grand concept autour de ce dernier au cas où cela arriverait.

18. Immutabilité (Immutability)

Immuable (Immutabilité) : Un objet est dit immuable lorsque son état interne (valeurs de ses attributs (attribut)) ne peut pas être modifié après sa création (instanciation).

Un type primitif est par nature immuable: lorsqu'on modifie sa valeur, en interne du langage Java il ne s'agit pas d'une modification de la valeur en elle-même, mais plutôt de la création d'une nouvelle valeur qui est affectée à la variable.

Un type référence n'est immuable que si son état (valeurs de ses attributs) ne peut pas être modifié après sa création, impliquant donc que l'ensemble de ses champs ait le modificateur final.

L'inverse de l'immutabilité est la mutabilité, à savoir le fait d'être mutable (et donc "changeable" / "modifiable").

```java
/**
 * Cette implémentation de la classe Voiture permet de créer des
 * objets immuables, car tous ses attributs sont déclarés finaux (final)
 * et par conséquent ne pourront pas être modifiés après
 * la création d'une instance de la classe.
 */
public final class Voiture {

    private final String marque;
    private final String modele;

    public Voiture(String marque, String modele) {
        this.marque = marque;
        this.modele = modele;
    }

    public final String getMarque() {
        return this.marque;
    }

    public final String getModele() {
        return this.modele;
    }

}
```

final : Modificateur impliquant que l'élément concerné ne puisse plus être modifié après sa création.

- *Pour un **type primitif**: sa valeur (primitive) ne pourra pas être changée après son **initialisation**.*
- *Pour un **type référence**: sa valeur (référence mémoire) ne pourra pas être changée après son **instanciation**, mais son état (valeurs de ses attributs) peut l'être si ce dernier n'est pas **immuable**.*
- *Pour une **méthode**: son corps ne pourra pas être redéfini (**Override**) par une classe qui en héritera, assurant ainsi un comportement unique dans toute l'arborescence (**héritage**).*
- *Pour une **classe**: elle devient non héritable et ne peut donc pas avoir de classe fille, elle est la dernière de sa lignée.*

```
/**
 * La classe Voiture possède le modificateur final et ne pourra donc
 * pas avoir de classe fille (extends Voiture)
 */
public final class Voiture {

    /**
     * Attribut primitif dont la valeur (primitive)
     * ne peut pas être changée
     */
    private final int anneeFabrication;

    /**
     * Attribut référence dont la valeur (référence mémoire de l'objet)
     * ne peut pas être changée
     */
    private final Moteur moteur;

    public Voiture(int anneeFabrication, Moteur moteur) {
        this.anneeFabrication = anneeFabrication;
        this.moteur = moteur;
    }

    // Méthode ne pouvant être redéfinie (Override)
    public final String getMarque() {
        return this.marque;
    }

}
```

record : Type de classe spécifiquement conçu pour simplifier l'implémentation d'objets immuables (Immutabilité) en générant automatiquement le code de base (boilerplate) dans la classe.

Bien que le mot-clé class ne soit pas utilisé, le résultat final sera bel et bien une classe immuable comme si vous l'aviez fait vous-même.

Un record va fournir, à la place du développeur, les implémentations des méthodes de base comme le getter et le setter de chaque attribut ainsi que equals(), hashCode(), et toString().

C'est une solution qui n'a pour but que de simplifier l'écriture du code, comme par exemple avec l'auto-boxing et l'auto-unboxing.

```
public record Coordonnees(int x, int y) { }

/**
 * Le (simple) code ci-dessus est équivalent au (long) code ci-dessous
 */

public class Coordonnees {

    private final int x;
    private final int y;

    public Coordonnees(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return this.x; }
    public int getY() { return this.y; }

    public int setX(int x) { this.x = x; }
    public int setY(int y) { this.y = y; }

    @Override
    public boolean equals(Coordonnees c) { // Code du equals }

    @Override
    public int hashCode() { // Code du hashCode }

    @Override
    public String toString() { // Code du toString }

}
```

19. Types Enveloppes (Wrappers)

Wrapper (Enveloppe) : Classe contenant (enveloppant) un type primitif et offrant diverses méthodes (méthode) propres à ce dernier afin d'améliorer sa manipulation.

Les types enveloppes consomment plus de mémoire que les types primitifs. Il faut donc les utiliser seulement si le type primitif concerné ne répond pas correctement à notre besoin.

Byte : Classe Enveloppe (Wrapper) du type primitif byte.

```
Byte distanceSalleDeSport = 2;
```

Short : Classe Enveloppe (Wrapper) du type primitif short.

```
Short rayonPlaneteTerre = 6371;
```

Integer : Classe Enveloppe (Wrapper) du type primitif int.

```
Integer distanceTerreLune = 384400;
```

Long : Classe Enveloppe (Wrapper) du type primitif long.

```
Long distanceSoleilNeptune = 4500000000L;
```

Float : Classe Enveloppe (Wrapper) du type primitif float.

```
Float tailleMoyenHumain = 1.75F;
```

Double : Classe Enveloppe (Wrapper) du type primitif double.

```
Double approximationPi = 3.141592653589793;
```

Character : Classe Enveloppe (Wrapper) du type primitif char.

```
Character note = 'A';
```

Boolean : Classe Enveloppe (Wrapper) du type primitif boolean.

```
Boolean like = true;
Boolean dislike = false;
```

L'intérêt du type enveloppe Boolean est important à retenir, car ce dernier étant un objet (et donc un type référence), il peut être affecté à la valeur spéciale null, offrant ainsi des possibilités ternaires (3) plutôt que binaires (2) :

- *L'utilisateur a-t-il validé son adresse email ?*
  - *true (vrai) : l'adresse email a été validée.*
  - *false (faux) : l'adresse email n'est pas encore validée.*
  - *null (rien) : l'utilisateur n'a pas fourni d'adresse email.*

Auto-boxing : Processus automatique par lequel Java convertit un type primitif en sa classe enveloppe (Wrapper) correspondante lorsqu'un objet est requis mais qu'une valeur primitive est fournie.

Fonctionnalité ajoutée au langage Java pour faciliter l'écriture du code. Il faut cependant garder en tête que ce processus de conversion automatique est coûteux en termes de performance.

```
Integer age = new Integer(25); // Approche dépréciée

Integer age = 25; // Usage de l'auto-boxing (primitif -> enveloppe)
```

Auto-unboxing : Processus automatique inverse de l'auto-boxing par lequel Java convertit un type enveloppe (Wrapper) en son type primitif correspondant lorsqu'une valeur primitive est requise mais qu'un objet est fourni.

Fonctionnalité ajoutée au langage Java pour faciliter l'écriture du code. Comme pour l'auto-boxing, il faut garder en tête que ce processus est coûteux en termes de performance.

```
Integer ageEnveloppe = 25; // Auto-boxing (primitif -> wrapper)

int agePrimitif = ageEnveloppe; // Auto-unboxing (wrapper -> primitif)
```

Casting (Conversion) : Conversion explicite d'un type vers un autre.

C'est une opération qui présente des risques et que les développeurs doivent considérer avant de l'utiliser. L'erreur la plus courante est la perte d'une partie des informations, comme par exemple en convertissant un float (nombre à virgule) en int (nombre entier), ce qui entraîne la perte des valeurs après la virgule (10.55 devient 10).

Pour un type primitif : on peut faire une conversion (casting) pour éviter la perte de données lors d'une opération arithmétique.

```
/**
 * On s'apprête à diviser la valeur des deux variables ci-dessous.
 * Le résultat attendu de 5 / 2 est donc 2.5
 */
int a = 5;
int b = 2;

// Division sans faire explicitement la conversion du résultat
float c = a / b;

// Division en faisant explicitement la conversion du résultat
float d = (float) a / b;

/**
 * La différence entre les deux résultats s'explique par l'ordre
 * d'exécution des instructions:
 * 1. Si une conversion est demandée, Java convertit les opérandes
 * 2. Java exécute l'opération arithmétique (a / b)
 *
 * Sans cette étape de conversion (casting), Java va réaliser
 * l'opération arithmétique sur deux entiers (int) et par
 * conséquent va tronquer le résultat:
 * * 2.5 devient 2 (int) puis devient 2.0 pour matcher le type float
 *
 * Avec cette étape de conversion (casting), Java va convertir
 * les opérandes (en float) puis réaliser l'opération arithmétique
 * manipulant ainsi des types compatibles avec les nombres à virgule.
 */
System.out.println("Résultat sans conversion : "  + c);
System.out.println("Résultat avec conversion : "  + d);
```

```
// Résultat (affichage) dans la console
Résultat sans conversion : 2.0
Résultat avec conversion : 2.5
```

Pour un type référence : on peut faire une conversion (casting) d'un type vers un autre, si les deux éléments partagent une hiérarchie de classe en commun. Requiert une vérification au préalable avec l'opérateur instanceof pour s'assurer de la réussite de la conversion, sans quoi une ClassCastException sera levée et arrêtera le programme.

```
public class Animal {

    public void manger() {
        System.out.println("Cet animal mange.");
    }

}

public class Chien extends Animal {

    public void aboyer() {
        System.out.println("Le chien aboie.");
    }

    public static void main(String[] args) {

        // Instanciation d'un Chien dans une variable Animal
        Animal animal = new Chien();

        // La méthode manger() est accessible par le type Animal
        animal.manger();

        // La méthode aboyer() n'est pas accessible par le type Animal
        animal.aboyer(); // Erreur de compilation (ligne à supprimer)

        /**
         * Vérification de l'appartenance à une hiérarchie commune.
         * La condition est valide car Chien extends Animal.
         */
        if (animal instanceof Chien) {

            // Conversion explicite du type Animal en type Chien
            Chien chien = (Chien) animal;

            // La méthode Chien.aboyer() est désormais accessible
            chien.aboyer();
        }
    }
}
```

20. Manipulation de Texte (String)

String : Classe permettant de représenter et de manipuler du texte.

Techniquement appelées des "chaînes de caractères" en raison de leur composition : des caractères (char) les uns à la suite des autres.

Rappel important : En Java, la valeur d'une chaîne de caractères (String) est entre " " (guillemets) et la valeur d'un caractère (char) est entre ' ' (apostrophes).

```
/**
 * Pour faciliter l'écriture du code en évitant d'avoir à faire appel
 * systématiquement à un constructeur, le langage Java permet une
 * instanciation simplifiée des chaînes de caractères (String)
 */

String nomFormation = "JavaCore";

/**
 * On peut voir la valeur "JavaCore" comme un tableau d'éléments char
 * * ['J', 'a', 'v', 'a', 'C', 'o', 'r', 'e']
 * Soit en valeurs décimales (table de caractères Unicode)
 * * [74 , 97 , 118, 97 , 67 , 111, 114, 101]
 */
```

Concaténation : Action de concaténer, à savoir le fait de joindre des chaînes de caractères (String) pour en former une nouvelle.

```
String nomFormation = "JavaCore";
String nomFormateur = "JirAWS";

// Concaténation des trois chaînes de caractères (String)
String concatenation = nomFormation + " par " + nomFormateur;

System.out.println(concatenation);
```

```
// Résultat (affichage) dans la console
JavaCore par JirAWS
```

Regex (Expression Régulière) : Technique (puissante) et flexible de recherche et de manipulation de chaînes de caractères (String) basée sur des motifs permettant de spécifier des règles d'identification des séquences de caractères.

Attention : Les expressions régulières ont une complexité cognitive très élevée. Même un développeur expérimenté aura besoin d'un temps d'analyse pour bien appréhender une regex complexe.

```
// Adresse mail à vérifier
String email = "contact@jiraws.com";

/**
 * Expression régulière basique pour la vérification de mail.
 * Elle correspond à un format d'adresse email basique : xxx@yyy.zzz
 */
String regex = "[\\w.-]+@\\w+\\.\\w+";

// Instanciation d'un compilateur de regex
Pattern pattern = Pattern.compile(regex);

/**
 * Instanciation d'un matcheur (comparateur) qui fera la vérification
 * entre la regex et l'adresse email fournies
 */
Matcher matcher = pattern.matcher(email);

/**
 * Cas où le matcheur trouve bien une correspondance
 * entre la regex et l'adresse email (String).
 */
if (matcher.matches()) {

    System.out.println(email + " est une adresse email valide.");

} else {

    System.out.println(email + " n'est pas une adresse email valide.");

}
```

```
// Résultat (affichage) dans la console
contact@jiraws.com est une adresse email valide
```

Text Blocks (Blocs de Texte) : Permet de définir des chaînes de caractères (String) sur plusieurs lignes, de manière lisible.

Particulièrement utile pour l'écriture de contenus spécifiques :

- *HTML : HyperText Markup Language*
- *JSON : JavaScript Object Notation*
- *SQL : Structured Query Language*

```
String basicHTML = """
    <html>
        <body>
            <p>Hello, World!</p>
        </body>
    </html>
    """;

System.out.println("Bloc de texte : ");
System.out.println(basicHTML);
```

```
// Résultat (affichage) dans la console
Bloc de texte :
<html>
    <body>
        <p>Hello, World!</p>
    </body>
</html>
```

String.length() : Retourne la longueur de la chaîne de caractères.

Souvent utilisé en la combinant avec une boucle for et la méthode String.charAt(int index) pour itérer sur chaque caractère.

```
String nomFormation = "JavaCore"; // Un texte composé de 8 caractères

System.out.println("Taille du texte : " + nomFormation.length());
```

```
// Résultat (affichage) dans la console
Taille du texte : 8
```

String.charAt(int index) : Renvoie le caractère (char) se trouvant à l'index (position) en argument dans la chaîne de caractères.

```
String nomFormation = "JavaCore";

System.out.println("Premier   caractère : " + nomFormation.charAt(0));
System.out.println("Second    caractère : " + nomFormation.charAt(1));
System.out.println("Troisième caractère : " + nomFormation.charAt(2));
System.out.println("Quatrième caractère : " + nomFormation.charAt(3));
System.out.println("Cinquième caractère : " + nomFormation.charAt(4));
System.out.println("Sixième   caractère : " + nomFormation.charAt(5));
System.out.println("Septième  caractère : " + nomFormation.charAt(6));
System.out.println("Huitième  caractère : " + nomFormation.charAt(7));
```

```
// Résultat (affichage) dans la console
Premier   caractère : J
Second    caractère : a
Troisième caractère : v
Quatrième caractère : a
Cinquième caractère : C
Sixième   caractère : o
Septième  caractère : r
Huitième  caractère : e
```

String.substring(int beginIndex, int endIndex) : Renvoie la partie de la chaîne de caractères entre les index (position) en argument.

```
String nomFormation = "JavaCore";

String java = nomFormation.substring(0, 4);

System.out.println("Nom découpé : " + java);
```

```
// Résultat (affichage) dans la console
Nom découpé : "Java"
```

String.split(String regex) : Sépare la chaîne de caractères pour chaque correspondance de l'expression en argument, et renvoie un tableau contenant les éléments séparés.

```
String nomFormation = "Java_Core";

String[] nomFormationSepare = nomFormation.split("_");

System.out.println("Nom séparé par '_' : " + nomFormationSepare);
```

```
// Résultat (affichage) dans la console
Nom séparé par '_' : ["Java", "Core"]
```

String.trim() : Renvoie une copie de la chaîne de caractères, sans les espaces vides au début et à la fin (s'il y en a).

Très pratique pour corriger une potentielle faute de frappe lorsqu'un utilisateur entre une information dans un formulaire.

```
String nomFormation = "   JavaCore   ";

System.out.println("Nom après trim : " + nomFormation.trim());
```

```
// Résultat (affichage) dans la console
Nom après trim : "JavaCore"
```

String.replaceAll(String target, String replacement) : Renvoie une chaîne de caractères, avec les correspondances de l'argument target remplacées par la valeur de l'argument remplacement.

```
String nomFormation = "JavaCore";

String futureFormation = nomFormation.replaceAll("Java", "Spring");

System.out.println("Nom de la future formation : " + futureFormation);
```

```
// Résultat (affichage) dans la console
Nom de la future formation : "SpringCore"
```

String.startsWith(String prefix) : Renvoie vrai (true) si la chaîne de caractères commence par le prefix en argument.

```
String nomFormation = "JavaCore";

boolean commenceParJava = nomFormation.startsWith("Java");
boolean commenceParCore = nomFormation.startsWith("Core");

System.out.println("Le nom commence par 'Java' : " + commenceParJava);
System.out.println("Le nom commence par 'Core' : " + commenceParCore);
```

```
// Résultat (affichage) dans la console
Le nom commence par 'Java' : true
Le nom commence par 'Core' : false
```

String.endsWith(String suffix) : Renvoie vrai (true) si la chaîne de caractères se termine par le suffix en argument.

```
String nomFormation = "JavaCore";

boolean termineParCore = nomFormation.endsWith("Core");
boolean termineParJava = nomFormation.endsWith("Java");

System.out.println("Le nom termine par 'Core' : " + termineParCore);
System.out.println("Le nom termine par 'Java' : " + termineParJava);
```

```
// Résultat (affichage) dans la console
Le nom termine par 'Core' : true
Le nom termine par 'Java' : false
```

String.toLowerCase() : Renvoie une copie de la chaîne de caractères en minuscule.

```
String nomFormation = "JavaCore";

String nomEnMinuscule = nomFormation.toLowerCase();

System.out.println("Nom en minuscule : " + nomEnMinuscule);
```

```
// Résultat (affichage) dans la console
Nom en minuscule : "javacore"
```

String.toUpperCase() : Renvoie une copie de la chaîne de caractères en majuscule.

```
String nomFormation = "JavaCore";

String nomEnMajuscule = nomFormation.toUpperCase();

System.out.println("Nom en majuscule : " + nomEnMajuscule);
```

```
// Résultat (affichage) dans la console
Nom en majuscule : "JAVACORE"
```

String.indexOf(String target) : Renvoie l'index (position) de la première occurrence de la chaîne spécifiée par le paramètre target.

```
String nomFormation = "JavaCore";

int positionA    = nomFormation.indexOf("a");
int positionCore = nomFormation.indexOf("Core");

System.out.println("Position de la lettre 'a'   : " + positionA);
System.out.println("Position du début de 'Core' : " + positionCore);
```

```
// Résultat (affichage) dans la console
Position de la lettre 'a'   : 1
Position du début de 'Core' : 4
```

String.lastIndexOf(String target) : Renvoie l'index (position) de la dernière occurrence de la chaîne spécifiée par le paramètre target.

```
String nomFormation = "JavaCore";

int positionA    = nomFormation.lastIndexOf("a");
int positionCore = nomFormation.lastIndexOf("Java");

System.out.println("Dernière position de la lettre 'a' : " + positionA);
System.out.println("Dernière position de 'Core' : " + positionCore);
```

```
// Résultat (affichage) dans la console
Dernière position de la lettre 'a' : 3
Dernière position de 'Core' : 4
```

String.equals(String str) : Renvoie vrai (true) si les deux chaînes de caractères sont identiques, caractère par caractère.

La classe String redéfinit (Override) la méthode héritée de la classe Mère / Reine (Object) Object.equals(Object obj), pour proposer une implémentation adaptée aux chaînes de caractères.

```
System.out.println("JavaCore".equals("JavaCore")); // true
System.out.println("JavaCore".equals("javacore")); // false
System.out.println("JavaCore".equals("abcdefgh")); // false
```

```
// Résultat (affichage) dans la console
true
false
false
```

21. Manipulation Numérique (Number)

BigInteger : Classe permettant de manipuler des nombres entiers de très grande taille, dépassant les limites du type primitif long.

Utilisé par des applications nécessitant une précision mathématique élevée, comme dans la cryptographie, le calcul de grands nombres premiers, ou pour des systèmes financiers.

```
BigInteger grandEntier1 = new BigInteger("12345678901234567890");
BigInteger grandEntier2 = new BigInteger("98765432109876543210");

// Multiplication de ces deux grands nombres
BigInteger produit = grandEntier1.multiply(grandEntier2);

System.out.println("Résultat : " + produit);

// Tentative de conversion du résultat en long
long conversion = produit.longValue();

System.out.println("Valeur tronquée (dépassement) : " + conversion);
```

```
// Résultat (affichage) dans la console
Résultat : 1219326311370217952237463801111263526900
Valeur tronquée (dépassement) : 1331246629686034420
```

BigDecimal : Classe permettant de manipuler des nombres décimaux de grande précision, dépassant les limites du type primitif double.

Essentielle dans les applications nécessitant une grande précision des calculs décimaux, comme en finance pour le calcul des taux d'intérêt, la facturation, et la comptabilité, où les erreurs d'arrondis des types à virgule flottante (float et double) ne sont pas acceptables.

```
BigDecimal grandDecimal1 = new BigDecimal("1234567890123456.123456789");
BigDecimal grandDecimal2 = new BigDecimal("9876543210987654.987654321");

// Multiplication de ces deux grands décimaux
BigDecimal produit = grandDecimal1.multiply(grandDecimal2);
System.out.println("Résultat : " + produit);

// Tentative de conversion du résultat en double
double conversion = produit.doubleValue();

System.out.println("Valeur tronquée (dépassement) : " + conversion);
```

```
// Résultat (affichage) dans la console
Résultat : 12193263113702173772138374308793.945269013112635269
Valeur tronquée (dépassement) : 1.2193263113702175E31
```

Integer.parseInt(String str) : Tente de convertir la chaîne de caractères passée en paramètre en valeur entière primitive (int), puis renvoie le résultat en cas de succès.

C'est une méthode très pratique pour convertir des chaînes de caractères en entiers, permettant ainsi de manipuler les informations avec le type le plus adapté.

Attention : Une tentative de conversion d'une valeur non convertible en valeur entière générera une erreur NumberFormatException.

```
int age1 = Integer.parseInt("25"); // age1 = 25
int age2 = Integer.parseInt("AZ"); // NumberFormatException
```

Integer.valueOf(String str) : Tente de convertir la chaîne de caractères passée en paramètre en type enveloppe entier (Integer), puis renvoie le résultat en cas de succès.

C'est une méthode similaire à Integer.parseInt(String str) avec pour seule différence le type de retour (Integer au lieu de int).

Attention: Une tentative de conversion d'une valeur non convertible en valeur entière générera une erreur NumberFormatException.

```
Integer age1 = Integer.valueOf("25"); // age1 = 25
Integer age2 = Integer.valueOf("AZ"); // NumberFormatException
```

Math : Classe Utilitaire contenant des méthodes (méthode) permettant de réaliser les opérations mathématiques, de la simple addition aux calculs trigonométriques plus complexes.

Une classe dont il faut se souvenir pour éviter l'implémentation de code inutile (et potentiellement défectueuse).

Math.min(int a, int b) : Renvoie la plus petite valeur des deux passées en argument.

```
int plusPetit = Math.min(4, 10);

System.out.println("Résultat : " + plusPetit);
```

```
// Résultat (affichage) dans la console
Résultat : 4
```

Math.max(int a, int b) : Renvoie la plus grande valeur des deux passées en argument.

```
int plusGrand = Math.max(4, 10);

System.out.println("Résultat : " + plusGrand);
```

```
// Résultat (affichage) dans la console
Résultat : 10
```

Math.random() : Renvoie une valeur aléatoire de type double, comprise entre 0 et 1.

```
double nombreAleatoire = Math.random();

System.out.println("Nombre aléatoire : " + nombreAleatoire);
```

```
// Résultat (affichage) dans la console
Nombre aléatoire : 0.9246716297487397
```

On peut facilement générer des nombres aléatoires entre 0 et 100 en combinant les méthodes Math.random() et Math.round(double a).

```
double nombreAleatoire = Math.random();

double multiplicationPar100 = nombreAleatoire * 100;

long resultatArrondi = Math.round(multiplicationPar100);

System.out.println("Nombre aléatoire sur 100 : " + resultatArrondi);
```

```
// Résultat (affichage) dans la console
Nombre aléatoire sur 100 : 88
```

Math.round(double a) : Renvoie la valeur entière arrondie la plus proche de la valeur passée en argument.

```
long valeurArrondie1 = Math.round(4.25);
long valeurArrondie2 = Math.round(4.75);

System.out.println("Valeur arrondie 1 : " + valeurArrondie1);
System.out.println("Valeur arrondie 2 : " + valeurArrondie2);
```

```
// Résultat (affichage) dans la console
Valeur arrondie 1 : 4
Valeur arrondie 2 : 5
```

Math.abs(int a) : Renvoie la valeur absolue correspondant à la valeur passée en argument.

```
int valeurAbsolue1 = Math.abs(-360);
int valeurAbsolue2 = Math.abs(3000);

System.out.println("Valeur absolue 1 : " + valeurAbsolue1);
System.out.println("Valeur absolue 2 : " + valeurAbsolue2);
```

```
// Résultat (affichage) dans la console
Valeur absolue 1 : 360
Valeur absolue 2 : 3000
```

Math.sqrt(double a) : Renvoie la valeur correspondant à la racine carrée (SQRT : Square Root) de la valeur passée en argument.

```
double racineCarreeDe50  = Math.sqrt(50);
double racineCarreeDe100 = Math.sqrt(100);

System.out.println("Racine carrée de 50  : " + racineCarreeDe50);
System.out.println("Racine carrée de 100 : " + racineCarreeDe100);
```

```
// Résultat (affichage) dans la console
Racine carrée de 50  : 7.0710678118654755
Racine carrée de 100 : 10.0
```

22. Dates et Temps (Dates and Times)

Date : Classe permettant de représenter un moment spécifique dans le temps, avec une précision à la milliseconde près.

Basée sur le nombre de millisecondes écoulées depuis le 1er Janvier 1970 à minuit UTC (Coordinated Universal Time).

```
Date maintenant = new Date();

System.out.println("Date actuelle : " + maintenant);
```

```
// Résultat (affichage) dans la console
Date actuelle : Fri Mar 15 18:18:18 CET 2024
```

LocalDate : Classe permettant de représenter une date sans heure, ni fuseau horaire (jour / mois / année).

Utilisée pour les anniversaires, les jours fériés, etc.

```
LocalDate aujourdHui = LocalDate.now();

System.out.println("Aujourd'hui : " + aujourdHui);
```

```
// Résultat (affichage) dans la console
Aujourd'hui : 2024-03-15
```

LocalTime : Classe permettant de représenter un horaire sans date, ni fuseau horaire (heure / minute / seconde).

Utilisée pour les heures d'ouverture, les horaires de trains, etc.

```
LocalTime maintenant = LocalTime.now();

System.out.println("Heure actuelle : " + maintenant);
```

```
// Résultat (affichage) dans la console
Heure actuelle : 18:20:49.586992600
```

LocalDateTime : Classe combinant LocalDate et LocalTime, permettant de représenter une date (jour / mois / année) avec un horaire (heure / minute / seconde), mais sans fuseau horaire.

Utilisée pour les dates entières ne nécessitant pas de fuseau horaire.

```
LocalDateTime maintenant = LocalDateTime.now();

System.out.println("Date et heure actuelles : " + maintenant);
```

```
// Résultat (affichage) dans la console
Date et heure actuelles : 2024-03-15T18:22:11.212587500
```

ZonedDateTime : Classe équivalente à LocalDateTime et intégrant la notion de fuseau horaire.

Utilisée par les applications dont les utilisateurs opèrent sur différents fuseaux horaires, évitant ainsi les erreurs d'horaires.

```
// Date complète avec le fuseau horaire en argument
ZonedDateTime dateParis = ZonedDateTime.now(ZoneId.of("Europe/Paris"));

System.out.println("Date et heure à Paris : " + dateParis);
```

```
// Résultat (affichage) dans la console
Date et heure à Paris : 2024-03-15T18:23:22.4611536+01:00 [Europe/Paris]
```

Duration : Classe utilisée pour représenter une quantité de temps en termes d'heures, de minutes et de secondes.

Elle permet de calculer le temps écoulé entre deux dates / instants.

```
LocalTime debut = LocalTime.of(10, 30); // 10H30
LocalTime fin   = LocalTime.of(17, 45); // 17H45

Duration duree = Duration.between(debut, fin);

System.out.println("Temps écoulé : " + duree.toSeconds() + " secondes");
```

```
// Résultat (affichage) dans la console
Temps écoulé : 26100 secondes
```

Gestion des Données 🧱
[Data Management]

23. Tableaux (Arrays)

Tableau (Array) : Structure de données permettant de stocker, sous la forme d'une liste, un nombre fixe d'éléments du même type.

C'est la forme primitive des suites d'éléments en Java, qui est à la base du fonctionnement des autres types similaires conceptuellement comme List<E>, Set<E> ou encore Queue<E>.

```
int[] intArray = {1, 2, 3, 4};

int premiereValeur = intArray[0];
int derniereValeur = intArray[3];

System.out.println("Première valeur du tableau : " + premiereValeur);
System.out.println("Dernière valeur du tableau : " + dernieреValeur);
```

```
// Résultat (affichage) dans la console
Première valeur du tableau : 1
Dernière valeur du tableau : 4
```

Il s'agit notamment du type utilisé pour le paramètre de la méthode main (entrée de programme), que l'on croise très souvent :

```
/**
 * La méthode main possède un paramètre qui en fait
 * un tableau de chaînes de caractères (String)
 *
 * Ce tableau contient les arguments passés lors de l'exécution
 * du fichier via une ligne de commande (java <classe> arg1 arg2)
 */
public static void main(String[] args) {

    // On peut récupérer les arguments passés en paramètre
    String arg1 = args[0];
    String arg2 = args[1];

    // Suite du code ...
}
```

24. Généricité (Generics)

Generics (Généricité) : Mécanisme permettant de définir des classes (classe), des interfaces (interface) et des méthodes (méthode) paramétrables avec un (ou plusieurs) type de données.

Peut aussi être lu "Multi Data Type" en raison de son objectif principal qui est de permettre l'implémentation de code s'adaptant à plusieurs (multi) types (type) de données (data).

C'est grâce à la généricité que les développeurs Java peuvent créer des listes de chaînes de caractères (ArrayList<String>) ou de tout autre type de données alors qu'il n'existe qu'une seule et unique classe ArrayList<E> générique.

```
/**
 * L'entête de déclaration de la classe ArrayList<E> contient
 * le paramètre de type <E> impliquant que la classe soit générique.
 *
 * C'est en cela que l'instanciation de cette classe prend la forme
 * * new ArrayList<String>();
 * où <String> remplace <E> lors de l'exécution du code.
 */
public class ArrayList<E> extends AbstractList<E>
                          implements List<E>, … { ... }

/**
 * La classe HashMap<K, V> possède une déclaration similaire
 * à la différence qu'elle est paramétrée par deux types au lieu d'un.
 *
 * Son instanciation est alors similaire, à un paramètre près
 * * new HashMap<String, Integer>();
 * où <String, Integer> remplace <K, V> au moment de l'exécution du code
 */
public class HashMap<K, V> extends AbstractMap<K, V>
                           implements Map<K, V>, … { ... }
```

25. Collections (Collections)

Collections : Classe Utilitaire contenant des méthodes permettant de manipuler, trier ou encore rechercher les éléments dans un type implémentant l'interface Collection<E> (List, Set, etc.).

C'est une classe dont vous aurez besoin en Java, tôt ou tard.

Collections.sort(List<T> list) : Tri les éléments de la liste passée en argument selon leur ordre naturel, ou celui d'un comparateur.

Existe également avec un paramètre supplémentaire de type Comparable pour trier les éléments selon ce dernier et non l'ordre naturel.

```
List<Integer> integerList = new ArrayList<>();
integerList.add(12);
integerList.add(254);
integerList.add(24);
integerList.add(65);
integerList.add(8);
integerList.add(982);

System.out.println("Ordre actuel : " + integerList);

// Tri la liste dans l'ordre naturel des éléments
Collections.sort(integerList);

System.out.println("Ordre après tri : " + integerList);
```

```
// Résultat (affichage) dans la console
Ordre actuel : [12, 254, 24, 65, 8, 982]
Ordre après tri : [8, 12, 24, 65, 254, 982]
```

Collection<E> : Interface générique (Généricité) déclarant les méthodes (méthode) de base pour travailler avec des suites d'éléments, également appelées des collections.

Une collection peut être vue comme une version grandement améliorée d'un tableau : sa taille est dynamique (et non fixe), elle offre une variété de méthodes de base permettant une manipulation plus précise de ses éléments, et ses implémentations (telles que List<E> et Set<E>) s'adaptent à des besoins bien spécifiques en programmation.

Collection.add(E element) : Ajoute l'élément en argument à la fin de la collection. *Existe également avec un paramètre supplémentaire de type int pour ajouter l'élément à une position précise.*

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("by JirAWS");

System.out.println("Liste initiale : " + texteList);

// Ajout de l'élément "JavaCore" à l'index 1 (2ème position)
texteList.add(1, "JavaCore");

System.out.println("Liste après ajout spécifique : " + texteList);
```

```
// Résultat (affichage) dans la console
Liste initiale : ["Formation", "by JirAWS"]
Liste après ajout spécifique : ["Formation", "JavaCore", "by JirAWS"]
```

Collection.remove(E element) : Supprime de la collection l'élément passé en argument, si ce dernier est présent dans la collection.

Existe également avec un paramètre de type int pour supprimer l'élément à une position précise.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");

System.out.println("Liste initiale : " + texteList);

texteList.remove("JavaCore");

System.out.println("Liste après suppression 1 : " + texteList);

texteList.remove(0); // Suppression de l'élément à l'index 0

System.out.println("Liste après suppression 2 : " + texteList);
```

```
// Résultat (affichage) dans la console
Liste initiale : ["Formation", "JavaCore"]
Liste après suppression 1 : ["Formation"]
Liste après suppression 2 : []
```

Collection.get(int index) : Renvoie l'élément se trouvant à l'index (position) passé en argument.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

String troisiemeElement = texteList.get(2); // "by JirAWS"

System.out.println("Liste actuelle : " + texteList);
System.out.println("Texte récupéré : " + troisiemeElement);
```

```
// Résultat (affichage) dans la console
Liste actuelle : ["Formation", "JavaCore", "by JirAWS"]
Texte récupéré : "by JirAWS"
```

Collection.contains(E element) : Renvoie vrai (true) si l'élément passé en argument est présent dans la collection.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

System.out.println("Liste actuelle : " + texteList);

boolean contientFormation = texteList.contains("Formation"); // true
boolean contientPython    = texteList.contains("Python");    // false

System.out.println("Contient 'Formation' : " + contientFormation);
System.out.println("Contient 'Python'    : " + contientPython);
```

```
// Résultat (affichage) dans la console
Liste actuelle : ["Formation", "JavaCore", "by JirAWS"]
Contient 'Formation' : true
Contient 'Python'    : false
```

Collection.size() : Renvoie le nombre d'éléments (taille) présents dans la collection.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

int tailleListe = texteList.size(); // 3

System.out.println("Liste actuelle : " + texteList);
System.out.println("Taille de la liste : " + tailleListe);
```

```
// Résultat (affichage) dans la console
Liste actuelle : ["Formation", "JavaCore", "by JirAWS"]
Taille de la liste : 3
```

Collection.clear() : Supprime tous les éléments de la collection.

Très utile pour réutiliser une collection déjà instanciée, évitant ainsi d'utiliser davantage de mémoire.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

System.out.println("Liste avant 'clear' : " + texteList);

texteList.clear();

System.out.println("Liste après 'clear' : " + texteList);
```

```
// Résultat (affichage) dans la console
Liste avant 'clear' : ["Formation", "JavaCore", "by JirAWS"]
Liste après 'clear' : []
```

Collection.isEmpty() : Renvoie vrai (true) si la collection est vide, autrement dit, si elle ne contient aucun élément.

Préférable à l'écriture manuelle d'une expression booléenne utilisant la méthode Collection.size() == 0;

```
List<String> textList = new ArrayList<>();

System.out.println("Liste avant les ajouts : " + textList);
System.out.println("Résultat de 'isEmpty' : " + textList.isEmpty());

textList.add("Formation");
textList.add("JavaCore");
textList.add("by JirAWS");

System.out.println("Liste après les ajouts : " + textList);
System.out.println("Résultat de 'isEmpty' : " + textList.isEmpty());
```

```
// Résultat (affichage) dans la console
Liste avant les ajouts : []
Résultat de 'isEmpty' : true

Liste après les ajouts : ["Formation", "JavaCore", "by JirAWS"]
Résultat de 'isEmpty' : false
```

Iterator : Interface spécifiquement conçue pour itérer sur les éléments d'un type Collection<E> (suite d'éléments).

Elle offre une approche d'itération optimisée et sécurisée, avec des méthodes (méthode) permettant de vérifier la présence d'éléments suivants, d'accéder à l'élément suivant et de supprimer des éléments durant l'itération, évitant ainsi les risques liés à l'utilisation d'une boucle for-each classique.

ListIterator : Une extension de l'interface Iterator qui permet de parcourir les éléments d'un type List<E> dans les deux sens, de les modifier durant l'itération et d'obtenir leurs index (position).

Permet notamment d'ajouter des éléments à la volée durant l'itération, sans craindre une ConcurrentModificationException comme avec une itération via boucle for-each.

Collection.iterator() : Renvoie un itérateur (Iterator) permettant une itération optimale sur les éléments de la collection.

Une approche souvent délaissée en faveur d'une boucle for-each pour les itérations simples ne nécessitant pas davantage de sécurité.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

// Obtention de l'itérateur pour la liste
Iterator<String> iterator = texteList.iterator();

// Parcours de la liste à l'aide de l'itérateur
while (iterator.hasNext()) {
    String texte = iterator.next();
    System.out.println(texte);
}
```

```
// Résultat (affichage) dans la console
"Formation"
"JavaCore"
"by JirAWS"
```

Comparable<T> : Interface générique (Généricité) déclarant une unique méthode pour comparer deux instances (objet) d'un même type.

Notamment utilisé par les méthodes de tri sur les types Collection.

```
public class Personne implements Comparable<Personne> {

    private int age;

    public Personne(int age) {
        this.age = age;
    }

    @Override
    public int compareTo(Personne autre) {
        return this.age - autre.age;
    }

}
```

26. Listes (Lists)

List<E> : Interface générique (Généricité) déclarant les méthodes (méthode) permettant la manipulation de suites (ou “listes”) d’éléments ordonnés du même type et pouvant contenir des doublons.

C’est le type de Collection<E> le plus couramment utilisé, au travers de la classe ArrayList<E> qui l’implémente.

```
public interface List<E> extends SequencedCollection<E> { ... }
```

ArrayList<E> : Classe générique (Généricité) implémentant l’interface List<E>, basée sur un tableau (array).

C’est la forme de List<E> la plus standard, pour les usages généraux.

```
List<String> texteList = new ArrayList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

System.out.println("Liste après les ajouts : " + texteList);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : ["Formation", "JavaCore", "by JirAWS"]
```

LinkedList<E> : Classe générique (Généricité) très similaire à ArrayList<E> mais dont les éléments sont doublement liés.

Chaque élément connaît l’élément qui le précède et le succède, ce qui rend ce type de liste particulièrement efficace pour les modifications (ajout / suppression / insertion) intensives.

```
List<String> texteList = new LinkedList<>();
texteList.add("Formation");
texteList.add("JavaCore");
texteList.add("by JirAWS");

System.out.println("Liste après les ajouts : " + texteList);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : ["Formation", "JavaCore", "by JirAWS"]
```

27. Ensembles (Sets)

Set<E> : Interface générique (Généricité) très similaire à List<E> mais qui ne permet pas les doublons d'éléments.

C'est l'interface de référence pour représenter des ensembles (Set) d'éléments distincts, sans ordre particulier.

```
public interface Set<E> extends Collection<E> { ... }
```

HashSet<E> : Classe générique (Généricité) implémentant l'interface Set<E>, basée sur une table de hachage (Hash), permettant ainsi un accès rapide aux éléments.

La forme d'ensemble (Set) la plus standard, pour les usages généraux.

La table de hachage (Hash) permet à une instance de HashSet de positionner les éléments contenus en fonction du résultat de leur hachage et ainsi de les retrouver grâce à cette valeur plutôt qu'en parcourant la suite d'éléments comme le fait un type List<E>.

```
Set<String> texteSet = new HashSet<>();

/**
 * Rappel important : l'ordre d'ajout des éléments n'est pas assuré dans
 * un HashSet, car il positionne les éléments en fonction du
 * résultat de leur hachage (Hash) et non de leur ordre d'insertion.
 */
texteSet.add("Formation");
texteSet.add("JavaCore");
texteSet.add("by JirAWS");

System.out.println("Set après les ajouts : " + texteSet);

// Tentative d'ajout d'un doublon
texteSet.add("Formation");

System.out.println("Set après ajout de doublon : " + texteSet);
```

```
// Résultat (affichage) dans la console
Set après les ajouts : ["Formation", "by JirAWS", "JavaCore"]
Set après ajout de doublon : ["Formation", "by JirAWS", "JavaCore"]
```

TreeSet<E> : Classe générique (Généricité) implémentant l'interface Set<E>, basée sur un arbre rouge-noir, garantissant ainsi que les éléments soient triés selon leur ordre naturel ou celui d'une implémentation de Comparable<T>.

Un TreeSet<E> peut être vu comme une ArrayList<E> à la différence qu'il empêche l'ajout de doublons d'éléments.

La notion d'arbre "rouge-noir" réfère aux arbres binaires: des structures d'éléments sous la forme d'arbre dont les branches ont (au mieux) toutes la même taille (les éléments étant les feuilles), ce qui permet à une machine (binaire) de réaliser des recherches très efficaces, et également de conserver l'ordre des éléments stockés dans le cas du TreeSet<E>.

```
Set<Integer> integerSet = new TreeSet<>();

/**
 * Rappel important : l'ordre naturel des éléments est assuré dans
 * un TreeSet, contrairement à un HashSet.
 */
integerSet.add(244);
integerSet.add(99);
integerSet.add(150);
integerSet.add(10);

System.out.println("Set après les ajouts : " + integerSet);

// Tentative d'ajout d'un doublon
integerSet.add(150);

System.out.println("Set après ajout de doublon : " + integerSet);
```

```
// Résultat (affichage) dans la console
Set après les ajouts : [10, 99, 150, 244]
Set après ajout de doublon : [10, 99, 150, 244]
```

28. Dictionnaires (Maps)

Map<K, V> : Interface générique (Généricité) déclarant les méthodes (méthode) permettant la manipulation de suite d'éléments au format Clé-Valeur (K, V : Key, Value).

Une Map conserve le concept fondamental des suites (List) d'éléments. Il s'agit d'une suite d'éléments, mais qui sont accessibles grâce à une clé associée plutôt qu'un index (position).

Les éléments similaires à Map en langage Python sont notamment appelés "Dictionnaire" : Mot -> Définition.

On peut voir ce guide comme une très grande instance de Map, en raison de sa structure : Mot-clé Java -> Définition.

```
public interface Map<K, V> { ... }
```

HashMap<K, V> : Classe générique (Généricité) implémentant l'interface Map, basée sur une table de hachage (Hash), permettant ainsi un accès rapide aux éléments.

C'est la forme de Map la plus standard, pour les usages généraux.

Comme avec la classe HashSet<E>, la classe HashMap<K, V> utilise le résultat du hachage (Hash) pour positionner les éléments en fonction de leur clé. Elle n'assure donc pas l'ordre d'insertion des éléments.

```
Map<String, Integer> langagesCreation = new HashMap<>();

/**
 * Rappel important : l'ordre d'ajout des clés n'est pas assuré dans
 * une HashMap, car il positionne les éléments en fonction du
 * résultat de leur hachage (Hash) et non de leur ordre d'insertion.
 */
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après les ajouts : " + langagesCreation);
```

```
// Résultat (affichage) dans la console
Map après les ajouts : {"Java" = 1995, "C" = 1972, "Python"= 1991}
```

TreeMap<K, V> : Classe générique (Généricité) implémentant l'interface Map, basée sur un arbre rouge-noir, garantissant ainsi que les clés soient triées selon leurs ordres naturels ou celui d'une implémentation de Comparable<T>.

La classe TreeMap<K, V> repose sur le même système d'arbre "rouge-noir" que la classe TreeSet<E>. L'explication sur les arbres binaires se trouve dans la définition de TreeSet<E>.

```
Map<String, Integer> langagesCreation = new HashMap<>();

/**
 * Rappel important : l'ordre naturel des clés est assuré dans
 * un TreeMap, contrairement à une HashMap.
 */
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après les ajouts : " + langagesCreation);
```

```
// Résultat (affichage) dans la console
Map après les ajouts : {"C" = 1972, "Java" = 1995, "Python" = 1991}

// On remarque que les entrées ont été triées alphabétiquement
```

Map.put(K key, V value) : Crée une entrée (un élément) dans la Map avec la clé (key) en lui associant la valeur (value) passée en argument, ce qui forme une paire clé-valeur.

Si la clé existe déjà, alors sa valeur associée existante est remplacée par la nouvelle passée en argument.

```
Map<String, Integer> langagesCreation = new HashMap<>();

// Clé : Nom du langage / Valeur : Année d'apparition
langagesCreation.put("Java", 1995);

System.out.println("Map après l'ajout : " + langagesCreation);
```

```
// Résultat (affichage) dans la console
Map après l'ajout : {"Java" = 1995}
```

Map.get(K key) : Renvoie la valeur associée à la clé passée en argument si elle existe dans la Map. Renvoie null si la clé n'existe pas ou qu'elle n'a pas de valeur associée.

```
Map<String, Integer> langagesCreation = new HashMap<>();

// Clé : Nom du langage / Valeur : Année d'apparition
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après les ajouts : " + langagesCreation);

Integer anneeCreationJava = langagesCreation.get("Java");

System.out.println("Apparition du langage Java : " + anneeCreationJava);
```

```
// Résultat (affichage) dans la console
Map après les ajouts : {"Java" = 1995, "C" = 1972, "Python"= 1991}
Apparition du langage Java : 1995
```

Map.remove(K key) : Supprime la clé passée en argument et sa valeur associée. La paire (élément) clé-valeur est supprimée.

Renvoie vrai (true) si la suppression est validée, faux (false) sinon.

```
Map<String, Integer> langagesCreation = new HashMap<>();

// Clé : Nom du langage / Valeur : Année d'apparition
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après les ajouts : " + langagesCreation);

langagesCreation.remove("Java");

System.out.println("Map après 'remove' : " + langagesCreation);
```

```
// Résultat (affichage) dans la console
Map après les ajouts : {"Java" = 1995, "C" = 1972, "Python"= 1991}
Map après 'remove' : {"C" = 1972, "Python"= 1991}
```

Map.keySet() : Renvoie une collection de type Set<E> contenant toutes les clés présentes dans la Map<K, V> (sans les valeurs).

Les clés d'une Map étant uniques, le type Set<E> est naturellement le plus approprié grâce à sa garantie d'unicité des éléments.

```
Map<String, Integer> langagesCreation = new HashMap<>();

// Clé : Nom du langage / Valeur : Année d'apparition
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après ajout : " + langagesCreation);

Set<String> langageSet = langagesCreation.keySet();

System.out.println("Set de clés : " + langageSet);
```

```
// Résultat (affichage) dans la console
Map après ajout : {"Java" = 1995, "C" = 1972, "Python" = 1991}
Set de clés : ["Java", "C", "Python"]
```

Map.values() : Renvoie une Collection<E> contenant toutes les valeurs présentes dans la Map (sans leurs clés).

```
Map<String, Integer> langagesCreation = new HashMap<>();

// Clé : Nom du langage / Valeur : Année d'apparition
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après ajout : " + langagesCreation);

Collection<Integer> anneeListe = langagesCreation.values();

System.out.println("Liste des valeurs : " + anneeListe);
```

```
// Résultat (affichage) dans la console
Map après ajout : {"Java" = 1995, "C" = 1972, "Python" = 1991}
Liste des valeurs : [1995, 1972, 1991]
```

Map.entrySet() : Renvoie une collection de type Set<E> contenant les paires clé-valeur (Entry<K, V>) de la Map.

Très utile lorsque l'on souhaite itérer avec une boucle for-each sur l'ensemble des éléments d'une Map, afin de les manipuler.

```
Map<String, Integer> langagesCreation = new HashMap<>();

// Clé : Nom du langage / Valeur : Année d'apparition
langagesCreation.put("Java", 1995);
langagesCreation.put("Python", 1991);
langagesCreation.put("C", 1972);

System.out.println("Map après ajout : " + langagesCreation);

// Conversion des entrées (éléments) de la Map en un ensemble (Set)
Set<Entry<String, Integer>> langagesSet = langagesCreation.entrySet();

System.out.println("Set d'Entrées : " + langagesSet);

// Itérations sur les éléments (Entry<X, Y>) de l'ensemble (Set)
for (Entry<String, Integer> langage : langagesSet) {
    System.out.println(langage.getKey() + " - " + langage.getValue() );
}
```

```
// Résultat (affichage) dans la console
Map après ajout : {"Java" = 1995, "C" = 1972, "Python" = 1991}
Set d'Entrées :   ["Java" = 1995, "C" = 1972, "Python" = 1991]

// Affichage de la boucle for-each
Java - 1995
C - 1972
Python - 1991
```

29. Files d'Attente (Queues)

Queue<E> : Interface générique (Généricité) partageant (avec List<E> et Set<E>) le concept de manipulation de suites d'éléments, en étant spécialement adaptée à la création de "files d'attentes" (Queue).

C'est un type de collection assez spécifique, conçu pour le traitement séquentiel d'éléments. Les éléments ajoutés (offer) en premiers seront les premiers à être extraits (poll) pour être traités. C'est ce qu'on appelle la méthode "FIFO", acronyme de "First In, First Out".

```java
// La classe LinkedList implémente l'interface Queue
Queue<Integer> fileAttente = new LinkedList<>();

// Ajout d'éléments - Équivalent de Collection.add(E element)
fileAttente.offer(1);
fileAttente.offer(2);
fileAttente.offer(3);

System.out.println("Queue après les ajouts : " + fileAttente);

/**
 * Extraction du prochain élément en attente
 * Équivalent de Collection.get(0) suivi de Collection.remove(0)
 */
Integer elementExtrait1 = fileAttente.poll(); // 1

System.out.println("Élément extrait : " + elementExtrait1);
System.out.println("Queue après extraction N°1 : " + fileAttente);

// Extraction du prochain élément en attente
Integer elementExtrait2 = fileAttente.poll(); // 2

System.out.println("Élément extrait : " + elementExtrait2);
System.out.println("Queue après extraction N°2 : " + fileAttente);
```

```
// Résultat (affichage) dans la console
Queue après les ajouts : [1, 2, 3]
Élément extrait : 1
Queue après extraction N°1 : [2, 3]
Élément extrait : 2
Queue après extraction N°2 : [3]
```

30. Flux (Streams)

Stream (Flux) : Un flux correspond à un ensemble d'éléments qui vont pouvoir être manipulés séquentiellement, les uns après les autres, ou parallèlement (parallélisme) avec un traitement en simultané.

C'est sur cette logique que repose l'interface Stream<E> (API), qui permet de réaliser des opérations sur les types Collection<E>.

Stream<E> (API) : Interface générique (Généricité) déclarant les méthodes (méthode) permettant de manipuler les éléments d'un type collection (List<E>, Set<E>, etc.) avec notamment des opérations de filtrage, de recherche, de mappage, ou encore de collecte.

C'est une interface incontournable pour les développeurs Java.

```
public interface Stream<T> extends BaseStream<T, Stream<T>> { ... }
```

Collection.stream() : Convertit une Collection<E> en Stream<E> permettant d'accéder aux méthodes (méthode) de cette interface.

On peut voir ça comme une "sorte" de transformation en boucle for-each permettant de traiter un flux (une suite) d'éléments un par un.

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(1);
integerList.add(25);
integerList.add(300);

System.out.println("Liste après les ajouts : " + integerList);

// Transformation de la liste en stream (flux)
Stream<Integer> integerStream = integerList.stream();

System.out.println("Stream : " + integerStream); // [1, 25, 300]
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : [1, 25, 300]
Stream : java.util.stream.ReferencePipeline$Head@b4c966a
```

Stream.filter(Predicate<T> predicate) : Renvoie un Stream<E> ne contenant que les éléments respectant le prédicat (condition) passé en argument.

Cette méthode peut être visualisée comme une boucle for-each qui va faire une itération sur chaque élément du flux, pour vérifier s'ils respectent la condition donnée et ne conserve que ceux pour lesquels cette dernière renvoie vrai (true).

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(1);
integerList.add(25);
integerList.add(300);

System.out.println("Liste après les ajouts : " + integerList);

/**
 * 1. Transformation de la liste en stream (flux)
 * 2. Filtrage des éléments pour ne garder que ceux dont la valeur
 *    est supérieure à 30
 */
Stream<Integer> intStream1 = integerList.stream()
                                        .filter(entier -> entier > 30);

System.out.println("Stream après 1er filtrage  : " + intStream1);

/**
 * La transformation en stream n'est plus nécessaire, car déjà faite
 * 1. Filtrage des éléments pour ne garder que ceux dont la valeur
 *    est supérieure à 500
 */
Stream<Integer> intStream2 = intStream1.filter(entier -> entier > 500);

System.out.println("Stream après 2ème filtrage : " + intStream2);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : [1, 25, 300]
Stream après 1er filtrage  : [25, 300]
Stream après 2ème filtrage : []
```

Stream.map(Function<T,R> mapper) : Applique la fonction passée en argument sur chaque élément du flux et renvoie un Stream<E> des éléments modifiés.

Cette méthode peut être visualisée comme une boucle for-each qui va faire une itération sur chaque élément du flux, appliquant l'opération (fonction) demandée et générant un nouveau flux avec les résultats.

La fonction passée en paramètre ne doit pas forcément être une opération de modification. Elle peut également être une opération de lecture (récupération) d'une valeur (comme un attribut), permettant ainsi de former un nouveau flux (stream) constitué de ces valeurs.

Le nom de la méthode "map" est emprunté au concept de la Programmation Fonctionnelle, où "mapper" signifie: "appliquer une fonction à chaque élément d'une collection pour en obtenir une nouvelle transformée".

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(1);
integerList.add(25);
integerList.add(300);

System.out.println("Liste après les ajouts : " + integerList);

/**
 * 1. Transformation de la liste en stream (flux)
 * 2. Multiplication par 2 de la valeur de chaque élément du Stream
 */
Stream<Integer> intStream = integerList.stream()
                                       .map(entier -> entier * 2);

System.out.println("Stream après filtrage : " + intStream);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : [1, 25, 300]
Stream après filtrage :  [2, 50, 600]
```

Stream.forEach(Function<T,R> mapper) : Permet d'effectuer une itération via une boucle for-each sur les éléments du Stream<E>.

Attention: Cette méthode ne renvoie rien (void), c'est-à-dire qu'elle ne renvoie pas de Stream<E> utilisable comme ce que font les autres méthodes vues dans ce guide.

Cette méthode est totalement comparable à l'implémentation d'une boucle for-each: elle itère sur chaque élément présent dans le Stream, et applique le bloc de code fourni en paramètre.

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(1);
integerList.add(25);
integerList.add(300);

System.out.println("Liste après les ajouts : " + integerList);

// 1. Transformation de la liste en stream (flux)
Stream<Integer> intStream = integerList.stream();

// 2. Itération sur chaque élément grâce à une boucle forEach
intStream.forEach(entier -> {

     System.out.println("Affichage de l'élément : " + entier);

});

System.out.println("Stream après les opérations : " + intStream);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : [1, 25, 300]
Affichage de l'élément : 1
Affichage de l'élément : 25
Affichage de l'élément : 300
Stream après les opérations : [1, 25, 300]
```

Stream.sorted() : Trie les éléments du Stream<E> en fonction de leur ordre naturel, ou de celui donné par un comparateur (Comparator<T>).

Il existe également la méthode surchargée (Overload) permettant de fournir un comparateur externe afin d'appliquer une logique de tri bien spécifique : Stream.sorted(Comparator<T> comparator).

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments dans un ordre non naturel (non croissant)
integerList.add(200);
integerList.add(25);
integerList.add(36);
integerList.add(44);

System.out.println("Liste après les ajouts : " + integerList);

/**
 * 1. Transformation de la liste en stream (flux)
 * 2. Tri des éléments du stream en fonction de leur ordre naturel
 */
Stream<Integer> intStream = integerList.stream()
                                       .sorted();

System.out.println("Stream après le tri    : " + intStream);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts : [200, 25, 36, 44]
Stream après le tri    : [25, 36, 44, 200]
```

Stream.toList() : Renvoie une collection de type List<E> à partir des éléments contenus dans le Stream<E>.

Particulièrement utile en sortie des opérations effectuées avec les méthodes (méthode) de Stream<E> pour réutiliser un type d'objet plus généraliste et compatible avec, par exemple, les bases de données.

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(200);
integerList.add(25);
integerList.add(36);
integerList.add(44);

System.out.println("Liste après les ajouts : " + integerList);

/**
 * 1. Transformation de la liste en stream (flux)
 * 2. Transformation du stream en liste
 */
List<Integer> liste = integerList.stream()
                                 .toList();

System.out.println("Liste générée à partir du stream : " + liste);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts  : [200, 25, 36, 44]
Liste générée à partir du stream : [200, 25, 36, 44]
```

Stream.distinct() : Renvoie une copie du Stream<E> ne contenant que des éléments uniques (distincts), après avoir supprimé les doublons.

On peut voir ça comme la conversion d'un type List<E> (acceptant les doublons) en Set<E> (garantissant l'unicité des éléments).

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(200);
integerList.add(25);
integerList.add(200);
integerList.add(44);
integerList.add(200);

System.out.println("Liste après les ajouts : " + integerList);

/**
 * 1. Transformation de la liste en stream (flux)
 * 2. Exclusion des éléments répétitifs
 */
Stream<Integer> integerStream = integerList.stream()
                                           .distinct();

System.out.println("Stream après 'distinct' : " + integerStream);
```

```
// Résultat (affichage) dans la console
Liste après les ajouts  : [200, 25, 200, 44, 200]
Stream après 'distinct' : [200, 25, 44]
```

Gestion des Exceptions ⚠️ [Exception Handling]

31. Objets Jetables (Throwables)

Throwable : Classe mère de toutes les erreurs possibles en Java. Elle possède deux sous-classes principales : Exception et Error.

Le mot "Throwable", qui se traduit en français par "Jetable", prend tout son sens dans la gestion des erreurs dans ce langage, avec notamment l'usage des mots-clés throw, throws ou encore catch, directement basés sur le verbe "jeter" et "attraper" en anglais.

Elle offre les méthodes (méthode) de base pour traiter les erreurs en Java, comme par exemple la méthode "printStackTrace()" qui affiche un résumé complet (StackTrace) de l'erreur survenue.

```
public class Throwable implements Serializable { ... }
```

StackTrace : Résumé d'une erreur (Throwable) survenue durant l'exécution d'un programme. Elle contient notamment le nom du type de l'erreur, un bref message d'explication et les lignes de code exécutées, ayant entraîné l'erreur.

Un élément essentiel pour les développeurs, qui joue un rôle crucial durant une phase de débogage. Apprendre à les lire et à les comprendre n'est pas une option pour devenir un développeur aguerri.

```
/**
 * Exemple de StackTrace lors d'une tentative de division par 0
 * Important : Le sens de lecture d'une StackTrace est de bas en haut
 * Type de l'erreur : Exception
 * Nom de l'erreur : ArithmeticException
 * Message de l'erreur : "/ by zero" ("Division par zero")
 * Nom du fichier concerné : Main.java
 * Ligne qui a déclenché l'erreur : at Main.methode3(Main.java:33)
 */
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Main.methode3(Main.java:33)
      at Main.methode2(Main.java:28)
      at Main.methode1(Main.java:13)
      at Main.main(Main.java:9)
```

try : Mot-clé permettant d'exécuter un bloc de code en demandant à Java de se "préparer" à une potentielle erreur (Throwable).

C'est un mécanisme à utiliser dans toutes les situations présentant un (ou plusieurs) risque potentiel, comme par exemple :

- *Une tentative de connexion à une base de données, qui pourrait être indisponible, en raison de problématiques techniques.*
- *Une tentative d'ouverture (lecture) d'un fichier, nécessitant que ce dernier soit bien présent à l'emplacement (Path) indiqué dans le programme, et que les permissions système autorisent le programme à manipuler le fichier concerné.*

Ce mot-clé se combine avec l'autre mot-clé catch, permettant de traiter l'erreur potentielle qui pourrait survenir.

```
/**
 * Code fictif et incomplet, pour mettre en lumière le mot-clé try
 * La suite du guide contient les définitions des autres mots-clés
 * relatifs aux erreurs, ainsi que des exemples de code plus complets.
 */

try {

    // Tentative de division par 0 (qui échoue systématiquement)
    int resultat = 10 / 0;

    System.out.println("Résultat de la division : " + resultat);

} catch(Exception e) {
    System.out.println("Une erreur est survenue durant la division.");
    e.printStackTrace();
}
```

```
/**
 * Le code présenté ci-dessus n'affiche rien, car le programme
 * est interrompu à la ligne générant une erreur.
 * L'instruction System.out.println est donc ignorée.
 */

/**
 * Cependant, une erreur (Throwable) sera belle et bien déclenchée
 * et le langage Java affichera une StackTrace dans la console.
 */
```

catch : Bloc de code associé à un bloc try capturant l'erreur (Throwable) survenue dans ce dernier, permettant ainsi de la gérer.

Le bloc catch contient en paramètre l'erreur survenue, représentée sous la forme d'un objet, qui est alors utilisable programmatiquement.

Attention : il est important de retenir que l'exécution de ce bloc est optionnel, car son exécution dépend du déclenchement d'une erreur dans le bloc try. Il est donc judicieux de ne placer à l'intérieur que le code relatif à un scénario d'erreur dans votre programme :

- *Gestion de l'erreur : une erreur n'est pas juste une information, c'est une perturbation du programme qui aurait pu interrompre l'exécution de ce dernier. Il faut donc faire une implémentation cohérente en lien avec les scénarios d'erreurs.*
- *Application du contexte : dans une application beaucoup plus poussée que les exemples de ce guide, le bloc catch peut permettre d'effectuer des opérations en réaction aux erreurs, comme par exemple avec l'envoi de notifications pour prévenir l'équipe de production qu'une erreur a été rencontrée.*

```
int resultat;

try {

    /**
     * Tentative de division par 0 qui échoue systématiquement
     * (dans cet exemple explicite) et qui va donc stopper
     * l'exécution du programme et passer dans le bloc catch
     * avec l'erreur en paramètre (qui est ici une Exception)
     */
    resultat = 10 / 0;

    System.out.println("Résultat de la division : " + resultat);

} catch(Exception e) {
    System.out.println("Une erreur est survenue durant la division.");
    e.printStackTrace();
}
```

```
// Résultat (affichage) dans la console
Une erreur est survenue durant la division.
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Main.methode1(Main.java:14)
```

finally : Bloc de code s'exécutant après un bloc try et ses blocs catch associés, indépendamment du fait que ces derniers aient été exécutés ou non.

On écrit à l'intérieur de ce bloc tout le code qu'on aurait souhaité exécuter quoi qu'il advienne dans les blocs try et catch, qu'une erreur (Throwable) soit déclenchée ou pas.

Bien que l'ajout de try-with-resources (avec Java 7) soit venu réduire les cas d'utilisations du bloc finally, il existe des cas où son usage s'avère très pratique. Un très bon exemple est celui d'un outil de chronométrage : il est démarré au début du bloc try pour chronométrer des étapes potentiellement lentes, et sera stoppé dans le bloc finally pour qu'il ne dépende pas de l'exécution potentielle d'un bloc catch.

```
int resultat = 0;

try {

    System.out.println("Démarrage de la division...");

    // Tentative de division par 0 (qui échoue systématiquement)
    resultat = 10 / 0;

    System.out.println("Ligne qui s'affiche si tout va bien.");

} catch(Exception e) {

    System.out.println("Une erreur est survenue durant la division.");
    e.printStackTrace();

} finally {
    System.out.println("Fin de la division.");
}

System.out.println("Résultat final : " + resultat);
```

```
// Résultat (affichage) dans la console
Démarrage de la division...
Une erreur est survenue durant la division.
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Main.methode5(Main.java:14)
Fin de la division.
Résultat final : 0
```

try-with-resources : Une autre forme de try, ayant le même objectif que ce dernier, mais prenant en paramètre des ressources de type AutoCloseable qui seront “automatiquement fermées”.

C’est une optimisation de code permettant d’assurer la fermeture de ressources nécessitant une ouverture, et gourmandes en mémoire, évitant ainsi aux développeurs d’avoir à le faire eux-mêmes.

Le try-with-resources a été ajouté en Java 7, pour :

- *Faciliter l’écriture du code : vous trouverez ci-dessous deux exemples de code, avec et sans l’usage de try-with-resources, témoignant avec évidence de l’intérêt de ce dernier.*
- *Optimiser le code : de nombreux problèmes de mémoire peuvent être évités grâce à l’usage de try-with-resources. Il suffit d’un matin à moitié réveillé pour qu’un développeur commette un simple oubli qui pourrait se transformer en une fuite de mémoire, entraînant une surconsommation de la machine hôte.*

```java
/**
 * Première approche - AVEC try-with-resources (Version optimisée)
 */
public void lireFichier(String path) {

    /**
     * Ouverture du flux de lecture (BufferedReader) du fichier dans
     * la variable "br" qui est alors utilisable dans le bloc try
     * mais pas ailleurs, à cause du mécanisme de Portée de Variable.
     *
     * La variable "br" qui est un type Closeable sera automatiquement
     * fermée après l’exécution des blocs try et catch.
     */
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {

        String ligne;

        while ((ligne = br.readLine()) != null) {
            System.out.println(ligne);
        }

    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

```java
/**
 * Seconde approche - SANS try-with-resources (Version non-optimisée)
 */
public void lireFichier(String path) {

    /**
     * Déclaration de la variable (pour le flux de lecture) en dehors
     * des blocs try / catch afin de respecter le mécanisme de
     * la Portée de Variable (et ainsi l'utiliser dans le finally).
     */
     BufferedReader br = null;

    try {

        // Ouverture manuelle du flux de lecture (BufferedReader)
        br = new BufferedReader(new FileReader(path));

        String ligne;

        while ((ligne = br.readLine()) != null) {
            System.out.println(ligne);
        }

    } catch (IOException e) {

        e.printStackTrace();

    } finally {

        /**
         * Fermeture manuelle du flux de lecture (BufferedReader)
         * dans le bloc finally, pour s'assurer de son exécution
         * et ainsi éviter les surconsommations de mémoire.
         */
        if (br != null) {

            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

        }
    }
}
```

throw : Instruction permettant de propager une erreur (Throwable) qui devra être traitée ailleurs dans le programme.

C'est l'instruction utilisée par tous les programmes qui déclenchent des erreurs (Throwable), contraignant les développeurs à faire usage des blocs try et catch pour les traiter.

Cette instruction est particulièrement utile lorsqu'on souhaite déclencher des erreurs spécifiques à nos programmes, en ayant fait au préalable une implémentation customisée de la classe Exception.

```
public void affichageDivision(int dividende, int diviseur) {

    /**
     * (Exemple de code totalement fictif)
     *
     * Pour éviter que la division ne puisse générer une erreur,
     * on prévient cette dernière avec une condition vérifiant les
     * cas limitants de l'opération (ici, une division).
     *
     * CustomArithmeticException a été inventée pour cet exemple.
     */
    if(dividende == 0 || diviseur == 0) {
        throw new CustomArithmeticException("Division par zéro !");
    }
    else {
        System.out.println("Résultat : " + (dividende / diviseur));
    }

}
```

throws : Mot-clé se plaçant dans la déclaration d'une méthode, et indiquant que cette dernière peut propager le ou les type(s) d'erreurs (Throwable) mentionnés.

Le programme appelant cette méthode sera alors contraint de faire usage des blocs try et catch pour traiter les erreurs, ou alors d'utiliser lui-même le mot-clé throws dans sa déclaration.

```
public void affichageDivision(int a, int b) throws ArithmeticException {

    System.out.println("Résultat : " + (a / b));

}
```

32. Exceptions (Exceptions)

Exception : Première classe fille de la classe Throwable, permettant de représenter les erreurs d'un programme exécuté par la JVM.

On précise ici la notion d'exécution "par la JVM", car c'est toute la différence entre les deux classes filles de la classe Throwable : une Exception est liée à l'exécution interne d'un programme, quand une Error concerne les problématiques liées à l'environnement extérieur de la JVM ou à ses défaillances techniques en interne.

La tentative de division par zéro est un exemple typique d'Exception : elle résulte d'une ArithmeticException, qui est due à la logique d'un programme qui ne prévient pas cette erreur (avec une condition par exemple) et donc qui est due au programme en lui-même.

```
public class Exception extends Throwable { ... }
```

RuntimeException : Classe de base pour les erreurs (Exception) de toutes sortes lancées durant l'exécution normale d'un programme.

Une tentative de manipulation d'une variable référence qui n'a pas été initialisée (null), lèvera une Exception nommée NullPointerException, une sous-classe de RuntimeException.

```
public class RuntimeException extends Exception { ... }
```

NullPointerException : Exception lancée lorsque l'on essaie de manipuler ou d'accéder à un membre d'une variable référence qui n'a pas encore été initialisée (null).

Une erreur très basique, mais qui a le mérite d'être très facilement retrouvable (débogage) dans le code avec l'aide d'une StackTrace.

```
public class NullPointerException extends RuntimeException { ... }
```

ArithmeticException : Exception lancée lors d'erreurs arithmétiques comme la division par zéro.

```
public class ArithmeticException extends RuntimeException { ... }
```

ClassCastException : Exception lancée lorsqu'une tentative de conversion (Casting) échoue.

Une tentative de conversion d'un élément de type List<E> en Map<K, V> déclenchera une ClassCastException car le type source (List<E>) n'a aucun lien d'héritage avec le type cible (Map<K, V>).

```
public class ClassCastException extends RuntimeException { ... }
```

IndexOutOfBoundsException : Classe de base pour les erreurs (Exception) qui concerne les tentatives d'accès à des indices (index) invalides.

Traduit par "Index en dehors des bornes" (ou limites), bornes qui sont établies par un début (index le plus petit) et une fin (index le plus grand) d'une suite d'éléments (collection ou tableau).

```
public class IndexOutOfBoundsException extends RuntimeException { ... }
```

ArrayIndexOutOfBoundsException : Exception lancée lorsqu'un programme tente d'accéder à un élément d'un tableau avec un indice (index) invalide.

Une tentative d'accès à l'élément en cinquième position (indice 4) d'un tableau n'en contenant que 3 (indice min = 0 / max = 2), déclenchera une ArrayIndexOutOfBoundsException.

```
public class ArrayIndexOutOfBoundsException extends
IndexOutOfBoundsException { ... }
```

ConcurrentModificationException : Exception lancée lorsqu'une suite d'éléments est modifiée alors qu'elle est en cours d'itération.

Une boucle for-each qui ajouterait ou supprimerait un élément d'une Collection<E> durant son itération sur cette dernière, entraînerait une ConcurrentModificationException car les éléments "chargés" (load) ne seraient plus les mêmes qu'au départ de la boucle.

```
public class ConcurrentModificationException extends RuntimeException
{ ... }
```

IOException : Classe de base pour les erreurs (Exception) liées aux Entrées / Sorties (IO : Input / Output).

Une tentative d'ouverture de fichier dans un programme n'étant pas autorisé (permission système) à le faire, déclenchera une IOException.

```
public class IOException extends Exception { ... }
```

FileNotFoundException : Exception lancée lorsqu'on essaie d'accéder à un fichier (File) qui n'existe pas sur le système.

Le problème peut également venir d'une étourderie durant l'écriture du chemin d'accès (Path) par le développeur.

```
public class FileNotFoundException extends IOException { ... }
```

ClassNotFoundException : Exception lancée lorsqu'on essaie de charger une classe qui n'existe pas.

Dans certains cas, cette erreur peut survenir à cause d'une mauvaise configuration du projet ou lorsqu'une bibliothèque importée contient une classe du même nom que celle dans le projet.

```
public class ClassNotFoundException extends ReflectiveOperationException
                                                                { ... }
```

SQLException : Classe de base pour les erreurs (Exception) liées aux bases de données SQL (Structured Query Language).

Une requête mal formulée envoyée par votre programme à une base de données déclenchera une SQLException.

```
public class SQLException extends Exception
                          implements Iterable<Throwable> { ... }
```

33. Erreurs Système (System Errors)

Error : Seconde classe fille de la classe Throwable, permettant de représenter les erreurs externes à la JVM, ou les erreurs graves causées par un dysfonctionnement de cette dernière.

La JVM étant installée sur une machine hôte (ordinateur ou serveur), elle partage les ressources (processeur, mémoire vive et stockage) de cette dernière avec les autres processus fonctionnant dessus.

Ce qu'il faut retenir, c'est que le bon fonctionnement de la JVM exécutant vos programmes en Java dépend fortement de l'environnement dans lequel elle est installée. Il existe beaucoup de facteurs dans un système informatique qui peuvent générer une erreur dans un programme. Bien qu'il soit fondamental d'anticiper ces aléas, la capacité de l'application à gérer directement de telles erreurs est souvent limitée, voire impossible, car ce type d'erreur est au-delà de sa portée d'action. C'est cette nette distinction entre les erreurs internes et externes à un programme qui a entraîné la création des deux sous-classes principales (Exception et Error) de la classe générale Throwable.

```
public class Error extends Throwable { ... }
```

OutOfMemoryError : Erreur qui se produit lorsque la JVM (Java Virtual Machine) ne dispose pas d'une quantité suffisante de mémoire vive pour faire fonctionner un programme.

Il faut veiller à ce que ce manque de mémoire vive ne soit pas dû à son propre programme, potentiellement trop gourmand.

```
public class OutOfMemoryError extends VirtualMachineError { ... }
```

StackOverflowError : Erreur qui se produit lorsqu'une récursivité excessive (et souvent sans fin) est exécutée.

Une fonction récursive mal implémentée (sans condition d'arrêt) résulte très souvent d'une StackOverflowError.

```
public class StackOverflowError extends VirtualMachineError { ... }
```

34. Ressources Gérables (Manageable Resources)

AutoCloseable : Interface déclarant une unique méthode dont la redéfinition (Override) devra être destinée à la libération des ressources utilisées par une instance.

Pour accéder (lecture) aux données, ou modifier (écriture) les données d'un fichier dans un programme, il faudra "ouvrir" un flux numérique vers ce dernier, permettant donc de faire transiter les données (en lecture, ou en écriture). Ce flux, qui consomme de la mémoire vive, nécessitera d'être fermé après exécution, pour libérer les ressources utilisées par cette "connexion" au fichier.

Un exemple d'application réelle est la classe `BufferedWriter`: elle ouvre un flux d'écriture vers un fichier cible, et nécessite donc d'être fermée afin de libérer l'espace mémoire.
Lors de la fermeture (close) d'une instance de cette classe:

- *Elle s'assure que toutes les données en attente (`Buffer`) d'écriture soient bien écrites dans le fichier, ou supprimées.*
- *Elle libère les ressources relatives au flux de connexion vers le fichier, avec par exemple l'identificateur de fichier.*
- *Elle marque l'objet comme fermé, pour qu'il déclenche (`throw`) une `IOException` si on tente de l'utiliser à nouveau.*

```
public interface AutoCloseable {

    void close() throws Exception;

}
```

Closeable : Interface fille de AutoCloseable, spécifique aux flux d'entrée / sortie, déclarant l'utilisation spécifique du type IOException plutôt qu'une simple Exception générale.

On l'utilise implicitement au travers de certaines classes comme `BufferedReader` dans un `try-with-resources`.

```
public interface Closeable extends AutoCloseable {

    void close() throws IOException;

}
```

Entrée / Sortie 🔁
[Input / Output]

35. Scanner (Scanner)

Scanner : Classe permettant de lire les données de différents types d'entrées comme l'entrée standard (Console IDE), un fichier (File) ou une simple chaîne de caractères (String).

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

System.out.print("Entrez votre nom : ");
String nom = scanner.nextLine(); // L'utilisateur écrit "JirAWS"

System.out.println("Bonjour, " + nom);

System.out.print("Entrez votre âge : ");
int age = scanner.nextInt(); // L'utilisateur écrit "30"

System.out.println("Vous avez " + age + " ans.");

System.out.print("Entrez votre année de naissance : ");

// Erreur volontaire : une chaîne non-convertible en nombre est entrée
int annee = scanner.nextInt(); // L'utilisateur écrit "AZERTY"

// Lignes ci-dessous ignorées à cause de l'Exception levée ci-dessus
System.out.println("Vous êtes né en " + annee);

scanner.close(); // Fuite de mémoire ! Ressource non fermée.
```

```
// Résultat (affichage) dans la console
Entrez votre nom : "JirAWS"
Bonjour, JirAWS
Entrez votre âge : "30"
Vous avez 30 ans.
Entrez votre année de naissance : "AZERTY"
Exception in thread "main" java.util.InputMismatchException
      at java.base/java.util.Scanner.throwFor(Scanner.java:947)
      at java.base/java.util.Scanner.next(Scanner.java:1602)
      at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
      at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
      at Main.main(Main.java:18)
```

36. Fichiers (Files)

File : Classe permettant de représenter sous la forme d'objet un fichier ou un dossier (répertoire) dans le système de la machine (ordinateur ou serveur) exécutant le programme.

Cette classe permet de manipuler des fichiers dans un programme, et notamment de lire et/ou écrire des données dans ces derniers à l'aide des classes FileReader et FileWriter.

```
/**
 * On admet ici qu'un fichier javacore_masterguide.txt se trouve bien
 * dans le dossier jiraws, lui-même dans le dossier home du système.
 */
File javaCoreGuide = new File("/home/jiraws/javacore_masterguide.txt");
```

Path : Classe permettant de représenter un chemin (path) vers une ressource dans le système (fichier ou dossier).

Bien que les chemins puissent être directement représentés avec du simple texte (String), la classe Path offre une approche beaucoup plus sécurisée, plus robuste et plus adaptée aux opérations sur les systèmes de fichiers. Elle offre notamment des méthodes (méthode) de recherche et de filtrage qui sont primordiales pour une application.

```
Path chemin = Paths.get("/home/jiraws/javacore_masterguide.txt");

System.out.println("Chemin d'accès : " + chemin);

/**
 * Une fois l'instance de Path créée, on peut directement accéder
 * au fichier vers lequel cette dernière pointe dans le système.
 */
File javaCoreGuide = chemin.toFile();
```

```
// Résultat (affichage) dans la console
Chemin d'accès : "/home/jiraws/javacore_masterguide.txt"
```

FileReader : Classe permettant de lire le contenu textuel d'un fichier (File), caractère (char) par caractère.

Offre un accès très simple à la lecture du contenu d'un fichier, mais consomme beaucoup de ressources à cause de la répétition d'opérations internes. Un FileReader va lire les caractères du contenu un par un, ce qui implique une forte répétition des opérations qui peuvent être optimisées avec un BufferedReader.

Malgré son manque d'optimisation, cette classe reste pertinente pour les besoins de lecture non fréquents. Si une application nécessite la lecture d'un fichier occasionnellement, l'impact sur les performances sera négligeable en comparaison d'une application ayant des besoins fréquents (plusieurs dizaines de fois par minute).

```
String path = "/home/jiraws/javacore_masterguide.txt";

/**
 * Utilisation d'un try-with-resources pour fermer automatiquement
 * le FileReader (reader) qui est de type Closeable.
 */
try (FileReader reader = new FileReader(path)) {

    int i;

    /**
     * Tant qu'il y a un caractère à lire dans le fichier, on exécute
     * la méthode .read() qui lira le prochain caractère
     *
     * Si le fichier contient une unique ligne "Bonjour", cette dernière
     * sera lue séquentiellement : 'B' 'o' 'n' 'j' 'o' 'u' 'r'
     */
    while ((i = reader.read()) != -1) {
        System.out.print((char) i);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

BufferedReader : Classe permettant de faciliter la lecture du contenu textuel d'un fichier (File), ligne par ligne, à partir d'un flux d'entrée comme un FileReader.

Le mise en tampon (Buffering), appliquée à la lecture de fichier, réduit le nombre d'opérations d'entrée/sortie en lisant les caractères par paquets plutôt qu'individuellement.
Cette approche limite grandement les interactions avec le système de stockage (Disque Dur / SSD).

```
String path = "/home/jiraws/javacore_masterguide.txt";

/**
 * Utilisation d'un try-with-resources pour fermer automatiquement
 * le BufferedReader (br) et le FileReader qui sont de type Closeable.
 /
try (BufferedReader br = new BufferedReader(new FileReader(path))) {

    int compteurLigne = 0;
    String ligne;

    /**
     * Tant qu'il y a une ligne à lire dans le fichier, on exécute
     * la méthode .readLine() qui lira la prochaine ligne
     *
     * Pour cet exemple, on admettra que le fichier lu contienne les
     * lignes suivantes : "Bonjour", "Bonne lecture !" et "Au revoir"
     */
    while ((ligne = br.readLine()) != null) {
        compteurLigne++;
        System.out.println("Ligne N°" + compteurLigne + " : " + ligne);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

```
// Résultat (affichage) dans la console
Ligne N°1 : "Bonjour"
Ligne N°2 : "Bonne lecture !"
Ligne N°3 : "Au revoir"
```

FileWriter : Classe permettant d'écrire du texte, caractère (char) par caractère, dans un fichier (File).

Offre un accès très simple à l'écriture de contenu dans un fichier, mais partage les mêmes enjeux de performances que FileReader.

```
String path = "/home/jiraws/javacore_masterguide.txt";

/**
 * Utilisation d'un try-with-resources pour fermer automatiquement
 * le FileWriter (fileWriter) qui est de type Closeable.
 /
try (FileWriter fileWriter = new FileWriter(path)) {

    // Instruction écrivant directement dans le fichier
    fileWriter.write("Hello JirAWS!");

} catch (IOException e) {
    e.printStackTrace();
}
```

BufferedWriter : Classe permettant de faciliter l'écriture de contenu textuel d'un fichier (File), ligne par ligne, à partir d'un flux de sortie comme un FileWriter.

Utilise le même principe de mise en tampon (Buffering) que le BufferedReader, permettant ainsi de limiter le nombre d'interactions avec le système de stockage (Disque Dur / SSD).

```
String path = "/home/jiraws/javacore_masterguide.txt";

/**
 * Utilisation d'un try-with-resources pour fermer automatiquement
 * le BufferedWriter (bw) et le FileWriter qui sont de type Closeable.
 /
try (BufferedWriter bw = new BufferedWriter(new FileWriter(path))) {

    bw.write("Welcome to JavaCore!");
    bw.newLine(); // Ajoute une nouvelle ligne
    bw.write("Goodbye, JirAWS!");

} catch (IOException e) {
    e.printStackTrace();
}
```

Avancé 🚀
[Advanced]

37. Programmation Fonctionnelle (Functional Programming)

Lambda Function (Fonction Lambda) : Mécanisme permettant de créer et d'utiliser une fonction dite "anonyme", c'est-à-dire une fonction définie "à la volée" n'ayant pas été déclarée (déclaration) explicitement au préalable dans le code.

Les fonctions Lambda perturbent souvent les développeurs débutants, notamment à cause du terme "anonyme". Les fonctions Lambda ne sont pourtant que des fonctions comme les autres, partageant le même but final qui est d'exécuter du code.

On les utilise souvent sans savoir qu'elles portent ce nom, comme par exemple avec l'interface Stream<E>.

```
List<Integer> integerList = new ArrayList<>();

// Ajout d'éléments
integerList.add(200);
integerList.add(25);
integerList.add(36);

System.out.println("Liste après les ajouts : " + integerList);

/**
 * 1. (stream)  Transformation de la liste en stream (flux)
 * 2. (forEach) Utilisation d'une expression Lambda pour itérer sur
 *              chaque élément du stream. Le bloc de code entre les
 *              parenthèses est une fonction lambda possédant un
 *              unique paramètre nommé ici "entier"
 */
integerList.stream()
           .forEach(entier -> {
               System.out.println("Entier actuel : " + entier);
           });
```

```
// Résultat (affichage) dans la console
Liste après les ajouts  : [200, 25, 36]
Entier actuel : 200
Entier actuel : 25
Entier actuel : 36
```

38. Fils d'Exécution (Threads)

Runnable : Interface déclarant une unique méthode "run()" dont l'implémentation sera exécutée par un thread (Thread).

*C'est l'élément clé de l'exécution en parallèle (**parallélisme**) en programmation Java. Un thread ne pourra lancer une exécution parallèle d'une classe que si cette dernière implémente (**implements**) Runnable.*

Le langage Java n'autorisant pas l'héritage de plusieurs classes (également appelé "héritage multiple"), cette interface pourra être implémentée par les classes ayant déjà hérité, et ne pouvant donc pas hériter de la classe Thread.

```java
@FunctionalInterface
public interface Runnable {

    // Unique méthode déclarée par l'interface Runnable.
    void run();

}
```

```java
public class MaTache implements Runnable {

    // Redéfinition de la méthode run() déclarée dans Runnable
    @Override
    public void run() {
        System.out.println("Code exécuté dans un thread séparé.");
    }

    public static void main(String[] args) {

        // Instanciation d'un Thread pour exécuter notre Classe Runnable
        Thread thread = new Thread(new MaTache());

        // Démarre l'exécution du thread et appel la méthode "run()"
        thread.start();
    }

}
```

```
// Résultat (affichage) dans la console
Code exécuté dans un thread séparé.
```

Thread : Classe permettant de gérer les exécutions en parallèle (parallélisme), aussi appelées “multi-tâches” ou “multi-thread”.

Un thread en Java ne peut exécuter que le code d’une classe ayant implémenté (implements) l’interface Runnable.

La classe Thread implémentant elle-même l’interface Runnable, une classe héritant de Thread devra redéfinir la méthode Runnable.run().

Chaque Thread créé peut exécuter un programme spécifique, permettant ainsi d’accélérer certains traitements (recherches, sauvegarde, etc.).

Plus la machine exécutant le programme est puissante, plus vous pourrez créer de threads sans concession de performance.

```
/**
 * La Classe Thread implémente l’interface Runnable.
 * Les classes héritant de Thread devront redéfinir “Runnable.run()”.
 */
public class Thread implements Runnable { … }
```

```
public class MaTache extends Thread {

    // Redéfinition de la méthode run() déclarée dans Runnable
    @Override
    public void run() {
        System.out.println("Code exécuté dans un thread séparé.");
    }

    public static void main(String[] args) {

        // Instanciation d'un Thread pour exécuter notre Classe Runnable
        Thread thread = new Thread(new MaTache());

        // Démarre l'exécution du thread et appel de la méthode "run()"
        thread.start();
    }

}
```

```
// Résultat (affichage) dans la console
Code exécuté dans un thread séparé.
```

volatile : Modificateur garantissant que les changements effectués sur la valeur d'une variable soient bien visibles (et considérés) par les différents threads (Thread) en cours d'exécution.

Dans un contexte multi-threads (avec plusieurs threads), chaque thread possède un petit emplacement mémoire dédié lui permettant de manipuler ses données et ainsi d'assurer son bon fonctionnement. Si un thread modifie la valeur d'une variable dans son espace mémoire dédié, les autres threads n'auront pas accès "immédiatement" à cette nouvelle valeur car il faudra attendre que la mise à jour ait bien été propagée dans chacun d'entre eux. Le risque est donc qu'un thread qui n'a pas encore été mis à jour utilise l'ancienne version de cette valeur, ce qui a de fortes chances de générer une erreur.

Le modificateur "volatile" est donc là pour garantir que la valeur de la variable soit stockée dans un emplacement mémoire général (partagé par tous les threads), assurant ainsi qu'ils accèdent tous à la même valeur, immédiatement et donc sans problème de synchronisation.

```
volatile boolean executionEnCours = true;
```

Pour comprendre l'intérêt du modificateur "volatile", on peut essayer d'utiliser une analogie dans le réel :

- *Cas **sans** l'utilisation de "volatile" : Vous travaillez sur un projet avec une équipe de 10 développeurs (10 threads). Votre manager, le directeur technique, vient vous transmettre une directive importante qu'il ne communiquera pas à vos collègues à cause de sa charge de travail importante. Vous êtes alors le seul détenteur de la dernière valeur de cette information, que les autres développeurs (threads) ne pourront donc pas considérer, ce qui entraînera inévitablement un problème de synchronisation entre les différents membres de l'équipe.*
- *Cas **avec** l'utilisation de "volatile" : Vous travaillez sur un projet avec une équipe de 10 développeurs (10 threads). Votre manager, le directeur technique, vient transmettre une information (variable) importante en faisant une annonce générale aux 10 développeurs de l'équipe. Tous les développeurs sont alors détenteurs de la dernière valeur de cette information, empêchant ainsi les problèmes de synchronisation entre les différents membres de l'équipe.*

synchronized : Modificateur garantissant qu'une méthode ou qu'un bloc de code ne puisse être exécuté que par un seul thread (Thread) à la fois, parmi tous ceux en cours d'exécution.

Il permet donc d'empêcher les exécutions simultanées, aussi appelées "exécutions concurrentes", évitant ainsi toutes les problématiques liées à la synchronisation.

Son utilisation est particulièrement pertinente pour éviter qu'un morceau de code (bloc de code, ou méthode entière) modifiant des informations importantes ne puisse être exécuté par plusieurs threads en même temps, ce qui entraînerait tôt ou tard des conflits.

En interne, un morceau de code (bloc de code, ou méthode entière) indiqué comme "synchronized" possède un unique verrou de sécurité qui ne pourra être accaparé que par un unique thread. Il pourra alors exécuter le code avec l'assurance d'être le seul à y avoir accès, tant qu'il aura ce verrou en sa possession. Une fois son travail terminé, le thread libérera le verrou qui sera alors disponible pour le prochain thread qui attendait de pouvoir exécuter ce code. Il faut donc en comprendre les avantages (sécurité), mais également les potentiels inconvénients (performances): le modificateur "synchronized" et son système d'exécution exclusif peuvent créer une "liste d'attente" de threads attendant pour leur tour.

```
public synchronized void methodeSynchronisee() {

    // Méthode protégée contre les exécutions concurrentes

    synchronized(this) {

        // Bloc de code protégé contre les exécutions concurrentes

    }

}
```

39. Annotations (Annotations)

Annotation : Mécanisme utilisant le symbole @ et permettant d'associer une ou plusieurs métadonnées à des éléments de code (classe, méthode, variable, etc.) dans le but d'informer le compilateur ou d'influencer le comportement de l'application.

L'utilisation d'annotations peut aller du simple marqueur, permettant de détecter les éléments dans un programme, jusqu'à l'implémentation d'une réelle logique programmatique avec des annotations possédant un ou plusieurs paramètre(s) et exécutant du code.

L'annotation @Override (redéfinition de méthode) informe le compilateur que l'implémentation de la méthode annotée remplace celle transmise au travers de l'héritage. Si une méthode est annotée avec @Override alors qu'elle n'existe pas dans une classe mère ou dans une interface, une erreur de compilation sera explicitement indiquée au développeur. Il pourra alors chercher la cause de cette erreur, pouvant être une simple faute de frappe dans le nom de la méthode, ou parfois même une tentative de redéfinition dans le mauvais fichier.

40. Réflexion (Reflection)

Reflection (Réflexion) : Permet à un programme d'examiner et de modifier son propre code à l'exécution.

C'est un concept avancé en programmation, qui est très abstrait pour les débutants mais qu'il est bon de connaître si on le rencontre.

```
try {

    // Créer une instance de la classe MyClass via la réflexion
    Class<?> myClass = Class.forName("MyClass");
    Object myObject = myClass.getDeclaredConstructor().newInstance();

    // Appelle la méthode helloWorld de MyClass via la réflexion
    Method method = myClass.getMethod("helloWorld");
    method.invoke(myObject);

} catch (Exception e) {
    e.printStackTrace();
}
```

41. Sérialisation (Serialization)

Serializable : Interface marquant une classe dont les instances (objet) peuvent être sérialisées. *Aucune méthode n'est requise pour implémenter cette interface, elle sert principalement de marqueur.*

```
public interface Serializable {

    /**
     * Cette interface ne déclare aucune méthode
     * car elle ne fait office que de “marqueur”
     * qui sera considéré par le langage Java.
     */

}
```

transient : Modificateur permettant d'ignorer un attribut durant le processus de sérialisation lorsqu'on ne souhaite pas le transmettre.

Exemple : Un mot de passe sera souvent exclu de la sérialisation.

Permet également d'écarter les données inutiles à l'envoi et ainsi réduire la taille du résultat à transférer.

```
public class Utilisateur implements Serializable {

    private String email;
    private transient String motDePasse;

}
```

Sérialisation : Processus de conversion d'un objet dans un format universel (comme JSON, XML, etc.) pouvant être transmis à une autre application. *Processus souvent utilisé sur Internet lorsque plusieurs applications développées différemment doivent communiquer ensemble, ayant donc recours à des formats de données de transition.*

Désérialisation : Processus inverse de la sérialisation, impliquant la reconstruction d'objet(s) à partir de données au format universel (comme JSON, XML, etc.). *Requiert une étape d'analyse (parsing) des données sérialisées afin de reconstituer correctement l'objet.*

Concepts et Principes de Développement 💡
[Development Concepts and Principles]

42. Bonnes Pratiques (Good Practices)

Cache : Espace mémoire d'une application dans lequel sont stockés temporairement des résultats d'opérations afin qu'elle n'ait pas à refaire le calcul si on lui redemande la même chose. *C'est un des premiers facteurs d'optimisation d'une application (en dehors de l'optimisation du code de cette dernière) et qui est utilisé par la quasi-totalité des solutions informatiques.*

Design Patterns (Patrons de Conception) : Solutions réutilisables pour des problèmes souvent rencontrés dans la conception logicielle.
Ils sont généralement divisés en trois catégories principales:

- *Design Patterns de Création : Singleton, Factory, Builder…*
- *Design Patterns Structurels : Composite, Bridge, Adapter…*
- *Design Patterns Comportementaux : Iterator, State, Command…*

L'apprentissage au moins théorique des Design Patterns est un très bon moyen d'améliorer son niveau de conception en développement.
Une certaine maîtrise du domaine est cependant nécessaire pour s'assurer de la bonne compréhension des tenants et des aboutissants.

Metadata (Métadonnées) : Donnée qui fournit des informations sur d'autres données. *Des informations comme l'auteur, le titre et la durée d'un fichier vidéo sont des métadonnées de ce dernier.*

Refactoring (Refactor) : Processus consistant à réorganiser le code source d'un programme dans le but d'améliorer sa lisibilité, sa maintenabilité et sa structure. *Implique souvent une réécriture complète du programme lorsqu'il a été mal conçu initialement ou que les besoins ont trop évolué avec le temps.*

Versioning (Gestion des Versions) : Principe visant à considérer chaque modification d'un projet de développement comme une nouvelle version de ce dernier, permettant ainsi de séparer logiquement et chronologiquement les différents ajouts de code. *Un principe indispensable pour le développement logiciel, utilisé par l'intégralité (sans exception ?) des projets commercialisés.*

43. Principes de Conception (Design Principles)

DRY (Don't Repeat Yourself) : Principe visant à réduire les répétitions de code. *La redondance (répétitions) d'une logique dans votre code devrait être factorisée avec des fonctions (`fonction`). C'est un grand principe qui doit guider les développeurs durant la conception de leurs programmes. Concevoir un programme en essayant au maximum de prévoir et ainsi d'éviter les répétitions de code aura, dans la plupart des cas, un impact très positif sur la cohérence de l'architecture finale de l'application.*

KISS (Keep It Simple, Stupid) : Principe de conception encourageant la simplicité et l'évitement de la complexité inutile. *Les solutions simples et efficaces sont souvent meilleures sur le long terme que les solutions complexes. Les logiciels complexes ne sont en fait souvent que des ensembles de petites fonctionnalités simples qui forment un tout compliqué. Plus le développeur est expérimenté, plus il aura tendance à utiliser le principe KISS et à le conseiller aux développeurs en devenir.*

YAGNI (You Aren't Gonna Need It) : Principe de développement qui recommande de ne pas ajouter une fonctionnalité avant qu'elle ne soit réellement nécessaire. *Est-ce que vous pensez que des gens auront besoin d'une fonctionnalité, ou est-ce que vous en êtes sûr ? Sur quoi vous appuyez-vous pour le justifier ? Les comportements des utilisateurs sont parfois inattendus, voire surprenants. Certaines fonctionnalités qui semblaient évidentes peuvent se retrouver totalement inutilisées par les utilisateurs finaux de l'application.*

Les développeurs en devenir sont souvent enthousiastes et se retrouvent à chasser plusieurs lapins en même temps. Si l'enthousiasme est un très bon signe, il faut cependant garder la tête froide. Chacune des fonctionnalités ajoutées est une complexité apportée dans l'application. Elles auront besoin d'être maintenues, améliorées et présentent toutes un potentiel risque d'erreur qui pourrait entraîner une consommation de temps pour les corriger.

SOLID : Ensemble de cinq principes de conception en Programmation Orientée Objet qui visent à rendre les logiciels plus flexibles, compréhensibles et maintenables.

Il est important de bien considérer que l'assimilation de ces cinq grands principes requiert une certaine expérience en développement.

Vous trouverez ci-dessous de brèves explications de chacun d'entre eux, qui vous sembleront très probablement vagues pour le moment. L'important est de garder les grandes idées en tête, pour être capable de les reconnaître quand vous les rencontrerez.

- *S - Single Responsibility Principle (SRP): Un élément ne doit avoir qu'un seul but. Une classe Voiture ne doit servir qu'à représenter et manipuler des "voitures", et ne doit donc contenir que les données et les comportements liés à ce concept.*
- *O - Open/Closed Principle (OCP): Les entités logicielles (classe, fonction, module, etc.) doivent être ouvertes à l'extension, mais fermées à la modification. L'ajout d'une nouvelle fonctionnalité ne doit pas entraîner une modification du code déjà existant, sauf dans le cas d'un refactor.*
- *L - Liskov Substitution Principle (LSP): Les instances (objet) d'une classe mère doivent pouvoir être remplacées par des instances d'une classe fille sans affecter le programme.*
- *I - Interface Segregation Principle (ISP): Une classe implémentant (implements) une interface ne doit pas se retrouver avec des méthodes héritées qui ne lui serviront pas. Mieux vaut faire plusieurs petites interfaces spécifiques qu'une seule grande générale qui ne conviendra pas à tout le monde.*
- *D - Dependency Inversion Principle (DIP): Les modules de haut niveau (logiques globales) ne devraient pas dépendre des modules de bas niveau (solutions techniques). Quand on veut construire une maison, on essaie au maximum de ne pas dépendre des matériaux : on commence par un plan abstrait de ce qu'on veut (logique globale), qu'on viendra adapter ensuite avec les matériaux (solutions techniques).*

Glossaire Général de la Programmation 📚
[General Programming Glossary]

Algorithme : Suite finie et ordonnée d'instructions (instruction) dans un programme qui sera exécuté pour résoudre un problème.

Le mot algorithme fait souvent peur car il est associé aux morceaux de code à la complexité élevée. Il n'en est pourtant rien: un code très simple et facilement compréhensible est également un algorithme.

API (Application Programming Interface) : Intermédiaire entre deux systèmes qui prend souvent la forme de fonctionnalités accessibles publiquement et permettant d'utiliser un service.

L'interface Stream<E> en Java est une API qui, au travers de ses méthodes très simples d'accès, permet de réaliser des opérations complexes sur les structures de données (Collection, List, Set, etc.).

Lorsqu'on se connecte à une application en utilisant un compte Google, Facebook, Microsoft, ou d'une autre compagnie, l'application concernée utilise une API d'authentification utilisateur mise à disposition par l'entreprise concernée pour "déléguer" ce travail (et ses enjeux).

Boilerplate : Terme faisant référence aux morceaux de code basiques mais nécessaires d'une implémentation et dont la logique est redondante, si ce n'est ennuyeuse, pour les développeurs.

Dans le cadre du principe d'Encapsulation en Java, l'implémentation des accesseurs (Getter) et des mutateurs (Setter) peut être considérée comme du code boilerplate, car ils sont présents dans chaque classe, et dans la majorité des cas leur intérêt n'est pas aussi important que le voudrait le principe d'Encapsulation.

Bibliothèque (Librairie) : Regroupement d'éléments (fonction, classe, interface, enum, etc.) répondant à des besoins spécifiques et que les développeurs peuvent utiliser.

En Java, la librairie Apache Commons propose un ensemble de solutions (fonction) répondant à toutes sortes de problématiques communes au développement, allant de la manipulation de chaînes de caractères (String) à la gestion de fichiers (File), en passant par des opérations mathématiques spécifiques et bien d'autres choses encore.

Framework : Semblable à une bibliothèque, mais qui impose un système de fonctionnement propre aux besoins auxquels il répond.

Le but final d'un framework est de créer un "cadre" de travail, c'est-à-dire un environnement de travail optimal pour répondre à des besoins, mais imposant certaines règles de fonctionnement.

Le framework Java Spring est, comme pour une bibliothèque, un regroupement d'éléments (fonction, classe, interface, enum, etc.) que les développeurs peuvent utiliser pour réaliser des applications web.

À la différence d'une bibliothèque, les fonctionnalités du framework Java Spring imposeront un mode de fonctionnement, avec des attentes spécifiques d'implémentations et une certaine direction quant à l'architecture de l'application.

Buffer (Tampon) : Zone temporaire de stockage en mémoire utilisée pour stocker des données en attendant qu'elles soient traitées.

Dans une entreprise de livraison qui possède des entrepôts de stockage de marchandises, on peut voir les préparateurs de commandes comme un exemple réel de "buffer" dans la vraie vie. Sans ces derniers, les livreurs devraient aller chercher eux-mêmes les marchandises à livrer dans l'entrepôt, ce qui leur ferait perdre un temps précieux.

Compilation : Étape de traduction d'un code source écrit par un développeur en un code plus facilement compréhensible, et donc plus efficacement exécutable, par une machine.

Même si la simple installation du JRE sur une machine peut lui permettre de comprendre et d'exécuter un programme écrit en Java, rien ne vaudra les performances obtenues dans son langage natif.

Interprétation : Exécution d'un programme sans compilation au préalable. Les instructions sont traduites (interprétées) en un langage machine seulement au moment de leur exécution.

Autrement dit, on peut dire qu'un programme interprété sera lu comme si c'était la première fois à chaque exécution, tandis qu'un programme compilé sera "appris par cœur" pour être plus efficace.

On constate une nette différence de performance en faveur des langages compilés, mais l'interprétation possède également ses avantages, offrant notamment une meilleure flexibilité et un débogage rapide.

Chiffrement (Encryption) : Conversion d'informations en données illisibles pour préserver leur confidentialité. Elles pourront être déchiffrées, et donc retransformées, dans leur état d'origine.

Les sites Internet utilisant HTTPS chiffrent les informations de vos échanges avec les serveurs web, évitant ainsi que ces dernières ne puissent être lues durant leur transport entre vous et les serveurs.

Hachage (Hash) : Conversion d'informations similaire au chiffrement, sauf qu'il n'y a pas de retour en arrière (déchiffrement) possible.

Les mots de passe, pour être stockés en base de données, ne sont pas simplement chiffrés, au risque qu'une faille ne les expose et qu'ils puissent être déchiffrés derrière. Ils seront plutôt hachés (hashed), ce qui assure que même en cas de faille exploitable, l'information soit pratiquement inutilisable par les pirates.

Débogage (Debugging) : Processus durant lequel un développeur essaie d'identifier la ou les sources d'une erreur (bug / bogue) dans un programme afin d'appliquer une correction.

Une compétence clé du métier de développeur, qui est source de blocages et de frustrations lorsqu'elle n'est pas maîtrisée.

Déploiement : Mise à disposition d'une application aux utilisateurs.

Le déploiement d'une application web consiste à l'héberger dans un serveur accessible sur Internet, impliquant donc plusieurs étapes comme la configuration de l'environnement d'exécution ou encore l'installation du code source de l'application.

Dépréciation (Deprecation) : Processus de marquage d'un élément comme obsolète et destiné à ne plus être utilisé dans le futur.

L'annotation @Deprecated sur un élément en Java sert d'avertissement aux développeurs, impliquant alors qu'ils doivent cesser de l'utiliser en faveur d'une alternative ajoutée plus récemment.

Intégration Continue (Continuous Integration) : Action d'intégrer fréquemment le travail d'une équipe de développement à un projet.

En général, au moins une fois par jour, en utilisant un outil de gestion des versions (Versioning) comme Git. Chaque intégration est automatiquement vérifiée par la construction du projet et l'exécution de tests, ce qui permet de détecter rapidement les erreurs.

IDE (Integrated Development Environment) : "Environnement de Développement Intégré", soit un logiciel fournissant un ensemble complet d'outils pour assister les développeurs dans leur métier.

Il suffit d'essayer l'expérience d'écrire un programme sur une application de traitement de texte basique comme Bloc-notes sur Windows pour se rendre compte de l'immense rôle que joue un IDE.

Instruction : Ligne de code demandant à la machine d'effectuer une opération spécifique. Se termine par un ; (point-virgule) en Java.

À la différence de son cousin Python dans lequel un saut de ligne indique la fin d'une instruction, les sauts de ligne en Java n'ont pour but que de rendre le code plus lisible pour les humains. Seul le ; est considéré comme une fin d'instruction pour le langage.

Implémentation : Résultat de l'action d'implémenter, c'est-à-dire le fait d'écrire du code. *Un développeur implémente des solutions pour répondre à des besoins. Une méthode contient une implémentation répondant à un besoin spécifique.*

Middleware (Intergiciel) : Système qui se positionne entre des applications ou des logiciels. *En passant une commande sur Amazon, on peut dire qu'Amazon se positionne en tant que "middleware" entre vous et le magasin qui vend le produit désiré.*

Overflow (Dépassement) : Situation problématique qui survient lorsqu'un dépassement de capacité (de "taille") est rencontré.

*En Java, le résultat de l'opération arithmétique 50 * 3 (= 150) générerait un dépassement (Overflow) s'il était affecté à une variable de type byte, car ce type ne peut recevoir que les valeurs comprises dans l'intervalle allant de -128 jusqu'à +127.*

Parallélisme : Exécution de plusieurs opérations en simultané, en parallèle. *Répartir le travail dans une équipe revient à paralléliser les tâches à accomplir pour arriver plus rapidement au but.*

Récursion (Récursivité) : Technique de programmation où une fonction s'appelle elle-même. *C'est une logique qui s'apparente à une boucle: l'exécution d'un bloc de code est répétée tant qu'une condition n'est pas remplie. Souvent utilisée pour parcourir un arbre d'information, permettant ainsi de ne pas avoir à connaître son étendue à l'avance.*

Formats de Fichiers 📄
[File Formats]

.java (Java) : Extension pour les fichiers contenant le code source d'un programme écrit en langage Java. *C'est le format de base avec lequel tout développeur Java travaille quotidiennement.*

.class (Classe) : Extension pour les fichiers bytecode Java générés par le compilateur (compilation) à partir des fichiers .java. *Le bytecode est une forme intermédiaire du code, prêt à être exécuté par la Machine Virtuelle Java (JVM).*

.jar (Java ARchive) : Format d'archive utilisé pour distribuer un ensemble de fichiers Java comme une seule unité d'application ou de bibliothèque. *Un fichier .jar peut être intégré à un projet en tant que bibliothèque externe, ou être exécuté comme une application en utilisant le JRE via une ligne de commande.*

.properties (Properties) : Format pour les fichiers contenant des propriétés utilisées pour configurer des applications Java.
Ils stockent des paires clé-valeur (comme une Map) qui peuvent être chargées dans l'application au moment de l'exécution.

.xml (XML) : "eXtensible Markup Language" est un langage de balisage conçu pour stocker et transporter des données. *Développé et recommandé comme un standard par le World Wide Web Consortium (W3C) à la fin des années 1990, le XML a été conçu pour être à la fois humainement lisible et facilement traité par les machines.*

.json (JSON) : "JavaScript Object Notation", un format de données utilisé pour le stockage et le transfert d'informations structurées entre des systèmes distincts. *À l'origine, le format JSON a été conçu pour faciliter le transfert de données entre le serveur et le navigateur web dans des applications basées sur JavaScript. L'efficacité et la praticité de ce format ont démocratisé son utilisation, allant même jusqu'à son intégration dans des cas d'usages où le langage JavaScript n'est pas impliqué.*

Historique des Versions 📅
[Version History]

Version 1.0.0 :
- Date de publication : Mars 2024
- Change Log :
  - Première version du document - Intégration de la majorité des éléments en lien avec la programmation Java.

Version 1.8.6 :
- Date de publication : Septembre 2024
- Change Log :
  - Majeur:
    - Adaptation de la colorimétrie pour la publication du guide au format papier *(disponible sur Amazon)*.
    - Ajout d'un sommaire alphabétique détaillé pour simplifier la navigation dans le document.
    - Correction d'une erreur sur la `boucle while`.
    - Correction du code sur la réflexion (`reflection`).
    - Correction du code sur le mot-clé `synchronized`.
    - Correction du code sur la méthode `Stream.distinct()`.
    - Correction du code sur la méthode `Map.entrySet()`.
    - Correction du code sur la méthode `Collection.contains(E element)`.
  - Mineur:
    - Adaptation du mode d'emploi pour la version papier.
    - Utilisation de la classe `ArrayList` au lieu de la classe `LinkedList` dans les exemples de codes où son usage n'était pas nécessaire.
    - Corrections de fautes d'orthographe et de grammaire.
    - Améliorations de certaines définitions.
    - Améliorations de certaines tournures de phrase.
    - Améliorations de certaines traductions.