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.

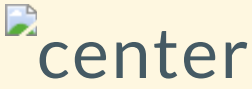
L'écriture de tests unitaires peut sembler fastidieuse au début, mais elle s'avère très bénéfique pour maintenir un code propre et fonctionnel.

Structures de données

Utilisation des API de Collections Courantes

Une collection est un groupe d'objets contenus dans un seul objet. Le framework de collections Java est un ensemble de classes dans `java.util` pour stocker des collections. Il y a quatre interfaces principales dans ce framework.

- **List** : Une liste est une collection ordonnée d'éléments qui accepte les doublons. Les éléments peuvent être accédés via un index `int`.
- **Set** : Un ensemble ne permet pas de doublons.
- **Queue** : Une file ordonne ses éléments pour le traitement. `Deque` est une sous-interface permettant l'accès aux deux extrémités.
- **Map** : Un tableau associatif lie des clés à des valeurs, sans doublons de clés. Les éléments sont des paires clé/valeur.



L'interface Collection, ses sous-interfaces et des classes qui les implémentent. Les interfaces sont représentées par des rectangles, et les classes par des boîtes arrondies.

Il est important de noter que `Map` n'implémente pas l'interface `Collection`. Elle fait néanmoins partie du framework car elle contient un groupe d'objets, bien qu'elle utilise des méthodes spécifiques à cause des paires clé/valeur.

Utilisation des API de Collections Courantes

Cette section présente les méthodes courantes que l'API Collections fournit aux classes implémentantes. Beaucoup de ces méthodes sont des méthodes utilitaires facilitant l'écriture et la lisibilité du code.

Nous utiliserons, par défaut, `ArrayList` et `HashSet` comme classes d'implémentation, mais ces méthodes s'appliquent à toute classe héritant de l'interface `Collection`.

Utilisation de l'Opérateur Diamant

Lors de la création d'une collection en Java, il est nécessaire de spécifier le type à contenir. Par exemple :

```
List<Integer> list = new ArrayList<Integer>();
```

Pour simplifier cette syntaxe, l'opérateur diamant (`<>`) permet de ne pas répéter le type à droite :

```
List<Integer> list = new ArrayList<>();
```

Cet opérateur ne peut être utilisé que dans l'assignation, pas dans les déclarations de variables.

Méthodes Courantes

- `add(E element)` : Ajoute un élément et retourne un booléen.
- `remove(Object object)` : Retire un élément correspondant et retourne un booléen.
- `isEmpty()`, `size()` : Vérifient si la collection est vide et donnent sa taille.

- `clear()` : Vide la collection.
- `contains(Object object)` : Vérifie si un élément est présent.
- `removeIf(Predicate<? super E> filter)` : Supprime les éléments selon une condition.
- `forEach(Consumer<? super T> action)` : Parcourt chaque élément de la collection.

Utilisation des Méthodes `hashCode` et `equals` dans le `add` et `remove`

Dans une collection comme `HashSet`, les méthodes `hashCode` et `equals` sont essentielles pour déterminer l'unicité des éléments :

- **`hashCode`** : Retourne un code de hachage pour l'objet, permettant de le localiser rapidement dans une structure basée sur des hachages.
- **`equals`** : Compare deux objets pour vérifier s'ils sont égaux.

Lorsqu'un élément est ajouté, Java utilise d'abord `hashCode` pour trouver une position, puis `equals` pour confirmer s'il existe déjà ou non.

Autres Méthodes

- **Itération** : Utilisation de `forEach`, de boucles ou d'un `Iterator`.
  `forEach` : Applique une action à chaque élément de la collection.

```
List<String> names = List.of("Alice", "Bob");
names.forEach(System.out::println);
```

```java
List<String> animals = new ArrayList<>();
      animals.add("Cat");
      animals.add("Dog");
      animals.add("Bird");

      // Création de l'itérateur
      Iterator<String> iterator = animals.iterator();

      // Parcours de la collection avec l'itérateur
      while (iterator.hasNext()) {
          String animal = iterator.next();
          System.out.println(animal);
      }
```

- **Comparaison** : Utilisation de `equals()` pour vérifier l'égalité des collections (en tenant compte de l'ordre ou non).

Ces concepts simplifient et optimisent la manipulation des collections en Java.

Listes en Java : Points Clés

1. **Caractéristiques des Listes**
   - Collection **ordonnée**, peut contenir des **doublons**.
   - Les éléments sont accessibles via un **index** (comme un tableau).
   - Les listes peuvent **changer de taille** dynamiquement (contrairement aux tableaux).
2. **Implémentations Communes**
   - **ArrayList** : Tableau redimensionnable, accès rapide aux éléments par index.
   - **LinkedList** : Doublement chaînée, accès rapide au début/fin.

3. **Création de Listes**
   - **ArrayList** : `new ArrayList<>();`
   - **LinkedList** : `new LinkedList<>();`
   - **Immuable** : `List.of("a", "b");`
4. **Méthodes Importantes**
   - `add(E element)` : Ajoute un élément.
   - `get(int index)` : Récupère un élément par index.
   - `remove(int index)` : Supprime un élément par index.

Exemples de Code

Création et Modification de Listes

```
List<String> list = new ArrayList<>();
list.add("Paris");
list.add(0, "Lyon");
list.set(1, "Marseille");
System.out.println(list.get(0)); // Affiche "Lyon"
list.remove("Lyon");
```

Liste Immuable avec Factory

```
List<String> immutableList = List.of("a", "b");
immutableList.add("c"); // UnsupportedOperationException
```

Utilisation de replaceAll()

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
numbers.replaceAll(x -> x * 2);
System.out.println(numbers); // [2, 4, 6]
```

Conversion de Liste en Tableau

```
List<String> list = new ArrayList<>(List.of("chien", "chat"));
String[] array = list.toArray(new String[0]);
System.out.println(Arrays.toString(array)); // [chien, chat]
```

Implémentations de `List` en Java : Points Principaux

1. **ArrayList**
   - **Description** : Basée sur un tableau redimensionnable.
   - **Usage** : Idéal pour des accès fréquents aux éléments par **index** et des opérations en **lecture**.
   - **Complexité** : Accès rapide en `O(1)` ; ajout/suppression en fin de liste en `O(1)` (sinon `O(n)`).
   - **Scénarios d'utilisation** :
     - Stockage de données fréquentes sans modifications importantes.

2. **LinkedList**
    - **Description** : Liste doublement chaînée.
    - **Usage** : Optimisée pour les **insertions** et **suppressions** aux débuts/fin de la liste.
    - **Complexité** : Accès en `O(n)` ; ajout/suppression aux extrémités en `O(1)`.
    - **Scénarios d'utilisation** :
        - File d'attente (FIFO) ou pile (LIFO).
        - Cas où les modifications fréquentes aux extrémités sont nécessaires.

3. **Vector**
    - **Description** : Similaire à `ArrayList`, mais **synchronisée**.
    - **Usage** : Utilisée dans des environnements **multi-threadés** pour garantir la sécurité des accès.
    - **Complexité** : Semblable à `ArrayList` mais avec des performances réduites à cause de la synchronisation.
    - **Scénarios d'utilisation** :
        - Applications héritées nécessitant des collections thread-safe.

4. **CopyOnWriteArrayList**
   - **Description** : Variante de `ArrayList` qui crée une **copie** de la liste à chaque modification.
   - **Usage** : Utile en **concurrence** lorsqu'il y a peu de modifications mais beaucoup de lectures.
   - **Complexité** : Lectures rapides (`O(1)`), mais modifications coûteuses (`O(n)`).
   - **Scénarios d'utilisation** :
     - Lecture majoritaire dans un environnement concurrent où l'immuabilité temporaire est essentielle.

Comparatif des Principales Implémentations

Implémentation	Accès aléatoire	Insertion/Suppression en fin	Thread-safe
`ArrayList`	`O(1)`	`O(1)` en fin, `O(n)` sinon	Non
`LinkedList`	`O(n)`	`O(1)` aux extrémités	Non
`Vector`	`O(1)`	`O(1)` en fin, `O(n)` sinon	Oui
`CopyOnWriteArrayList`	`O(1)`	`O(n)` (copie complète)	Oui

Conseils de Choix

- **ArrayList** : Usage général si accès rapide aux éléments et peu de modifications.
- **LinkedList** : Privilégier si les opérations fréquentes sont aux extrémités de la liste.
- **Vector** : Si une version synchronisée de `ArrayList` est requise.
- **CopyOnWriteArrayList** : En environnement concurrent si la majorité des opérations sont en lecture.

Points Principaux : Travail avec les Sets en Java

1. **Création de Sets immuables**
   - `Set<Character> letters = Set.of('z', 'o', 'o');`
   - `Set<Character> copy = Set.copyOf(letters);`
2. **Comportement des Sets**
   - Les méthodes traditionnelles (`add()`, `remove()`, `contains()`, etc.) fonctionnent pour les Sets.
   - Les Sets ne permettent pas de doublons.

Quelques Implémentations de `Set` en Java

1. **Set Immuables et Copies**
   - **Créer un Set immuable** :
     ```
     Set<Character> letters = Set.of('z', 'o', 'o');
     ```
   - **Faire une copie d'un Set** :
     ```
     Set<Character> copy = Set.copyOf(letters);
     ```
   - **Remarque** : Ces méthodes créent des sets immuables, ce qui les rend non modifiables après création.
2. **HashSet**
   - **Description** : Implémentation basée sur une **table de hachage**.
   - **Usage** : Rapide pour les opérations d'**ajout**, **suppression** et

Utilisation de l'interface Map

- **Définition :** Utilisez une Map pour identifier des valeurs par une clé (ex : liste de contacts).
- **Types de Map :**
  - **HashMap :** Utilise un tableau de hachage. Accès rapide, mais pas d'ordre.
  - **TreeMap :** Utilise une structure d'arbre trié. Les clés sont toujours triées, mais ajout plus lent.

Méthodes de Map

- **Création :**
  - `Map.of("key1", "value1", "key2", "value2");`
  - `Map.ofEntries(Map.entry("key1", "value1"), Map.entry("key2", "value2"));`

Méthodes importantes

Méthode	Description
`public void clear()`	Supprime toutes les clés et valeurs de la map.
`public boolean containsKey(Object key)`	Vérifie si la clé est dans la map.
`public V get(Object key)`	Retourne la valeur mappée par la clé ou null si absente.
`public V getOrDefault(Object key, V defaultValue)`	Retourne la valeur mappée ou une valeur par défaut.
	Ajoute ou remplace une paire

Itération sur une Map

- Utiliser `forEach((k, v) -> ...)` pour parcourir les paires clé/valeur.
- Accéder aux clés et valeurs via `entrySet()`.

Remplacement et fusion de valeurs

- `replace(K key, V value)` : Remplace la valeur pour la clé donnée.
- `merge(K key, V value, BiFunction<K, V, V> mapper)` : Ajoute ou fusionne une valeur avec logique personnalisée.

Exemples

- **Utilisation de `putIfAbsent()`:** Ajoute une valeur si la clé n'est pas déjà présente.
- **Comportement de `merge()`:**
  - Si la clé a une valeur nulle, remplace-la.
  - Si la clé a une valeur non nulle, applique la fonction de fusion.

Exemples d'implémentation de Map en Java

1. HashMap

Spécificités :

- **Performance :** Accès en temps constant (O(1)) pour les opérations de recherche, insertion et suppression.
- **Ordre :** Les éléments ne sont pas ordonnés.
- **Clés nulles :** Accepte une clé nulle et plusieurs valeurs nulles.

Utilisation :

Utilisez `HashMap` lorsque la rapidité d'accès est essentielle et que l'ordre des éléments n'a pas d'importance.

Exemple :

```
Map<String, Integer> scores = new HashMap<>();
scores.put("Alice", 90);
scores.put("Bob", 85);
scores.put("Charlie", 92);
```

TreeMap

Spécificités :

- Performance : Accès en temps logarithmique (O(log n)).
- Ordre : Les clés sont triées selon leur ordre naturel ou selon un comparateur fourni.
- Clés nulles : Ne permet pas de clés nulles.

Utilisation :

Utilisez TreeMap lorsque vous avez besoin d’un accès trié aux éléments et que vous voulez effectuer des opérations comme la recherche de la clé minimale ou maximale.

LinkedHashMap

Spécificités:

- Performance : Accès en temps constant (O(1)), comme HashMap.
- Ordre : Maintient l'ordre d'insertion des éléments.
- Clés nulles : Accepte les clés nulles.

Utilisation :

Utilisez LinkedHashMap lorsque vous avez besoin d'une Map qui conserve l'ordre d'insertion tout en offrant des performances de recherche rapides.

WeakHashMap

Spécificités :

- Garbage Collection : Permet aux clés d'être collectées par le ramasse-miettes si elles ne sont plus référencées ailleurs.
- Utilisation de mémoire : Idéal pour les caches où les objets peuvent être libérés lorsque la mémoire est nécessaire.

Utilisation :

Utilisez WeakHashMap pour des caches ou des mappages temporaires où vous ne voulez pas empêcher les objets de mémoire d'être collectés.

la Comparaison en Java

Implémentation de `Comparable`

```
public class MissingDuck implements Comparable<MissingDuck> {
   private String name;

   public int compareTo(MissingDuck quack) {
      if (quack == null)
         throw new IllegalArgumentException("Poorly formed duck!");
      if (this.name == null && quack.name == null)
         return 0;
      else if (this.name == null) return -1;
      else if (quack.name == null) return 1;
      else return name.compareTo(quack.name);
   }
}
```

- **Classe `MissingDuck` :**
  - Implémente l'interface `Comparable`.
  - La méthode `compareTo` gère les objets null et compare les noms.
  - Si un nom est null, il est trié en premier.

Consistance de `compareTo()` et `equals()`

```
public class Product implements Comparable<Product> {
   private int id;
   private String name;

   public int hashCode() { return id; }
   public boolean equals(Object obj) {
      if(!(obj instanceof Product)) return false;
      var other = (Product) obj;
      return this.id == other.id;
   }
   public int compareTo(Product obj) {
      return this.name.compareTo(obj.name);
   }
}
```

- **Consistance** :
  - `x.equals(y)` doit être vrai si `x.compareTo(y)` est 0.
  - La classe `Product` montre une incohérence :
    - `equals()` compare les IDs.
    - `compareTo()` compare les noms, qui ne sont pas uniques.

Utilisation de `Comparator`

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Duck implements Comparable<Duck> {
   private String name;
   private int weight;

   public String toString() { return name; }

   public int compareTo(Duck d) {
      return name.compareTo(d.name);
   }

   public static void main(String[] args) {
      Comparator<Duck> byWeight = new Comparator<Duck>() {
         public int compare(Duck d1, Duck d2) {
            return d1.getWeight() - d2.getWeight();
         }
      };

      var ducks = new ArrayList<Duck>();
      ducks.add(new Duck("Quack", 7));
      ducks.add(new Duck("Puddles", 10));
      Collections.sort(ducks); // Tri par nom
      System.out.println(ducks); // [Puddles, Quack]
      Collections.sort(ducks, byWeight); // Tri par poids
      System.out.println(ducks); // [Quack, Puddles]
   }
}
```

- **Classe `Duck` :**
  - Peut être triée par nom ou poids.
  - `Comparator` défini pour trier par poids.
  - Exemples d'implémentation : classe interne, expression lambda, ou méthode de référence.

Comparaison de `Comparable` et `Comparator`

Caractéristique	Comparable	Comparator
Nom du package	`java.lang`	`java.util`
Méthode	`compareTo()`	`compare()`
Nombre de paramètres	1	2
Déclaration lambda	Non	Oui

Tri par plusieurs champs

```
public class Squirrel {
   private int weight;
   private String species;

   // Assume getters/setters/constructors fournis
}

public class MultiFieldComparator implements Comparator<Squirrel> {
   public int compare(Squirrel s1, Squirrel s2) {
      int result = s1.getSpecies().compareTo(s2.getSpecies());
      if (result != 0) return result;
      return s1.getWeight() - s2.getWeight();
   }
}
```

- Exemple de `Squirrel` :
  - `MultiFieldComparator` compare d'abord par espèce, puis par poids.
  - Utilisation de méthodes de chaîne comme `thenComparingInt()`.

Méthodes utilitaires pour `Comparator`

Méthode	Description
`comparing(function)`	Compare par les résultats d'une fonction.
`reversed()`	Inverse l'ordre du `Comparator` chainé.
`thenComparing(function)`	Utilise ce `Comparator` si le précédent est 0.

Lambdas et Interfaces Fonctionnelles

Expressions Lambda

- Une expression lambda est une fonction anonyme utilisée pour implémenter une méthode d'une interface fonctionnelle.
- Syntaxe : `(argument1, argument2) -> expression` ou `(argument1, argument2) -> { statements; }`

```
// Exemple d'une lambda
Runnable run = () -> System.out.println("Hello, Lambda!");
run.run(); // Affiche "Hello, Lambda!"
```

Interfaces Fonctionnelles

- Une interface fonctionnelle est une interface qui ne contient qu'une seule méthode abstraite.
- Les annotations @FunctionalInterface peuvent être utilisées pour indiquer une interface fonctionnelle.

exemple

```
@FunctionalInterface
interface MyFunctionalInterface {
    void execute();
}
```

Utilisation des Lambdas avec les Interfaces Fonctionnelles

Les lambdas simplifient la mise en œuvre des interfaces fonctionnelles.

```
// Implémentation de l'interface fonctionnelle
MyFunctionalInterface myFunc = () -> System.out.println("Executing!");
myFunc.execute(); // Affiche "Executing!"
```

Paramètres et Types de Retour

Les lambdas peuvent accepter des paramètres et retourner des valeurs.

exemple:

```
@FunctionalInterface
interface Adder {
    int add(int a, int b);
}

Adder adder = (a, b) -> a + b;
System.out.println(adder.add(5, 3)); // Affiche 8
```

Utilisation avec les Collections

Les lambdas sont souvent utilisées avec les API de collection, comme forEach, map, filter, etc.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name)); // Affiche chaque nom
```

Références de Méthodes

Les références de méthode peuvent être utilisées pour désigner des méthodes existantes au lieu d'utiliser une expression lambda.

exemple:

```
class Printer {
   static void print(String message) {
      System.out.println(message);
   }
}

Consumer<String> printer = Printer::print;
printer.accept("Hello, Method Reference!"); // Affiche "Hello, Method Reference!"
```

Composition des Fonctions

Les lambdas permettent de composer plusieurs fonctions ensemble.

```
@FunctionalInterface
interface StringProcessor {
    String process(String input);
}

StringProcessor toUpperCase = String::toUpperCase;
StringProcessor addExclamation = s -> s + "!";
StringProcessor combined = toUpperCase.andThen(addExclamation);
System.out.println(combined.process("hello")); // Affiche "HELLO!"
```

Interfaces Fonctionnelles Courantes en Java

1. `Consumer<T>`
   - **Description**: Opération qui prend un argument et ne retourne aucun résultat.
   - **Méthode Abstraite**: `void accept(T t)`
   - **Exemple**:

```
Consumer<String> print = s -> System.out.println(s);
print.accept("Hello, Consumer!"); // Affiche "Hello, Consumer!"
```

2. `Supplier<T>`
   - **Description**: Opération qui fournit un résultat sans accepter d'argument.
   - **Méthode Abstraite**: `T get()`
   - **Exemple**:

```
Supplier<String> supplier = () -> "Hello, Supplier!";
System.out.println(supplier.get()); // Affiche "Hello, Supplier!"
```

3. `Function<T, R>`
   - **Description**: Fonction qui prend un argument et produit un résultat.
   - **Méthode Abstraite**: `R apply(T t)`
   - **Exemple**:

```
Function<String, Integer> stringLength = s -> s.length();
System.out.println(stringLength.apply("Hello")); // Affiche 5
```

4. `Predicate<T>`
   - **Description**: Fonction qui prend un argument et retourne un booléen.
   - **Méthode Abstraite**: `boolean test(T t)`
   - **Exemple**:

```
Predicate<String> isEmpty = s -> s.isEmpty();
System.out.println(isEmpty.test("")); // Affiche true
```

5. `UnaryOperator<T>`
   - **Description**: Spécialisation de `Function` qui retourne un résultat du même type.
   - **Méthode Abstraite**: `T apply(T t)`
   - **Exemple**:

```
UnaryOperator<Integer> increment = x -> x + 1;
System.out.println(increment.apply(5)); // Affiche 6
```

6. `BinaryOperator<T>`
   - **Description**: Spécialisation de `BiFunction` qui retourne un résultat du même type.
   - **Méthode Abstraite**: `T apply(T t1, T t2)`
   - **Exemple**:

```
BinaryOperator<Integer> add = (a, b) -> a + b;
System.out.println(add.apply(3, 4)); // Affiche 7
```

7. `BiConsumer<T, U>`
   - **Description**: Opération qui prend deux arguments et ne retourne aucun résultat.
   - **Méthode Abstraite**: `void accept(T t, U u)`
   - **Exemple**:

```
BiConsumer<String, Integer> greet = (name, age) ->
    System.out.println("Hello " + name + ", you are " + age + " years old.");
greet.accept("Alice", 30); // Affiche "Hello Alice, you are 30 years old."
```

Modifier et combiner des appels

Interface	Méthode	Type de Retour	Paramètres
`Consumer`	`andThen()`	`Consumer`	`Consumer`
`Function`	`andThen()`	`Function`	`Function`
`Function`	`compose()`	`Function`	`Function`
`Predicate`	`and()`	`Predicate`	`Predicate`
`Predicate`	`negate()`	`Predicate`	—
`Predicate`	`or()`	`Predicate`	`Predicate`

Exemples d'utilisation

1. Utilisation de `Predicate`:

```
Predicate<String> egg = s -> s.contains("egg");
   Predicate<String> brown = s -> s.contains("brown");

   Predicate<String> brownEggs = egg.and(brown);
   Predicate<String> otherEggs = egg.and(brown.negate());
```

2. **Utilisation de `Predicate`**:

```
Consumer<String> c1 = x -> System.out.print("1: " + x);
Consumer<String> c2 = x -> System.out.print(", 2: " + x);

Consumer<String> combined = c1.andThen(c2);
combined.accept("Annie"); // Affiche "1: Annie, 2: Annie"
```

3. **Utilisation de `Function`**:

```
Function<Integer, Integer> before = x -> x + 1;
Function<Integer, Integer> after = x -> x * 2;

Function<Integer, Integer> combined = after.compose(before);
System.out.println(combined.apply(3)); // Affiche 8
```

API Streams

API Streams

- Utilisée pour la programmation fonctionnelle.
- Ne pas confondre avec les flux `java.io`, traités dans un autre chapitre.

Optional

- Représente un conteneur qui peut contenir une valeur ou être vide, pour exprimer l'absence de valeur.
- Création avec `Optional.empty()` ou `Optional.of(value)`.

- **Méthode pour Calculer la Moyenne**

```
public static Optional<Double> average(int... scores) {
    if (scores.length == 0) return Optional.empty();
    int sum = 0;
    for (int score: scores) sum += score;
    return Optional.of((double) sum / scores.length);
}
```

Exemples d’Appel:

- System.out.println(average(90, 100)); // Optional[95.0]
- System.out.println(average()); // Optional.empty

Vérification de la Valeur dans un Optional

Utiliser isPresent() pour vérifier la valeur et get() pour l’obtenir.

```
Optional<Double> opt = average(90, 100);
if (opt.isPresent())
   System.out.println(opt.get()); // 95.0
```

Méthodes Utiles de Optional

- **get()**
  - Vide: Lance une exception
  - Contient une valeur: Retourne la valeur
- **ifPresent(Consumer)**
  - Vide: Ne fait rien
  - Contient une valeur: Appelle le Consumer avec la valeur

- **isPresent()**
  - Vide: Retourne false
  - Contient une valeur: Retourne true
- **orElse(T other)**
  - Vide: Retourne l’autre paramètre
  - Contient une valeur: Retourne la valeur

- **orElseGet(Supplier)**
  - Vide: Retourne le résultat du Supplier
  - Contient une valeur: Retourne la valeur
- **orElseThrow()**
  - Vide: Lance NoSuchElementException
  - Contient une valeur: Retourne la valeur

Avantages de Optional par Rapport à null

- **Clarté**: Signale explicitement l'absence de valeur dans l'API.
- **Style fonctionnel**: Permet d'utiliser des méthodes comme `ifPresent()` au lieu d'instructions if classiques.
- **Chaining**: Possibilité de chaîner les appels sur Optional.

Utilisation des Streams

Introduction

- Un **stream** en Java est une séquence de données qui permet de traiter des collections de manière fonctionnelle.
- Une **pipeline de stream** est un ensemble d'opérations qui s'exécutent sur un stream pour produire un résultat.

Compréhension du Flux de Pipeline

- **Métaphore de l'assemblage** :
  - Pensez à une chaîne d'assemblage où chaque étape représente une opération sur les données.
  - Exemple : Pour créer des panneaux pour une exposition animale au zoo, chaque étape a un travail spécifique :
    - **Poste 1** : Retirer un panneau de la boîte.
    - **Poste 2** : Peindre le panneau.
    - **Poste 3** : Utiliser le pochoir pour le nom de l'animal.
    - **Poste 4** : Placer le panneau terminé dans une boîte pour l'exposition.

Caractéristiques des Pipelines

- **Évaluation paresseuse** :
  - Les opérations intermédiaires ne s'exécutent qu'à la demande de l'opération terminale. Cela signifie que le traitement n'est pas effectué tant que ce n'est pas nécessaire.
- **Éléments traités** :
  - Chaque élément est manipulé par une opération, puis il est « consommé » et ne peut plus être utilisé dans les étapes suivantes.
- **Flux fini vs. infini** :
  - Les flux peuvent être limités (ex. : une boîte de panneaux) ou infinis (ex. : un cycle de jour/nuit), mais une fois traités, ils

Différences entre Opérations Intermédiaires et Terminales

Aspect	Opérations Intermédiaires	Opérations Terminales
Nécessaires dans la pipeline ?	Non	Oui
Multiples dans la pipeline ?	Oui	Non
Type de retour	Type de stream	Type de résultat
Exécutées lors de l'appel ?	Non	Oui

Rôle de Java

- Java agit comme un **superviseur** de la chaîne d'assemblage, en attendant l'opération terminale pour démarrer l'exécution des opérations.
- Cela permet une gestion efficace des ressources et un traitement optimal des données.

Exemple d'Utilisation

1. **Source** : Prendre des panneaux de la boîte.
2. **Opération intermédiaire** : Peindre le panneau. (Cette opération ne commence qu'après que le panneau a été retiré.)

Utilisation des Streams en Java

Un **Stream** en Java est une séquence de données. Un **pipeline de Stream** est constitué des opérations qui s'exécutent sur un Stream pour produire un résultat.

Création de Streams

- **Stream.empty()**
  - **Type** : Fini
  - **Description** : Crée un Stream qui ne contient aucun élément.
  - **Exemple** :

```
Stream<String> emptyStream = Stream.empty();
```

- **Stream.of(varargs)**
  - **Type** : Fini
  - **Description** : Crée un Stream contenant les éléments passés en arguments.
  - **Exemple** :

```
Stream<String> stream = Stream.of("a", "b", "c");
```

- **coll.stream()**
  - **Type** : Fini
  - **Description** : Crée un Stream à partir d'une Collection existante (List, Set, etc.).
  - **Exemple** :

```
List<String> list = Arrays.asList("one", "two", "three");
Stream<String> streamFromList = list.stream();
```

- **coll.parallelStream()**
  - **Type** : Fini
  - **Description** : Crée un Stream pouvant s'exécuter en parallèle pour une meilleure performance.
  - **Exemple** :

```
List<String> list = Arrays.asList("one", "two", "three");
Stream<String> parallelStream = list.parallelStream();
```

- **Stream.generate(supplier)**
  - **Type** : Infini
  - **Description** : Crée un Stream en appelant le Supplier pour générer chaque élément.
  - **Exemple** :

```
Stream<Double> randomStream = Stream.generate(Math::random).limit(5);
```

- **Stream.iterate(seed, unaryOperator)**
  - **Type** : Infini
  - **Description** : Crée un Stream où chaque élément est généré à partir de l'élément précédent via une fonction.
  - **Exemple** :

```
Stream<Integer> integerStream = Stream.iterate(0, n -> n + 2).limit(5);
```

- **Stream.iterate(seed, predicate, unaryOperator)**
  - **Type** : Fini ou infini
  - **Description** : Crée un Stream à partir d'un seed, appliquant une fonction pour générer le prochain élément jusqu'à ce que le Predicate retourne false.
  - **Exemple** :

```
Stream<Integer> limitedStream = Stream.iterate(0, n -> n < 10, n -> n + 1);
```

Opérations Terminales

Les opérations terminales sont celles qui déclenchent le traitement des données dans un Stream et produisent un résultat ou un effet secondaire. Une fois qu'une opération terminale est exécutée, le Stream ne peut plus être utilisé.

- **count()**
  - **Signature** : `public long count()`
  - **Description** : Compte le nombre d'éléments dans le Stream.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
System.out.println(s.count()); // Affiche : 3
```

- **min()**
  - **Signature** :

    `public Optional<T> min(Comparator<? super T> comparator)`
  - **Description** : Trouve l'élément minimum selon un comparateur.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
Optional<String> min = s.min((s1, s2) -> s1.length() - s2.length());
min.ifPresent(System.out::println); // Affiche : ape
```

- **max()**
  - **Signature** :

    `public Optional<T> max(Comparator<? super T> comparator)`
  - **Description** : Trouve l'élément maximum selon un comparateur.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
Optional<String> max = s.max((s1, s2) -> s1.length() - s2.length());
max.ifPresent(System.out::println); // Affiche : monkey
```

- **findAny()**
  - **Signature** : `public Optional<T> findAny()`
  - **Description** : Trouve et retourne un élément du Stream, n'importe lequel.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.findAny().ifPresent(System.out::println); // Affiche : peut varier
```

- **findFirst()**
  - **Signature** : `public Optional<T> findFirst()`
  - **Description** : Retourne le premier élément du Stream.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");
s.findFirst().ifPresent(System.out::println); // Affiche : monkey
```

- **allMatch()**
  - **Signature** :

    ```
    public boolean allMatch(Predicate<? super T> predicate)
    ```

  - **Description** : Vérifie si tous les éléments correspondent au Predicate.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
boolean allStartWithM = s.allMatch(s1 -> s1.startsWith("m")); // false
```

- **anyMatch()**
  - **Signature** :

    ```
    public boolean anyMatch(Predicate<? super T> predicate)
    ```

  - **Description** : Vérifie si au moins un élément correspond au Predicate.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
boolean anyStartWithM = s.anyMatch(s1 -> s1.startsWith("m")); // true
```

- **noneMatch()**
  - **Signature** :

    ```
    public boolean noneMatch(Predicate<? super T> predicate)
    ```

  - **Description** : Vérifie si aucun élément ne correspond au Predicate.
  - **Exemple** :

```
Stream<String> s = Stream.of("monkey", "ape", "bonobo");
boolean noneStartWithZ = s.noneMatch(s1 -> s1.startsWith("z")); // true
```

Itération

- **forEach()**
  - **Signature** : `public void forEach(Consumer<? super T> action)`
  - **Description** : Applique une action (Consumer) à chaque élément du Stream.
  - **Exemple** :

```
Stream<String> s = Stream.of("Monkey", "Gorilla", "Bonobo");
s.forEach(System.out::print); // Affiche : MonkeyGorillaBonobo
```

Réduction

- **reduce()**
  - **Signatures** :
    - `public T reduce(T identity, BinaryOperator<T> accumulator)`
      - **Description** : Réduit le Stream à une seule valeur en utilisant un accumulateur.

- **Exemple** :

```java
Stream<String> stream = Stream.of("w", "o", "l", "f");
String word = stream.reduce("", String::concat);
System.out.println(word); // Affiche : wolf
```

```
- `public Optional<T> reduce(BinaryOperator<T> accumulator)`
  - **Description** : Réduit le Stream à une seule valeur sans valeur d'identité.
  - **Exemple** :
    ```java
    Stream<Integer> numbers = Stream.of(1, 2, 3, 4);
    Optional<Integer> sum = numbers.reduce(Integer::sum);
    sum.ifPresent(System.out::println); // Affiche : 10
    ```
```

Collecte

- **collect()**
  - **Signature** :

```
public <R> R collect(Supplier<R> supplier,
                     BiConsumer<R, ? super T> accumulator,
                     BiConsumer<R, R> combiner)
```

  - **Description** : Collecte les éléments du Stream dans une collection ou un autre type.

- **Exemple** :

```
List<String> collected = Stream.of("a", "b", "c")
                               .collect(Collectors.toList());
System.out.println(collected); // Affiche : [a, b, c]
```

Points Importants sur les Opérations Intermédiaires Courantes

- **Opérations Intermédiaires**
  - Produisent un **Stream** comme résultat.
  - Peuvent traiter des **streams infinis**.
  - Traitement **lazy** : les éléments sont produits uniquement lorsque nécessaire.

- **Filtrage**
  - **filter()** : Renvoie un Stream avec des éléments correspondant à une expression.

```
public Stream<T> filter(Predicate<? super T> predicate)
```

- **Suppression des Doublons**
  - **distinct()** : Renvoie un stream sans valeurs dupliquées.

```
public Stream<T> distinct()
```

- **Restriction par Position**
  - **limit()** : Limite le Stream à un nombre maximum d'éléments.
  - **skip()** : Ignore un nombre spécifié d'éléments.

```
public Stream<T> limit(long maxSize)
public Stream<T> skip(long n)
```

- **Mapping**
  - **map()** : Crée une correspondance un-à-un entre les éléments dans le stream et les éléments du prochain étape.

```
public <R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

- **Utilisation de flatMap**
  - **flatMap()** : Transforme chaque élément en un Stream et aplatit le résultat.

```
public <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

- **Concatenation de Streams**
  - **Stream.concat()** : Combine deux streams en un seul.

```
Stream.concat(one, two)
```

- **Tri**
  - **sorted()** : Renvoie un stream avec les éléments triés. Peut utiliser un comparateur.

```
public Stream<T> sorted()
public Stream<T> sorted(Comparator<? super T> comparator)
```

- **Prendre un Aperçu**
  - **peek()** : Permet d'exécuter une opération de stream sans changer le stream, utile pour le débogage.

```
public Stream<T> peek(Consumer<? super T> action)
```

Statistiques sur les streams

- **Calcul de la valeur maximale** :
  - Méthode `max(IntStream ints)` utilise `OptionalInt` pour récupérer le max d'un `IntStream`.
  - Si vide, lance une `RuntimeException`.

- **Calcul de la plage (range)** :
  - `range(IntStream ints)` utilise `IntSummaryStatistics` pour obtenir plusieurs statistiques.
  - Vérifie si le stream est vide et lance une exception si c'est le cas.
  - Retourne la différence entre la valeur max et min.

- **Statistiques disponibles avec `IntSummaryStatistics` :**
  - `getCount()` : nombre de valeurs.
  - `getAverage()` : moyenne des valeurs.
  - `getSum()` : somme des valeurs.
  - `getMin()` : plus petite valeur.
  - `getMax()` : plus grande valeur.

Concepts avancés des pipelines de Stream

- **Calculs de moyenne :**
  - `averagingDouble`, `averagingInt`, `averagingLong` : Moyenne pour types primitifs (Double, Int, Long).
  - **Résultat** : Double.

exemple

```
var ohMy = Stream.of("lions", "tigers", "bears");
TreeSet<String> result = ohMy
    .filter(s -> s.startsWith("t"))
    .collect(Collectors.toCollection(TreeSet::new));
System.out.println(result); // [tigers]
```

Concepts avancés de pipeline de Stream

Lien entre les Streams et les Données

- Les streams sont évalués paresseusement.
- Exemple : Ajouter un élément après la création d'un stream augmente le compte si le pipeline n'est pas encore exécuté.

```
var cats = new ArrayList<String>();
cats.add("Annie");
cats.add("Ripley");
var stream = cats.stream();
cats.add("KC");
System.out.println(stream.count()); // Affiche 3
```

Chaining des Optionals

- Simplifie l'écriture en évitant les structures imbriquées.
- Exemple : `map()`, `filter()` et `ifPresent()` pour traiter les Optionals.
- Utilisation de `flatMap()` pour éviter les Optional imbriqués.

```
private static void threeDigit(Optional<Integer> optional) {
    optional.map(n -> "" + n)
            .filter(s -> s.length() == 3)
            .ifPresent(System.out::println);
}
```

Exceptions Vérifiées et Interfaces Fonctionnelles

- La plupart des interfaces fonctionnelles ne supportent pas les exceptions vérifiées.
- Solutions : transformer l'exception en non vérifiée ou créer un wrapper avec `try/catch`.

Solution 1 : Transformer en exception non vérifiée.

```
Supplier<List<String>> s = () -> {
    try {
        return ExceptionCaseStudy.create();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
};
```

Solution 2 : Créer un wrapper avec try/catch.

```
private static List<String> createSafe() {
    try {
        return ExceptionCaseStudy.create();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Utilisation d'un Spliterator

- Divise une source de données pour un traitement parallèle.
- Utile pour répartir équitablement des tâches, comme diviser de la nourriture pour des enfants.

```
List<String> animals = List.of("Chien", "Chat", "Lapin", "Canard");
Spliterator<String> spliterator1 = animals.spliterator();
Spliterator<String> spliterator2 = spliterator1.trySplit();

spliterator1.forEachRemaining(System.out::println); // Traite une partie
spliterator2.forEachRemaining(System.out::println); // Traite l'autre partie
```

Utilisation d'un Spliterator

- Un **Spliterator** permet de contrôler le traitement des éléments d'une collection ou d'un stream en les divisant.
- Comparaison : un Spliterator est comme une sacoche de nourriture pour deux enfants, où chacun reçoit une partie de la nourriture.

Caractéristiques des Spliterators

- Dépendent de la source de données.
- Un Spliterator de collection est basique, tandis qu'un Spliterator de stream peut être parallèle ou infini.
- L'évaluation d'un stream est paresseuse.

Méthodes Clés d'un Spliterator

Méthode	Description
`Spliterator<T> trySplit()`	Renvoie un Spliterator contenant idéalement la moitié des données.
`void forEachRemaining(Consumer<T> c)`	Traite les éléments restants dans le Spliterator.
`boolean tryAdvance(Consumer<T> c)`	Traite un élément unique du Spliterator si disponible.

Exemple de Spliterator

- Exemple de division en trois sacs :

```
var stream = List.of("bird- ", "bunny- ", "cat- ", "dog- ", "fish- ", "lamb- ", "mouse- ");
Spliterator<String> originalBagOfFood = stream.spliterator();
Spliterator<String> emmasBag = originalBagOfFood.trySplit();
emmasBag.forEachRemaining(System.out::print);  // Affiche : bird- bunny- cat-

Spliterator<String> jillsBag = originalBagOfFood.trySplit();
jillsBag.tryAdvance(System.out::print);        // Affiche : dog-
jillsBag.forEachRemaining(System.out::print);  // Affiche : fish-

originalBagOfFood.forEachRemaining(System.out::print); // Affiche : lamb- mouse-
```

Exemple avec un stream infini

Traitement d'un stream infini :

```
var originalBag = Stream.iterate(1, n -> ++n).spliterator();
Spliterator<Integer> newBag = originalBag.trySplit();

newBag.tryAdvance(System.out::print); // Affiche : 1
newBag.tryAdvance(System.out::print); // Affiche : 2
newBag.tryAdvance(System.out::print); // Affiche : 3
```

Remarque : Utiliser forEachRemaining() sur un stream infini serait une mauvaise idée.

Collecte des Résultats

Vous avez presque terminé d'apprendre sur les flux. Le dernier sujet aborde la collecte des résultats. Vous avez vu l'opération terminale `collect()`. Il existe de nombreux collecteurs prédéfinis, y compris ceux présentés dans le tableau ci-dessous. Ces collecteurs sont disponibles via des méthodes statiques de la classe `Collectors`.

Il y a également un collecteur appelé `reducing()`, qui est un cas général à utiliser si les collecteurs précédents ne répondent pas à vos besoins.

Utilisation des Collecteurs de Base

De nombreux collecteurs fonctionnent de la même manière. Par exemple :

```
var ohMy = Stream.of("lions", "tigers", "bears");
String result = ohMy.collect(Collectors.joining(", "));
System.out.println(result); // lions, tigers, bears
```

Les collecteurs prédéfinis se trouvent dans la classe Collectors, et non dans l'interface Collector.

Exemples de Collecteurs de Groupement/Partitionnement

Collecteur	Description	Valeur retournée lorsqu'il est passé à `collect`
`averagingDouble(ToDoubleFunction f)`	Calcule la moyenne pour les types primitifs	`Double`
`counting()`	Compte le nombre	`Long`

Collecteur	Description	Valeur retournée lorsqu'il est passé à `collect`
`groupingBy(Function f)`	Crée une carte groupant par la fonction spécifiée	`Map<K, List<T>>`
`joining(CharSequence cs)`	Crée une chaîne unique avec le délimiteur	`String`
`partitioningBy(Predicate p)`	Crée une carte groupant par un prédicat	`Map<Boolean, List<T>>`

Collecte dans un Map:

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<String, Integer> map = ohMy.collect(
   Collectors.toMap(s -> s, String::length));
System.out.println(map); // {lions=5, bears=5, tigers=6}
```

Lorsque vous créez une Map, vous devez spécifier deux fonctions : une pour la clé et une pour la valeur.

Gestion des Clés Dupliquées

Si vous essayez de mapper des longueurs de noms d'animaux aux noms eux-mêmes, voici un exemple incorrect :

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, String> map = ohMy.collect(Collectors.toMap(
   String::length,
   k -> k)); // FAUX
```

Cela soulève une exception de clé dupliquée. Pour éviter cela, vous devez spécifier comment gérer les doublons :

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, String> map = ohMy.collect(Collectors.toMap(
   String::length,
   k -> k,
   (s1, s2) -> s1 + "," + s2));
System.out.println(map); // {5=lions,bears, 6=tigers}
```

Groupement et Partitionnement

Pour regrouper des noms par longueur :

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, List<String>> map = ohMy.collect(
   Collectors.groupingBy(String::length));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

Le partitionnement divise en deux groupes

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Boolean, List<String>> map = ohMy.collect(
   Collectors.partitioningBy(s -> s.length() <= 5));
System.out.println(map); // {false=[tigers], true=[lions, bears]}
```

Différence entre type objet et type référence

Type Objet

- **Définition** : Un type objet est une instance d'une classe qui encapsule des données et des comportements. Il peut avoir des méthodes et des attributs.

- **Exemple** :

```
class Person {
    String name;

    Person(String name) {
        this.name = name; // Initialisation de l'attribut
    }

    void introduce() {
        System.out.println("Je m'appelle " + name);
    }
}

Person p = new Person("Alice"); // `p` est un type objet.
p.introduce(); // Appel d'une méthode sur l'objet
```

Type Référence

- **Définition** : Un type référence est une référence à un objet en mémoire. Cela signifie qu'il ne contient pas l'objet lui-même, mais plutôt l'adresse mémoire où l'objet est stocké.

- **Exemple** :

```
Person p1 = new Person("Alice");
Person p2 = p1; // `p2` référence le même objet que `p1`.

p2.introduce(); // Affiche "Je m'appelle Alice"
p1.name = "Bob"; // Modification via p1
p2.introduce(); // Affiche "Je m'appelle Bob" (modification réfléchie)
```

Polymorphisme : Concept et Utilisation

Concept

- **Définition** : Le polymorphisme est la capacité d'une méthode à être appelée de différentes manières selon l'objet qui l'invoque. Il est principalement utilisé en POO pour permettre la flexibilité et la réutilisabilité du code.
- **Types** :
  - **Polymorphisme de sous-type** : Permet d'utiliser une méthode d'une sous-classe dans le contexte d'une super-classe.
  - **Polymorphisme paramétrique** : Utilisé dans les génériques, permet de définir des classes et méthodes avec des types

Utilisation

- **Exemple** :

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark"); // Redéfinition de la méthode
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Meow"); // Redéfinition de la méthode
    }
}

Animal myDog = new Dog();
Animal myCat = new Cat();
```

Programmation Orientée Objet (POO)

Concepts Clés

1. **Classes** : Définissent la structure d'un objet. Elles contiennent des attributs (données) et des méthodes (comportements).
2. **Objets** : Instances créées à partir des classes. Ils contiennent des valeurs spécifiques pour leurs attributs.
3. **Héritage** : Permet à une classe d'hériter des propriétés d'une autre classe. Cela favorise la réutilisation du code.
4. **Interfaces** : Contrats que les classes peuvent implémenter. Elles définissent des méthodes que les classes doivent fournir.

Exemple de Classe et Héritage

```java
class Vehicle {
    void start() {
        System.out.println("Vehicle started");
    }
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car started"); // Méthode redéfinie
    }
}

class Truck extends Vehicle {
    void start() {
        System.out.println("Truck started"); // Méthode redéfinie
    }
}

// Utilisation
Vehicle myCar = new Car();
Vehicle myTruck = new Truck();

myCar.start(); // Affiche "Car started"
myTruck.start(); // Affiche "Truck started"
```

Encapsulation, Héritage, Polymorphisme

Encapsulation

- Définition : Restriction de l'accès aux composants internes d'un objet. Cela signifie que les attributs d'un objet sont cachés (private) et accessibles uniquement via des méthodes publiques.
- Avantages : Protège les données, contrôle l'accès, et améliore la maintenance du code.

exemple:

```
class BankAccount {
    private double balance; // Attribut privé

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount; // Modification sécurisée
        }
    }

    public double getBalance() {
        return balance; // Accès contrôlé
    }
}

BankAccount account = new BankAccount();
account.deposit(100);
System.out.println(account.getBalance()); // Affiche 100.0
```

Héritage et Polymorphisme

Exemple de l’héritage avec polymorphisme :

```
class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}

class Cat extends Animal {
    void makeSound() {
        System.out.println("Meow"); // Comportement spécifique
    }
}

Animal myCat = new Cat();
myCat.makeSound(); // Affiche "Meow"
```

Casting d'Objets

Définition : Conversion d'un type d'objet à un autre. Cela nécessite que l'objet soit effectivement d'un type compatible.

exemple:

```
class Animal {}
class Dog extends Animal {}

Animal a = new Dog(); // Upcasting
Dog d = (Dog) a; // Downcasting, nécessite une vérification

if (a instanceof Dog) {
    Dog d = (Dog) a; // Sécurise le downcasting
}
```

Utilisation de l’opérateur instanceof

instanceof est un opérateur qui permet de vérifier si un objet est une instance d’une classe ou d’une interface. Cela évite les exceptions de type lors des castings.

exemple:

```
Animal a = new Dog();

if (a instanceof Dog) {
    System.out.println("C'est un chien"); // Vérifie le type avant de caster
} else {
    System.out.println("Ce n'est pas un chien");
}

// Utilisation d'instanceof avec une interface
interface Flyable {}
class Bird implements Flyable {}

Flyable f = new Bird();
if (f instanceof Bird) {
    System.out.println("C'est un oiseau");
}
```

Pattern Matching avec instanceof

Introduit dans **Java 16**, le pattern matching avec **instanceof** simplifie le code en combinant vérification de type et casting en une seule expression.

exemple:

```
if (a instanceof Dog d) { // Vérification et casting en un seul
    System.out.println("C'est un chien et son nom est " + d.name); // Accès direct à l'attribut
} else {
    System.out.println("Ce n'est pas un chien");
}
```

Avantages

- Lisibilité : Réduit le code répétitif.
- Sécurité : Évite les erreurs de casting en s’assurant que l’objet est du bon type.

Les classes abstract

- Une **classe abstraite** est une classe qui ne peut pas être instanciée directement et qui peut contenir des méthodes abstraites (sans implémentation) ainsi que des méthodes concrètes (avec implémentation).
- Elle sert de modèle pour d'autres classes.

Utilisation des Classes Abstraites : Une classe abstraite devient utilisable lorsqu'elle est étendue par une sous-classe concrète, qui doit implémenter toutes les méthodes abstraites héritées.

Exemple de Classe Abstraite

```
abstract class Animal {
    abstract void makeSound(); // Méthode abstraite

    void eat() { // Méthode concrète
        System.out.println("L'animal mange.");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof");
    }
}
```

Dans cet exemple, **Animal** est une classe **abstract**. La méthode makeSound() doit être implémentée par toute sous-classe, comme Dog, tandis que eat() peut être utilisée telle quelle.

Quand Utiliser des Classes Abstraites ?

Utilisez des classes abstraites lorsque :

- Vous avez des classes partageant des comportements communs, mais qui ne peuvent pas être instanciées.
- Vous souhaitez fournir une base partielle avec des méthodes partiellement implémentées pour des sous-classes.
- Vous avez besoin de définir des méthodes que les sous-classes doivent implémenter.

Intérêts des Classes Abstraites en POO

Réutilisation de Code : Les classes abstraites permettent de centraliser des comportements communs, évitant ainsi la duplication de code.
Encapsulation : Elles offrent un niveau d'encapsulation, permettant de contrôler la visibilité des méthodes et attributs.
Flexibilité : Les classes abstraites peuvent contenir des méthodes avec une implémentation par défaut, offrant une certaine flexibilité pour les sous-classes.

Type Abstract

Les classes abstraites et les interfaces sont toutes deux considérées comme des types abstraits, mais seules les interfaces utilisent des modificateurs implicites.

Déclaration de Méthode

```
abstract class Husky {     // 'abstract' requis dans la déclaration de classe
   abstract void play();   // 'abstract' requis dans la déclaration de méthode
}

interface Poodle {         // 'abstract' optionnel dans la déclaration d'interface
   void play();            // 'abstract' optionnel dans la déclaration de méthode
}
```

Les deux définitions de méthode sont abstraites. Cependant, la classe Husky ne se compile pas si la méthode play() n'est pas marquée comme abstraite, tandis que la méthode de l'interface Poodle se compile avec ou sans le modificateur abstrait

Les exceptions

Adaptation aux Changements : Les applications doivent gérer des situations imprévues, comme des données invalides ou des déconnexions de base de données, et être conçues pour s'adapter à des besoins futurs, comme la localisation.

Compréhension des Exceptions : Les erreurs de programme peuvent résulter de divers problèmes (erreur de code, connexion Internet perdue, valeurs non supportées par des méthodes). Les exceptions permettent de gérer ces situations.

Rôle des Exceptions : Les exceptions signalent qu'un problème est survenu. Le programme peut soit traiter l'exception, soit la transmettre à la méthode appelante. Cela reflète la gestion des erreurs dans le code.

Codes de Retour vs. Exceptions : Les exceptions sont préférées aux codes de retour pour signaler des erreurs. Bien que des codes de retour soient parfois utilisés (ex. retourner -1), il est recommandé d'utiliser le cadre d'exceptions fourni par Java pour une gestion des erreurs plus claire et efficace.

Types d'Exceptions

- **Exception** : Un événement qui modifie le flux du programme. La classe `Throwable` est la superclasse de toutes les exceptions en Java.
- **Catégories** :
  - **Checked Exceptions** : Doivent être déclarées ou gérées. Elles héritent de `Exception` mais pas de `RuntimeException`. Exemple : `IOException`.
  - **Unchecked Exceptions** : N'ont pas besoin d'être déclarées ou gérées. Elles incluent `RuntimeException` et `Error`. Exemple : `NullPointerException`.

catégorie d'exception


center

Checked Exceptions

- **Définition** : Exceptions anticipées, nécessitant une gestion explicite.
- **Règle de Gestion** : Les exceptions doivent être soit gérées avec des blocs `try` et `catch`, soit déclarées dans la signature de la méthode.
- **Exemple** :

```
void fall(int distance) throws IOException {
    if (distance > 10) {
        throw new IOException();
    }
}
```

Gestion des Checked Exceptions

Gestion avec try-catch

```
void fall(int distance) {
    try {
        if (distance > 10) {
            throw new IOException();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Note : Le bloc catch peut attraper IOException car il hérite de Exception.

Unchecked Exceptions

Exceptions non obligatoires à gérer ou déclarer. Souvent inattendues, mais pas nécessairement fatales.

Exemple : Une NullPointerException peut survenir dans ce code :

```
void fall(String input) {
    System.out.println(input.toLowerCase()); // Peut lancer NullPointerException
}
```

Remarque : Déclarer des exceptions non vérifiées est redondant car elles ne nécessitent pas de gestion obligatoire.

Lancer une exception en Java

- Le code Java peut lancer une exception, y compris le code que vous écrivez.
- Les exceptions fournies par Java peuvent inclure des exceptions fictives (ex. : `MyCustomException`).

Types d'Exceptions

1. **Exceptions causées par le code incorrect** :
   - Exemple :

```
String[] animals = new String[0];
System.out.println(animals[0]);  // ArrayIndexOutOfBoundsException
```

   - **Remarque** : Cette exception se produit car le tableau est vide.

2. **Exceptions lancées explicitement** :
    - Exemple :

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");
```

Utilisation des Mots-Clés `throw` et `throws`

- `throw` : Utilisé pour lancer une nouvelle exception ou relancer une exception existante.
- `throws` : Utilisé à la fin d'une déclaration de méthode pour indiquer les exceptions gérées par cette méthode.

Exemples pour lancer des Exceptions

- **Création et Lancement d'une Exception** :

```
var e = new RuntimeException();
throw e;  // Lancement de l'exception
```

Attention aux Erreurs de Compilation

Exemples incorrects:

```
throw RuntimeException();  // NE COMPILERA PAS
```

Raison : Manque le mot-clé new.

```
try {
    throw new RuntimeException();
    throw new ArrayIndexOutOfBoundsException();  // NE COMPILERA PAS
} catch (Exception e) {}
```

Raison : La seconde ligne est inaccessibles car la première lance déjà une exception.

Appel de Méthodes qui Lancent des Exceptions

Lors de l'appel d'une méthode qui lance une exception, les règles sont les mêmes que dans une méthode.

Problème de Compilation

```
class NoMoreCarrotsException extends Exception {}

public class Bunny {
   public static void main(String[] args) {
      eatCarrot();  // NE COMPILERA PAS
   }
   private static void eatCarrot() throws NoMoreCarrotsException {}
}
```

Raison : NoMoreCarrotsException est une exception vérifiée (checked exception) et doit être gérée ou déclarée.

Solutions de Compilation

1. Déclaration dans main()

```
public static void main(String[] args) throws NoMoreCarrotsException {
    eatCarrot();
}
```

2. Gestion avec try-catch

```
public static void main(String[] args) {
    try {
        eatCarrot();
    } catch (NoMoreCarrotsException e) {
        System.out.print("sad rabbit");
    }
}
```

Remarque : La méthode eatCarrot() déclare une exception sans la lancer, ce qui oblige le compilateur à exiger la gestion de l'exception.

Code Inaccessible

Code problématique:

```
public void bad() {
    try {
        eatCarrot();
    } catch (NoMoreCarrotsException e) {  // NE COMPILERA PAS
        System.out.print("sad rabbit");
    }
}

private void eatCarrot() {}
```

Raison : eatCarrot() ne peut pas lancer d'exception vérifiée, donc le bloc catch est

Types d'Exceptions à Reconnaître

1. **RuntimeException**
2. **Checked Exception**
3. **Error**

RuntimeException

- **Caractéristiques** : Exceptions non vérifiées (unchecked) qui n'ont pas besoin d'être gérées ou déclarées.
- **Sources** : Lancées par le programmeur ou la JVM.

Exemples Courants

Exception	Description
ArithmeticException	Lancée lors d'une division par zéro.
ArrayIndexOutOfBoundsException	Lancée lors de l'accès à un index illégal dans un tableau.
ClassCastException	Lancée lors d'un cast impossible.

Exception	Description
NullPointerException	Lancée lors d'un appel sur une référence nulle.
IllegalArgumentException	Lancée par le programmeur pour indiquer un argument illégal.
NumberFormatException	Sous-classe d'IllegalArgumentException, lancée lors d'une conversion de String en type numérique invalide.

Détails sur les Exceptions

1. **ArithmeticException**

```
int answer = 11 / 0;  // Lance ArithmeticException
```

2. **ArrayIndexOutOfBoundsException**

```
int[] countsOfMoose = new int[3];
System.out.println(countsOfMoose[-1]);  // Lance ArrayIndexOutOfBoundsException
```

3. **ClassCastException**

```
Object obj = "moose";
Integer number = (Integer) obj;  // Lance ClassCastException
```

4. NullPointerException

```
public void hop(String name, Integer jump) {
   System.out.print(name.toLowerCase() + " " + jump.intValue());
}
new Frog().hop(null, 1);  // Lance NullPointerException
```

5. IllegalArgumentException

```
public void setNumberEggs(int numberEggs) {
   if (numberEggs < 0)
      throw new IllegalArgumentException("# eggs must not be negative");
}
```

Checked Exception

Caractéristiques : Exceptions vérifiées (checked) qui doivent être gérées ou déclarées.

Exemples Courants :

Exception	Description
FileNotFoundException	Sous-classe de IOException, lancée lorsqu'un fichier est introuvable.
NotSerializableException	Sous-classe de IOException, lancée lors d'une tentative de sérialisation d'un objet non sérialisable.
SQLException	Lancée lors d'erreurs liées à des opérations sur une base de données.

Error

Caractéristiques : Exceptions non vérifiées (unchecked) étendant la classe Error, lancées par la JVM.

Remarque : Les erreurs ne devraient pas être gérées ou déclarées. Elles sont rares, mais peuvent inclure :

Erreur	Description
OutOfMemoryError	Indique que la JVM ne peut pas allouer plus de mémoire.
StackOverflowError	Indique qu'une méthode a dépassé la profondeur de la pile.
NoClassDefFoundError	Indique que la JVM ne peut pas trouver une classe requise.

Utilisation des déclarations try et catch

- **try** : Sépare la logique pouvant lancer une exception de celle qui la gère.
- Si une exception se produit dans le bloc **try**, l'exécution passe au bloc **catch** correspondant.
- Les accolades sont obligatoires pour les blocs **try** et **catch**.

Chaining des blocs catch

- Les exceptions peuvent être chaînées.
- Ordre des blocs **catch** important : un bloc plus spécifique doit être placé avant un bloc plus général.
- Exemple :

```
catch (AnimalsOutForAWalk e) { }
catch (ExhibitClosed e) { }
```

Multi-catch

Permet de capturer plusieurs types d’exceptions dans le même bloc.

Syntaxe :

```
catch (Exception1 | Exception2 e) { }
```

Une seule variable d’exception par bloc multi-catch.

Bloc finally

S’exécute toujours, que l’exception soit lancée ou non.

Permet de libérer des ressources ou effectuer des actions de nettoyage.

Exemple :

```
try { /* code */ }
catch (Exception e) { /* gestion d'exception */ }
finally { /* code exécuté en dernier */ }
```

- Les blocs catch doivent être correctement ordonnés pour éviter le code inaccessible.
- Un bloc finally peut exister sans bloc catch.
- Un bloc try doit toujours être suivi d’au moins un catch ou finally.

Introduction aux génériques

Les génériques permettent de définir des classes, interfaces et méthodes avec des types paramétrés, permettant de travailler avec n'importe quel type tout en garantissant la sécurité des types.

Définir une classe générique

Exemple d'une classe générique `Box` qui peut contenir un objet de n'importe quel type :

```
public class Box<T> {
    private T value;
    public void setValue(T value) { this.value = value; }
    public T getValue() { return value; }
}
```

Création d’un enregistrement générique

Les génériques peuvent aussi être utilisés avec des enregistrements (records) :

```
public record Container<T>(T item) {
    public T getItem() {
        if (item == null) throw new IllegalStateException("Item absent");
        return item;
    }
}
```

Exemple d’utilisation :

```
Container<String> container = new Container<>("Bonjour");
```

Comprendre les wildcards non bornées

Les wildcards (?) représentent n'importe quel type. Exemple de méthode pour afficher une liste de n'importe quel type :

```
public static void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}
```

Utilisation de la wildcard non bornée

Exemple d’utilisation de la méthode printList :

```
List<String> list = new ArrayList<>();
list.add("Java");
printList(list); // Fonctionne avec List<String>
```

Wildcard avec borne supérieure

Une wildcard avec une borne supérieure (? extends T) restreint le type aux sous-types de T. Exemple avec des numéros :

```
public static double sum(List<? extends Number> numbers) {
    double total = 0;
    for (Number number : numbers) {
        total += number.doubleValue();
    }
    return total;
}
```

Exemple de wildcard avec borne supérieure

```
List<Integer> intList = new ArrayList<>();
intList.add(10);
sum(intList);  // Fonctionne avec List<Integer>
```

Wildcard avec borne inférieure

Une wildcard avec une borne inférieure (? super T) permet de travailler avec des supertypes de T :

```
public static void addString(List<? super String> list) {
    list.add("Hello");
}
```

Exemple de wildcard avec borne inférieure

```
List<Object> objList = new ArrayList<>();
addString(objList); // Fonctionne avec List<Object>
```

Pourquoi utiliser des bornes ?

Les bornes supérieures permettent de lire des objets d'un type spécifique et de ses sous-types, tandis que les bornes inférieures sont utilisées pour ajouter des objets d'un type spécifique et de ses sous-types.

Combiner des génériques avec des collections

Les génériques sont souvent utilisés avec les collections. Exemple de création d'une List générique avec un type Integer :

```
List<Integer> numbers = new ArrayList<>();
numbers.add(10);
numbers.add(20);
```

Erreur lors de l’utilisation de List avec un type spécifique

```
List<Object> objects = new ArrayList<>();
List<Integer> integers = new ArrayList<>();
objects = integers;  // Erreur de compilation
```

Les types génériques sont invariants en Java, ce qui empêche l’assignation de List<Integer> à List<Object>.

Génériques et héritage

Les génériques ne permettent pas de faire de conversion automatique entre types, même si un type est un sous-type d'un autre.

```
List<Number> numbers = new ArrayList<>();
List<Integer> integers = new ArrayList<>();
// numbers = integers; // Erreur : types incompatibles
```

Utilisation des génériques avec des interfaces

Les interfaces génériques permettent également d'imposer des types lors de l'implémentation. Exemple :

```
interface Pair<T, U> {
    T getFirst();
    U getSecond();
}
```

Exemple de classe générique avec une méthode

```
public class Pair<T, U> {
    private T first;
    private U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
    }
}
```

Utilisation d’un Map avec des génériques

Les Map génériques permettent de définir des paires clé-valeur avec des types spécifiques :

```
Map<String, Integer> ageMap = new HashMap<>();
ageMap.put("Alice", 30);
ageMap.put("Bob", 25);
```

Génériques avec des méthodes

Les méthodes génériques permettent de définir des méthodes flexibles qui peuvent fonctionner avec différents types :

```
public static <T> void printArray(T[] array) {
    for (T element : array) {
        System.out.println(element);
    }
}
```

Création d’une méthode générique avec des bornes

Les méthodes génériques peuvent aussi inclure des bornes pour restreindre les types acceptés. Exemple :

```
public static <T extends Number> double add(T a, T b) {
    return a.doubleValue() + b.doubleValue();
}
```

Génériques avec des exceptions

Les génériques peuvent être utilisés pour gérer des exceptions spécifiques, comme pour des exceptions liées à un type :

```
public class InvalidTypeException<T> extends Exception {
    public InvalidTypeException(String message) {
        super(message);
    }
}
```

- Utilisez les génériques pour garantir la sécurité des types.
- Les wildcards permettent une grande flexibilité, mais attention aux contraintes.
- Combinez les génériques avec les collections pour des structures de données robustes.

La concurrence

Threads et Multithreading

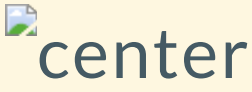
- Les opérations disque/réseau sont plus lentes que les opérations CPU, ce qui peut bloquer le système.
- Le multithreading permet de gérer plusieurs tâches simultanément, améliorant les performances.
- Les systèmes d'exploitation utilisent la gestion multithread pour éviter les blocages.

Concepts de Base des Threads

- Un **thread** est la plus petite unité d'exécution planifiable par le système d'exploitation.
- Un **processus** est un groupe de threads partageant le même espace mémoire.
- Un processus peut être **monothread** (1 thread) ou **multithread** (plusieurs threads).

Mémoire Partagée et Communication

- Les threads partagent des variables statiques, des instances et des variables locales transmises.
- Les variables statiques sont accessibles par tous les threads d’un processus.
- Les threads communiquent directement via l’espace mémoire partagé, évitant des copies redondantes.


center

Tâches et Threads

- Une **tâche** est une unité de travail qu'un thread exécute séquentiellement.
- Un thread peut exécuter plusieurs tâches indépendantes, mais une seule à la fois.
- Les expressions lambda simplifient l'écriture des tâches dans la programmation Java.

Comprendre la Concurrence des Threads

- La **concurrence** permet d'exécuter plusieurs threads/processus simultanément.
- Le **thread scheduler** attribue des cycles CPU aux threads (ex: planification round-robin).
- Les interruptions ou priorités de threads influencent l'ordre d'exécution.
- Exemple simple :

```
Runnable task = () -> System.out.println("Task executed");
new Thread(task).start();
```

Création et Gestion des Threads

- **Runnable** est une interface fonctionnelle pour définir des tâches de threads.
- Exemple : `new Thread(() -> System.out.print("Hello")).start();`
- L'exécution asynchrone ne garantit pas l'ordre des tâches entre threads.

Différences entre start() et run()

- **start()** exécute une tâche sur un thread séparé.

```
new Thread(() -> System.out.println("Thread started")).start();
```

- **run()** exécute la tâche dans le thread actuel, bloquant le programme.

```
new Thread(() -> System.out.println("Running task")).run();
```

- Toujours utiliser `start()` pour les tâches multithread.

Meilleures Pratiques avec Runnable

- Utiliser des **expressions lambda** pour simplifier la création de tâches.

```
Runnable printTask = () -> System.out.println("Hello from thread");
new Thread(printTask).start();
```

- Étendre la classe `Thread` uniquement si des méthodes supplémentaires doivent être surchargées.

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Custom thread logic");
    }
}
new MyThread().start();
```

- Préférer `Runnable` pour une intégration facile avec l'API Concurrency.

```
Runnable task = () -> {
    for (int i = 0; i < 3; i++) {
        System.out.println("Task execution: " + i);
    }
};
new Thread(task).start();
```

Types de Threads en Java

- **Threads système** :
  - Créés par la JVM, exécutent des tâches en arrière-plan.
  - Exemple : **Garbage Collector**.
- **Threads définis par l'utilisateur** :
  - Créés par le développeur pour des tâches spécifiques.
  - Par défaut, un programme a un thread utilisateur : celui qui exécute `main()`.

- **Threads virtuels** *(introduits en Java 21)* :
  - Légers, gérés directement par la JVM.
  - Permettent des millions de threads simultanés.
  - Utilisent un modèle basé sur les continuations.
  - Parfaits pour les applications à forte concurrence.

Threads Daemon

- **Threads daemon** :
  - N'empêchent pas la JVM de s'arrêter.
  - Exemple : Garbage Collector (thread daemon par défaut).
- **Threads utilisateur** :
  - La JVM attend leur fin pour se terminer.

Exemple : Thread utilisateur vs daemon

```
public class Zoo {
    public static void pause() {
        try {
            Thread.sleep(10_000); // Attend 10 secondes
        } catch (InterruptedException e) {}
        System.out.println("Thread terminé !");
    }

    public static void main(String[] args) {
        var job = new Thread(() -> pause());
        // job.setDaemon(true); // Activer pour un daemon
        job.start();
        System.out.println("Main terminé !");
    }
}
```

Sortie (sans daemon) :

```
Main terminé !
(Thread attend 10 secondes)
Thread terminé !
```

Sortie (avec daemon) :

```
Main terminé !
```

Threads Virtuels : Détails

- Fournissent une alternative efficace aux threads traditionnels.
- Évitent les problèmes liés à la limitation des threads OS.
- Transparence pour les développeurs : **API identique à Thread classique**.
- Idéal pour les serveurs hautement concurrentiels.

Création d'un thread virtuel

```
public static void main(String[] args) {
    var virtualThread = Thread.ofVirtual().start(() -> {
        System.out.println("Thread virtuel en exécution !");
    });
    System.out.println("Main terminé !");
}
```

Exemple : Avantages des threads virtuels

Sans thread virtuel (classique)

```
public class ClassicalThreads {
    public static void main(String[] args) {
        for (int i = 0; i < 10_000; i++) {
            new Thread(() -> System.out.println("Thread classique")).start();
        }
    }
}
```

- Limité par les ressources système.
- Les threads OS consomment beaucoup de mémoire.

Avec threads virtuels

```
public class VirtualThreads {
    public static void main(String[] args) {
        for (int i = 0; i < 10_000; i++) {
            Thread.ofVirtual().start(() -> System.out.println("Thread virtuel"));
        }
    }
}
```

- Supporte des millions de threads.
- Réduit l’utilisation des ressources système.

Cycle de vie des Threads

États des Threads

1. NEW : Thread créé mais pas encore démarré.
2. RUNNABLE : Prêt à s'exécuter (mais pas forcément en cours d'exécution).
3. TERMINATED : Thread terminé ou exception non interceptée.
4. États d'attente :
   - BLOCKED : En attente d'accéder à une ressource synchronisée.
   - WAITING : Attend indéfiniment une notification.
   - TIMED_WAITING : Attend une durée spécifique (ex. : sleep()).

Points importants sur les Threads Virtuels

- Performants : Réduction de la surcharge des threads OS.
- Simplicité : Pas besoin de changer les paradigmes existants.
- Idéal pour :
  - Serveurs HTTP hautement concurrentiels.
  - Applications nécessitant un grand nombre de connexions.

États d'un Thread

```
public class CheckResults {
    private static int counter = 0;

    public static void main(String[] args) {
        new Thread(() -> {
            for (int i = 0; i < 1_000_000; i++) counter++;
        }).start();

        while (counter < 1_000_000) {
            System.out.println("Pas encore atteint");
        }
        System.out.println("Atteint : " + counter);
    }
}
```

Amélioration avec sleep()

Mauvaise pratique : Boucle infinie

```
while (counter < 1_000_000) {
    System.out.println("Pas encore atteint");
}
```

Utiliser Thread.sleep():

```
while (counter < 1_000_000) {
    System.out.println("Pas encore atteint");
    try {
        Thread.sleep(1_000); // Pause de 1 seconde
    } catch (InterruptedException e) {
        System.out.println("Interrompu !");
    }
}
```

Avantage: le CPU pour d'autres tâches.

Points importants

- Par défaut, les threads définis par l'utilisateur ne sont pas daemon.
- Utiliser les threads daemon pour des tâches non essentielles.
- Éviter les boucles infinies pour vérifier des conditions, préférer Thread.sleep().

Exemple de Polling sans Sleep

Voici un exemple de polling où un thread modifie une variable partagée et le thread principal attend que cette variable atteigne un certain seuil :

```
public class CheckResults {
   private static int counter = 0;
   public static void main(String[] args) {
      new Thread(() -> {
         for (int i = 0; i < 1_000_000; i++) counter++;
      }).start();
      while (counter < 1_000_000) {
         System.out.println("Not reached yet");
      }
      System.out.println("Reached: " + counter);
   }
```

Problèmes

- Ce programme utilise un while() pour vérifier la valeur de counter.
- Il peut imprimer “Not reached yet” plusieurs fois, voire des millions de fois.
- Cela consomme des ressources CPU sans raison, car le thread principal tourne indéfiniment.

Amélioration avec Thread.sleep()

En introduisant la méthode Thread.sleep(), on permet au thread principal de faire une pause et de libérer le CPU, ce qui améliore les performances en évitant un usage excessif du processeur.

```java
public class CheckResultsWithSleep {
   private static int counter = 0;
   public static void main(String[] args) {
      new Thread(() -> {
         for (int i = 0; i < 1_000_000; i++) counter++;
      }).start();
      while (counter < 1_000_000) {
         System.out.println("Not reached yet");
         try {
            Thread.sleep(1_000);  // Pause de 1 seconde
         } catch (InterruptedException e) {
            System.out.println("Interrupted!");
         }
      }
      System.out.println("Reached: " + counter);
   }
}
```

Explication

- Le Thread.sleep(1000) permet au thread principal de faire une pause de 1 seconde à chaque itération du while.
- Cela libère le CPU pour d’autres tâches et évite un travail inutile.

Limites du Polling avec Sleep

Bien que l'utilisation de sleep() réduise l'utilisation des ressources CPU, plusieurs problèmes subsistent :

- Incertitude du nombre d'exécutions : Le programme ne garantit pas combien de fois le while() sera exécuté avant que la condition counter < 1_000_000 ne soit remplie.
- Précision du délai : L'exécution de la boucle peut être retardée en raison d'autres processus ayant une priorité plus élevée.
- Accès aux ressources partagées : Si plusieurs threads accèdent à des ressources partagées, il existe un risque d'obtenir des valeurs inattendues à cause de problèmes de synchronisation.

Amélioration avec Thread.interrupt()

Une autre amélioration consiste à permettre au thread secondaire d’interrompre le thread principal une fois le travail terminé, évitant ainsi un délai inutile.

```java
public class CheckResultsWithSleepAndInterrupt {
   private static int counter = 0;
   public static void main(String[] args) {
      final var mainThread = Thread.currentThread();
      new Thread(() -> {
         for (int i = 0; i < 1_000_000; i++) counter++;
         mainThread.interrupt(); // Interruption du thread principal
      }).start();
      while (counter < 1_000_000) {
         System.out.println("Not reached yet");
         try {
            Thread.sleep(1_000);  // Pause de 1 seconde
         } catch (InterruptedException e) {
            System.out.println("Interrupted!");
         }
      }
      System.out.println("Reached: " + counter);
   }
}
```

Explication

- Le thread secondaire appelle mainThread.interrupt() une fois que le compteur atteint 1 million.
- Cela réveille le thread principal de son état TIMED_WAITING et le réactive, ce qui permet de terminer l'exécution sans attendre le temps de pause complet.
- L'interruption déclenche une exception InterruptedException qui est gérée par le thread principal.

Le polling avec sleep() permet d'améliorer l'efficacité d'un programme multithreadé en réduisant l'usage excessif de CPU. Toutefois, il reste important de gérer correctement l'accès aux ressources partagées et d'améliorer la réactivité du programme en utilisant des mécanismes comme l'interruption de threads.

Créer des Threads avec l'API de Concurrence

Java inclut le package `java.util.concurrent`, également connu sous le nom d'API de Concurrence, pour faciliter la gestion des threads. Cette API comprend l'interface `ExecutorService`, qui définit des services permettant de créer et gérer des threads.

Vous devez d'abord obtenir une instance de l'interface `ExecutorService`, puis envoyer des tâches à traiter. Ce cadre propose de nombreuses fonctionnalités utiles, telles que le pooling de threads et la planification. Il est recommandé d'utiliser ce cadre dès que vous devez créer et exécuter une tâche séparée, même pour un seul thread.

Introduction à l'Executor à Thread Unique

Puisque `ExecutorService` est une interface, comment obtenir une instance de celle-ci ? L'API de Concurrence inclut la classe `Executors`, qui peut être utilisée pour créer des instances de `ExecutorService`.

Voici un exemple d'utilisation avec deux instances de `Runnable` :

```
ExecutorService service = Executors.newSingleThreadExecutor();
try {
    System.out.println("début");
    service.execute(printInventory);
    service.execute(printRecords);
    service.execute(printInventory);
    System.out.println("fin");
} finally {
    service.shutdown();
}
```

Dans cet exemple, la méthode newSingleThreadExecutor() crée le service. Contrairement à notre exemple précédent avec quatre threads, ici nous avons seulement deux threads (un principal et un autre pour l'exécution des tâches). Cela réduit la variation dans l'ordre d'exécution des tâches.

Arrêter un Executor de Thread

Une fois que vous avez terminé d'utiliser un thread executor, il est important d'appeler la méthode shutdown(). Un thread executor crée un thread non-deamon lors de la première tâche exécutée. Si shutdown() n'est pas appelé, l'application ne se terminera jamais.

Le processus de fermeture de l'executor de thread consiste à rejeter toutes les nouvelles tâches soumises tout en continuant d'exécuter celles déjà envoyées. Pendant cette période, la méthode isShutdown() renverra true, tandis que isTerminated() renverra false. Une fois toutes les tâches terminées, les deux méthodes renverront true.

Arrêter un Executor de Thread

Pour arrêter un `ExecutorService`, vous devez appeler la méthode `shutdown()` ou `shutdownNow()`.

- `shutdown()` : Arrête l'`ExecutorService` de manière ordonnée. Les tâches en cours d'exécution continuent, mais aucune nouvelle tâche n'est acceptée.
- `shutdownNow()` : Arrête immédiatement l'`ExecutorService`. Les tâches en cours sont annulées et aucune nouvelle tâche n'est acceptée.

Exemple d'arrêt ordonné avec `shutdown()`

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

public class ShutdownExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre des tâches
        executor.submit(() -> System.out.println("Tâche 1 en cours"));
        executor.submit(() -> System.out.println("Tâche 2 en cours"));

        // Arrêter l'executor de manière ordonnée
        executor.shutdown();

        // Vérification si l'executor est arrêté
        if (executor.isShutdown()) {
            System.out.println("L'executor a été arrêté.");
        }
    }
}
```

Exemple d’arrêt immédiat avec shutdownNow()

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

public class ShutdownNowExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre des tâches
        executor.submit(() -> System.out.println("Tâche 1 en cours"));
        executor.submit(() -> System.out.println("Tâche 2 en cours"));

        // Arrêter immédiatement l'executor
        List<Runnable> tâchesNonTerminées = executor.shutdownNow();

        // Vérification des tâches non terminées
        System.out.println("Tâches non terminées : " + tâchesNonTerminées);
    }
}
```

Soumettre des Tâches

Vous pouvez soumettre des tâches à une instance de ExecutorService de plusieurs façons. La méthode execute() que nous avons présentée précédemment hérite de l'interface Executor, que ExecutorService implémente. Cette méthode permet d'exécuter une tâche de type Runnable de manière asynchrone.

Cependant, la méthode submit() de ExecutorService offre l'avantage de retourner une instance de Future. Cela permet de déterminer si la tâche est terminée et d'obtenir un résultat générique après son exécution.

Exemple de soumission avec execute() (pour Runnable)

La méthode execute() est utilisée pour soumettre des tâches de type Runnable sans retour de valeur.

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

public class ExecuteExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre des tâches à l'executor
        executor.execute(() -> System.out.println("Exécution de la tâche 1"));
        executor.execute(() -> System.out.println("Exécution de la tâche 2"));

        // Arrêter l'executor après l'exécution des tâches
        executor.shutdown();
    }
```

Exemple de soumission avec submit() (pour Runnable et Callable)

La méthode submit() retourne un objet Future, permettant de suivre l’exécution de la tâche.

Avec Runnable :

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

public class SubmitRunnableExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre une tâche Runnable
        Future<?> future = executor.submit(() -> {
            System.out.println("Exécution de la tâche 1");
        });

        // Attendre la fin de l'exécution
        future.get();

        // Arrêter l'executor après l'exécution des tâches
        executor.shutdown();
    }
}
```

Avec Callable (avec retour de valeur) :

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

public class SubmitCallableExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre une tâche Callable qui retourne une valeur
        Future<Integer> future = executor.submit(() -> {
            return 42; // Retourne une valeur
        });

        // Obtenir le résultat de la tâche
        Integer result = future.get();
        System.out.println("Résultat de la tâche : " + result);

        // Arrêter l'executor après l'exécution des tâches
        executor.shutdown();
    }
}
```

Gestion des Exceptions lors de la soumission des tâches

Lors de l’utilisation de Callable, vous pouvez gérer les exceptions directement.

Exemple avec Callable et gestion d’exception

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

public class CallableWithExceptionExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Soumettre une tâche Callable qui lance une exception
        Future<Integer> future = executor.submit(() -> {
            if (true) throw new Exception("Une erreur est survenue");
            return 42;
        });

        try {
            // Essayer de récupérer le résultat de la tâche
            Integer result = future.get();
        } catch (ExecutionException e) {
            System.out.println("Erreur lors de l'exécution de la tâche : " + e.getCause().getMessage());
        }

        // Arrêter l'executor après l'exécution des tâches
        executor.shutdown();
    }
}
```

Attendre les Résultats

Comment savoir quand une tâche soumise à un ExecutorService est terminée ? Comme mentionné précédemment, la méthode submit() retourne une instance de Future, qui peut être utilisée pour vérifier l'état d'une tâche.

```
Future<?> future = service.submit(() -> System.out.println("Hello"));
```

Voici un exemple d'utilisation de Future pour attendre un résultat avec un délai :

```
import java.util.concurrent.*;
public class CheckResults {
   private static int counter = 0;
   public static void main(String[] unused) throws Exception {
      ExecutorService service = Executors.newSingleThreadExecutor();
      try {
         Future<?> result = service.submit(() -> {
            for(int i = 0; i < 1_000_000; i++) counter++;
         });
         result.get(10, TimeUnit.SECONDS);  // Retourne null pour Runnable
         System.out.println("Terminé !");
      } catch (TimeoutException e) {
         System.out.println("Non terminé dans le temps imparti");
      } finally {
         service.shutdown();
      }
   }
}
```

Dans cet exemple, **result.get(10, TimeUnit.SECONDS)** attend que la tâche soit **terminée** pendant 10 secondes, et lève une exception **TimeoutException** si la tâche n'est pas **terminée** à temps.

L’interface Callable

L’interface Callable est similaire à Runnable, mais elle permet de retourner un résultat et peut lever des exceptions vérifiées. Voici sa définition :

```
@FunctionalInterface public interface Callable<V> {
    V call() throws Exception;
}
```

La méthode submit() accepte des objets Callable et retourne un Future<T>. Contrairement à Runnable, où la méthode get() retourne toujours null, avec Callable, la méthode get() retourne un résultat correspondant au type générique de la tâche.

```
var service = Executors.newSingleThreadExecutor();
try {
   Future<Integer> result = service.submit(() -> 30 + 11);
   System.out.println(result.get());   // Affiche 41
} finally {
   service.shutdown();
}
```

Ce code soumet une tâche de type Callable et affiche le résultat lorsque la tâche est terminée.

Sécurité des threads

La sécurité des threads garantit l'exécution sécurisée d'un objet par plusieurs threads en même temps. Comme les threads partagent un environnement et un espace mémoire, il faut organiser l'accès aux données pour éviter des résultats invalides ou inattendus.

Problème d'accès concurrent aux données

Imaginons que nous ayons un programme pour compter les moutons dans un zoo. Chaque travailleur du zoo exécute une tâche concurrente pour ajouter un mouton et rapporter le total. Voici un exemple de code pour cette simulation :

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

public class SheepManager {
    private int sheepCount = 0;

    private void incrementAndReport() {
        System.out.print((++sheepCount) + " ");
    }

    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(20);
        try {
            SheepManager manager = new SheepManager();
            for (int i = 0; i < 10; i++)
                service.submit(() -> manager.incrementAndReport());
        } finally {
            service.shutdown();
        }
    }
}
```

Ce programme peut produire des sorties inattendues, comme :

```
1 9 8 7 3 6 6 2 4 5
```

Le problème réside dans l'incrémentation de sheepCount. Lorsqu'un thread lit la valeur avant qu'un autre ne la modifie, les deux threads peuvent écrire la même valeur, ce qui mène à une perte de comptage.

Mot-clé volatile

Le mot-clé **volatile** garantit l'accès **cohérent** aux données dans la mémoire. Il permet de s'assurer qu'une seule modification de variable est effectuée à la fois. Cependant, l'usage de **volatile** n'assure pas la sécurité complète des **threads**, car des opérations complexes (comme ++sheepCount) restent non sécurisées.

```
private volatile int sheepCount = 0;
private void incrementAndReport() {
    System.out.print((++sheepCount) + " ");
}
```

Même avec volatile, ce code reste vulnérable aux conditions de concurrence. Le problème provient du fait que l'opération ++ est composée de deux actions distinctes (lecture et écriture).

Classes atomiques

Les classes atomiques fournissent une méthode pour effectuer des opérations atomiques, assurant qu'une opération complète est effectuée sans interruption par un autre thread. Par exemple, AtomicInteger permet de garantir qu'une variable entière est modifiée de manière atomique.

Exemple avec AtomicInteger :

```
private AtomicInteger sheepCount = new AtomicInteger(0);

private void incrementAndReport() {
    System.out.print(sheepCount.incrementAndGet() + " ");
}
```

Ce code garantit que le comptage des moutons sera toujours correct, sans perdre de valeurs à cause de l'exécution concurrente. La sortie ressemblera à :

```
1 2 3 4 5 6 7 8 9 10
```

Résumé des classes atomiques

Voici quelques classes atomiques importantes dans l'API Java Concurrency :

Nom de la classe	Description
`AtomicBoolean`	Valeur booléenne modifiable de manière atomique
`AtomicInteger`	Entier modifiable de manière atomique
`AtomicLong`	Long modifiable de manière atomique

Améliorer l'accès avec des blocs

`synchronized`

Les classes atomiques sont idéales pour protéger une seule variable, mais ne sont pas suffisantes lorsqu'il faut exécuter une série de commandes ou appeler une méthode. Par exemple, elles ne permettent pas de mettre à jour deux variables atomiques simultanément. Comment améliorer cela afin que chaque travailleur puisse incrémenter et rapporter les résultats dans l'ordre ?

La technique la plus courante consiste à utiliser un **monitor** pour synchroniser l'accès. Un monitor, également appelé **verrou**, est une structure qui prend en charge l'exclusion mutuelle, c'est-à-dire la propriété qu'à tout moment, au plus un thread exécute une section de code donnée.

Exemples de méthodes atomiques courantes

Méthode	Description
`get()`	Récupère la valeur actuelle
`set(newValue)`	Définit la nouvelle valeur
`getAndSet(newValue)`	Définit la nouvelle valeur et retourne l'ancienne

Méthode	Description
`incrementAndGet()`	Incrémente numériquement et retourne la nouvelle valeur
`getAndIncrement()`	Incrémente numériquement après avoir retourné l'ancienne valeur
`decrementAndGet()`	Décrémente numériquement et retourne la nouvelle valeur
`getAndDecrement()`	Décrémente numériquement après avoir retourné l'ancienne valeur

Bloc `synchronized` en Java

En Java, n'importe quel objet peut être utilisé comme un **monitor**, avec le mot-clé `synchronized` :

```
var manager = new SheepManager();
synchronized(manager) {
    // Travail à effectuer par un thread à la fois
}
```

Un bloc **synchronisé** garantit qu'un **seul thread exécute une section de code à la fois**. Si un **thread** essaie d'entrer dans un bloc synchronized alors qu'un autre thread y est déjà, il passe dans un état **bloqué** jusqu'à ce que le verrou soit **libéré**.

Pour synchroniser l'accès entre plusieurs threads, chaque thread doit avoir accès au même objet. Si chaque thread synchronise sur un objet différent, le code ne sera pas sécurisé pour les threads.

Exemple : Gestion des threads avec synchronized

Voici un exemple de gestion de threads où l'incrémentation et l'affichage des résultats sont effectués dans l'ordre :

```
for(int i = 0; i < 10; i++) {
    synchronized(manager) {
        service.submit(() -> manager.incrementAndReport());
    }
}
```

Cependant, cela ne règle pas le problème, car bien que les threads soient créés un par un, ils peuvent encore s'exécuter simultanément, ce qui ne garantit pas l'ordre des résultats.

Solution correcte : synchronisation de l'exécution des threads

```
public class SheepManager {
    private int sheepCount = 0;

    private void incrementAndReport() {
        synchronized(this) {
            System.out.print((++sheepCount) + " ");
        }
    }

    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(20);
        try {
            var manager = new SheepManager();
            for(int i = 0; i < 10; i++) {
                service.submit(() -> manager.incrementAndReport());
            }
        } finally {
            service.shutdown();
        }
    }
}
```

Résultat

Lors de l’exécution de ce code, l’affichage sera toujours dans l’ordre :

```
1 2 3 4 5 6 7 8 9 10
```

Les threads attendent chacun leur tour dans le bloc synchronisé, garantissant que chaque incrémentation et affichage se fait dans l’ordre.

Synchronisation sur une méthode

Il est également possible de synchroniser directement une méthode en utilisant le mot-clé synchronized :

```
void sing() {
   synchronized(this) {
      System.out.print("La la la!");
   }
}
```

Ou de manière plus concise :

```
synchronized void sing() {
    System.out.print("La la la!");
}
```

Synchronisation statique

Pour la synchronisation des méthodes statiques, le monitor utilisé sera l'objet de la classe elle-même, comme ceci :

```
static synchronized void sing() {
   System.out.print("La la la!");
}
```

Le framework de verrouillage (Lock)

L'interface Lock offre plus de fonctionnalités que le mot-clé synchronized, comme la possibilité de tester la disponibilité du verrou sans attendre indéfiniment. L'exemple suivant montre l'utilisation d'un verrou réentrant (ReentrantLock) :

```
Lock lock = new ReentrantLock();
try {
    lock.lock();
    // Code protégé
} finally {
    lock.unlock();
}
```

L’utilisation de **try/finally** garantit que le verrou sera toujours libéré, même si une exception se produit.

Méthodes de l’interface Lock

L'interface `Lock` inclut quatre méthodes importantes que vous devez connaître.

Méthode	Description
`void lock()`	Demande un verrou et bloque jusqu'à ce qu'il soit acquis.
`void unlock()`	Libère le verrou.
`boolean tryLock()`	Demande un verrou et retourne immédiatement un résultat booléen indiquant si le verrou a été acquis.
`boolean tryLock(long timeout, TimeUnit unit)`	Demande un verrou et attend pendant un délai spécifié ou jusqu'à ce qu'il soit acquis. Retourne un booléen indiquant si le verrou a été acquis.

Utilisation de `tryLock()`

La méthode `tryLock()` permet de tenter d'acquérir un verrou et retourne immédiatement un résultat booléen indiquant si le verrou a été acquis. Contrairement à la méthode `lock()`, elle ne bloque pas si un autre thread détient déjà le verrou, elle retourne immédiatement.

Voici un exemple de mise en œuvre avec `tryLock()` :

```
Lock lock = new ReentrantLock();
new Thread(() -> printHello(lock)).start();

if (lock.tryLock()) {
    try {
        System.out.println("Verrou acquis, exécution du code protégé");
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("Impossible d’acquérir le verrou, je fais autre chose");
}
```

Lorsque ce code est exécuté, il peut afficher soit le message de réussite (if), soit le message d'échec (else), **en fonction de l'ordre d'exécution des threads**. Dans tous les cas, **"Hello"** sera imprimé car l'appel à **lock()** dans printHello() **attendra** indéfiniment que le verrou devienne disponible.

Il est courant d'utiliser tryLock() dans une structure conditionnelle (if) afin de s'assurer que le verrou est bien acquis avant de libérer le verrou avec unlock().

tryLock(long, TimeUnit)

L’interface Lock inclut une version surchargée de tryLock(long, TimeUnit), qui fonctionne comme un mélange entre lock() et tryLock(). Si un verrou est disponible, cette méthode le retourne immédiatement. Si le verrou n’est pas disponible, elle attend pendant un délai spécifié avant de vérifier à nouveau si le verrou est disponible. Cette méthode retourne un booléen pour indiquer si le verrou a été acquis ou non.

Utilisation des Collections Concurrentes

L'API de Concurrence inclut des interfaces et des classes qui vous aident à coordonner l'accès aux collections partagées par plusieurs tâches. Ces collections font partie du *Java Collections Framework*.

Comprendre les erreurs de cohérence mémoire

Les classes de collections concurrentes visent à résoudre les erreurs courantes de cohérence mémoire. Une erreur de cohérence mémoire se produit lorsque deux threads ont des vues incohérentes des données censées être identiques. L'objectif est que les écritures effectuées par un thread soient disponibles pour un autre thread accédant à la collection après l'écriture.

Exemple de `ConcurrentModificationException`

Lorsqu'un thread tente de modifier une collection non concurrente, la JVM peut lancer une `ConcurrentModificationException`. Exemple avec `HashMap` :

```
var foodData = new HashMap<String, Integer>();
foodData.put("penguin", 1);
foodData.put("flamingo", 2);
for (String key : foodData.keySet())
    foodData.remove(key);
```

Ce code lève une ConcurrentModificationException lors de la deuxième itération du boucle, car l'itérateur sur keySet() n'est pas mis à jour correctement après la suppression du premier élément. En utilisant ConcurrentHashMap, cette exception est évitée :

```
var foodData = new ConcurrentHashMap<String, Integer>();
```

Travailler avec les classes concurrentes

Il est recommandé d'utiliser une collection concurrente chaque fois que plusieurs threads modifient une collection en dehors d'un bloc ou d'une méthode synchronisée, même si vous ne vous attendez pas à un problème de concurrence. Sans collections concurrentes, plusieurs threads accédant à une collection pourraient entraîner des exceptions ou pire, des données corrompues !

Si la collection est immuable (et contient des objets immuables), les collections concurrentes ne sont pas nécessaires. Les objets immuables peuvent être accédés par n'importe quel nombre de threads sans nécessité de synchronisation.

Passer une référence de collection concurrente

Lors du passage d'une collection concurrente, il est conseillé de passer une référence d'interface non concurrente, comme pour un HashMap où l'on passe souvent une référence Map :

```
Map<String, Integer> map = new ConcurrentHashMap<>();
```

Classes concurrentes courantes

Nom de la classe	Interfaces Java Collections	Trié ?	Blocage ?
ConcurrentHashMap	Map, ConcurrentMap	Non	Non
ConcurrentLinkedQueue	Queue	Non	Non
ConcurrentSkipListMap	Map, SortedMap, NavigableMap, ConcurrentMap, ConcurrentNavigableMap	Oui	Non
ConcurrentSkipListSet	Set, SortedSet, NavigableSet	Oui	Non

Nom de la classe	Interfaces Java Collections	Trié ?	Blocage ?
CopyOnWriteArrayList	List	Non	Non
CopyOnWriteArraySet	Set	Non	Non
LinkedBlockingQueue	Queue, BlockingQueue	Non	Oui

Comportement des classes CopyOnWrite

Les classes CopyOnWrite créent une copie de la collection chaque fois qu'une référence est ajoutée, supprimée ou modifiée, et mettent ensuite à jour la référence de la collection d'origine pour pointer vers cette copie. Elles sont couramment utilisées pour garantir qu'un itérateur ne voit pas les modifications de la collection.

```java
List<Integer> favNumbers = new CopyOnWriteArrayList<>(List.of(4, 3, 42));
for (var n : favNumbers) {
    System.out.print(n + " ");  // 4 3 42
    favNumbers.add(n + 1);
}
```

Le code affiche “4 3 42”, même si des éléments sont ajoutés pendant l’itération, car l’itérateur n’est pas modifié.

Classes CopyOnWrite et mémoire

Les classes CopyOnWrite peuvent consommer beaucoup de mémoire, car une nouvelle structure de collection est créée chaque fois que la collection est modifiée. Elles sont donc utiles dans des environnements multithread où les lectures sont beaucoup plus fréquentes que les écritures.

LinkedBlockingQueue

La classe LinkedBlockingQueue implémente l'interface concurrente BlockingQueue. Elle fonctionne comme une queue régulière, mais elle inclut des versions surchargées des méthodes offer() et poll() qui prennent un délai d'attente. Ces méthodes attendent (ou bloquent) jusqu'à un certain temps pour compléter une opération.

Problèmes de threading

- Un **problème de threading** survient lorsque deux threads ou plus interagissent de manière inattendue.
- Les **problèmes de threading** peuvent entraîner des blocages, des attentes infinies, ou de l'inaccessibilité.
- L'API de Concurrence aide à éviter ces problèmes mais ne les élimine pas entièrement.

Liveness et ses problèmes

- **Liveness** : capacité d'une application à continuer d'exécuter des tâches en temps voulu.
- Problèmes de liveness : lorsque des threads sont bloqués ou attendent indéfiniment, rendant l'application non réactive.

- Trois types de problèmes de liveness :
  1. **Deadlock** : deux threads bloqués, attendant l'un l'autre.
  2. **Starvation** : un thread ne peut jamais accéder aux ressources partagées.
  3. **Livelock** : threads bloqués, mais actifs, dans une boucle infinie de tentatives.

Exemple de Deadlock, Starvation et Livelock

- **Deadlock** : Deux threads se bloquent mutuellement en attendant des ressources.
  - Exemple : deux renards, Foxy et Tails, attendant chacun de l'autre la ressource pour finir leur repas.

- **Starvation** : Un thread est constamment privé d'accès à une ressource partagée.
  - Exemple : Foxy attend sans fin, car d'autres renards prennent toujours la nourriture avant elle.

- **Livelock** : Les threads restent actifs mais bloqués dans une boucle infinie de tentatives.
  - Exemple : Foxy et Tails alternent sans fin entre la nourriture et l'eau sans jamais finir leur repas.

Flux Parallèles (Parallel Streams)

Les flux parallèles permettent de traiter les éléments d'un flux de manière concurrente à l'aide de plusieurs threads. Contrairement aux flux sériels, où un seul élément est traité à la fois, les flux parallèles améliorent les performances en utilisant plusieurs threads simultanément.

2. Création de Flux Parallèles

Il existe deux manières principales de créer un flux parallèle en Java :

- À partir d'un flux existant :

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
Stream<Integer> parallelStream = list.stream().parallel();
```

- Directement depuis une collection :

```
List<Integer> list = List.of(1, 2, 3, 4, 5);
Stream<Integer> parallelStream = list.parallelStream();
```

Vérification de l'État Parallèle

La méthode isParallel() permet de tester si un flux est parallèle. Elle retourne true si le flux est parallèle, sinon false.

```
Stream<Integer> serialStream = List.of(1, 2, 3, 4).stream();
Stream<Integer> parallelStream = List.of(1, 2, 3, 4).parallelStream();

System.out.println(serialStream.isParallel()); // false
System.out.println(parallelStream.isParallel()); // true
```

Décomposition Parallèle

La décomposition parallèle consiste à diviser une tâche en morceaux plus petits pouvant être traités en parallèle. Cela peut améliorer les performances si la décomposition est bien optimisée.

```
private static int doWork(int input) {
    try { Thread.sleep(1000); } catch (InterruptedException e) {}
    return input * input; // Exemple simple : calcul du carré de l'input
}

List<Integer> results = List.of(1, 2, 3, 4, 5)
     .parallelStream()
     .map(w -> doWork(w))
     .collect(Collectors.toList());
System.out.println(results);
```

Exécution en Flux Série

Exemple d'exécution en flux série:

```
List.of(1, 2, 3, 4, 5)
    .stream()
    .map(w -> doWork(w))
    .forEach(s -> System.out.print(s + " "));
```

Cela prend environ 5 secondes pour chaque élément.

Exécution en Flux Parallèles

Lorsque vous remplacez .stream() par .parallelStream(), l'exécution se fait en parallèle. Le temps d'exécution est réduit, mais l'ordre des résultats peut être différent.

```
List.of(1, 2, 3, 4, 5)
    .parallelStream()
    .map(w -> doWork(w))
    .forEach(s -> System.out.print(s + " "));
```

Cela peut être exécuté plus rapidement, mais l'ordre des résultats n'est pas garanti.

Garantie d'Ordre avec forEachOrdered()

Si vous avez besoin de garantir l'ordre des éléments même avec un flux parallèle, utilisez forEachOrdered() :

```
List.of(1, 2, 3, 4, 5)
    .parallelStream()
    .map(w -> doWork(w))
    .forEachOrdered(s -> System.out.print(s + " "));
```

Cela maintient l'ordre d'origine mais perd certains avantages de performance.

Réductions Parallèles

Les opérations de réduction sur des flux parallèles peuvent donner des résultats inattendus si l'accumulateur n'est pas conçu correctement. Par exemple, une soustraction dans un accumulateur parallèle peut produire des résultats incohérents.

```
List.of(1, 2, 3, 4, 5, 6)
    .parallelStream()
    .reduce(0, (a, b) -> a - b); // Résultat inattendu, produit un résultat incorrect en parallèle
```

Combiner les Résultats avec reduce()

Le reduce() avec trois arguments (identité, accumulateur, combinateur) permet de combiner les résultats efficacement en parallèle.

```
Stream.of("w", "o", "l", "f")
    .parallel()
    .reduce("", String::concat); // Résultat : "wolf"
```

Cela permet de combiner les résultats même dans un environnement parallèle.

Utilisation de collect() en Parallèle

Le collect() en parallèle nécessite également l'utilisation d'un accumulateur et d'un combinateur adaptés pour éviter des exceptions de modification concurrente.

```
Stream<String> stream = Stream.of("w", "o", "l", "f").parallel();
SortedSet<String> set = stream.collect(ConcurrentSkipListSet::new, Set::add, Set::addAll);
System.out.println(set); // Résultat : [f, l, o, w]
```

Cela garantit que les éléments sont collectés correctement, même en parallèle, sans violer la sécurité concurrente.

Connexion à une Base de Données

Construction d'une URL JDBC

Pour se connecter à une base de données, il faut construire une URL JDBC. Elle comporte trois parties :

1. **Protocole** : Toujours `jdbc`.
2. **Sous-protocole** : Nom du SGBD (par ex., `mysql`, `postgresql`).
3. **Sous-nom** : Informations spécifiques comme l'adresse, le port et le nom de la base.

Exemples :

```
jdbc:hsqldb:file:zoo
jdbc:postgresql://localhost/zoo
jdbc:mysql://localhost:3306/zoo?useSSL=false
```

Obtenir une Connexion à une Base de Données

Deux principales méthodes :

- DriverManager : Utilisé pour les tests ou les applications simples.
- DataSource : Préféré en production pour ses fonctionnalités avancées (ex., pool de connexions).

Exemple avec DriverManager:

```
import java.sql.*;

public class TestConnect {
    public static void main(String[] args) throws SQLException {
        try (Connection conn = DriverManager.getConnection("jdbc:hsqldb:file:zoo")) {
            System.out.println(conn);
        }
    }
}
```

Exemple avec un utilisateur et un mot de passe:

```
import java.sql.*;

public class TestExternal {
    public static void main(String[] args) throws SQLException {
        try (Connection conn = DriverManager.getConnection(
                "jdbc:postgresql://localhost:5432/zoo",
                "username",
                "Password123")) {
            System.out.println(conn);
        }
    }
}
```

Note : Ne jamais coder les mots de passe directement dans le code ; utilisez des fichiers de configuration sécurisés.

Gestion des Exceptions et Fermeture des Ressources

Les connexions doivent être fermées pour libérer les ressources. Utilisez try-with-resources.

Exemple :

```
try (Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/zoo", "user", "password")) {
    System.out.println("Connexion réussie : " + conn);
} catch (SQLException e) {
    e.printStackTrace();
}
```

Résultat attendu:

```
Connexion réussie : com.mysql.jdbc.JDBC4Connection@1a2b3c4d
```

Vérification des Pilotes

Assurez-vous que le pilote JDBC est dans le classpath. Pour des versions anciennes de JDBC, utilisez Class.forName().

Exemple:

```
Class.forName("org.postgresql.Driver");
try (Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost/zoo", "user", "password")) {
    System.out.println("Connexion établie.");
}
```

Attention : Cette méthode n'est plus nécessaire avec les pilotes JDBC modernes.

Travailler avec `PreparedStatement`

- En Java, trois interfaces permettent d'exécuter des requêtes SQL :
  - **Statement**
  - **PreparedStatement**
  - **CallableStatement**
- `PreparedStatement` et `CallableStatement` étendent `Statement`.

Différences principales :

- **Statement** : Exécute directement une requête SQL sans paramètres.
- **PreparedStatement** : Accepte des paramètres pour améliorer performance et sécurité.
- **CallableStatement** : Utilisé pour les procédures stockées dans la base de données.

Avantages de `PreparedStatement`

- **Performance** : Réutilisation d'un plan optimisé pour exécuter la requête.
- **Sécurité** : Protection contre les attaques par injection SQL.
- **Lisibilité** : Simplifie les requêtes avec paramètres.
- **Flexibilité future** : Adapté pour les requêtes sans ou avec paramètres.

Obtenir un `PreparedStatement`

Exemple simple :

```
try (PreparedStatement ps = conn.prepareStatement(
    "SELECT * FROM exhibits")) {
    // Travail avec ps
}
```

Attention : Une requête SQL est obligatoire lors de la création.

```
try (var ps = conn.prepareStatement()) {
    // NE COMPILE PAS
}
```

Exécuter un PreparedStatement

Modifier des données avec executeUpdate()

- Utilisé pour les requêtes DELETE, INSERT ou UPDATE.
- Retourne le nombre de lignes affectées.

Exemple:

```
var insertSql = "INSERT INTO exhibits VALUES(10, 'Deer', 3)";
try (var ps = conn.prepareStatement(insertSql)) {
    int result = ps.executeUpdate();
    System.out.println(result); // Nombre de lignes affectées
}
```

Lire des données avec executeQuery()

- Utilisé pour les requêtes SELECT.
- Retourne un ResultSet contenant les résultats.

Exemple:

```
var sql = "SELECT * FROM exhibits";
try (var ps = conn.prepareStatement(sql);
     ResultSet rs = ps.executeQuery()) {
    // Traiter les résultats
}
```

Méthode générique : execute()

- Exécute une requête et retourne un booléen.
- Si vrai : Requête SELECT → Utiliser getResultSet().
- Si faux : Requête UPDATE/INSERT/DELETE → Utiliser getUpdateCount().

Exemple:

```
boolean isResultSet = ps.execute();
if (isResultSet) {
    try (ResultSet rs = ps.getResultSet()) {
        System.out.println("Requête SELECT exécutée");
    }
} else {
    int result = ps.getUpdateCount();
    System.out.println("Mise à jour exécutée : " + result);
}
```

Travailler avec des paramètres

- Les paramètres sont définis avec ? dans la requête SQL.
- Utilisation des méthodes setType() pour lier les valeurs.

```
String sql = "INSERT INTO names VALUES(?, ?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 6);           // Premier paramètre
    ps.setInt(2, 1);           // Deuxième paramètre
    ps.setString(3, "Edith"); // Troisième paramètre
    ps.executeUpdate();
}
```

Récapitulatif des méthodes

Méthode	DELETE	INSERT	SELECT	UPDATE
`ps.execute()`	Oui	Oui	Oui	Oui
`ps.executeQuery()`	Non	Non	Oui	Non
`ps.executeUpdate()`	Oui	Oui	Non	Oui

Méthode	Retour pour SELECT	Retour pour DELETE/INSERT/UPDATE
`ps.execute()`	`true` (ResultSet)	`false`
`ps.executeQuery()`	`ResultSet`	n/a
`ps.executeUpdate()`	n/a	Nombre de lignes affectées

Updating Multiple Records

Ajouter plusieurs enregistrements avec `PreparedStatement`

```
var sql = "INSERT INTO names VALUES(?, ?, ?)";
try (var ps = conn.prepareStatement(sql)) {
    ps.setInt(1, 20);
    ps.setInt(2, 1);
    ps.setString(3, "Ester");
    ps.executeUpdate();

    ps.setInt(1, 21);
    ps.setString(3, "Elias");
    ps.executeUpdate();
}
```

- Le PreparedStatement conserve les valeurs des paramètres déjà définis.
- Seuls les paramètres modifiés doivent être réinitialisés avant chaque exécution.

Batching Statements

Exécuter plusieurs requêtes en un seul appel réseau

Avantages :

- Réduction des appels réseau, donc gain de performance.
- Utile pour insérer de nombreuses lignes en minimisant le temps d’exécution.

```
public static void register(Connection conn, int firstKey,
   int type, String... names) throws SQLException {
   var sql = "INSERT INTO names VALUES(?, ?, ?)";
   var nextIndex = firstKey;

   try (var ps = conn.prepareStatement(sql)) {
      ps.setInt(2, type);

      for (var name : names) {
         ps.setInt(1, nextIndex);
         ps.setString(3, name);
         ps.addBatch();
         nextIndex++;
      }
      int[] result = ps.executeBatch();
      System.out.println(Arrays.toString(result));
   }
}
```

Appel de la méthode:

```
register(conn, 100, 1, "Elias", "Ester");
```

- Sortie : [1, 1] (chaque élément correspond à une ligne insérée).
- Idéal pour des opérations massives en définissant une taille de lot appropriée.

Récupérer des données d'un ResultSet

```
String sql = "SELECT id, name FROM exhibits";
var idToNameMap = new HashMap<Integer, String>();

try (var ps = conn.prepareStatement(sql);
     ResultSet rs = ps.executeQuery()) {
   while (rs.next()) {
      int id = rs.getInt("id");
      String name = rs.getString("name");
      idToNameMap.put(id, name);
   }
   System.out.println(idToNameMap);
}
```

Comportement du curseur :

- Commence avant la première ligne.
- Se déplace vers la ligne suivante avec rs.next().
- Retourne false lorsqu'il n'y a plus de lignes.

Exemple de sortie :
{1=African Elephant, 2=Zebra}

Accès aux colonnes dans un ResultSet

Accéder aux colonnes en utilisant :
- Le nom : rs.getInt("id");
- L’index (commençant à 1) : rs.getInt(1);

Exemple:

```
var sql = "SELECT count(*) FROM exhibits";
try (var ps = conn.prepareStatement(sql);
     var rs = ps.executeQuery()) {
   if (rs.next()) {
      int count = rs.getInt(1);
      System.out.println(count);
   }
}
```

** Règles importantes :**

- Toujours appeler rs.next() avant d’accéder aux données.
- Utiliser un nom ou un index de colonne invalide génère une SQLException.

Méthode	Type de retour
`getBoolean`	`boolean`
`getDouble`	`double`
`getInt`	`int`
`getLong`	`long`
`getString`	`String`
`getObject`	`Object`

Exemple:

```
var sql = "SELECT id, name FROM exhibits";
try (var ps = conn.prepareStatement(sql);
     var rs = ps.executeQuery()) {
   while (rs.next()) {
      Object idField = rs.getObject("id");
      Object nameField = rs.getObject("name");
      if (idField instanceof Integer id) System.out.println(id);
      if (nameField instanceof String name) System.out.println(name);
   }
}
```

getObject() retourne dynamiquement le type correspondant le plus adapté.

Introduction aux procédures stockées

Les procédures stockées sont des morceaux de code compilés et stockés dans la base de données, souvent utilisés pour des requêtes SQL complexes. Elles réduisent les aller-retour réseau et permettent aux experts en bases de données de gérer cette partie du code. Cependant, elles sont spécifiques aux bases de données et peuvent compliquer la maintenance de l'application.

Syntaxe pour appeler une procédure

Pour appeler une procédure stockée, la syntaxe suivante est utilisée :

```
String sql = "{call procedure_name()}";
```

Cette syntaxe utilise des accolades {} pour entourer l'appel à la procédure.

Exemple: appeler une procédure stockée:

```
String sql = "{call read_e_names()}";
try (CallableStatement cs = conn.prepareCall(sql);
     ResultSet rs = cs.executeQuery()) {
    while (rs.next()) {
        System.out.println(rs.getString(3));
    }
}
```

Dans cet exemple, la procédure read_e_names() ne prend aucun paramètre et retourne un ResultSet.

Passer un paramètre IN

Pour appeler une procédure avec un paramètre d'entrée (IN), la syntaxe suivante est utilisée :

```
var sql = "{call read_names_by_letter(?)}";
try (var cs = conn.prepareCall(sql)) {
    cs.setString("prefix", "Z");
    try (var rs = cs.executeQuery()) {
        while (rs.next()) {
            System.out.println(rs.getString(3));
        }
    }
}
```

Ici, le ? est utilisé pour indiquer un paramètre, et la méthode setString

Appeler une procédure avec un paramètre OUT

Une procédure peut aussi retourner un paramètre de sortie (OUT).
Exemple :

```
var sql = "{?= call magic_number(?) }";
try (var cs = conn.prepareCall(sql)) {
    cs.registerOutParameter(1, Types.INTEGER);
    cs.execute();
    System.out.println(cs.getInt("num"));
}
```

L'utilisation de registerOutParameter() est essentielle pour enregistrer le paramètre de sortie.

Travailler avec un paramètre INOUT

Un paramètre INOUT peut être utilisé à la fois comme entrée et sortie.

Exemple :

```
var sql = "{call double_number(?)}";
try (var cs = conn.prepareCall(sql)) {
    cs.setInt(1, 8);  // IN
    cs.registerOutParameter(1, Types.INTEGER);  // OUT
    cs.execute();
    System.out.println(cs.getInt("num"));
}
```

Ici, le même paramètre est utilisé pour l'entrée et la sortie.

Types de paramètres d'une procédure stockée

Table 15.8 récapitule les différents types de paramètres pour une procédure stockée.

Type de Paramètre	IN	OUT	INOUT
Utilisé pour l'entrée	Oui	Non	Oui
Utilisé pour la sortie	Non	Oui	Oui
Doit définir la valeur	Oui	Non	Oui
Doit appeler `registerOutParameter()`	Non	Oui	Oui
Peut inclure `?=`	Non	Oui	Oui

Options supplémentaires pour CallableStatement

Lors de la création d'un `PreparedStatement` ou d'un `CallableStatement` , vous pouvez spécifier des options supplémentaires. Voici les types et modes de concurrence de `ResultSet` :

Types de ResultSet :

- `ResultSet.TYPE_FORWARD_ONLY` : Parcours du `ResultSet` uniquement ligne par ligne.
- `ResultSet.TYPE_SCROLL_INSENSITIVE` : Parcours du `ResultSet` dans n'importe quel ordre sans voir les changements dans la base.
- `ResultSet.TYPE_SCROLL_SENSITIVE` : Parcours du `ResultSet` dans n'importe quel ordre avec les changements reflétés dans la base.

Modes de Concurrence de ResultSet :

- `ResultSet.CONCUR_READ_ONLY` : Le `ResultSet` ne peut pas être mis à jour.
- `ResultSet.CONCUR_UPDATABLE` : Le `ResultSet` peut être mis à jour.

Utilisation des options dans CallableStatement

Ces options sont des valeurs entières et doivent être passées comme paramètres supplémentaires après le SQL :

```
conn.prepareCall(sql, ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Contrôler les données avec des transactions

Commit et Rollback

- **Commit** : Enregistre les modifications apportées à la base de données.
- **Rollback** : Annule les modifications effectuées depuis le début de la transaction.
- Exemple :
  - **Autocommit désactivé** : `conn.setAutoCommit(false);`
  - **Rollback en cas d'échec** : `conn.rollback();`
  - **Commit si valide** : `conn.commit();`

Cas particuliers avec l'Autocommit

- **setAutoCommit(true)** : Valide automatiquement les changements après chaque instruction.
- **Fermeture de la connexion sans Commit/Rollback** : Comportement indéfini ; les modifications peuvent ou non être validées. Il est important de toujours effectuer un commit ou un rollback à la fin d'une transaction.

Marquer avec des Savepoints

- **Savepoint** : Marque un point dans une transaction auquel vous pouvez revenir.
- Exemple :
  - `Savepoint sp1 = conn.setSavepoint();`
  - `conn.rollback(sp1);`
  - Rollback vers un savepoint spécifique, pas toute la transaction.

Méthodes des APIs de transaction

Méthode	Description
`setAutoCommit(boolean)`	Définit le mode auto-commit
`commit()`	Valide les modifications
`rollback()`	Annule toutes les modifications
`rollback(Savepoint)`	Annule jusqu'à un savepoint spécifique
`setSavepoint()`	Crée un savepoint
`setSavepoint(String)`	Crée un savepoint nommé

Fermeture des ressources de base de données

- **Ordre de fermeture** : Fermez toujours d'abord le `ResultSet`, puis le `PreparedStatement`, et enfin la `Connection`.
- **Fermeture automatique** : La fermeture d'une `Connection` ferme automatiquement les ressources associées (`PreparedStatement`, `ResultSet`).

Écriture d'une fuite de ressources

- Incorrect : Déclarer des ressources avant `try-with-resources` provoque une fuite de ressources si le bloc `try` n'est pas exécuté.
  Mauvais exemple :

```
try (conn; ps; rs) { ... }
```

Éviter les Fuites de Ressources

- **Utilisation correcte** : Déclarez les ressources à l'intérieur du bloc `try-with-resources` pour garantir leur fermeture correcte, même en cas d'exception.
- **Exemple correct** :

```
try (Connection conn = DriverManager.getConnection(url);
     PreparedStatement ps = conn.prepareStatement(query);
     ResultSet rs = ps.executeQuery()) {
     // Traitement des données
}
```

Gestion des Exceptions SQL

- SQLException : Cette exception est lancée lorsqu'il y a une erreur pendant l'interaction avec la base de données.

Exemple :

```
try {
    // Opération sur la base de données
} catch (SQLException e) {
    System.out.println("Erreur de base de données : " + e.getMessage());
}
```

Exceptions enchaînées : La SQLException peut contenir des exceptions supplémentaires, accessibles via getNextException().

Gestion des Connexions et des Transactions

- Pool de connexions : Pour une meilleure efficacité, utilisez un pool de connexions afin de réutiliser les connexions à la base de données.
- Gestion des transactions : Utilisez les transactions pour regrouper plusieurs opérations en une seule unité de travail, garantissant l'atomicité.

Exemple de pool de connexions:

```
DataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:yourdatabase");
dataSource.setUsername("username");
dataSource.setPassword("password");
Connection conn = dataSource.getConnection();
```

Garantir l’Isolation des Transactions

- Niveaux d’isolation des transactions : Contrôlez la visibilité des modifications de données effectuées par une transaction vis-à-vis des autres transactions.
  - READ_COMMITTED : Permet de lire les données validées.
  - REPEATABLE_READ : Garantit que les données lues par une transaction ne peuvent pas être modifiées par une autre transaction avant la fin de celle-ci.
  - SERIALIZABLE : Garantit que les transactions sont exécutées les unes après les autres, offrant le plus haut niveau d’isolation.

Exemple :

```
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

Utilisation des Prepared Statements

PreparedStatement : Utilisez les PreparedStatement pour exécuter des requêtes paramétrées, ce qui aide à prévenir les attaques par injection SQL.

Exemple :

```
String query = "SELECT * FROM users WHERE username = ?";
try (PreparedStatement ps = conn.prepareStatement(query)) {
    ps.setString(1, "username123");
    ResultSet rs = ps.executeQuery();
}
```

Travailler avec les Transactions dans un Environnement Multi-Thread

- Sécurité des threads : Les connexions à la base de données ne sont pas thread-safe par défaut. Assurez-vous que chaque thread ait sa propre connexion.
- Pool de connexions : Un pool de connexions permet de gérer l'accès concurrent à la base de données en fournissant à chaque thread une connexion séparée.

Opérations Courantes sur les Bases de Données dans les Transactions

- Insertion : Ajouter de nouvelles données dans la base de données.
- Mise à jour : Modifier des données existantes.
- Suppression : Supprimer des données de la base de données.
- Sélection : Interroger des données depuis la base de données.

Bonnes Pratiques

- Gérez toujours les exceptions pour garantir l’intégrité de la base de données.
- Utilisez des transactions pour maintenir l’atomicité et la cohérence.
- Libérez les ressources de la base de données pour éviter les fuites de mémoire.
- Utilisez le pool de connexions pour améliorer les performances et la scalabilité.