Getting started with Java

Pourquoi apprendre le Java

Java est un langage de programmation, développé par **Sun microsystems** en 1991, qui a marqué l'histoire du développement logiciel. Depuis sa première release (1.0) en **1995**, il a évolué pour devenir l'un des outils les plus puissants et polyvalents au service des développeurs.

Aujourd'hui, si vous avez utilisé des applications web, des systèmes d'entreprise, ou même des applications mobiles, il est fort probable que vous ayez interagi avec du code Java sans même vous en rendre compte.

Avec sa syntaxe claire inspirée du C et sa capacité à fonctionner sur diverses plateformes grâce à la machine virtuelle Java (JVM), Java continue de jouer un rôle clé dans le monde de la technologie. En 2024, Java reste attractif avec des millions de développeurs dans le monde et une adoption massive dans divers secteurs.

- **4ème** langage le plus utilisé en juillet 2024 selon l'index TIOBE.
  Source: Index TIOBE
- **36%** des développeurs professionnels utilisent Java.
  Source: JetBrains
- **52%** des services Web sont développés en Java.
  Source: Etat de l'écosystème des développeurs 2020

Caractéristiques de Java

Caractéristique	Description
Java est interprété	Java est compilé en bytecode, qui est interprété par la JVM, permettant une exécution sur différentes plateformes.
Java est portable	Grâce à la JVM, Java peut s'exécuter sur n'importe quelle plate-forme sans modification du code source.
Java est orienté objet	Java utilise des concepts de la programmation orientée objet tels que les classes et les objets pour structurer le code.

Caractéristique	Description
Java est simple	La syntaxe de Java est conçue pour être claire et facile à apprendre, inspirée du C mais sans ses complexités.
Java est fortement typé	Java impose des règles strictes sur les types de données, réduisant les erreurs et augmentant la sécurité du code.
Java assure la gestion de la mémoire	La gestion de la mémoire est automatisée par le ramasse-miettes (garbage collector) qui libère la mémoire inutilisée.

Caractéristique	Description
Java est sûr	Java inclut des mécanismes de sécurité tels que le contrôle d'accès aux ressources, la gestion des exceptions, et le modèle de sécurité basé sur des permissions. Par exemple, le gestionnaire de sécurité de Java (Security Manager) contrôle l'accès aux fichiers et aux réseaux.
Java est économe	Java est conçu pour une exécution efficace avec une gestion optimisée des ressources et des performances adaptées aux environnements variés.

Caractéristique	Description
Java est multitâche	Java prend en charge la programmation multithread, permettant l'exécution simultanée de plusieurs tâches.

Scope du cours

Applis et applets:

Il existe 2 types de programmes avec la version standard de Java :

- les applets,
- les applications.

Les applets sont *deprecated* depuis la version 9 de Java, ce cours ne les couvrira pas.

Les différentes éditions Java:

Les types d'applications qui peuvent être développées en Java sont nombreux et variés :

- Applications desktop
- Applications web : servlets/JSP, portlets, applets
- Applications pour appareil mobile
- Applications pour appareil embarqué
- Applications pour carte à puce
- Applications temps réel

Différentes éditions sont définies pour des cibles distinctes selon le besoin:

- Java SE: Java Standard Edition,
- Java EE: Java Enterprise Edition,
- Java ME: Java Micro Edition.

Ce cours se basera sur la **version 21** de l'édition **Java SE**.

C'est quoi Java ?

JDK : L'outil du développeur

Le **Java Development Kit (JDK)** est le kit de développement qui contient tous les outils nécessaires pour **créer** des applications Java. C'est le composant essentiel pour un développeur. Voici ce qu'il inclut :

- **javac** : Le **compilateur** Java, qui convertit le code source (.java) en **bytecode** (.class).
- Des outils pour le débogage, la documentation et la gestion des applications Java.
- Le **JRE** (Java Runtime Environment), qui permet d'exécuter les programmes.

Le JDK est donc utilisé pour **écrire** et **compiler** les programmes. Il est indispensable pour tout développeur Java, car c'est lui qui permet de passer du code source au bytecode.

JRE : L'environnement d'exécution

Le **Java Runtime Environment (JRE)** est nécessaire pour faire fonctionner les programmes Java. Il ne contient pas d'outils de développement, mais uniquement ce qui est nécessaire pour **exécuter** une application. Le JRE comprend :

- La **JVM** (Java Virtual Machine), qui exécute le bytecode.
- Des bibliothèques de classes précompilées pour que le programme puisse fonctionner.

Le JRE est ce que l'on installe sur une machine lorsqu'on veut **lancer une application Java** déjà développée. C'est l'environnement d'exécution utilisé par les utilisateurs finaux.

La JVM : Le cœur de l'exécution Java

La **Java Virtual Machine (JVM)** est ce qui permet à Java d'être un langage **indépendant de la plateforme**. En effet, la JVM est une machine virtuelle qui interprète le **bytecode** (le code généré par le compilateur) et le transforme en instructions compréhensibles par la machine hôte (Windows, Linux, MacOS, etc.).

Fonctionnement de la JVM :

1. La JVM **charge** le bytecode produit par la compilation.
2. Elle l'**interprète** ou le compile dynamiquement en **code machine** grâce à un processus appelé **compilation Just-In-Time (JIT)**, ce qui permet d'améliorer les performances d'exécution.
3. Enfin, elle exécute le programme, en s'assurant de la gestion de la mémoire (via le **garbage collector**) et de la sécurité.

Ainsi, un programme Java peut être exécuté sur n'importe quel système équipé d'une JVM, sans que le code ait besoin d'être modifié.

Gestion de la mémoire dans la JVM

La JVM gère automatiquement la mémoire pour les programmes Java, en s'occupant du processus de **gestion de la mémoire dynamique**. Cela signifie que les développeurs n'ont pas besoin de libérer manuellement la mémoire comme dans d'autres langages (ex : C/C++). La JVM utilise une technologie appelée **Garbage Collector** pour libérer la mémoire inutilisée.

Garbage Collector : Gestion automatique de la mémoire

Le **Garbage Collector (GC)** est responsable de la récupération de la mémoire occupée par les objets qui ne sont plus accessibles (c'est-à-dire, qui ne sont plus utilisés par le programme).

Limitations du Garbage Collector

Bien que le **Garbage Collector** soit un puissant mécanisme pour gérer la mémoire, il présente certaines **limites** :

- **Manque de mémoire (OutOfMemoryError)** : Si trop d'objets sont chargés dans la mémoire, ou si la mémoire allouée à la JVM (Heap) est insuffisante, le GC ne peut pas empêcher un manque de mémoire. Cela entraîne des erreurs **OutOfMemoryError**, même si le GC fait son travail.

- **Fuites de mémoire** : Le GC ne peut pas détecter les fuites de mémoire liées à des erreurs de programmation. Si un objet reste référencé quelque part alors qu'il n'est plus utilisé, le GC ne pourra pas le collecter, créant ainsi une **fuite de mémoire**.
- **Temps de pause** : Même si des efforts sont faits pour optimiser les performances, certaines applications critiques peuvent être affectées par des pauses liées à la collecte de la mémoire, surtout si la configuration du GC n'est pas adaptée.

Le Garbage Collector, bien que très utile, n'est donc pas une solution magique. Une **mauvaise gestion des références d'objets** peut toujours causer des problèmes de mémoire dans une application Java.

Processus de fonctionnement du Garbage Collector :

1. **Marquage** : Le GC identifie les objets qui ne sont plus utilisés par le programme.
2. **Libération de la mémoire** : Ces objets sont alors supprimés, et la mémoire qu'ils occupaient est libérée.
3. **Compaction (si nécessaire)** : Après avoir libéré la mémoire, le GC peut réorganiser les objets encore utilisés pour éviter la fragmentation de la mémoire.

Types de Garbage Collectors dans la JVM :

La JVM propose plusieurs algorithmes de GC, chacun ayant des objectifs différents :

- **Serial GC** : Simple et adapté pour les petites applications avec un seul thread.
- **Parallel GC** : Utilise plusieurs threads pour collecter les objets morts, augmentant ainsi la vitesse du GC sur les systèmes multi-thread.

- **G1 GC (Garbage First)** : Collecte en priorité les zones où il y a le plus d'objets non utilisés. C'est un GC adapté pour des applications nécessitant une gestion plus prédictive de la mémoire.
- **ZGC** et **Shenandoah** : GCs qui minimisent les temps de pause même dans les applications avec de très grands volumes de mémoire (tuning avancé).

Les espaces mémoire dans la JVM :

La mémoire utilisée par la JVM est divisée en plusieurs zones :

- **Heap** : C'est la mémoire principale où sont stockés les objets créés par l'application. La majorité des opérations du Garbage Collector se concentre sur cet espace.
- **Stack** : Chaque thread a son propre espace Stack où sont stockées les variables locales et les appels de méthodes.
- **Method Area** : Stocke les informations sur les classes, comme le bytecode, les constantes et les variables statiques.

Le processus de compilation en Java

Le processus de création d'un programme Java passe par plusieurs étapes :

1. **Écriture du code source** : Le développeur écrit le code dans un fichier `.java`. Ce code est lisible et compréhensible par un humain.

```java
// Exemple d'une classe Java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

2. **Compilation** : Le compilateur (`javac`) transforme ce code source en **bytecode**, qui est un langage intermédiaire, indépendant de la machine, stocké dans des fichiers `.class` .

   Contrairement aux langages comme C ou C++, Java ne produit pas de fichier exécutable directement (comme un `.exe`), mais un bytecode qui peut être exécuté sur n'importe quelle machine dotée d'une JVM.

Exécution du code Java

L'exécution d'un programme Java se fait en deux phases principales :

1. **Chargement et interprétation** : La JVM charge le bytecode en mémoire et commence à l'interpréter. Chaque ligne du bytecode est traduite en instructions que le processeur de la machine peut comprendre.

2. **Compilation Just-In-Time (JIT)** : Pour accélérer les performances, la JVM peut **compiler certaines parties du bytecode en code natif** lors de l'exécution. Cela permet d'améliorer considérablement la vitesse des programmes Java en répétant moins de calculs.

Grâce à ce mécanisme, Java allie **portabilité** (avec le bytecode indépendant de la plateforme) et **performance** (avec la compilation JIT).

Récapitulatif

- Le **JDK** est utilisé par les développeurs pour écrire et compiler le code Java.
- Le **JRE** permet d'exécuter les applications Java.
- La **JVM** est responsable de l'exécution du bytecode, assurant l'indépendance de la plateforme et gérant les performances et la mémoire.

Ces trois composants travaillent ensemble pour permettre à Java d'être un langage robuste, sécurisé et surtout portable, utilisé dans une multitude de domaines, des applications d'entreprise aux systèmes embarqués.

Les Environnements de Développement Intégré (IDE) pour Java

Un **IDE (Integrated Development Environment)** est un logiciel qui fournit des outils pour faciliter le développement de programmes en Java. Les IDE sont largement utilisés par les développeurs pour écrire, tester, et déboguer leur code. Voici les IDE les plus populaires pour Java .

1. IntelliJ IDEA

- **Caractéristiques** :
  - Puissant, avec de nombreuses fonctionnalités pour le développement Java.
  - Support avancé pour la refactorisation, le débogage, et l'intégration avec des outils de versioning (Git).
  - Versions : Communauté (gratuite) et Ultimate (payante, avec plus de fonctionnalités).

2. Eclipse

- **Caractéristiques** :
  - Gratuit et open-source.
  - Large écosystème de plugins pour ajouter des fonctionnalités.
  - Utilisé aussi bien pour des petits projets que pour des applications complexes.

3. NetBeans

- **Caractéristiques** :
  - IDE open-source développé par Apache.
  - Intégration native avec les outils de développement Java, comme les serveurs d'applications (GlassFish, Tomcat).
  - Bon support pour le débogage et la gestion de projets Java standards.

Classes

En Java, les classes sont les blocs de construction fondamentaux, définissant les caractéristiques des composants du programme. D'autres blocs incluent les interfaces, records et enums.

Pour utiliser les classes, il faut créer des objets, qui sont des instances en mémoire représentant l'état du programme. Une référence est une variable pointant vers un objet.

Éléments minimaux d'une classe Java

1. **Modificateur d'accès**
   Définit la visibilité de la classe (ex. : `public`, `private`, `protected` ou sans modificateur).

2. **Mot-clé `class`**
   Indique que l'élément défini est une classe.

3. **Nom de la classe**
   Identifiant unique de la classe, qui doit commencer par une lettre majuscule par convention.

4. **Corps de la classe**
   Contient le code de la classe, défini entre accolades `{}`.

Un exemple minimal de classe contenu dans un fichier Hello.java

```
class Hello {


}
```

- Les classes Java se composent principalement de deux éléments : les méthodes (ou fonctions) et les champs (ou variables), appelés membres de la classe. Les variables conservent l'état du programme, tandis que les méthodes manipulent cet état.

- Les développeurs doivent organiser ces éléments pour créer un code utile et compréhensible pour les autres.

Attributs et Méthodes en Java

Classe `Animal`

```java
public class Animal {
    String name; // Attribut pour stocker le nom de l'animal

    public String getName() { // Méthode pour obtenir le nom
        return name; // Retourne la valeur de l'attribut name
    }

    public void setName(String newName) { // Méthode pour définir le nom
        name = newName; // Modifie l'attribut name avec la nouvelle valeur
    }
}
```

Attributs

- **Définition** : Les attributs (ou champs) sont des variables déclarées dans une classe qui définissent l'état ou les caractéristiques d'un objet.
- **Exemple** :
  - Dans la classe `Animal`, l'attribut `name` est défini à la ligne 2 comme un `String`. Cela signifie que `name` peut contenir des valeurs textuelles, telles que "Chat" ou "Chien". L'attribut stocke des informations sur l'objet `Animal`.

Méthodes

- **Définition** : Les méthodes sont des opérations définies dans une classe, permettant d'interagir avec les attributs et de réaliser des actions. Elles manipulent l'état de l'objet.
- **Exemples** :
  - La méthode `getName()` retourne la valeur de l'attribut `name`. Elle a un type de retour `String`, indiquant qu'elle renvoie une chaîne de caractères.
  - La méthode `setName(String newName)` permet de modifier l'attribut `name`. Elle a un paramètre nommé `newName`, de type `String`, ce qui signifie qu'elle attend une chaîne de caractères comme argument, et elle ne renvoie rien (`void`).

Signature de la Méthode

- La signature de la méthode inclut le nom de la méthode et les types de ses paramètres. Par exemple, pour la méthode `numberVisitors(int month)`, le nom est `numberVisitors`, et il y a un paramètre `month` de type `int`, ce qui en fait la signature de la méthode : `numberVisitors(int)`.

Les commentaires

Types de Commentaires

- Java propose trois types de commentaires :
  - **Commentaire sur une ligne** : commence par `//` et tout le texte après sur la même ligne est ignoré par le compilateur.
  - **Commentaire multi-lignes** : commence par `/*` et se termine par `*/`. Utilisé pour écrire des commentaires sur plusieurs lignes.
  - **Commentaire Javadoc** : commence par `/**` et se termine par `*/`. Utilisé pour générer de la documentation via l'outil Javadoc.

Utilisation des Commentaires

- Les commentaires servent à rendre le code plus lisible. La Javadoc, en particulier, est utilisée pour documenter des méthodes ou des classes de façon structurée pour d'autres développeurs.

Exemple de Code avec les Trois Types de Commentaires

```java
public class Animal {
    /* Commentaire multi-lignes :
     * attribut pour stocker le nom de l'animal
     */
    // Commentaire sur une ligne : attribut pour stocker le nom de l'animal
    String name;


    public String getName() {
        return name; // Retourne la valeur de l'attribut name
    }

    /**
     * Méthode pour définir le nom
     * @param newName Le nouveau nom de l'animal
     */
    public void setName(String newName) {
        name = newName; // Modifie l'attribut name avec la nouvelle valeur
    }
```

Les fichiers .java

1. **Organisation des Classes**
   - En général, chaque classe Java est définie dans son propre fichier `.java`.
   - Une classe de haut niveau (top-level) est souvent publique, mais ce n’est pas une obligation en Java.

2. **Règles pour les Fichiers**
    - Si un fichier contient plusieurs types, seul un de ces types de haut niveau peut être public.
    - Si une classe est publique, son nom doit correspondre au nom du fichier. Par exemple, `public class Animal` doit être dans un fichier nommé `Animal.java`.

Exemples de Code

Dans `Animal.java`:

```
class Animal {
    String name; // Attribut privé pour stocker le nom
}
```

Dans `Animal.java`:

```
public class Animal {
    private String name; // Attribut privé pour stocker le nom
}

class Animal2 {
    // Une deuxième classe sans modifier d'accès public
}
```

La Méthode main()

- La méthode `main()` est le point d'entrée d'une application Java, permettant à la JVM d'exécuter le code.
- Exemple de la méthode `main()` la plus simple :

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

Commande `javac`

- **Fonction :** Utilisée pour compiler le code source Java écrit dans un fichier `.java`.
- **Syntaxe :**

```
javac NomDuFichier.java
```

- **Résultat :** Génère un fichier de bytecode avec l'extension `.class`, qui contient des instructions compréhensibles par la Java Virtual Machine (JVM).
- **Exemple :**

```
javac Hello.java
```

Commande `java`

- **Fonction :** Utilisée pour exécuter un programme Java qui a été compilé. Elle lance la JVM et exécute le bytecode.
- **Syntaxe :**

```
java NomDeLaClasse
```

- **Remarque :** N'incluez pas l'extension `.class` lors de l'exécution.
- **Exemple :**

```
java Hello
```

  - Cela exécute le programme contenu dans `Hello.class`.

Passage de paramêtres

Modification du Programme Hello:

- On modifie la classe `Hello` pour afficher les deux premiers arguments passés à la méthode `main()` :

```
public class Hello {
    public static void main(String[] args) {
        System.out.println(args[0]); // Premier argument
        System.out.println(args[1]); // Deuxième argument
    }
}
```

Nous allons de nouveau utiliser les commandes **javac** et **java** pour compiler et exécuter notre classe Hello.
Ce qui changera sera le passage de paramètres lors de l'exécution de la classe.

```
 javac Hello.java
 java Hello Hello World
 Hello
 World
```

** Quelques erreurs possibles lors de l'exécution de Hello.java:**

- Si vous exécutez la classe sans fournir suffisamment d’arguments :

```
java Hello Bonjour
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 1 out of bounds for length 1
```

- Si le fichier Hello.java n’est pas trouvé lors de la compilation:

```
javac Hello.java
error: file not found: Hello.java
```

Commande Simplifiée:

Vous pouvez exécuter votre programme directement avec la commande suivante :

```
java Hello.java Hello World!!!
Hello
World!!!
```

- Avec Compilation : Lorsque vous compilez d’abord le code avec javac, vous omettez l’extension .java lors de l’exécution.
- Sans Compilation Explicite : En utilisant java Zoo.java, vous incluez l’extension .java. Cela permet de lancer le programme sans passer par une étape de compilation explicite.

Les Packages

Packages

- Java organise les classes en **packages**, un peu comme des dossiers pour organiser des documents.
- Les packages permettent d'éviter les conflits de noms et de structurer le code de manière logique.
- Exemple : `java.util` contient des classes utilitaires comme `Random` .

Exemple d'Import

- Si vous utilisez une classe sans préciser le package, une erreur de compilation apparaît.

```java
import java.util.Random; // On précise où trouver la classe Random
public class RandomPrinter {
    public static void main(String[] args) {
        Random r = new Random();
        System.out.println(r.nextInt(10));  // Affiche un nombre entre 0 et 9
    }
}
```

Imports et wildcards

- Cela n'importe que les classes du package spécifié, pas les sous-packages.
- On peut utiliser * pour importer toutes les classes d'un package:

```
import java.util.*; // Importe toutes les classes du package java.util
```

Imports Redondants

- Certains imports sont automatiques (ex : java.lang) et d’autres peuvent être redondants si une classe est déjà importée explicitement.
- Évitez d’importer des classes du même package où se trouve votre classe, Java les détecte automatiquement.

Conflits de Nommage

Pourquoi les Conflits de Nommage ?

- Les packages permettent d'avoir plusieurs classes avec le même nom, situées dans des packages différents.
- Exemple courant : `java.util.Date` et `java.sql.Date`.

Résoudre les Conflits de Nommage

- Si une classe se trouve dans plusieurs packages, il faut préciser lequel utiliser :

```
import java.util.Date; // On choisit la classe java.util.Date

import java.util.Date; // Prend le dessus sur les imports génériques
import java.sql.*; // Importe les autres classes nécessaires

import java.util.Date; // Prend le dessus sur les imports génériques
import java.sql.*; // Importe les autres classes nécessaires
```

Utilisation de Deux Classes du Même Nom

- Si vous avez besoin des deux versions de la classe, utilisez le nom de classe complet :

```
public class Conflicts {
    java.util.Date date;
    java.sql.Date sqlDate;
}
```

- Cela permet de différencier clairement les deux classes dans votre code.

Ordre des Éléments dans une Classe

Ordre Correct des Éléments

- L'ordre des éléments dans une classe suit des règles précises. Voici l'acronyme à retenir : **PIC** (Package, Import, Class).

Élément	Exemple	Obligatoire ?	Position
Déclaration de package	`package abc;`	Non	Première ligne du fichier
Déclarations d'import	`import java.util.*;`	Non	Après la déclaration de

```java
import java.util.*;
package structure;          // ERREUR : ordre incorrect
String name;                // ERREUR : déclaration hors classe
public class Meerkat { } // ERREUR
```

Problèmes :

- La déclaration de package doit précéder les imports.
  - Les champs et méthodes doivent être à l'intérieur d'une classe.

Exemple de Classe

```java
package structure;       // Le package doit être le premier élément (hors commentaire)
import java.util.*;      // L'import vient après le package
public class Meerkat {   // Puis vient la déclaration de la classe
   double weight;        // Les champs et méthodes peuvent être dans n'importe quel ordre
   public double getWeight() {
      return weight;
   }
   double height;        // Un autre champ (pas besoin d'être groupé)
}
```

Création d'un objet

Appel des Constructeurs

- Pour créer une instance d'une classe, utilisez `new` suivi du nom de la classe et des parenthèses :

```
Car p = new Car();
```

- Park : type de l’objet.
- p : variable de référence.
- new Park() : création de l’objet.

Définition d’un Constructeur

Un constructeur ressemble à une méthode mais :

- Il porte le même nom que la classe.
- Il n’a pas de type de retour.

Si aucun constructeur n’est défini, le compilateur en génère un par défaut.

```
public class Car {
    public Car() {
        System.out.println("dans le constructeur");
    }
}
```

Initialisation des Champs

- Champs initialisés directement à la déclaration ou dans le constructeur :

```
public class Car {
    int door = 3;  // Initialisation directe
    String fuel;

    public Car(String fuel) {
        this.fuel = fuel;  // Initialisation dans le constructeur
    }
}
```

Lecture et Écriture Directe de Variables d'Instance

- Il est possible de lire et écrire directement des variables d'instance depuis le code appelant.
- Exemple : une voiture ayant un compteur de kilomètres :

```
public class Car {
    int door = 3;  // Initialisation directe
    String fuel;

    public Car(String fuel) {
        this.fuel = fuel;  // Initialisation dans le constructeur
    }
}
```

Remarque: Dans cet exemple, la méthode main() agit en tant qu'appelant et modifie directement la variable mileage de l'instance myCar.

Accès Direct aux Champs

On peut accéder directement aux valeurs des champs déjà initialisés pour en initialiser un autre :

```
public class CarModel {
 String brand = "Toyota";
 String model = "Corolla";
 String fullModel = brand + " " + model;
}
```

- Les lignes 2 et 3 écrivent dans les champs brand et model.
- La ligne 4 lit les valeurs de ces champs et les utilise pour initialiser fullModel.

Les types de Données en Java

- Les applications Java contiennent deux types de données principaux :
  - **Types primitifs**
  - **Types de référence**

Les Types Primitifs

- Java possède huit types de données primitifs.
- Chaque type a une taille spécifique en bits et un intervalle de valeurs.
- La table ci-dessous résume ces types.

Types Primitifs en Java

Mot-clé	Type	Valeur Min	Valeur Max	Valeur par défaut	Exemple
boolean	true ou false	n/a	n/a	false	true
byte	Valeur entière sur 8 bits	-128	127	0	123
	Valeur entière				80

Le Cas Spécial de `String`

- `String` n'est **pas** un type primitif.
- Java inclut un support intégré pour les littéraux de chaîne et les opérateurs, ce qui conduit à le confondre parfois avec un type primitif.

Points Clés

- Les types `byte`, `short`, `int` et `long` sont utilisés pour les valeurs entières sans virgule.
- Les types numériques utilisent le double de bits du type plus petit correspondant (ex. `short` utilise le double de bits de `byte`).
- Tous les types numériques sont **signés**, réservant un bit pour les valeurs négatives.

Littéraux en Java

- Lorsqu'un nombre est écrit dans le code, il est appelé **littéral**. Par défaut, Java le considère comme un `int`.
- Ajoutez `L` pour un `long` ou `f` pour un `float` pour indiquer explicitement le type.

Types Numériques Spécifiés

- **Octal** : Préfixé par `0` (ex: `017`).
- **Hexadécimal** : Préfixé par `0x` ou `0X` (ex: `0xFF`).
- **Binaire** : Préfixé par `0b` ou `0B` (ex: `0b10`).

Utilisation de l'Underscore

Les littéraux peuvent contenir des underscores pour plus de lisibilité :

```
int million = 1_000_000; // Lisible
```

Utilisation de l'Underscore

Règles à suivre :

- Positions autorisées : N’importe où entre les chiffres.
- Positions interdites :
- Au début ou à la fin d’un nombre.
- Avant ou après un point décimal.
- Avant le suffixe d’un nombre (comme L ou F).

Distinction entre Types Primitifs et Types Référence

Différences Clés :

- **Noms des Types :**
  - Les types primitifs utilisent des noms en **minuscules** (ex: `int`, `char`).
  - Les types référence (classes) commencent par une **majuscule** (ex: `String`).
- **Méthodes :**
  - Les types référence peuvent appeler des méthodes (ex: `length()`

- **Valeur null :**
  - Seuls les types référence peuvent être assignés à null.
  - Les types primitifs ne peuvent pas être initialisés avec null.

```
int val = null;     // Erreur : int est un primitif
String nom = null; // Valide : String est un type référence
```

Utilisation de Wrapper pour les Primitifs :

Si vous souhaitez utiliser null avec un primitif, utilisez la classe wrapper correspondante (ex: Integer au lieu de int).

Les Classes Wrapper en Java

Définition

- Chaque type primitif a une **classe wrapper** correspondante.
- Permet de manipuler des primitives comme des objets.

Création de Wrappers

- `valueOf()` : Convertit une chaîne de caractères en wrapper.

```
int primitif = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
```

Classes Wrapper et la Classe Number

- Les classes numériques (ex: Integer, Double) étendent la classe Number.
- Méthodes disponibles :
- byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue()

```
Double apple = Double.valueOf("200.99");
System.out.println(apple.byteValue());   // -56
System.out.println(apple.intValue());    // 200
System.out.println(apple.doubleValue()); // 200.99
```

Type Primitif	Classe Wrapper	Hérite de `Number`	Exemple
`boolean`	`Boolean`	Non	`Boolean.valueOf(true)`
`byte`	`Byte`	Oui	`Byte.valueOf((byte) 1)`
`short`	`Short`	Oui	`Short.valueOf((short) 1)`
`int`	`Integer`	Oui	`Integer.valueOf(1)`
`long`	`Long`	Oui	`Long.valueOf(1L)`
`float`	`Float`	Oui	`Float.valueOf(1.0f)`
`double`	`Double`	Oui	`Double.valueOf(1.0)`
`char`	`Character`	Non	`Character.valueOf('c')`

Remarques :

- Les conversions peuvent entraîner des pertes de précision.
- Exemples :
- Valeur hors portée pour un type (byte par exemple) : résultat inattendu.
  - Troncature des valeurs à virgule (décimales ignorées).

Blocs de Texte en Java

Java permet de créer des chaînes multilignes appelées blocs de texte, définis à l'aide de trois guillemets doubles (`"""`). Les blocs de texte facilitent l'écriture et la lecture des chaînes qui s'étendent sur

Caractères d’Échappement

- " : Permet d’utiliser " dans une chaîne.
- \n : Ajoute une nouvelle ligne.

Comparaison entre Chaînes Régulières et Blocs de Texte

Formatage	Chaîne Régulière	Bloc de Texte
`\"`	`"`	`"`
`\"\"\"`	Invalide	`"""`
Espace (fin de ligne)	Espace	Ignoré
`\s`	Deux espaces	Deux espaces
`\` (fin de ligne)	Invalide	Omets la nouvelle ligne pour cette ligne

Exemples :

Bloc de Texte Simple :

```
String chaine = "\"Support de cours\"\n   Java - Débutant";
```

Bloc de Texte Multiligne Valide :

```
String pyramid = """
     *
    * *
   * * *
""";
```

Produit quatre lignes, avec les étoiles alignées selon les espaces essentiels.

Utiliser \ pour Supprimer la Nouvelle Ligne :

```
String block = """
   doe \
   deer
""";
```

Produit une seule ligne : "doe deer".

Exemple avec \n et Espaces :

```
String block = """
   doe \n
   deer
""";
```

Produit quatre lignes, y compris une nouvelle ligne explicite pour \n.

Exemple Complexe :

```
String block = """
    "doe\"\"\"
    \"deer\""""
""";
System.out.print("*" + block + "*");
```

```
"doe"""
"deer"""
```

Déclaration de Variables

Une variable est un nom pour un morceau de mémoire qui stocke des données. Lors de la déclaration d'une variable, il est nécessaire d'indiquer le type de la variable ainsi que son nom. L'attribution d'une valeur à une variable s'appelle l'initialisation d'une variable. Voici un exemple de déclaration et d'initialisation d'une variable en une seule ligne :

```
String myValue = "a custom string";
```

Java a des règles précises concernant les noms d'identifiants. Un identifiant est le nom d'une variable, d'une méthode, d'une classe, d'une interface ou d'un package. Voici les règles pour des identifiants légaux :

- Les identifiants doivent commencer par une lettre, un symbole monétaire ou un symbole _.
- Les identifiants peuvent inclure des chiffres mais ne peuvent pas commencer par eux.
- Un seul souligné _ n'est pas autorisé comme identifiant.
- Le nom ne peut pas être un mot réservé de Java.

Mots réservés	Mots réservés	Mots réservés	Mots réservés	Mots réservés
abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	final	finally	float
for	goto*	if	implements	import

Mots réservés	Mots réservés	Mots réservés	Mots réservés	Mots réservés
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Les mots réservés `const` et `goto` ne sont pas réellement utilisés en Java. Ils sont réservés pour éviter toute confusion avec d'autres langages.

CamelCase et Snake_case

Java a des conventions pour rendre le code lisible et cohérent. Par exemple :

- CamelCase : Utilisé pour les noms de méthodes et de variables, avec la première lettre de chaque mot en majuscule (ex : toUpper()).
- Snake_case : Utilisé pour les constantes, avec des underscores séparant les mots (ex : NUMBER_FLAGS).

Déclaration de Variables Multiples

Vous pouvez déclarer et initialiser plusieurs variables dans la même instruction. Par exemple :

```
void initValues() {
   String s1, s2;
   String s3 = "yes", s4 = "no";
}
```

Ici, quatre variables de type String sont déclarées : s1, s2, s3, et s4.

Initialisation des Variables

Avant d'utiliser une variable, elle doit avoir une valeur. Certaines variables obtiennent cette valeur automatiquement, d'autres doivent être spécifiées par le programmeur. Voici les différences entre les valeurs par défaut pour les variables locales, d'instance et de classe.

Création de Variables Locales

Une variable locale est définie à l'intérieur d'un constructeur, d'une méthode ou d'un bloc d'initialisation. Par exemple :

```
final int y = 10; // y ne peut pas être modifié
```

Si vous essayez de modifier y, cela déclenche une erreur de compilation

Variables Locales Finales

Le mot-clé final peut être appliqué aux variables locales et équivaut à déclarer des constantes dans d'autres langages. Voici un exemple :

```
final int[] favoriteNumbers = new int[10];
favoriteNumbers[0] = 10; // Cela est valide

favoriteNumbers = = new int[10]; //Cela est invalide
```

Le type `var`

Vous avez la possibilité d'utiliser le mot-clé `var` au lieu du type lors de la déclaration de variables locales dans certaines conditions. Pour utiliser cette fonctionnalité, il suffit de taper `var` au lieu du type primitif ou de référence. Voici un exemple :

```
public class CustomClass {
   public void whatTypeAmI() {
      var name = "Hello";
      var size = 7;
   }
}

Cependant, var ne peut être utilisé que pour les variables locales et ne fonctionne pas avec les variables d'instance.

---

**Type Inference de var**

var indique au compilateur de déterminer le type de la variable à la compilation. Par exemple :
```

Règles supplémentaires

var ne peut pas être utilisé avec null sans type, car le compilateur ne peut pas déterminer le type.

De plus, var ne peut pas être utilisé pour des paramètres de méthode ou des variables d’instance.

var dans le monde réel

L’utilisation de var améliore la lisibilité du code en simplifiant les déclarations longues :

```
var pileOfPapersToFile = new PileOfPapersToFileInFilingCabinet();
```

Il est recommandé d’utiliser var de manière judicieuse pour maintenir la clarté du code.

Gestion de la portée des variables

Variables locales

Les variables locales sont déclarées à l'intérieur d'un bloc de code

Limitation de la portée

Les variables locales n'ont pas de portée plus large que celle de la méthode. Par exemple :

```
public void eatIfHungry(boolean hungry) {
   if (hungry) {
      int bitesOfCheese = 1;
   } // bitesOfCheese sort de portée ici
   System.out.println(bitesOfCheese); // NE COMPILERA PAS
}
```

- hungry a une portée de méthode entière.
- bitesOfCheese a une portée limitée au bloc if.

Bloc de portée

Chaque bloc de code (délimité par des accolades {}) a sa propre portée. Les blocs peuvent contenir d'autres blocs. Les variables d'un bloc englobant peuvent être référencées dans un bloc intérieur, mais pas l'inverse.

```
public void eatMore(boolean hungry, int amountOfFood) {
   int roomInBelly = 5;
   if (hungry) {
      var timeToEat = true;
      while (amountOfFood > 0) {
         int amountEaten = 2;
         roomInBelly = roomInBelly - amountEaten;
         amountOfFood = amountOfFood - amountEaten;
      }
   }
```

Application de la portée aux classes

Les variables d'instance sont disponibles dès leur définition jusqu'à ce que l'objet soit éligible pour la collecte des ordures. Les variables de classe (ou statiques) sont accessibles dès leur déclaration et restent en portée pendant la durée de vie du programme.

Exemples de portée dans une classe:

```
public class Mouse {
   final static int MAX_LENGTH = 5;
   int length;
   public void grow(int inches) {
      if (length < MAX_LENGTH) {
         int newSize = length + inches;
         length = newSize;
```

Règles de portée

- Variables locales : en portée de la déclaration à la fin du bloc.
- Paramètres de méthode : en portée pendant la durée de la méthode.
- Variables d'instance : en portée de la déclaration jusqu'à ce que l'objet soit éligible pour la collecte des ordures.
- Variables de classe : en portée de la déclaration jusqu'à la fin du programme.

Les opérateurs

Définition des opérateurs

- **Opérateur** : symbole spécial appliqué à un ensemble de variables, valeurs ou littéraux (appelés **opérandes**) qui retourne un résultat.
- **Opérande** : valeur ou variable sur laquelle l'opérateur est appliqué.
- **Résultat** : sortie de l'opération.

Exemples d'opérateurs

- **Addition (+)** et **Soustraction (-)** : opérateurs de base connus.
- **Opérateurs d'affectation (=)** : utilisés pour stocker le résultat dans une variable.

Types d'opérateurs en Java

Classification des opérateurs

Java prend en charge trois types d'opérateurs :

- **Unaires** : Appliqués à un seul opérande.
- **Binaires** : Appliqués à deux opérandes.
- **Ternaires** : Appliqués à trois opérandes.

Exemple d'opération

```
var c = a + b; // c reçoit le résultat de l'opération
```

Évaluation des opérateurs

L'évaluation des opérateurs ne suit pas toujours un ordre de gauche à droite. Exemple :

```
int cookies = 4;
double reward = 3 + 2 * - - cookies;
System.out.print("reward values: "+reward);
```

- cookies est décrémenté à 3.
- Le résultat est multiplié par 2, puis 3 est ajouté, résultant en reward = 9.0.
- Valeurs finales : reward = 9.0, cookies = 3.

Priorité des opérateurs

Définition de la priorité des opérateurs

La priorité des opérateurs détermine l'ordre d'évaluation. En Java, cela suit les règles mathématiques.

Exemple d'expression:

```
var perimeter = 2 * height + 2 * length;
```

Parenthèses ajoutées pour clarification :

```
var perimeter = ((2 * height) + (2 * length));
```

L’opérateur de multiplication (*) a une priorité plus élevée que l’addition (+).

Symboles d'opérateurs et exemples	Évaluation
Post-opérateurs unaires	expression++, expression-- (Gauche à droite)
Pré-opérateurs unaires	++expression, --expression (Gauche à droite)
Autres opérateurs unaires	-, !, ~, +, (type) (Droite à gauche)

Symboles d'opérateurs et exemples	Évaluation
Cast (Type)	(type)reference (Droite à gauche)
Multiplication/division/modulus	*, /, % (Gauche à droite)
Addition/soustraction	+, - (Gauche à droite)
Opérateurs de décalage	<<, >>, >>> (Gauche à droite)
Opérateurs relationnels	<, >, <=, >=, instanceof (Gauche à droite)

Symboles d'opérateurs et exemples	Évaluation
Égalité/inégalité	==, != (Gauche à droite)
ET logique	& (Gauche à droite)
OU exclusif logique	^ (Gauche à droite)
OU inclusif logique	
ET conditionnel	&& (Gauche à droite)
OU conditionnel	

Symboles d'opérateurs et exemples	Évaluation
Opérateurs ternaires	boolean expression ? expression1 : expression2 (Droite à gauche)
Opérateurs d'affectation	=, +=, -=, *=, /=, %=, &=, ^=,
Opérateur flèche	-> (Droite à gauche)

Unary Operators

Définition des opérateurs unaires

Un opérateur unaire nécessite un seul opérande ou variable. Ils effectuent des tâches simples comme l'incrémentation d'une variable numérique ou l'inversion d'une valeur booléenne.

opérateurs unaires

- **Complément logique** (`!a`) : Inverse la valeur d'un booléen.
- **Complément bit à bit** (`~b`) : Inverse tous les bits d'un nombre.
- **Plus** (`+c`) : Indique qu'un nombre est positif.
- **Négation ou moins** (`-d`) : Indique qu'un nombre est négatif ou inverse une expression.
- **Incrément** (`++e` ou `f++`) : Augmente une valeur de 1.
- **Décrément** (`--f` ou `h--`) : Diminue une valeur de 1.
- **Cast** (`(String)i`) : Convertit une valeur en un type spécifique.

Opérateurs de complément et de négation

- **Complément logique** : Inverse la valeur d'un booléen (ex : `false` devient `true`).
- **Complément bit à bit** : Appliqué aux types numériques entiers, il inverse les bits. Par exemple : `~3` donne `-4`.

Exemples de non-compilation

- Les opérateurs unaires doivent être appliqués au bon type de variable. Par exemple :
  - `int pelican = !5;` // Erreur de compilation.
  - `boolean penguin = -true;` // Erreur de compilation.

Opérateurs d'incrément et de décrément

- **Pré-incrément** (`++w`) : Incrémente et retourne la nouvelle valeur.
- **Pré-décrément** (`--x`) : Décrémente et retourne la nouvelle valeur.
- **Post-incrément** (`y++`) : Incrémente et retourne l'ancienne valeur.
- **Post-décrément** (`z--`) : Décrémente et retourne l'ancienne valeur.

Différence entre pré- et post-incrément/décrément

- L'ordre d'attachement des opérateurs (`++` ou `--`) à une variable impacte la valeur retournée.
- Exemple :

```
int parkAttendance = 0;
System.out.println(++parkAttendance);  // Affiche 1
System.out.println(parkAttendance--);  // Affiche 1 avant décrément
```

Note : Les opérateurs d'incrément et de décrément sont couramment utilisés et nécessitent une compréhension claire pour éviter des erreurs.

Arithmetic Operators

Opérateurs Arithmétiques

Les opérateurs arithmétiques s'appliquent aux valeurs numériques (cf. tableau ci-dessous).

Opérateur	Exemple	Description
Addition	`a + b`	Addition de deux valeurs numériques
Soustraction	`c - d`	Soustraction de deux valeurs numériques
Multiplication	`e * f`	Multiplication de deux valeurs numériques
Division	`g / h`	Division d'une valeur numérique par une autre

Priorité des Opérateurs

Les opérateurs multiplicatifs (`*`, `/`, `%`) ont une priorité plus élevée que les opérateurs additifs (`+`, `-`). Exemple :

```
int price = 2 * 5 + 3 * 4 - 8;
// Réduction : 2 * 5 + 3 * 4 - 8 => 10 + 12 - 8 => 14
```

Parenthèses et Priorité

Les parenthèses modifient la priorité des opérations. Exemple :

```
int price = 2 * ((5 + 3) * 4 - 8);
// Réduction : 2 * (8 * 4 - 8) => 2 * (32 - 8) => 2 * 24 => 48
```

Validité des Parenthèses

Les parenthèses doivent être équilibrées. Exemples non valides :

```
long value1 = 1 + ((3 * 5) / 3;         // ERREUR
int value2 = (9 + 2) + 3) / (2 * 4;   // ERREUR
```

Division et Modulus

Le modulus (%) donne le reste d'une division :

```
System.out.println(11 / 3);  // 3
System.out.println(11 % 3);  // 2
```

Pour les valeurs entières, la division donne la valeur entière inférieure la plus proche. Le modulus est le reste.

Promotion Numérique

Java promeut les types primitifs selon des règles :

- Si deux types diffèrent, le plus petit est promu au plus grand.
- Un type intégral est promu à un type flottant si nécessaire.
- Les types byte, short, char sont promus à int lorsqu'ils sont utilisés avec un opérateur binaire.
- Le résultat d'une opération conserve le type promu.

Exemple de promotion:

```
int x = 1;
long y = 33;
var z = x * y; // z est de type long

double x = 39.21;
float y = 2.1f;
var z = x + y; // z est de type double

short x = 10;
short y = 3;
var z = x * y; // z est de type int

short w = 14;
float x = 13;
double y = 30;
var z = w * x / y; // z est de type double
```

Assignation des Valeurs

Lors de l’utilisation des opérateurs arithmétiques, il est crucial de suivre la promotion des types de données. Faites attention aux erreurs de compilation liées aux conversions de types.

Comparaison des Valeurs en Java

Les opérateurs de comparaison permettent de vérifier si deux valeurs sont égales, si une valeur numérique est inférieure ou supérieure à une autre, ou de réaliser des opérations booléennes. Vous avez probablement déjà utilisé plusieurs de ces opérateurs dans votre expérience de développement.

Opérateurs d'Égalité

En Java, la détermination de l'égalité peut être complexe, car il y a une différence entre “deux objets sont identiques” et “deux objets sont équivalents”. Cette distinction ne s'applique pas aux types primitifs numériques et booléens.

Les opérateurs d'égalité incluent :

- `==` : Vérifie si deux valeurs sont égales.
- `!=` : Vérifie si deux valeurs sont différentes.

Ces opérateurs s'appliquent aux valeurs numériques, booléennes, et aux objets (y compris les chaînes de caractères et `null`). Cependant, il est interdit de mélanger ces types, ce qui entraînerait une erreur de compilation.

Exemples :

```
boolean singe = true == 3;          // NE COMPILERA PAS
boolean gorille = 10.2 == "Koko"; // NE COMPILERA PAS
```

Attention aux Opérateurs d’Égalité

Le compilateur génère une erreur si vous essayez de comparer des types incompatibles. Des erreurs peuvent survenir si des opérateurs d’assignation sont confondus avec des opérateurs d’égalité.

exemple:

```
boolean ours = false;
boolean polaire = (ours = true);
System.out.println(polaire);  // Affiche true
```

Ici, l’opérateur d’assignation = attribue true à ours, et la valeur est également assignée à polaire.

Comparaison d’Objets

L’opérateur d’égalité compare les références des objets, pas les objets eux-mêmes. Deux références sont égales si elles pointent vers le même objet ou toutes deux vers null.

Exemple:

```
var lundi = new File("planning.txt");
var mardi = new File("planning.txt");
var mercredi = mardi;
System.out.println(lundi == mardi);    // false
System.out.println(mardi == mercredi); // true
```

Bien que lundi et mardi contiennent des informations identiques, elles représentent des objets différents car le mot-clé new a été utilisé.

Comparaison de null

En Java, comparer null à une autre valeur n'est pas toujours faux :

```
System.out.print(null == null);  // Affiche true
```

Dans un cas où une variable est null, l'utilisation de instanceof retourne toujours false.

Opérateurs Relationnels

Les opérateurs relationnels comparent deux expressions numériques

Exemple:

```
int gibbon = 2, loup = 4, autruche = 2;
System.out.println(gibbon < loup);          // true
System.out.println(gibbon >= autruche);     // true
System.out.println(gibbon > autruche);      // false
```

Opérateur instanceof

L'opérateur instanceof vérifie si un objet est une instance d'une classe ou interface spécifique. Cet opérateur est utile pour déterminer le type d'un objet au moment de l'exécution, particulièrement lorsque Java utilise le polymorphisme.

```
Integer heureZoo = Integer.valueOf(9);
Number nombre = heureZoo;
Object objet = heureZoo;
System.out.println(heureZoo instanceof Integer); // true
System.out.println(nombre instanceof Number);    // true
System.out.println(objet instanceof Object);     // true
```

instanceof et Polymorphisme

L'opérateur instanceof est souvent utilisé avant un cast pour s'assurer que l'objet peut être converti sans erreur de compilation. C'est une bonne pratique en Java.

```
public void ouvrirZoo(Number heure) {
   if (heure instanceof Integer)
      System.out.print((Integer)heure + " O'clock");
   else
      System.out.print(heure);
}
```

Logical Operators

Les opérateurs logiques sont utilisés pour effectuer des opérations sur des valeurs booléennes. Ils incluent les opérateurs `&`, `|`, et `^`, comme décrit dans le tableau ci-dessous. Ces opérateurs peuvent être appliqués aux types de données booléens et numériques.

Opérateurs Logiques

Opérateur	Exemple	Description
AND logique	`a & b`	Renvoie `true` uniquement si les deux valeurs sont `true`.
OR inclusif logique	`c	d`
OR exclusif logique	`e ^ f`	Renvoie `true` uniquement si l'une des valeurs est `true` et l'autre est `false`.

Tables de Vérité

- **AND** : `true` uniquement si les deux opérandes sont `true`.
- **OR Inclusif** : `true` si au moins un opérande est `true`.
- **OR Exclusif** : `true` si les opérandes sont différents.

Exemple

```
boolean eyesClosed = true;
boolean breathingSlowly = true;

boolean resting = eyesClosed | breathingSlowly;
boolean asleep = eyesClosed & breathingSlowly;
boolean awake = eyesClosed ^ breathingSlowly;
System.out.println(resting);  // true
System.out.println(asleep);   // true
System.out.println(awake);    // false
```

Essayez de changer les valeurs pour observer les résultats.

Opérateurs Conditionnels

Exemples et Optimisations

Éviter les NullPointerException

```
if (duck != null && duck.getAge() < 5) {
    // Do something
}
```

Si duck est null, l'évaluation s'arrête avant d'appeler getAge(), évitant une NullPointerException.

Effets de Bord

L'opérateur conditionnel peut éviter des effets de bord si le côté droit n'est jamais évalué.

Exemple:

```
int rabbit = 6;
boolean bunny = (rabbit >= 6) || (++rabbit <= 7);
System.out.println(rabbit); // 6
```

Le côté droit (++rabbit <= 7) n'est pas évalué car le côté gauche est true, donc la valeur de rabbit reste inchangée.

Opérateur Ternaire

L'opérateur ternaire est un opérateur conditionnel qui utilise trois opérandes et a la forme suivante :

```
booleanExpression ? expression1 : expression2
```

Description

- **Premier opérande** : Doit être une expression booléenne.
- **Deuxième et troisième opérandes** : Peuvent être n'importe quelles expressions qui renvoient une valeur.

L'opérateur ternaire est une version condensée d'une instruction `if/else` qui retourne une valeur. Par exemple :

```
int owl = 5;
int food = owl < 2 ? 3 : 4;
System.out.println(food);  // Affiche 4
```

Parenthèses pour la lisibilité

Il est recommandé d'ajouter des parenthèses pour améliorer la lisibilité, surtout lorsque plusieurs opérateurs ternaires sont utilisés :

```
int food1 = owl < 4 ? owl > 2 ? 3 : 4 : 5;
int food2 = (owl < 4 ? ((owl > 2) ? 3 : 4) : 5);
```

Types de données différents

Les expressions expression1 et expression2 n'ont pas besoin d'avoir le même type de données, mais cela peut poser problème si elles sont combinées avec un opérateur d'affectation :

```
int stripes = 7;
System.out.print((stripes > 5) ? 21 : "Zebra");  // Compile
int animal = (stripes < 9) ? 3 : "Horse";        // Ne compile pas
```

Effets de bord non exécutés

Comme avec les opérateurs conditionnels, l'opérateur ternaire peut contenir des effets de bord non exécutés, car une seule des expressions sera évaluée à l'exécution :

```
int sheep = 1;
int zzz = 1;
int sleep = zzz < 10 ? sheep++ : zzz++;
System.out.print(sheep + "," + zzz);  // Affiche 2,1
```

Si la condition change :

```
int sheep = 1;
int zzz = 1;
int sleep = sheep >= 10 ? sheep++ : zzz++;
System.out.print(sheep + "," + zzz);  // Affiche 1,2
```

Dans cet exemple, seule l'une des variables sera incrémentée, en fonction de l'évaluation de la condition.

Contrôle du Flux de Programme

Java permet de créer des programmes intelligents capables de prendre des décisions en fonction des conditions rencontrées à l'exécution. Ce chapitre présente les différentes structures de contrôle de flux en Java, y compris les instructions `if/else`, les expressions et instructions `switch`, les boucles, ainsi que les instructions `break` et `continue`.

Création de Structures de Décision

Les opérateurs Java permettent de créer des expressions complexes, mais ils sont limités pour contrôler le flux de programme de manière dynamique. Pour exécuter du code en fonction de conditions déterminées à l'exécution, Java utilise des structures comme `if` et `else`, et des fonctionnalités plus récentes comme la correspondance de motifs (pattern matching).

Instructions et Blocs

Une instruction en Java est une unité d'exécution complète se terminant par un point-virgule (`;`). Les instructions de contrôle de flux permettent de fragmenter l'exécution en prenant des décisions, en utilisant des boucles, et en effectuant des sauts dans le code. Elles permettent à l'application d'exécuter sélectivement certains segments de code.

Ces instructions peuvent s'appliquer à une seule expression ou à un bloc de code, qui est un groupe de zéro ou plusieurs instructions entourées d'accolades (`{}`). Par exemple, les deux extraits suivants sont équivalents :

```
// Instruction simple
value++;

// Instruction à l'intérieur d'un bloc
{
   value++;
}
```

Exemple avec une Instruction de Décision

Une instruction ou un bloc peut être la cible d'une structure de décision. Par exemple, nous pouvons ajouter une condition if aux deux exemples précédents :

```
// Instruction simple
if(ticketsTaken > 1)
   patrons++;

// Instruction à l'intérieur d'un bloc
if(ticketsTaken > 1)
{
   patrons++;
}
```

L'instruction if

Souvent, nous voulons exécuter un bloc de code uniquement dans certaines conditions. L'instruction `if` permet à notre application d'exécuter un bloc particulier de code si une expression booléenne évalue à `true` à l'exécution.

Structure de l'instruction if

```
if (booleanExpression) {
    // bloc de code
}
```

- Mot-clé if : Indique le début de l’instruction conditionnelle.
- Parenthèses : Requises pour encapsuler l’expression booléenne.
- Accolades : Nécessaires pour les blocs de plusieurs instructions, optionnelles pour une seule instruction.

Exemple:

```
if(hourOfDay < 11) {
    System.out.println("Good Morning");
    morningGreetingCount++;
}
```

Attention à l'intendation e taux accolades:

```
if(hourOfDay < 11)
    System.out.println("Good Morning");
    morningGreetingCount++; // S'exécute toujours
```

L’instruction else

Pour afficher un message différent lorsque hourOfDay est 11 heures ou plus, nous pouvons utiliser l’instruction else.

Structure de l’instruction else

```
if (booleanExpression) {
    // Branche si vrai
} else {
    // Branche si faux
}
```

Exemple de if else

```
if(hourOfDay < 11) {
   System.out.println("Good Morning");
} else {
   System.out.println("Good Afternoon");
}
```

Cette approche évite l'évaluation redondante de hourOfDay.

Utilisation de else if

Pour des conditions plus complexes, nous pouvons enchaîner des instructions if avec else if :

```
if(hourOfDay < 11) {
   System.out.println("Good Morning");
} else if(hourOfDay < 15) {
   System.out.println("Good Afternoon");
} else {
   System.out.println("Good Evening");
}
```

Le processus continuera jusqu'à ce qu'une condition soit vraie.

Raccourcir le Code avec le Pattern Matching

Java 16 a introduit le pattern matching avec les instructions `if` et l'opérateur `instanceof`. Cela permet de contrôler le flux du programme en exécutant un code qui répond à certains critères, tout en réduisant le code redondant.

Différence entre Pattern Matching et Expressions Régulières

- **Pattern Matching** : Technique pour contrôler le flux, liée à `if` et `instanceof`.
- **Expressions Régulières** : Utilisées pour le filtrage, mais conceptuellement différentes.

Utilisation du Pattern Matching

Exemple sans Pattern Matching

```
void compareIntegers(Number number) {
   if(number instanceof Integer) {
      Integer data = (Integer)number;
      System.out.print(data.compareTo(5));
   }
}
```

Exemple avec Pattern Matching

```
void compareIntegers(Number number) {
    if(number instanceof Integer data) {
        System.out.print(data.compareTo(5));
    }
}
```

Variable data : Appelée variable de pattern, évitant le ClassCastException en cas de vérification réussie.

Réassignation de Variables de Pattern

Bien que possible, la réassignation d'une variable de pattern est déconseillée.

```
if(number instanceof Integer data) {
    data = 10; // Mauvaise pratique
}
```

L'utilisation du modificateur final pour empêcher la réassignation est possible mais non recommandée.

Utilisation d'Expressions avec Pattern Matching

Les variables de pattern peuvent être utilisées dans des expressions :

```
void printIntegersGreaterThan5(Number number) {
   if(number instanceof Integer data && data.compareTo(5) > 0)
      System.out.print(data);
}
```

Sous-types et Limites

Le type de la variable de pattern doit être un sous-type strict de la variable d'origine.

Exemple:

```
Integer value = 123;
if(value instanceof Integer data) {}  // NE COMPILERA PAS
```

Bien que le compilateur permette if(value instanceof List) {}, ce n'est pas lié.

Portée du Flux

La portée du flux signifie que la variable n'est accessible que si le compilateur peut déterminer son type de manière définitive.

Exemple de non-compilation

```
void printIntegersOrNumbersGreaterThan5(Number number) {
   if(number instanceof Integer data || data.compareTo(5) > 0)
      System.out.print(data); // NE COMPILERA PAS
}
```

Ici, data n'est pas défini si number n'est pas un Integer.

Exemples de Portée

```
void myMethod(Number number) {
   if (number instanceof Integer value)
      System.out.print(value.intValue());
   System.out.print(value.intValue()); // NE COMPILERA PAS
}
```

Le code ci-dessus ne compile pas car data n’est plus dans le scope après l’instruction if.

Portée et Branches else

La logique peut être réécrite pour clarifier la portée :

```
void printOnlyIntegers(Number number) {
   if (number instanceof Integer data)
      System.out.print(data.intValue());
   else
      return;
}
```

L'important est que le compilateur détermine que data est dans le scope uniquement lorsque number est un Integer.

Les instructions `switch` en Java

Exemples de l'utilisation de `switch`

```java
public void printDayOfWeek(int day) {
    switch (day) {
        case 0:
            System.out.print("Sunday");
            break;
        case 1:
            System.out.print("Monday");
            break;
        case 2:
            System.out.print("Tuesday");
            break;
        case 3:
            System.out.print("Wednesday");
            break;
        case 4:
            System.out.print("Thursday");
            break;
        case 5:
            System.out.print("Friday");
            break;
        case 6:
            System.out.print("Saturday");
            break;
        default:
            System.out.print("Invalid value");
            break;
    }
}
```

Remarques sur le break

- Le break termine le switch et retourne le contrôle au processus englobant.
- Sans break, le code exécute toutes les branches suivant le case correspondant.

Exemple sans break:

```
public void printSeason(int month) {
    switch(month) {
        case 1, 2, 3:    System.out.print("Winter");
        case 4, 5, 6:    System.out.print("Spring");
        default:         System.out.print("Unknown");
        case 7, 8, 9:    System.out.print("Summer");
        case 10, 11, 12: System.out.print("Fall");
    }
}
```

```
WinterSpringUnknownSummerFall
```

Types de données acceptés dans switch

- Types primitifs : int, byte, short, char.
- Classes wrapper : Integer, Byte, Short, Character.
- Types supplémentaires : String, valeurs enum, var.

Exemples de valeurs non valides:

```
final int getGrass() { return 4; }
void feedAnimals() {
    final int bananas = 1;
    int apples = 2;
    int numberOfAnimals = 3;
    final int cookies = getGrass();
    switch(numberOfAnimals) {
        case bananas:
        case apples:          // NE COMPILE PAS
        case getGrass():  // NE COMPILE PAS
        case cookies :        // NE COMPILE PAS
        case 3 * 5 :         // COMPILE
    }
}
```

L’expression switch

```
public void printDayOfWeek(int day) {
    var result = switch(day) {
        case 0 -> "Sunday";
        case 1 -> "Monday";
        case 2 -> "Tuesday";
        case 3 -> "Wednesday";
        case 4 -> "Thursday";
        case 5 -> "Friday";
        case 6 -> "Saturday";
        default -> "Invalid value";
    };
    System.out.print(result);
}
```

Règles des expressions switch

- Toutes les branches doivent retourner un type de données cohérent.
- Les branches qui ne sont pas des expressions doivent retourner une valeur avec yield.
- Une branche par défaut est requise si toutes les valeurs possibles ne sont pas gérées.

Attention aux points-virgules

- Les expressions case nécessitent un point-virgule.
- Les blocs case ne doivent pas utiliser de point-virgule à la fin.

Gestion de toutes les valeurs possibles

- Un switch qui retourne une valeur doit couvrir toutes les entrées possibles.
- Solutions :
  - Ajouter une branche par défaut.
  - Pour les enum, ajouter une branche pour chaque valeur.

Boucles `while` en Java

- Les boucles permettent d'exécuter des instructions plusieurs fois.
- Utiliser des chaînes d'instructions `if` pour des tâches répétitives est inefficace.
- Une boucle exécute un bloc de code tant qu'une condition est vraie.

Exemple de Boucle `while`

```
int counter = 0;
while (counter < 10) {
    double price = counter * 10;
    System.out.println(price);
    counter++;
}
```

Structure d’une Boucle while

Syntaxe:

```
while (expressionBoolean) {
    // Corps de la boucle
}
```

La condition est évaluée avant chaque itération.

attention: la boucle ne s'éxecutera jamais sila condition est fausse dès le début.

Boucle do/while

Caractéristiques

- La boucle do/while garantit que le corps de la boucle s’exécute au moins une fois.
- Syntaxe :

```
do {
    // Corps
} while (expressionBoolean);
```

exemple:

```
int lizard = 0;
do {
   lizard++;
} while (false);
System.out.println(lizard);  // 1
```

Boucles Infinies

- Une boucle infinie ne se termine jamais.
- Exemple de boucle infinie :

```
int value1 = 2;
int value2 = 5;
while (value1 < 10)
   value2++;
```

Points d'attention:

- Toujours s’assurer que les conditions de terminaison des boucles sont atteintes.
- Vérifier que les variables changent à chaque itération pour éviter les boucles infinies.

Boucles for

Bien que les instructions while et do/while soient puissantes, certaines tâches sont courantes en développement logiciel, comme itérer sur une instruction un nombre spécifique de fois. Pour simplifier cela, nous utilisons les **boucles for**, qui permettent d'effectuer des tâches avec moins de code répétitif.

La boucle for

Une boucle for de base comprend :

- Un **bloc d'initialisation**
- Une **expression booléenne conditionnelle**
- Une **instruction de mise à jour**

Structure

```
for (initialisation; expressionBooléenne; instructionMiseÀJour) {
    // Corps
}
```

- Chaque section est séparée par un point-virgule.
- Les variables déclarées dans le bloc d'initialisation sont limitées à cette portée.

Exemple

```
for(int i = 0; i < 5; i++) {
   System.out.print(i + " ");
}
```

```
Sortie : 0 1 2 3 4
```

Remarques:
Points clés

- Utilisez les boucles for pour des tâches avec un nombre connu d’itérations.
- Comprenez la portée des variables dans les boucles pour éviter les erreurs de compilation.

Instruction break

L'instruction `break` permet de sortir prématurément d'une boucle (`while`, `do/while`, ou `for`). Si elle est utilisée sans étiquette, elle termine la boucle la plus proche. Cependant, avec un paramètre d'étiquette, elle peut interrompre une boucle englobante.

Exemple

```
public class FindInMatrix {
   public static void main(String[] args) {
      int[][] list = {{1,13},{5,2},{2,2}};
      int searchValue = 2;
      int positionX = -1;
      int positionY = -1;
      PARENT_LOOP: for(int i=0; i<list.length; i++) {
         for(int j=0; j<list[i].length; j++) {
            if(list[i][j] == searchValue) {
               positionX = i;
               positionY = j;
               break PARENT_LOOP; // Sortie de la boucle externe
            }
         }
      }
   }
}
```

Remarques sur l’instruction break

- Si l’on utilise simplement break;, seule la boucle interne est terminée.
- Si break est omis, la recherche continue jusqu’à la fin de la structure, retournant potentiellement la dernière occurrence.

Instruction continue

L'instruction continue termine l'itération actuelle de la boucle et passe à l'itération suivante, tout en vérifiant à nouveau la condition de la boucle.

```
public class CleaningSchedule {
   public static void main(String[] args) {
      CLEANING: for(char stables = 'a'; stables <= 'd'; stables++) {
         for(int leopard = 1; leopard < 4; leopard++) {
            if(stables == 'b' || leopard == 2) {
               continue CLEANING; // Passe à l'itération suivante de la boucle externe
            }
            System.out.println("Cleaning: " + stables + "," + leopard);
         }
      }
   }
}
```

la sortie est:

```
Cleaning: a,1
Cleaning: c,1
Cleaning: d,1
```

Variations de l’instruction continue

- Si l’étiquette est omise, continue ne s’applique qu’à la boucle interne.
- Sans continue, toutes les valeurs sont traitées.

Instruction return

Les instructions return peuvent également servir à sortir des boucles, rendant le code plus lisible et facilitant la réutilisation. L'exemple suivant illustre l'utilisation de return pour remplacer break.

Exemple:

```
public class FindInMatrixUsingReturn {
   private static int[] searchForValue(int[][] list, int v) {
      for (int i = 0; i < list.length; i++) {
         for (int j = 0; j < list[i].length; j++) {
            if (list[i][j] == v) {
               return new int[] {i, j}; // Sortie rapide de la fonction
            }
         }
      }
      return null; // Valeur non trouvée
   }
}
```

Sortie similaire à break, mais le code est plus lisible et maintenable.

Tests Unitaires en Java avec JUnit

Les tests unitaires sont une pratique essentielle dans le développement logiciel qui vise à vérifier le bon fonctionnement des unités de code, généralement des méthodes ou des classes. JUnit est un framework populaire pour écrire et exécuter des tests unitaires en Java.

Concepts Clés

Qu'est-ce qu'un test unitaire ?

Un test unitaire est un morceau de code qui teste une unité spécifique d'un programme. Les tests unitaires permettent de :

- Valider le comportement d'une méthode ou d'une classe.
- Détecter rapidement les erreurs lors du développement.
- Faciliter les modifications du code en garantissant que les fonctionnalités existantes ne sont pas affectées.

Pourquoi utiliser JUnit ?

JUnit offre plusieurs avantages :

- **Simplicité** : Écrire des tests est simple et direct.
- **Annotations** : Utilisation d'annotations pour définir les méthodes de test, facilitant la lecture et la compréhension du code.
- **Intégration** : JUnit s'intègre facilement avec des outils de construction comme Maven et Gradle, ainsi qu'avec des IDEs comme IntelliJ et Eclipse.
- **Rapports** : Génération de rapports détaillés sur l'exécution des tests.

Structure d'un Test JUnit

Un test JUnit typique comprend plusieurs éléments clés :

Annotations JUnit :

- `@Test` : Indique qu'une méthode est un test.
- `@Before` : Exécutée avant chaque test pour préparer l'environnement.
- `@After` : Exécutée après chaque test pour nettoyer l'environnement.
- `@BeforeClass` et `@AfterClass` : Exécutées une fois avant ou après tous les tests d'une classe.

Assertions : Utilisées pour vérifier les résultats des tests.

- `assertEquals(expected, actual)` : Vérifie que les valeurs sont égales.
- `assertTrue(condition)` : Vérifie qu'une condition est vraie.
- `assertFalse(condition)` : Vérifie qu'une condition est fausse.
- `assertNull(object)` : Vérifie qu'un objet est nul.
- `assertNotNull(object)` : Vérifie qu'un objet n'est pas nul.

Exemple de Test JUnit

Voici un exemple simple d'un test unitaire utilisant JUnit :

Code de la classe à tester

```
package com.example;

public class Calculatrice {
    public int additionner(int a, int b) {
        return a + b;
    }
}
```

Code du test unitaire

```java
package com.example;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatriceTest {
    private Calculatrice calc;

    @Before
    public void setUp() {
        calc = new Calculatrice(); // Initialisation de l'objet avant chaque test
    }

    @Test
    public void testAdditionner() {
        assertEquals(5, calc.additionner(2, 3)); // Teste la méthode d'addition
    }

    @Test
    public void testAdditionnerAvecNégatif() {
        assertEquals(1, calc.additionner(3, -2)); // Teste avec un nombre négatif
    }
}
```

Exécution des Tests

Les tests peuvent être exécutés de différentes manières :

- IDE : La plupart des IDEs offrent des options pour exécuter des tests JUnit directement à partir de l'éditeur.
- Ligne de commande : Utilisation de Maven ou Gradle pour exécuter les tests via la ligne de commande.

```
mvn test
```

Les tests unitaires avec **JUnit** sont un aspect essentiel du développement **Java** qui permettent de garantir la **qualité** et la **fiabilité** du code.

L'écriture de tests unitaires peut sembler fastidieuse au début, mais elle s'avère très bénéfique pour maintenir un code propre et fonctionnel.