

# Java

## Introduction

La syntaxe de Java sert de fondement à la compréhension de la programmation dans ce langage.

En maîtrisant la syntaxe de base, vous pourrez écrire, lire et déboguer du code Java efficacement.

Ce module introduit des éléments syntaxiques clés qui facilitent le développement de programmes structurés et orientés objet.

## Objectifs pédagogiques

- Identifier les structures syntaxiques de base en Java
- Écrire du code Java simple et structuré
- Comprendre l'organisation d'un programme Java
- Appliquer les conventions de nommage standard

## Bases de la syntaxe Java

La syntaxe Java repose sur quelques principes fondamentaux :

- Chaque programme commence par une classe avec une méthode `main`.
- Les instructions se terminent par un point-virgule `;`.
- Les blocs de code sont délimités par des accolades `{}`.

Un programme simple suit généralement cette structure.

Voici l'exemple d'un programme Java basique.

## Exemple

Premier programme en Java :

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

- `public class HelloWorld` : Définit une classe publique nommée `HelloWorld`.
- `public static void main(String[] args)` : Point d'entrée du programme.
- `System.out.println()` : Invoque une méthode de sortie pour afficher du texte.

# Variables et Types de Données

Java est un langage fortement typé.

Les variables doivent être déclarées avant utilisation.

- Types primitifs : `int` , `double` , `char` , `boolean`
- Types de référence : `String` , `Array` , `Object`

Déclaration d'une variable :

```
int age = 25;
```

Ce code crée une variable entière `age` initialisée à 25.

# Structures de Contrôle

Java utilise des structures de contrôle pour orienter le flux d'exécution :

- Conditionnelles : `if` , `else` , `switch`
- Boucles : `for` , `while` , `do-while`

Exemple d'une structure conditionnelle :

```
if (age >= 18) {  
    System.out.println("Adult");  
} else {  
    System.out.println("Minor");  
}
```

Le bloc `if` exécute une instruction basée sur la condition.

# Exemples de code

Manipuler les structures de contrôle basiques :

Exemple de boucle `for` :

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

Exemple de `switch` :

```
int day = 3;  
switch (day) {  
    case 1:  
        System.out.println("Monday");  
        break;  
    case 2:  
        System.out.println("Tuesday");  
        break;  
    default:  
        System.out.println("Other day");  
}
```

# Exercices Pratiques

- Déclarer et utiliser des variables :** Écrire un programme qui déclare trois variables de types différents et manipule leurs valeurs.
- Implémenter des structures conditionnelles :** Écrire un programme qui utilise une instruction `if-else` pour vérifier si un nombre est pair ou impair.
- Boucles et itérations :** Créer un programme qui génère et imprime les 10 premiers nombres de Fibonacci.

**Critères d'évaluation :** Code fonctionnel et lisible, utilisation correcte des structures syntaxiques.

## Synthèse et Ouverture

La maîtrise de la syntaxe Java est cruciale pour progresser dans la programmation orientée objet.

Ces bases vous permettent de structurer efficacement vos programmes.

En avançant, nous explorerons des concepts plus avancés tels que les classes et les objets, améliorant ainsi la modularité et la réutilisabilité du code.

## Introduction

Dans cette section, nous allons explorer le concept de "Classes" en Java, qui est un élément fondamental de la programmation orientée objet.

Les classes constituent le modèle à partir duquel les objets sont créés.

Elles définissent les propriétés et les comportements d'un objet.

## Objectifs pédagogiques

- Comprendre le rôle des classes dans la programmation Java
- Savoir créer et utiliser des classes en Java
- Apprendre à définir des attributs et méthodes au sein d'une classe
- Maîtriser les concepts de constructeur et d'instanciation

## Les classes en Java

Les classes en Java constituent la pierre angulaire de la programmation orientée objet.

Elles permettent de créer des types de données personnalisés qui encapsulent des données et des méthodes.

- **Définition :** Une classe est un modèle définissant les propriétés (attributs) et comportements (méthodes) d'un objet.
- **Structure :** Une classe se compose généralement de variables d'instance, de méthodes et d'un ou plusieurs constructeurs.

Un exemple simple de classe :

```
public class Voiture {  
    String couleur;  
    String marque;  
  
    public void démarrer() {  
        System.out.println("La voiture démarre.");  
    }  
}
```

## Attributs d'une classe

Les attributs, ou variables d'instance, dans une classe définissent l'état des objets créés à partir de cette classe.

- **Déclaration** : Les attributs sont déclarés à l'intérieur de la classe, mais en dehors de toute méthode.
- **Types** : Les attributs peuvent être de n'importe quel type de données supporté par Java (int, String, boolean, etc.).

Exemple d'attributs pour la classe Voiture :

```
String couleur;  
String marque;
```

## Méthodes d'une classe

Les méthodes définissent le comportement d'une classe.

Elles permettent d'effectuer des actions en utilisant les attributs de la classe ou de fournir un service à d'autres classes.

- **Syntaxe** : Les méthodes ont une syntaxe similaire aux fonctions, comprenant un type de retour, un nom, et des paramètres optionnels.
- **Visibilité** : Une méthode peut être publique, privée, ou protégée, affectant son accessibilité depuis d'autres classes.

Exemple de méthode pour la classe Voiture :

```
public void démarrer() {  
    System.out.println("La voiture démarre.");  
}
```

## Constructeurs en Java

Les constructeurs sont des méthodes spéciales utilisées pour initialiser les objets.

Ils portent le même nom que la classe et n'ont pas de type de retour.

- **Par défaut** : Java fournit un constructeur par défaut si aucun n'est explicitement défini.
- **Personnalisé** : Un constructeur peut être personnalisé pour initialiser des attributs avec des valeurs spécifiques.

Exemple de constructeur :

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

# Exemple de classe complète

Voyons un exemple de classe complète et fonctionnelle, avec attributs, méthodes et constructeur :

```
public class Voiture {  
    String couleur;  
    String marque;  
  
    // Constructeur  
    public Voiture(String couleur, String marque) {  
        this.couleur = couleur;  
        this.marque = marque;  
    }  
  
    // Méthode pour démarrer la voiture  
    public void démarrer() {  
        System.out.println("La voiture de couleur " + couleur + " démarre.");  
    }  
}
```

## Exercices pratiques

- Créer une classe** : Créez une classe `Animal` avec des attributs `nom` et `age`, une méthode `manger()`, et un constructeur pour initialiser les attributs.
- Instancier des objets** : Utilisez la classe `Animal` pour créer plusieurs objets avec différents noms et âges.

Appelez les méthodes pour démontrer leur fonctionnement.

- Étendre la classe** : Ajoutez une méthode `dormir()` à la classe `Animal` et testez son intégration.

Évaluation : Vérifiez si les objets sont correctement créés et si les méthodes produisent l'output attendu.

## Synthèse

Les classes en Java sont essentielles pour structurer et organiser le code dans un style orienté objet.

Elles favorisent la réutilisation et l'extensibilité du code via l'encapsulation de données et de comportements spécifiques.

En maîtrisant les classes, vous acquérez une compétence clé pour développer des applications robustes et bien organisées en Java.

**Dans les prochaines leçons, nous explorerons comment les classes interagissent avec d'autres concepts tels que l'héritage et le polymorphisme.**

## Objets

### Introduction

Dans le paradigme de programmation orientée objet (POO), les objets sont au cœur de la modularité et de la réutilisabilité du code.

En Java, un objet est une instance d'une classe et encapsule des états et comportements.

Cette section explore en détail comment manipuler et utiliser des objets pour structurer efficacement vos programmes.

## Objectifs pédagogiques

- Décrire le concept d'objet en POO.
- Créer et manipuler des objets en Java.
- Utiliser des méthodes d'instance pour interagir avec les objets.
- Comprendre la différence entre une classe et un objet.

## Les objets en Java

Les objets en Java sont créés à partir de classes, qui définissent les propriétés (attributs) et actions possibles (méthodes) de ces objets.

Un objet représente une instance concrète de ces définitions, permettant l'exécution de comportement spécifique à cette instance.

### 1. Création d'objets : Utilisation du mot-clé `new`.

- Associez une classe à une variable pour créer un objet.
- Exemple : `MonObjet obj = new MonObjet();`

### 2. Attributs et méthodes :

- Les attributs définissent l'état de l'objet.
- Les méthodes définissent les comportements.

## Exemple de classe et d'objet

Considérons une classe `Voiture` :

```
public class Voiture {  
    String marque;  
    int vitesse;  
  
    public Voiture(String marque) {  
        this.marque = marque;  
        this.vitesse = 0;  
    }  
  
    public void accelerer(int increment) {  
        this.vitesse += increment;  
    }  
}
```

Créons un objet de cette classe :

```
Voiture maVoiture = new Voiture("Toyota");  
maVoiture.accelerer(50);
```

- L'objet `maVoiture` est une instance de `Voiture` avec un état unique.
- La méthode `accelerer` modifie cet état.

## Exercices pratiques

### 1. Création et manipulation :

- Créez une classe `Animal` avec des attributs `nom` et `age`.

- Instanciez un objet de cette classe et imprimez ses attributs.

## 2. Interagir avec des méthodes :

- Ajoutez une méthode `vieillir` à la classe `Animal` qui incrémente l'`age`.
- Testez cette méthode sur votre instance.

### Critères d'évaluation :

- Vérification de la création correcte des objets,
- Utilisation appropriée des méthodes pour modifier l'état des objets.

## Synthèse

Les objets sont essentiels pour structurer la logique en programmation Java.

En encapsulant l'état et le comportement, ils simplifient la gestion et la manipulation des données.

Maîtriser leur création et utilisation est crucial pour développer des applications Java robustes.

La prochaine étape pourrait inclure l'exploration des patrons de conception pour améliorer encore davantage la réutilisation du code et l'organisation des objets.

## Introduction

Les interfaces en Java sont des ensembles de méthodes abstraites qui définissent un comportement que les classes peuvent implémenter.

Elles jouent un rôle clé dans l'abstraction et la programmation orientée objet en aidant à définir des contrats sans implémentation détaillée, promouvant ainsi la flexibilité et la réutilisabilité du code.

## Objectifs pédagogiques

- Comprendre le concept et l'usage des interfaces en Java
- Distinguer entre interfaces et classes abstraites
- Implémenter des interfaces dans les classes
- Concevoir des architectures modulaires utilisant des interfaces

## Qu'est-ce qu'une Interface ?

Une interface en Java est un type de référence similaire à une classe, mais qui ne peut contenir que des variables finales (constantes) et des méthodes abstraites.

Depuis Java 8, les interfaces peuvent également contenir des méthodes par défaut et statiques, qui fournissent une implémentation que les classes peuvent utiliser.

## Rôle des Interfaces

Les interfaces permettent d'atteindre l'abstraction et la séparation des contrats d'implémentation.

Elles définissent un ensemble de méthodes qu'une classe doit implémenter, ce qui permet à différentes classes d'implémenter la même interface tout en fournissant leurs propres implémentations des méthodes.

## Syntaxe d'une Interface

Une interface est déclarée avec le mot clé `interface` :

```
public interface Animal {  
    void makeSound();  
    void move();  
}
```

Dans cet exemple, toute classe implémentant `Animal` doit fournir des implémentations concrètes pour `makeSound` et `move`.

## Implémentation d'une Interface

Pour qu'une classe implémente une interface, elle utilise le mot clé `implements` :

```
public class Dog implements Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof");  
    }  
  
    @Override  
    public void move() {  
        System.out.println("Runs");  
    }  
}
```

La classe `Dog` implémente l'interface `Animal` et fournit des implémentations concrètes des deux méthodes.

## Interfaces vs Classes Abstraites

Contrairement aux interfaces, les classes abstraites peuvent avoir des méthodes avec implémentation et des variables d'instance.

Cependant, elles ne permettent pas la "multiple inheritance" qu'une interface peut simuler en permettant la classe à implémenter plusieurs interfaces.

## Exemples d'utilisation

Les interfaces sont souvent utilisées dans les API pour fournir une flexibilité maximale.

Par exemple, `List`, `Map`, et `Set` dans le package `java.util` sont des interfaces, laissant aux classes comme `ArrayList`, `HashMap`, et `HashSet` l'implémentation des détails internes.

# Exemples de Code Annotés

Imaginez que vous concevez une application pour un zoo :

```
interface Flyer {  
    void fly();  
}  
  
class Bird implements Flyer {  
    @Override  
    public void fly() {  
        System.out.println("Flies in the sky");  
    }  
}  
  
class Plane implements Flyer {  
    @Override  
    public void fly() {  
        System.out.println("Flies using engines");  
    }  
}
```

Ici, Bird et Plane implémentent tous deux l'interface Flyer , illustrant l'abstraction du vol.

## Exercices Pratiques

**1. Création d'Interfaces :** Créez une interface Vehicle avec des méthodes startEngine et stopEngine .

Implémentez cette interface dans deux classes : Car et Boat .

**2. Interface Avancée :** Ajoutez une méthode par défaut à Vehicle qui imprime Checking vehicle status .

Implémentez-la dans vos classes précédemment créées.

**Critères d'évaluation :** Votre solution doit démontrer une compréhension de l'abstraction, la réutilisation de code, et l'utilisation correcte des interfaces et méthodes par défaut.

## Synthèse et Ouverture

Les interfaces jouent un rôle majeur dans la conception de Java, fournissant une manière de garantir qu'un ensemble de classes implémente certaines méthodes.

**Cela permet de concevoir des systèmes modulaires et extensibles. À l'avenir, explorez comment les interfaces fonctionnent en conjonction avec les lambdas et les streams en Java 8 pour tirer parti davantage des interfaces fonctionnelles.**

## Introduction

Dans cette section dédiée au langage Java, nous allons revoir les fondations essentielles pour bien comprendre la programmation orientée objet.

Les sous-notions abordées incluront la syntaxe de base, la structure de classes et d'objets, ainsi que les concepts clés comme les interfaces, l'héritage et le polymorphisme.

Nous explorerons également des principes de programmation importants tels que les principes SOLID, DRY et KISS pour écrire un code propre et maintenable.

## Vue d'ensemble des sous-notions

Voici comment les différentes sous-notions s'articulent :

- **Syntaxe, Classes, Objets** : Les bases de la programmation orientée objet.
- **Interfaces, Héritage, Polymorphisme** : Constructions avancées pour la réutilisation et l'extensibilité du code.
- **Encapsulation, Surcharge et redéfinition, Portée des variables** : Contrôle de l'accès et des opérations sur les données.
- **Principes SOLID, DRY, KISS** : Bonnes pratiques pour la qualité du code.
- **Exercice pratique** pour mettre en application les principes SOLID.

## Fondations et rappels

### Introduction aux Fondations

Dans ce module, nous explorerons les fondations essentielles du développement en Java.

Nous aborderons les bases du langage avec ses structures et principes clés.

Nous examinerons également les APIs standards qui sont indispensables au développement Java moderne.

Enfin, nous découvrirons les principes de la programmation fonctionnelle introduits avec Java 8.

### Vue d'ensemble des sous-notions

1. **Rappel des bases du langage** : Comprendre comment les concepts fondamentaux de Java comme les classes, objets et interfaces s'articulent.
2. **APIs standards** : Étudier les collections et les APIs I/O pour une gestion efficace des données.
3. **Programmation fonctionnelle (Java 8)** : Explorer les lambdas, Streams et autres innovations apportées par Java 8.

## Modularité avec Java 9

### Introduction

Avec Java 9, le projet Jigsaw a introduit un système de modularité révolutionnaire.

Ce système permet d'organiser le code Java en modules distincts, améliorant la gestion et l'évolutivité des applications.

La modularité offre des avantages comme une encapsulation stricte et une meilleure gestion des dépendances.

# Objectifs pédagogiques

- Comprendre les concepts de base de la modularité en Java 9.
- Apprendre à créer et gérer des modules Java.
- Savoir identifier et résoudre les problèmes courants de dépendance.
- Maîtriser les outils et fonctionnalités fournis par Java pour la modularité.

## Concept de Modularité

La modularité permet de diviser une application en unités distinctes nommées *modules*.

Chaque module contient des *packages* et des *ressources*, et définit de manière explicite quels autres modules il utilise et quels packages il exporte.

- **Module** : Conteneur logique de packages.
- **Module Descriptor (`module-info.java`)** : Fichier de déclaration des modules.
- **Exports** : Déclaration des packages accessibles aux autres modules.
- **Requires** : Spécifie les modules dont le module actuel dépend.

## Création d'un Module

Pour créer un module, il faut un fichier `module-info.java` à la racine du répertoire du module.

```
module com.example.myapp {  
    requires java.base;  
    exports com.example.myapp.api;  
}
```

- `module com.example.myapp` : Définit le nom unique du module.
- `requires java.base` : Import implicite du module de base Java.
- `exports com.example.myapp.api` : Rend le package accessible aux autres modules.

## Avantages de la Modularité

- **Encapsulation stricte** : Seuls les packages explicitement exportés sont accessibles.
- **Clarté des dépendances** : Les dépendances entre modules sont clairement spécifiées.
- **Chargement dynamique** : Permet à l'application de ne charger que les modules nécessaires à l'exécution.

Ces avantages permettent de construire des applications plus robustes et maintenables, avec un démarrage plus rapide, car seuls les modules nécessaires sont chargés en mémoire.

## Exemple 1 : Module Basique

Voyons un exemple simple où un module `com.example.app` utilise un autre module `com.example.utils`.

- **com.example.app/module-info.java**

```
module com.example.app {  
    requires com.example.utils;  
}
```

- **com.example.utils/module-info.java**

```
module com.example.utils {  
    exports com.example.utils;  
}
```

Ce setup montre comment un module client déclare ses dépendances et comment un module fournisseur expose ses fonctionnalités.

## Exemple 2 : Résolution de Conflits

Supposons que `com.example.utils` et `com.example.logging` exposent chacun un utilitaire nommé `Logger`, menant à un conflit.

- Utilisez `requires` avec `as` pour résoudre le conflit :

```
module com.example.app {  
    requires com.example.utils;  
    requires com.example.logging as log;  
}
```

En spécifiant `as`, il est possible de différencier et d'utiliser les deux modules en conflit.

## Exercices Pratiques

1. **Créer un module simple** : Mettez en place un module `com.example.library` qui exporte une classe `Book`.
  - Vérifiez qu'une classe `BookApp` dans `com.example.app` puisse utiliser `Book`.
2. **Gestion des dépendances** : Ajoutez un module `com.example.network` dépendant de `com.example.library`.
  - Modifiez le module de la bibliothèque pour qu'il nécessite le module `java.net.http`.

Critères de succès : Les modules doivent compiler et s'exécuter correctement, en assurant une séparation des packages et une gestion des dépendances nette.

## Synthèse

Le système de modularité introduit avec Java 9 permet une organisation claire et optimisée du code Java.

En utilisant les modules efficacement, nous bénéficions d'une meilleure encapsulation et gestion des dépendances.

La maîtrise de ces concepts ouvre la voie à la création d'applications plus robustes et évolutives, préparant le terrain pour les versions futures de Java.

## Introduction

L'inférence de type local introduite avec Java 10 permet de simplifier le code en remplaçant explicitement le type d'une variable par le mot-clé `var`.

Cela favorise la lisibilité et la concision du code, tout en améliorant la capacité du développeur à se concentrer sur la logique plutôt que sur la syntaxe de type.

## Objectifs pédagogiques

- Comprendre l'usage de l'inférence de type local avec `var`.
- Appliquer `var` pour simplifier le code Java existant.

- Identifier les situations adaptées et inadaptées pour l'utilisation de `var`.
- Évaluer les impacts de l'inférence de type sur la lisibilité et la maintenance du code Java.

## Qu'est-ce que l'inférence de type ?

L'inférence de type en Java 10 introduit `var` pour permettre au compilateur de déterminer le type de la variable.

Par conséquent, vous n'avez pas besoin de déclarer explicitement le type, ce qui simplifie le code.

Bien que `var` ne soit utilisable que dans des contextes locaux, il conserve la sécurité de type fournie par le compilateur.

## Cas d'utilisation appropriés

`var` est idéal lorsque le type est évident à partir du contexte, par exemple :

- Lorsque le type est dupliqué dans l'initialisation (`Map<String, List<Integer>> map = new HashMap<>()`).
- Avec des déclarations locales de `for` améliorées.
- Initialisation avec des méthodes dont le type de retour est évident.

## Limitations et précautions

Même si `var` simplifie le code, il peut nuire à la lisibilité quand le type n'est pas évident.

Il convient de l'éviter :

- Lorsque le type n'est pas évident.
- En cas de déclaration non initialisée.
- Pour des déclarations de champs ou de méthodes, `var` étant limité aux variables locales.

## Exemples pratiques

Déclaration explicite :

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

Avec `var` :

```
var scores = new HashMap<String, Integer>();
```

Autre exemple :

```
var list = List.of("Java", "Kotlin", "Scala");
```

Comme vous le voyez, `var` simplifie significativement la déclaration surtout lorsque le type est complexe.

# Exercices pratiques

## 1. Simplification de code :

Transformez les déclarations explicites en déclarations avec `var` :

```
ArrayList<String> names = new ArrayList<>();  
HashMap<Integer, String> idMap = new HashMap<>();
```

## 2. Discernement :

Identifiez le cas où l'utilisation de `var` n'est pas appropriée dans le code suivant et justifiez votre choix :

```
var calculator = getCalculator();  
var value; // Déclaration inappropriée
```

# Synthèse et ouverture

L'inférence de type local avec `var` est un ajout puissant de Java 10 qui augmente la lisibilité et la concision du code tout en conservant une forte sécurité de type.

En avançant, explorez comment d'autres qualités améliorées du langage dans les versions ultérieures peuvent construire sur ces bases pour rendre Java encore plus expressif et convivial.

# Introduction

Les expressions switch introduites dans Java 12 représentent une évolution significative par rapport au switch traditionnel.

Elles offrent une syntaxe plus concise et expressive, améliorant la lisibilité et réduisant le risque d'erreurs, telles que l'oubli des break.

Cette nouvelle fonctionnalité contribue à rendre le code Java plus moderne et fonctionnel.

# Objectifs pédagogiques

- Comprendre et utiliser les expressions switch.
- Différencier les expressions switch des instructions switch traditionnelles.
- Implémenter des expressions switch pour simplifier le code.
- Analyser les avantages en termes de lisibilité et sécurité.

# Syntaxe des expressions switch

Les expressions switch en Java 12 se présentent avec une syntaxe compacte.

Contrairement aux switch classiques, elles peuvent retourner une valeur.

L'utilisation de "case" et des flèches "->" permet de définir les actions.

```
int numLetters = switch (day) {  
    case "MONDAY", "FRIDAY", "SUNDAY" -> 6;  
    case "TUESDAY" -> 7;  
    case "THURSDAY", "SATURDAY" -> 8;  
    case "WEDNESDAY" -> 9;  
    default -> day.length();  
};
```

## Différences avec le switch classique

Les expressions switch éliminent le besoin de break.

Elles permettent d'attribuer le résultat directement à une variable.

Cela réduit le risque d'oublier un break, prévenant ainsi les "fall-through" non désirés.

## Avantages des expressions switch

1. **Lisibilité accrue** : Syntaxe plus claire, surtout pour les cas nombreux.
2. **Sécurité accrue** : Moins de risque d'erreur d'exécution liée au manque de break.
3. **Expressivité** : Possibilité de retourner une valeur, facilitant l'affectation.

## Exemples pratiques

### Exemple 1 : Calcul de jours

```
String dayType = switch (day) {  
    case "SATURDAY", "SUNDAY" -> "Weekend";  
    default -> {  
        System.out.println("It's a weekday");  
        yield "Weekday";  
    };  
};
```

### Exemple 2 : Calcul basé sur l'âge

```
int category = switch (age) {  
    case 0 -> "Infant";  
    case 1, 2 -> "Toddler";  
    default -> "Older";  
};
```

# Exercices pratiques

## Exercice 1 : Catégorisation

**Écrivez une expression switch pour catégoriser les mois en "Hiver", "Printemps", "Été", "Automne".**

## Exercice 2 : Jour de la semaine

**Créez une expression switch qui détermine si un jour est en semaine ou le weekend.**

### Critères d'évaluation

- Utiliser la nouvelle syntaxe avec flèches.
- Retourner une valeur basé sur les cases.

## Synthèse

Les expressions switch modernisent Java en offrant une syntaxe simplifiée et sécurisée.

Elles favorisent un code plus propre et expressif.

L'apprentissage de cette fonctionnalité est crucial pour écrire du code Java moderne et efficace, préparant le terrain pour un usage avancé dans les prochaines éditions de Java.

## Introduction

Le Garbage Collector G1 (Garbage-First) est introduit pour améliorer la gestion de la mémoire dans les applications Java.

Conçu pour remplacer le traditionnel CMS (Concurrent Mark-Sweep), G1 cible les pauses courtes et prévisibles, une valeur essentielle pour les applications nécessitant une haute performance et une gestion efficace de la mémoire.

## Objectifs pédagogiques

- Comprendre le rôle et le fonctionnement du Garbage Collector G1.
- Identifier les avantages et limites du GC G1 par rapport à d'autres collecteurs.
- Configurer G1 dans une application Java.
- Analyser et optimiser la performance avec le GC G1.

## Fonctionnement du GC G1

Le GC G1 fonctionne en divisant la mémoire en plusieurs régions, par opposition à un format contigu.

Lors du processus de collection, G1 cible d'abord les régions avec le plus de déchets (récupération de mémoire mort), d'où son nom "Garbage-First".

Cela permet une gestion plus efficace et moins intrusive de la mémoire.

Avantages :

- Régions indépendantes permettent des pauses plus courtes.
- Priorisation des régions maximise l'efficacité de la collection.
- Offre des options de tuning pour adapter la performance aux besoins spécifiques des applications.

## Configurer le GC G1

Pour activer le GC G1, ajoutez l'option suivante lors de l'exécution de votre application Java :

```
java -XX:+UseG1GC -jar MonApplication.jar
```

Principaux paramètres de configuration :

- `-XX:MaxGCPauseMillis` : Objectif de durée maximale pour les pauses de garbage collection.
- `-XX:InitiatingHeapOccupancyPercent` : Seuil de déclenchement de la collection basée sur le pourcentage d'occupation du tas.
- `-XX:G1HeapRegionSize` : Taille des régions dans le tas, réglable selon les besoins de l'application.

## Exemples de configuration

Pour un ajustement spécifique, considérer les besoins de production :

```
java -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:InitiatingHeapOccupancyPercent=45 -XX:G1HeapRegionSize=8M -jar MonApplication.jar
```

Cet exemple configure G1 pour des pauses maximales de 200ms, avec un déclenchement à 45% d'occupation du tas et une taille de région de 8Mo pour équilibrer performances et utilisation mémoire.

## Exercices pratiques

1. **Configuration Basique:** Activez le GC G1 dans une application Java et observez les résultats.
2. **Optimisation Avancée:** Ajustez les paramètres de `MaxGCPauseMillis` et `InitiatingHeapOccupancyPercent`, et évaluez l'impact sur la performance.

Pour chaque exercice, documentez les configurations appliquées et les résultats observés en termes de réduction de pauses et de performance globale.

## Synthèse

Le Garbage Collector G1 est une avancée significative pour gérer les besoins de mémoires intenses et les pauses minimales. À travers sa gestion des régions et son approche priorisée, il répond aux attentes des applications modernes. À l'avenir, explorez des techniques avancées de tuning pour perfectionner encore la performance de votre application Java.

# Introduction

L'évolution de Java entre les versions 8 et 12 a introduit des améliorations significatives.

Ces versions ont favorisé la modularité du code grâce à Project Jigsaw (Java 9), simplifié la syntaxe avec l'inférence de type local (var, Java 10), et amélioré le contrôle du flux avec les expressions switch (Java 12).

## Vue d'ensemble

Les avancées de Java 8 à 12 incluent l'intégration de nouveaux Garbage Collectors (G1 et ZGC), et invitent à explorer la modularité via un exercice pratique de création d'un mini-module Java avec dépendances internes.

## Introduction

L'évolution du langage Java de la version 8 à la 21 marque des avancées significatives en syntaxe, modularité et gestion des ressources.

Ce parcours sera exploré à travers trois phases clés : Java 8 à 12, Java 13 à 17, et Java 18 à 21. Chaque segment introduit des fonctionnalités transformatrices façonnant le développement moderne en Java.

## Vue d'ensemble des évolutions

- **Java 8 à 12** : Introduction de la modularité avec Project Jigsaw et de nouvelles expressions avec `switch`.
- **Java 13 à 17** : Amélioration de la manipulation de texte avec Text Blocks et introduction de `Records` pour simplifier les données immuables.
- **Java 18 à 21** : Focus sur la performance et la concurrence avec les Virtual Threads et de nouveaux patterns.

Ces étapes démontrent les efforts continus de Java pour répondre aux besoins changeants des développeurs.

## Introduction

Les types génériques bornés en Java sont essentiels pour créer des classes et des méthodes flexibles tout en garantissant la sécurité.

Ils permettent de restreindre les types pouvant être utilisés avec un paramètre générique, grâce aux mots-clés `extends` et `super`.

Cette fonctionnalité est cruciale pour écrire du code réutilisable et robuste, tout en évitant les erreurs au moment de la compilation.

## Objectifs pédagogiques

- Comprendre et utiliser les bornes supérieures et inférieures en Java.
- Appliquer les concepts de `extends` et `super` dans des classes et méthodes génériques.
- Maîtriser l'écriture de code générique réutilisable tout en assurant la sécurité de type.

# Bornes supérieures avec extends

L'utilisation de `extends` dans les types génériques permet de définir une "borne supérieure".

Cela limite les types pouvant être utilisés à une classe spécifique ou à ses sous-classes.

- Syntaxe : `<T extends ClassName>` .
- Utile pour implémenter des interfaces ou des classes abstraites.

Exemple : Les collections qui acceptent des objets de types homogènes.

# Bornes inférieures avec super

Le mot-clé `super` est utilisé pour définir une "borne inférieure".

Cela contraint l'utilisation d'un certain type ou de ses superclasses.

- Syntaxe : `<T super ClassName>` .
- Pratique pour consommer des instances d'un type ou de ses sous-types dans les méthodes.

Exemple : Algorithme de sorting qui fonctionne avec un `Comparator` .

# Exemples de extends

```
class Box<T extends Number> {  
    private T t;  
  
    public void set(T t) {  
        this.t = t;  
    }  
  
    public T get() {  
        return t;  
    }  
}
```

  

```
Box<Integer> integerBox = new Box<>();  
Box<Double> doubleBox = new Box<>();
```

- Ici, `Box` ne peut accepter que des types qui sont ou dérivent de `Number` .

# Exemples de super

```
public static void addNumbers(List<? super Integer> list) {  
    list.add(new Integer(42));  
}
```

- Cette méthode accepte des listes dont le type de base est `Integer` ou toute superclasse de `Integer` , permettant l'ajout d'entiers.

# Travaux pratiques

1. Créer une classe `ShapeBox<T extends Shape>` pouvant stocker des types dérivés de `Shape` .
  - Implémentez une méthode pour calculer l'aire totale des formes contenues.
2. Écrire une méthode générique `public static void copy(List<? super T> dest, List<T> src)` pour copier tous les éléments de `src` à `dest` .

Consignes : Utiliser les bornes génériques pour manager la sécurité des types.

## Synthèse

Les types génériques bornés permettent de restreindre les paramètres tout en augmentant la flexibilité. `extends` offre des bornes supérieures, restreignant les types à une hiérarchie spécifique, tandis que `super` facilite l'opération sur des hiérarchies de superclasses.

Ces concepts sont vitaux pour l'écriture de code générique performant et maintenable.

Pour aller plus loin, explorez les motifs de conception qui tirent parti des génériques en combinaison avec les collections.

## Introduction aux Wildcards

Les wildcards en Java sont une fonctionnalité puissante qui permet de rendre vos programmes plus flexibles lorsqu'il s'agit de travailler avec des collections génériques.

Ils servent à définir des limites de compatibilité dans les types paramétrisés.

En comprenant et utilisant les wildcards, vous pouvez manipuler des collections plus diversifiées tout en maintenant la sécurité de type.

## Objectifs pédagogiques

- Comprendre le concept de wildcards en Java
- Savoir utiliser les wildcards avec `? extends` et `? super`
- Identifier quand utiliser les wildcards pour la flexibilité du code
- Manipuler des collections génériques avec des wildcards

## Concept de Wildcards

Les wildcards en Java sont représentés par le symbole `?`.

Ils permettent de réaliser des opérations polymorphiques sur des collections.

Les wildcards sont utiles pour traiter des objets d'un type inconnu dans un contexte de typage paramétré et consistent en deux principaux types :

- **Upper Bounded Wildcards (`? extends T`)** : Permet de lire des objets de la collection en garantissant qu'ils sont d'un certain type ou sous-type.
- **Lower Bounded Wildcards (`? super T`)** : Permet d'écrire des objets dans la collection tout en garantissant qu'ils sont d'un certain type ou supertype.

# Upper Bounded Wildcards ( ? extends T )

Avec `? extends T`, vous pouvez lire de la collection, mais ne pas y écrire.

Cela signifie que vous pouvez effectuer des opérations qui nécessitent la lecture d'éléments, en vous assurant que l'élément est un type T ou un sous-type de T.

Par exemple, cela peut être utilisé pour calculer la somme d'une liste de nombres où la liste peut être composée d'Integer, Double, ou n'importe quelle sous-classe de Number.

# Lower Bounded Wildcards ( ? super T )

À l'inverse, `? super T` permet d'ajouter des éléments à la collection, garantissant seulement qu'ils sont d'un type T ou d'un supertype de T.

Cette contrainte est utile pour implémenter des algorithmes de consommation de données sur une collection.

Par exemple, si vous voulez ajouter des instances de Integer à une collection de Numbers, vous pouvez déclarer la collection avec `? super Integer`.

## Exemple avec Upper Bounded Wildcards

```
public static double sumOfList(List<? extends Number> list) {  
    double sum = 0.0;  
    for (Number n : list) {  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

Dans cet exemple, `List<? extends Number>` autorise la méthode à accepter des List de Integer, Double, etc., mais vous ne pouvez pas ajouter dans cette liste.

## Exemple avec Lower Bounded Wildcards

```
public static void addElements(List<? super Integer> list) {  
    list.add(1);  
    list.add(2);  
    list.add(3);  
}
```

Cet exemple montre `List<? super Integer>`, qui permet d'ajouter des entiers à la collection.

Les wildcards garantissent que le type dans la liste est Integer ou un supertype, comme Number ou Object.

## Exercice Pratique

1. **Exercice 1 :** Écrivez une méthode qui prend en paramètre une `List<? extends Animal>` et affiche le son spécifique de chaque animal.
  - **Indication :** Utilisez la classe de base `Animal` avec une méthode `makeSound()`.

**2. Exercice 2 :** Créez une méthode capable d'ajouter plusieurs types d'éléments (par exemple, des `Integer` et des `Double`) dans une `List<? super Number>`.

- **Critères d'évaluation :** Vérifiez que la méthode fonctionne avec différents types qui sont des sous-classes de `Number`.

## Synthèse et Ouverture

Les wildcards améliorent la flexibilité et l'interopérabilité des méthodes génériques en Java.

En maîtrisant `? extends` et `? super`, vous pouvez écrire des fonctions plus dynamiques et compatibilité-tolérantes.

La prochaine étape pour approfondir vos compétences avancées en Java consiste à explorer comment combiner différentes structures de données ou utiliser des patterns de design modernes avec la puissance de la généréricité.

## Introduction

La notion de `Comparable` en Java joue un rôle crucial dans la gestion des collections triées.

Elle permet à un objet de se comparer à d'autres objets pour définir un ordre naturel, facilitant ainsi des opérations comme le tri et la recherche.

Comprendre `Comparable` offre une meilleure prise en main des collections Java.

## Objectifs pédagogiques

- Comprendre le rôle de l'interface `Comparable`.
- Implémenter `Comparable` dans ses classes Java.
- Utiliser le tri naturel dans les collections Java.
- Analyser les différences entre `Comparable` et `Comparator`.

## Contenu détaillé

### Comprendre Comparable

L'interface `Comparable` est une interface générique de Java utilisée pour imposer un ordre naturel sur les objets d'une classe.

Elle contient une méthode unique : `compareTo`.

Cette méthode compare l'objet courant à un autre objet du même type.

### Méthode compareTo

La signature de la méthode `compareTo` est `int compareTo(T o)`.

Elle doit retourner :

- Un nombre négatif si l'objet courant est inférieur à l'objet spécifié.
- Zéro si l'objet courant est égal à l'objet spécifié.

- Un nombre positif si l'objet courant est supérieur à l'objet spécifié.

## Implémentation pratique

### Exemple simple

Supposons une classe `Person` avec un attribut `name`.

Nous souhaitons trier des objets `Person` par nom :

```
public class Person implements Comparable<Person> {  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public int compareTo(Person other) {  
        return this.name.compareTo(other.name);  
    }  
  
    // Getter, Setter et autres méthodes  
}
```

### Exemple avec tri

Une liste de personnes peut être triée facilement grâce à `Comparable` :

```
List<Person> people = Arrays.asList(new Person("Alice"), new Person("Bob"), new Person("Charlie"));  
Collections.sort(people);  
// La liste est maintenant triée par ordre alphabétique
```

## Comparaison avec `Comparator`

Le `Comparator` est une interface fonctionnelle utilisée pour définir plusieurs façons de comparer deux objets.

Contrairement à `Comparable`, `Comparator` peut être implémenté séparément de la classe des objets à comparer.

Il est souvent utilisé lorsque l'ordre d'un objet ne peut pas être défini dans la classe elle-même ou si plusieurs ordres sont nécessaires.

## Exercices pratiques

## **Exercice 1 : Implémenter Comparable**

Créez une classe Book avec des attributs title et author .

**Implémentez Comparable pour trier par title .**

### **Critères d'évaluation**

- La méthode compareTo doit comparer correctement les title .
- Vérifiez le bon tri avec une collection de Book .

## **Exercice 2 : Comparaison multiple**

**Associer un Comparator pour trier d'abord par author , puis par title si les auteurs sont identiques.**

### **Critères d'évaluation**

- Implémentation correcte de Comparator .
- Vérifiez le tri avec une collection appropriée.

## **Synthèse**

La maîtrise de Comparable et Comparator permet une flexibilité et une efficacité accrues dans la manipulation des collections triées en Java.

Tandis que Comparable définit un ordre naturel, Comparator offre la possibilité de multiples classements.

**Pour aller plus loin, explorez l'intégration de ces concepts avec les nouvelles fonctionnalités des Streams pour obtenir un code toujours plus performant et élégant.**

## **Comparator**

### **Introduction**

Le Comparator en Java est une interface fonctionnelle qui permet de définir un ordre de tri personnalisé pour des collections d'objets.

Contrairement à Comparable, qui impose un ordre naturel, Comparator offre une flexibilité pour trier les objets différemment selon les besoins.

Cette fonctionnalité est essentielle pour manipuler efficacement des collections avec des critères de tri spécifiques.

### **Objectifs pédagogiques**

- Comprendre l'interface Comparator.
- Appliquer Comparator pour trier des collections.
- Différencier Comparator de Comparable.

- Implémenter des tris personnalisés en Java.
- Utiliser des méthodes de référence pour simplifier le code.

## Contenu détaillé

### Interface Comparator :

- Comparator est une interface fonctionnelle introduite dans Java 1.2.
- Utilisée pour définir l'ordre relatif de deux objets.
- Contient une méthode unique `compare(T o1, T o2)`.

### Déférence avec Comparable :

- Comparable impose un ordre naturel (implémenté par l'objet).
- Comparator permet des ordres multiples (implémenté indépendamment).

### Méthodes clés :

- `compare` : Compare deux objets pour définir leur ordre.
- `reversed` : Inverse l'ordre du tri.
- `thenComparing` : Chaine plusieurs critères de tri.

## Contenu détaillé (suite)

### Avantages de Comparator :

- Flexibilité pour multiples critères de tri sans modifier les objets.
- Facilité de réutilisation pour différents types de tri.

### Implémentation :

- Peut être implémenté en tant que classe anonyme ou via expression lambda.
- Compatible avec les méthodes de tri des collections (`sort`, `sorted`).

## Exemples de code

```
// Exercice de tri des chaînes par longueur
Comparator<String> lengthComparator = (s1, s2) -> Integer.compare(s1.length(), s2.length());

// Exercice de tri des personnes par âge
Comparator<Person> ageComparator = Comparator.comparingInt(Person::getAge);

List<Person> people = Arrays.asList(new Person("Alice", 30), new Person("Bob", 25));
Collections.sort(people, ageComparator);
```

## Exemple annoté

```
// Exemple avec classe anonyme
Comparator<String> cmp = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
};

// Utilisation avec lambda
Comparator<String> cmpLambda = (s1, s2) -> s1.compareTo(s2);
```

- L'utilisation de lambdas réduit le code accidentel.
- `compareTo` est utilisé pour comparer les objets String eux-mêmes.

## Exercices pratiques

### Exercice 1 :

- Implémentez un Comparator pour trier une liste de produits par prix.

### Exercice 2 :

- Tri des étudiants par notes, puis par ordre alphabétique croissant.

### Critères d'évaluation :

- Fonctionnalité de tri respectée.
- Utilisation adéquate de Comparator.
- Simplicité et lisibilité du code.

## Synthèse

Le Comparator offre une solution puissante pour le tri sur mesure des collections Java.

Sa flexibilité facilite la gestion de cas complexes où plusieurs critères sont envisagés.

La compréhension et l'utilisation efficace des Comparators permettent de manipuler les données de manière précise et intégrée dans l'écosystème Java.

**Pour les prochaines étapes, nous explorerons les autres fonctionnalités avancées des collections Java, telles que Streams et Collectors.**

## Tri personnalisé

## Introduction

Dans ce module, nous explorerons le tri personnalisé en Java, une fonction clé lorsque l'ordre naturel d'une collection d'objets ne suffit pas.

En utilisant les interfaces `Comparable` et `Comparator`, nous pouvons définir des critères de tri personnalisés pour répondre aux besoins spécifiques de nos applications Java.

# Objectifs pédagogiques

- Comprendre et appliquer l'interface `Comparable`.
- Utiliser l'interface `Comparator` pour des tris personnalisés.
- Maîtriser le tri de collections avec des critères multiples.
- Implémenter des solutions de tri pratiques et optimales en Java.

## Usage de Comparable

L'interface `Comparable` est utilisée pour définir l'ordre naturel d'une classe donnée.

En implémentant la méthode `compareTo()`, un objet peut être comparé à un autre objet du même type.

Cela est utile pour les tris simples où une seule logique de comparaison est nécessaire.

## Utilisation de Comparator

Le `Comparator` est une interface fonctionnelle qui offre plus de flexibilité qu'une implémentation `Comparable`.

Vous pouvez créer plusieurs comparateurs pour une seule classe, permettant des critères de tri variés selon les besoins spécifiques. `Comparator` est idéal pour des conditions complexes ou multiples.

## Définitions de Comparateurs

`Comparator` permet de définir des comparaisons par fonctionnalités lambda ou par des classes anonymes.

Cela permet de créer un comparateur à la volée, augmentant la flexibilité et réduisant le besoin de code API supplémentaire.

Une approche commune est d'utiliser les méthodes statiques comme `Comparator.comparing()`.

## Exemple de code : Comparable

Considérons une classe `Person` :

```
public class Person implements Comparable<Person> {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    public int compareTo(Person other) {  
        return Integer.compare(this.age, other.age);  
    }  
}
```

Ce code trie les `Person` par âge.

## Exemple de code : Comparator

Pour une approche différente, imaginons un tri par nom :

```
Comparator<Person> nameComparator = new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
};
```

Ou avec une expression lambda pour la même tâche :

```
Comparator<Person> nameComparatorLambda =
    Comparator.comparing(Person::getName);
```

## Exemple Combiné

Nous pouvons combiner plusieurs `Comparators` pour comparer par nom d'abord, puis par âge :

```
Comparator<Person> combinedComparator =
    Comparator.comparing(Person::getName)
        .thenComparing(Person::getAge);
```

Cette combinaison est puissante pour des tris multicritères.

## Exercices pratiques

### Exercice 1 : Tri simple

- Créez une liste de personnes et triez-la en utilisant `Comparable`.

### Exercice 2 : Tri multiple

- Implémentez un `Comparator` pour trier la même liste d'abord par âge, puis par nom.

## Critères d'évaluation

- Respect du contrat des interfaces.
- Correctitude et lisibilité du code.

## Synthèse

Nous avons vu comment personnaliser le tri en Java grâce aux interfaces `Comparable` et `Comparator`.

Ces outils permettent de répondre à des besoins de tri spécifiques et complexes, offrant ainsi une grande flexibilité dans la gestion des collections.

Dans les modules futurs, nous aborderons d'autres aspects avancés des collections et la programmation fonctionnelle avec Java Streams.

## Introduction

La notion de Stream parallèle en Java est essentielle pour exploiter la puissance des processeurs multi-cœurs.

Les Streams parallèles permettent de traiter les collections de données de manière concurrente, améliorant ainsi les performances des opérations lourdes.

Cette approche est particulièrement efficace pour les tâches de calcul intensif.

## Objectifs pédagogiques

- Comprendre le concept de Stream parallèle en Java.
- Apprendre à créer et manipuler des Streams parallèles.
- Identifier les avantages et les inconvénients des Streams parallèles.
- Écrire et exécuter des programmes Java exploitant les Streams parallèles de manière efficace.

## Concept de Stream parallèle

Les Streams en Java, introduits avec Java 8, facilitent le traitement des collections.

Les Streams parallèles partitionnent automatiquement le flux en sous-flux traités en parallèle.

- Utilisation du framework Fork/Join pour le parallélisme.
- Bénéficie immédiatement des architectures multi-cœurs.
- Favorise un code concis et lisible pour les opérations en parallèle.

## Avantages et Inconvénients

Les Streams parallèles accélèrent les traitements mais peuvent introduire de la complexité :

- **Avantages** : Traitement plus rapide grâce au parallélisme.
- **Inconvénients** : Overhead de la gestion de threads, complexité du débogage.

L'usage inapproprié des Streams parallèles peut causer des problèmes de performance.

## Créer un Stream Parallèle

Pour créer un Stream parallèle, on utilise la méthode `parallelStream()` sur une collection ou `stream().parallel()`.

Voici comment convertir un Stream séquentiel en parallèle :

```
List<String> list = Arrays.asList("a", "b", "c");
list.parallelStream().forEach(System.out::println);
```

L'exécution des éléments se fait en parallèle, l'ordre de sortie n'est pas garanti.

# Exemples de code

Considérons le calcul de la somme des carrés des nombres d'une liste en utilisant un Stream parallèle :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sumOfSquares = numbers.parallelStream()
    .mapToInt(n -> n * n)
    .sum();
System.out.println("Sum of squares: " + sumOfSquares);
```

Cet exemple calcule la somme des carrés de manière efficace grâce au parallélisme.

## Exercices pratiques

1. **Exercice 1 :** Utilisez un Stream parallèle pour filtrer et compter les mots d'un fichier texte ayant plus de 5 lettres.
  - Critère : Utiliser les méthodes `filter()` et `count()`.
  - Critères d'évaluation : Exactitude du résultat, bonne utilisation du parallélisme.
2. **Exercice 2 :** Écrire un programme pour multiplier simultanément les éléments d'un tableau d'entiers par un facteur donné en utilisant Streams parallèles.
  - Critères d'évaluation : Efficacité du programme, code clair et concise.

## Synthèse et ouverture

Les Streams parallèles offrent une manière puissante de traiter les données de manière concurrente, tirant parti des architectures multi-coeurs.

Cependant, il est crucial de comprendre où et comment les utiliser pour éviter les pièges potentiels liés à la gestion des threads.

En poursuivant l'exploration des fonctionnalités modernes de Java, intégrer les Collectors pour regrouper et modifier les résultats pourrait être la prochaine étape dans l'apprentissage approfondi des Streams.

## Introduction

Le `Collectors.groupingBy` en Java est une fonctionnalité puissante du Stream API qui permet de regrouper des éléments en fonction d'une clé donnée.

Cela offre une flexibilité considérable pour manipuler et analyser des collections de données, rendant le traitement des données plus clair et efficace.

## Objectifs pédagogiques

- Comprendre le concept de regroupement avec `groupingBy`.
- Appliquer `Collectors.groupingBy` à des collections de données en Java.
- Utiliser des Collectors complémentaires pour des agrégations avancées.

# Contenu détaillé

Le `Collectors.groupingBy` est utilisé pour collecter les éléments d'un Stream en une Map, où les clés sont les résultats de l'application d'une fonction de classification et les valeurs sont des listes d'objets de l'élément de Stream.

- `groupingBy` est utile pour créer des sous-groupes au sein d'une collection.
- Utilisation typique : transformation d'un Stream d'objets en une Map structurée.

Le `Collectors.groupingBy` supporte plusieurs formes :

1. `Collectors.groupingBy(classifier)` .
2. `Collectors.groupingBy(classifier, downstream)` .
3. `Collectors.groupingBy(classifier, supplier, downstream)` .

Chaque forme augmente en complexité et en fonctionnalités.

## Contenu détaillé (suite)

1. **Basic Grouping** : Regroupement basique avec une clé simple.
2. **Complex Grouping** : Utilisation de `downstream collector` pour effectuer des opérations supplémentaires sur les groupes.
3. **Performance Tuning** : Choix d'une `supplier` pour un Map personnalisé afin de contrôler l'implémentation sous-jacente.

Les classes fréquemment utilisées incluent `Collectors`, `Stream` et `Function`.

Le concept de lambdas simplifie souvent l'utilisation.

## Exemples de code

### Exemple de Grouping Basique

```
List<String> items = Arrays.asList("apple", "banana", "orange", "watermelon");
Map<Integer, List<String>> lengthMap = items.stream()
    .collect(Collectors.groupingBy(String::length));
System.out.println(lengthMap);
```

Cet exemple regroupe des chaînes de caractères par la longueur.

### Exemple de Grouping Complexé

```
Map<Integer, Set<String>> lengthMap = items.stream()
    .collect(Collectors.groupingBy(String::length, Collectors.toSet()));
System.out.println(lengthMap);
```

Ici, nous utilisons `toSet` pour éviter les doublons au sein des groupes.

## Exemple pratique

## Regroupement avec une Opération de Comptage

```
List<String> items = Arrays.asList("apple", "banana", "orange", "apple", "orange", "watermelon");
Map<String, Long> countMap = items.stream()
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
System.out.println(countMap);
```

Ce code compte la fréquence des éléments dans la liste initiale, regroupant par identité.

## Exercices pratiques

1. Implémentez un regroupement de personnes par longueur de nom à partir d'une liste d'objets `Person`.
2. Modifiez pour utiliser un `Collector` supplémentaire qui concatène les noms.
3. Évaluez la performance en utilisant un `supplier` pour une Map dédiée de haute performance.

Critères d'évaluation : Utilisation correcte des méthodes, efficacité du code, et clarté de la solution.

## Synthèse et ouverture

En conclusion, `Collectors.groupingBy` est un outil essentiel pour organiser des données complexes en sous-groupes logiques.

Il améliore la lisibilité et l'efficacité du code Java moderne.

Pour aller plus loin, explorez comment `groupingBy` peut être combiné avec d'autres opérations de Stream pour réaliser des analyses de données encore plus poussées.

## Collectors.mapping

## Introduction

Dans le cadre de l'utilisation des streams en Java, `Collectors.mapping` est une fonction puissante qui permet de transformer les éléments d'un Stream lors de la phase de collecte.

Elle est particulièrement utile pour appliquer une fonction de mapping sur les éléments collectés, facilitant ainsi l'extraction et la transformation des données dans les collections.

## Objectifs pédagogiques

- Comprendre le rôle de `Collectors.mapping` dans les flux de données.
- Appliquer `Collectors.mapping` pour transformer et collecter des données.
- Mettre en œuvre des cas concrets d'utilisation de `Collectors.mapping`.
- Comparer `Collectors.mapping` avec d'autres méthodes de collecte.

# Utilisation de Collectors.mapping

Collectors.mapping est utilisé avec collect pour transformer un stream.

Le mapping applique une fonction sur chaque élément avant de collecter les résultats.

Par exemple, vous pouvez extraire et collecter uniquement certains attributs d'objets.

- **Syntaxe principale:** Utilise Collectors.mapping conjointement avec une clé de regroupement pour transformer des données dans un Map .
- **Pourquoi l'utiliser ?** Simplifie la transformation et la collecte de données en une seule opération fluide.
- **Nature des transformations:** Permet des opérations comme la conversion de types, l'extraction de propriétés ou la transformation en chaînes.

## Exemples de code

### Exemple de base

Supposons que nous ayons une liste d'objets Person et que nous souhaitons collecter tous les noms en majuscules :

```
List<Person> people = ...;
Map<String, List<String>> namesByCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity,
        Collectors.mapping(person -> person.getName().toUpperCase(),
        Collectors.toList())));
```

Ici, Collectors.mapping transforme les noms en majuscules avant de les collecter dans une liste par ville.

### Comparaison avec d'autres méthodes

#### Sans Collectors.mapping

```
List<String> names = people.stream()
    .map(person -> person.getName().toUpperCase())
    .collect(Collectors.toList());
```

#### Avec Collectors.mapping

Permet une intégration directe dans des collectes plus complexes, comme le regroupement.

## Exercices pratiques

## Exercice 1

**Objectif:** Utiliser `Collectors.mapping` pour extraire et formater des données.

- Créez une liste d'objets `Book` avec des attributs `title` et `author`.
- Utilisez `Collectors.mapping` pour collecter les titres de tous les livres en minuscules, regroupés par année de publication.

## Exercice 2

**Objectif:** Intégrer `Collectors.mapping` dans un processus de transformation.

- Modifiez l'exercice précédent pour inclure à la fois la transformation des titres et l'ajout d'une longueur de chaque titre après le nom de l'auteur.

**Critères d'évaluation:**

- Correction des résultats : les titres doivent être en minuscules et groupés correctement.
- Utilisation efficace de `Collectors.mapping`.

## Synthèse et perspectives

Le `Collectors.mapping` élargit les possibilités de transformation lors de la collecte des streams en Java, apportant flexibilité et précision aux opérations de traitement des données.

Il s'intègre bien dans des chaînes de traitement plus complexes, surtout lors de l'utilisation conjointe avec d'autres collecteurs.

En pratique, maîtriser cette fonctionnalité permet d'écrire un code plus concis et performant.

Dans la suite de vos apprentissages, explorez comment `Collectors.mapping` s'intègre avec d'autres opérations liées aux streams pour augmenter l'efficacité de vos manipulations de données.

## Introduction

Le `Collectors.teeing` est une fonctionnalité avancée introduite dans Java 12, qui enrichit les capacités de traitement des collections avec les Streams.

Elle permet de combiner les résultats de deux collectors distincts en un résultat final, apportant une grande flexibilité et puissance dans le traitement des données.

Cette fonctionnalité est particulièrement utile lorsque vous devez effectuer deux opérations distinctes sur un même Stream et réunir les résultats.

## Objectifs pédagogiques

- Comprendre le concept de `Collectors.teeing`.
- Apprendre à créer des pipelines de Stream complexes.
- Savoir combiner les résultats de plusieurs opérations sur un même Stream.
- Pouvoir implémenter du code Java efficace avec `Collectors.teeing`.

# Fonctionnement du Collectors.teeing

Le `Collectors.teeing` prend trois arguments : deux Collectors et une fonction de combinaison.

Chaque Collector traite les éléments du Stream et la fonction combine leurs résultats.

Cela permet d'extraire des données de manière flexible et efficace.

Il nécessite, cependant, d'avoir une bonne compréhension des Collectors et de la manière dont les Streams fonctionnent dans Java.

## Pédagogie et Utilité

Utiliser `Collectors.teeing` évite le traitement multiple de données en permettant de réaliser deux opérations en parallèle sur le même Stream.

Cela conduit non seulement à un code plus lisible et maintenable mais aussi potentiellement à des gains de performance.

## Exemple : Somme et Moyenne

Considérons un Stream d'entiers; nous souhaitons calculer la somme et la moyenne des nombres en une seule passe.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
double result = numbers.stream()
    .collect(Collectors.teeing(
        Collectors.summingInt(Integer::intValue),
        Collectors.averagingInt(Integer::intValue),
        (sum, avg) -> sum + avg));
```

Dans cet exemple, `Collectors.summingInt` calcule la somme et `Collectors.averagingInt` calcule la moyenne.

La fonction de combinaison associe les deux résultats.

## Exemple : Min et Max

Vous pouvez également utiliser `Collectors.teeing` pour obtenir simultanément le minimum et le maximum d'un ensemble de données.

```
List<Integer> numbers = List.of(23, 4, 15, 42, 8);
String result = numbers.stream()
    .collect(Collectors.teeing(
        Collectors.minBy(Integer::compareTo),
        Collectors.maxBy(Integer::compareTo),
        (min, max) -> "Min: " + min.orElseThrow() + ", Max: " + max.orElseThrow()));
```

Ici, `Collectors.minBy` et `Collectors.maxBy` identifient respectivement le minimum et le maximum, que la fonction de combinaison réunit.

## Exercice Pratique

1. **Objectif :** Utiliser `Collectors.teeing` pour déterminer le nombre total de mots et la longueur moyenne des mots dans un texte.

2. **Consignes :**

- Créez un Stream à partir d'une liste de mots.

- Utilisez `Collectors.teeing` pour obtenir le total de mots et leur longueur moyenne.
- Affichez les résultats combinés dans une phrase bien structurée.

**Critères d'évaluation :** Vérification de l'utilisation correcte de `Collectors.teeing`, et de l'exactitude des résultats.

## Synthèse

Le `Collectors.teeing` est un outil puissant dans Java pour traiter des streams en parallèle et combiner les résultats sans devoir retraverser les données.

Grâce à sa flexibilité et ses capacités de combinaison, `Collectors.teeing` aide à simplifier le code tout en améliorant potentiellement la performance.

Cela ouvre la voie à des pratiques de traitement de données plus efficaces dans des applications Java avancées.

## Exercice : Trier et Grouper un Dataset Complexé

### Introduction

Dans cet exercice, nous explorerons les techniques avancées de Java pour manipuler un dataset complexe en utilisant des structures de données modernes et efficaces.

Nous nous concentrerons sur la capacité à trier et regrouper des données grâce à l'utilisation de comparateurs et des API de flux de Java.

Cette compétence est cruciale pour travailler avec de grandes quantités de données de manière performante et structurée.

### Objectifs Pédagogiques

- Comprendre et utiliser `Comparator` pour le tri personnalisé.
- Appliquer les opérations de regroupement avec `Collectors.groupingBy`.
- Manipuler les données avec les Streams Java pour améliorer la lisibilité et la performance du code.
- Concevoir des solutions modulaires permettant une extension facile des comportements de tri et de regroupement.

### Contenu Détailé

Dans cet exercice, nous allons d'abord revoir comment utiliser des `Comparators` en Java.

Ces objets permettent de définir des ordres personnalisés pour n'importe quel type d'objet.

En parallèle, vous apprendrez à utiliser des collecteurs permettant un regroupement efficace des données.

Nous appliquerons ces concepts afin de résoudre des cas concrets de manipulation de données.

# Comparators et Trie

Un `Comparator` permet de définir un ordre personnalisé de tri.

En Java, vous pouvez implémenter l'interface `Comparator` et la méthode `compare`.

Considérez cet exemple où nous trions une liste de personnes par nom et âge :

```
List<Person> people = ...;
people.sort(Comparator.comparing(Person::getName).thenComparing(Person::getAge));
```

Dans cet exemple, nous utilisons `Comparator.comparing` avec une méthode de référence pour trier d'abord par nom, puis par âge.

# Utilisation des Streams et GroupingBy

Les Streams en Java permettent un traitement fluide et efficace des collections de données. `Collectors.groupingBy` est particulièrement utile pour regrouper des objets sous certaines propriétés.

Par exemple, pour regrouper une liste de transactions par type :

```
Map<String, List<Transaction>> transactionsByType =
    transactions.stream()
        .collect(Collectors.groupingBy(Transaction::getType));
```

Ce code regroupe toutes les transactions par leur type en utilisant un seul appel fluide de méthode.

# Exemples Pratiques

Essayons un exemple simple pour solidifier la compréhension.

Considérez une classe `Produit` avec des attributs pour le nom et le prix.

Nous trierons une liste de produits par prix de manière décroissante et les regrouperons par catégorie :

```
List<Produit> produits = ...;
Map<String, List<Produit>> produitsParCategorie = produits.stream()
    .sorted(Comparator.comparing(Produit:: getPrix).reversed())
    .collect(Collectors.groupingBy(Produit:: getCategorie));
```

Ce code montre comment combiner tri et regroupement pour organiser des produits par prix décroissant par catégorie.

# Exercices Progressifs

- 1. Tri de Personnes** : Donnez une liste d'objets `Personne`, triez la liste par nom puis par âge.
- 2. Regroupement de Commandes** : Pour des objets `Commande`, groupez-les par statut (e.g., livré, en attente) et après, triez chaque groupe par date de commande.
- 3. Analyse de Données** : En partant d'un dataset de `Ventes`, regroupez-les par région et triez chaque groupe de manière décroissante par le montant de la vente.

Critères d'évaluation : efficacité du code et clarté des solutions proposées.

# Synthèse

Cet exercice a permis de découvrir comment trier et regrouper efficacement des datasets complexes en Java grâce aux `Comparators` et à l'API Stream.

Vous devriez maintenant être à l'aise pour appliquer ces techniques à vos propres projets, améliorant ainsi la performance et la lisibilité de votre code.

**Pour aller plus loin, explorez les opérations additionnelles proposées par les Streams et investiguez des cas d'usage réels de gestion de données volumineuses.**

## Introduction

Dans le cadre de la programmation Java avancée, l'utilisation de la généricité et des collections joue un rôle crucial.

Ce module explore diverses techniques et concepts pour manipuler les structures de données de manière flexible et efficace.

Nous approfondirons des notions telles que les types génériques bornés, les wildcards, et les interfaces `Comparable` et `Comparator`.

De plus, nous aborderons les techniques de tri personnalisé et l'utilisation de flux parallèles pour optimiser les traitements de données.

## Vue d'ensemble

Les sous-notions se décomposent en plusieurs aspects :

- **Types génériques bornés** : Restriction des types pour assurer la sécurité de type.
- **Wildcards** : Introduction de flexibilité dans les méthodes génériques.
- **Comparable et Comparator** : Interfaces cruciales pour le tri d'objets.
- **Tri personnalisé** : Personnalisation des critères de tri.
- **Stream parallèle** : Amélioration des performances de traitement.
- **Collectors** : Utilisation avancée pour regrouper et transformer les données (groupingBy, mapping, teeing).
- **Exercice pratique** : Applications réelles pour structurer et manipuler de vastes datasets de manière efficace.

## Introduction

Les Records en Java, introduits dans la version 14, simplifient la création de classes immuables.

Ils réduisent significativement le code boilerplate en générant automatiquement les méthodes comme `equals()`, `hashCode()`, et `toString()`.

Dans cette section avancée, nous explorerons comment optimiser l'utilisation des Records et les bonnes pratiques pour tirer le meilleur parti de ces fonctionnalités.

## Objectifs pédagogiques

- Comprendre l'utilité des Records en Java
- Maîtriser la syntaxe et les bonnes pratiques d'utilisation des Records

- Différencier les Records des classes traditionnelles
- Implémenter des Records dans des scénarios avancés
- Appliquer les Records pour améliorer la concision et la clarté du code

## Records en Java

Un Record en Java est une classe immuable qui représente de façon concise des données simples.

Lorsqu'un Record est déclaré, Java génère automatiquement des implémentations pour plusieurs méthodes.

Cela permet d'économiser du temps et d'éviter des erreurs classiques lors de la réécriture de ces méthodes.

- Les Records sont définis avec le mot-clé `record`.
- Ils ne peuvent pas répondre au même besoin qu'une classe si des comportements spécifiques sont demandés.
- Ils sont immuables par nature.

## Création et structure

Déclarer un Record est simple.

Voici sa structure de base:

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

Ce code définit un Record appelé `Point` avec deux composants `x` et `y`.

- `x` et `y` sont les champs immuables du Record.
- Les méthodes `equals()`, `hashCode()`, et `toString()` sont générées automatiquement.
- Les composants peuvent également avoir des annotations et peuvent préciser des comportements avec des méthodes personnalisées.

## Avantages des Records

L'utilisation des Records offre plusieurs avantages:

- **Concision:** Réduction significative du code nécessaire pour créer des objets simples.
- **Immuabilité:** Promotion de la sécurité des threads grâce à l'immutabilité.
- **Lisibilité:** Structure claire qui met en avant les données sans ajouter de complexité.

Ces caractéristiques font des Records un bon choix pour modéliser des données claires et standardisées, telles que des données de configuration ou de transfert.

## Différences avec les Classes

Les Records ne doivent pas être utilisés pour tout par défaut.

Leur immuabilité peut être une contrainte dans certains cas.

- Les Records ne peuvent pas étendre d'autres classes (mais peuvent implémenter des interfaces).
- Ils sont principalement utilisés pour des données contenantes et pas pour des logiques complexes.
- Comparativement, les classes traditionnelles offrent plus de flexibilité sur la gestion des comportements.

# Exemples de Code

Utilisation de Record pour un objet simple:

```
public record Product(String name, double price) {  
    public Product {  
        if(price < 0) {  
            throw new IllegalArgumentException("Price cannot be negative.");  
        }  
    }  
}
```

Ce Record `Product` impose une validation sur le prix pour garantir qu'il est positif.

La validation est effectuée dans le constructeur compact du Record.

# Exemples de Scénarios

- **Transfert de données** : En simplifiant les échanges au sein de systèmes distribués par leur clarté et leur concision.
- **Configurations appliquées** : En encapsulant les configurations immuables utilisées dans les applications.

Les Records permettent de facilement réutiliser ces données avec peu de possibilités d'erreur.

# Exercices pratiques

1. **Créer un Record simple**: Écrire un Record pour un objet `Rectangle` avec des validations pour s'assurer que la longueur et la largeur sont positives.
2. **Étendre un Record** : Implémentez une interface existante avec un Record.
3. **Refactoriser le code existant** : Trouvez une classe immuable existante et réécrivez-la en tant que Record pour observer la réduction de code.

Évaluez chaque exercice en vous assurant que les règles d'immutabilité et de validation sont correctement implémentées.

# Synthèse et Ouverture

Les Records en Java simplifient la gestion des données immuables dans les applications.

En réduisant le code indispensable pour gérer les objets simples, ils augmentent la lisibilité et la sécurité. À l'avenir, les fonctionnalités des Records peuvent être étendues pour fournir des modèles plus flexibles, rendant Java encore plus efficace pour le développement moderne.

Anticipez l'apprentissage des Sealed Interfaces pour explorer plus de capacités de Java 17 et au-delà.

# Sealed interface

## Introduction

Les `sealed interfaces`, introduites en Java 17, permettent de restreindre les classes qui peuvent les implémenter.

Elles fournissent un contrôle granulé sur l'héritage, augmentant ainsi la sécurité et la clarté du code.

En utilisant les `sealed interfaces`, les développeurs peuvent définir un ensemble limité de sous-types autorisés, améliorant ainsi l'intégrité de la hiérarchie des types.

## Objectifs pédagogiques

- Comprendre le concept de `sealed interfaces` en Java.
- Savoir comment les implémenter dans un projet.
- Apprendre à définir des sous-types autorisés pour des interfaces scellées.
- Analyser les avantages en termes de sécurité et de maintenance du code.

## Contenu détaillé

Le concept de `sealed interfaces` vise à offrir un contrôle précis sur quelle classe peut les implémenter.

Une `sealed interface` peut spécifier quels sont les sous-types autorisés en utilisant le mot-clé `permits`.

Cela empêche les extensions imprévues et renforce l'invariant de subclassing.

Les sous-types d'une `sealed interface` doivent être marqués comme `final`, `non-sealed` ou rester `sealed`.

- `final` : Empêche tout sous-typage.
- `non-sealed` : Permet un classement descendant non restreint.
- `sealed` : Maintient des restrictions pour les sous-classes.

## Avantages des Sealed Interfaces

Les `sealed interfaces` offrent plusieurs avantages :

- **Sécurité renforcée** : Protègent l'intégrité de la hiérarchie des classes.
- **Lisibilité améliorée** : Rendent explicites les relations d'héritage.
- **Maintenance facilitée** : Réduisent les risques d'extensions non intentionnelles du code.

Leurs implémentations permettent un modèle d'héritage plus contrôlé et prévisible, particulièrement utile dans de grands projets où la cohérence est cruciale.

## Exemples de code

```
// Définition d'une sealed interface
public sealed interface Shape
    permits Circle, Square { }

// Implémentation finale d'un sous-type
public final class Circle implements Shape {
    // Implementation spécifique à Circle
}

// Implémentation d'un sous-type non-sealed
public non-sealed class Square implements Shape {
    // Implementation spécifique à Square
    // D'autres sous-types peuvent dériver de Square
}
```

Cet exemple montre comment la `sealed interface Shape` limite ses implémentations à `Circle` et `Square`, garantissant que pas d'autres classes ne peuvent implémenter `Shape` sans être déclarées.

## Exercices pratiques

- Définir une Sealed Interface :** Créez une interface scellée `Transport` permettant uniquement les classes `Car` et `Bike` comme sous-types.
- Ajout de sous-types et exploration :** Ajoutez un nouveau sous-type non-sealed `Skateboard` qui étend `Transport` et implémentez des méthodes spécifiques pour ce sous-type.
- Analyse Comparative :** Réalisez une analyse des avantages d'utiliser les `sealed interfaces` par rapport aux interfaces classiques dans un court essai.

Critères d'évaluation : Exactitude du code, compréhension des concepts, qualités des réflexions et des analyses produites.

## Synthèse

Les `sealed interfaces` enrichissent significativement Java en offrant une nouvelle manière de structurer les hiérarchies de classes.

Ces interfaces fournissent sécurité et maintenance, tout en garantissant que seul un ensemble prédéfini de sous-types peut les implémenter.

Pour les développeurs cherchant à contrôler le design et l'intégrité de leurs systèmes, les `sealed interfaces` constituent un ajout précieux et puissant. À l'avenir, explorez comment elles peuvent interagir avec d'autres fonctionnalités modernes de Java comme les records pour construire des applications robustes.

## Introduction

L'immutabilité est un concept clé en programmation qui garantit que les objets ne peuvent pas être modifiés après leur création.

En Java, cela assure non seulement la sécurité des données contre les changements accidentels mais aide aussi à simplifier la gestion des états dans les applications multithread.

## Objectifs pédagogiques

- Comprendre le concept d'immutabilité et son importance
- Distinguer les objets immuables des objets muables
- Mettre en œuvre l'immutabilité dans les classes Java

- Utiliser l'immédiateté pour améliorer la sécurité et la performance des applications

## Concept de l'Immutabilité

En Java, un objet immuable est un objet dont l'état ne peut pas être modifié après qu'il a été créé.

Pour qu'une classe soit immuable, il faut :

- Déclarer toutes les variables d'instance comme `final`.
- Ne fournir que des méthodes "getter", pas de "setter".
- Initialiser les variables d'instance dans le constructeur, sans exposer de références modifiables.

## Avantages de l'Immutabilité

Les objets immuables offrent plusieurs avantages :

- **Plus sûr en multithreading** : Pas de souci avec les conditions de concurrence.
- **Simplicité** : Plus facile à comprendre car aucun état ne change.
- **Optimisation** : Possibilité de les mettre en cache pour éviter des créations répétées.

## Créer une Classe Immuable

Pour créer une classe immuable en Java, suivez les principes :

- Rendez la classe `final` pour empêcher d'autres classes de la sous-classer.
- Rendez les champs `final` et `private`.
- Fourni un constructeur paramétré pour initialiser les attributs.
- Aucun "setter" : ne modifiez jamais l'état après la création.
- Si l'objet contient des références à des objets modifiables, assurez-vous de copier chaque objet à l'intérieur de votre classe.

## Exemple de Classe Immuable

```
public final class ImmutablePoint {  
    private final int x;  
    private final int y;  
  
    public ImmutablePoint(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}
```

Cette classe `ImmutablePoint` offre un bon exemple d'immortalité.

## Utiliser les Collections Immuables

Java 9 a introduit des méthodes pour créer des collections immuables :

```
List<String> list = List.of("A", "B", "C");
Set<String> set = Set.of("X", "Y", "Z");
Map<String, String> map = Map.of("key1", "value1", "key2", "value2");
```

Ces collections ne peuvent pas être modifiées après leur création, rendant leur état sûr et stable.

## Exercices Pratiques

1. **Créer une classe immuable** : Conception d'une classe `ImmutableUser` avec des champs `name` et `email`.
2. **Collections immuables** : Convertir une Liste modifiable existante en Liste immuable.
3. **Discussion** : Quels sont les scénarios où l'immortalité peut limiter la flexibilité ?

## Synthèse

L'immortalité en Java joue un rôle crucial dans le développement sécurisé et efficace d'applications modernes.

Elle contribue à réduire les erreurs liées à la concurrence et à assurer une gestion plus simple des états.

En tant que pratique de programmation, cela renforce la robustesse et la maintenabilité de vos codes, préfigurant des concepts explorés plus avant dans des domaines comme le design de software et la manipulation des données massives.

## Introduction

La thread-safety en Java est essentielle dans les applications multi-thread où plusieurs threads peuvent accéder simultanément aux mêmes données.

Assurer la sécurité des threads permet d'éviter des incohérences et des erreurs d'accès concurrentiel, garantissant ainsi une exécution prévisible et fiable de votre programme.

C'est crucial pour développer des applications performantes et robustes.

## Objectifs pédagogiques

- Comprendre le concept de thread-safety en Java.
- Identifier les problèmes courants de concurrence.
- Appliquer des techniques pour développer des applications thread-safe.
- Utiliser les classes et API de Java pour la synchronisation des threads.

# Concepts de Thread-safety

La thread-safety signifie que plusieurs threads peuvent accéder à une ressource partagée sans entraîner d'état incohérent.

Un objet est dit thread-safe si son état est toujours cohérent, même lorsque plusieurs threads le modifient simultanément.

Les problèmes courants incluent les conditions de course, où l'ordre des opérations conduit à un comportement erroné.

## Problèmes de Concurrence

Les problèmes de concurrence surviennent souvent lorsque des threads tentent de mettre à jour ou de lire des données à partir de ressources partagées.

Les conditions de course et les interblocages sont parmi les erreurs classiques.

Il est crucial d'encadrer l'accès à ces ressources pour éviter les corruptions de données et les comportements imprévisibles.

## Techniques de Synchronisation

Java propose différentes techniques pour assurer la sécurité des threads, incluant :

- Les blocs synchronisés avec le mot-clé `synchronized`.
- L'utilisation de verrous explicites (`java.util.concurrent.locks.Lock`).
- Les objets `ThreadLocal` pour des variables locales de thread.
- Les collections concurrentes fournies par Java pour manipuler en sécurité des collections partagées.

## Exemples de Code (Synchronized)

```
public class Counter {  
    private int count = 0;  
  
    public synchronized void increment() {  
        count++;  
    }  
  
    public synchronized int getCount() {  
        return count;  
    }  
}
```

Cet exemple montre un compteur simple où les méthodes sont synchronisées pour éviter les incohérences lorsque plusieurs threads accèdent simultanément à l'objet Counter.

# Exemples de Code (Lock)

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private int count = 0;
    private Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        return count;
    }
}
```

Dans cet exemple, `ReentrantLock` est utilisé pour un contrôle explicite du verrouillage, offrant plus de flexibilité que le mot-clé `synchronized`.

## Exercice 1 : Bloc Synchronized

1. Créez une classe Java avec un compteur partageable entre threads.
2. Utilisez le mot-clé `synchronized` pour protéger la méthode d'incrémentation.
3. Comparez le comportement du compteur avec et sans synchronisation en créant plusieurs threads.

Critères d'évaluation : Cohérence des valeurs de compteur dans des environnements multi-thread.

## Exercice 2 : Utilisation de Locks

1. Imitez l'exercice précédent en remplaçant `synchronized` par `ReentrantLock`.
2. Testez le code avec plusieurs threads et commentez les résultats.

Critères d'évaluation : Compréhension des bénéfices de l'utilisation de locks par rapport à `synchronized`.

## Synthèse

Pour assurer la sécurité des threads en Java, il est essentiel de comprendre et d'utiliser correctement les techniques de synchronisation.

En choisissant judicieusement entre `synchronized`, locks et collections concurrentes, les développeurs peuvent prévenir les problèmes de concurrence et garantir le bon fonctionnement de leurs applications.

**Les prochaines étapes incluront l'exploration avancée des outils Java pour optimiser les performances multithread.**

## Pattern builder immuable

### Introduction

Le pattern builder immuable est une construction avancée en Java permettant de créer des objets de manière flexible et sécurisée tout en garantissant leur immutabilité.

Dans un monde où le parallélisme et la sécurité des données sont cruciaux, ce pattern offre une solution élégante pour éviter les effets de bord imprévus et les bugs liés à la mutabilité des objets.

### Objectifs pédagogiques

- Comprendre le concept de pattern builder immuable
- Construire des objets complexes en garantissant leur immutabilité
- Utiliser les builders pour une construction fluide et lisible
- Appliquer des techniques pour rendre le code thread-safe

### Contenu détaillé

Le pattern builder immuable vise à combiner la fluidité et la lisibilité avec l'immuabilité.

Cette pratique permet de créer des objets dont l'état ne peut pas être modifié après leur création.

L'immuabilité élimine une classe d'erreurs liées aux états changeants.

Le pattern est particulièrement utile lors de la construction d'objets ayant de nombreuses options de configuration.

### Création d'un Builder

Pour créer un builder immuable en Java, il faut d'abord définir une classe statique interne statique nommée `Builder` dans la classe cible.

Cette classe contiendra les mêmes champs que la classe cible.

Ensuite, une méthode `build()` retourne une nouvelle instance de l'objet cible.

Elle initialisera les valeurs à partir du builder.

Chaque option de construction sera définie via une méthode qui retourne `this`, permettant le chaînage de méthodes.

### Avantages du Pattern

1. **Immuabilité et sécurité:** Une fois construit, l'objet ne peut être modifié.
2. **Lisibilité accrue:** Construction intuitive grâce aux méthodes enchaînées.
3. **Facilité d'ajout de nouvelles options:** Ajouter de nouvelles options de construction sans affecter le code existant.

## Exemple de code 1

Prenons l'exemple d'une classe `Person` utilisant un builder immuable :

```
public final class Person {  
    private final String name;  
    private final int age;  
  
    private Person(Builder builder) {  
        this.name = builder.name;  
        this.age = builder.age;  
    }  
  
    public static class Builder {  
        private String name;  
        private int age;  
  
        public Builder name(String name) {  
            this.name = name;  
            return this;  
        }  
  
        public Builder age(int age) {  
            this.age = age;  
            return this;  
        }  
  
        public Person build() {  
            return new Person(this);  
        }  
    }  
}
```

Ici, le nom et l'âge sont définis dans le builder et finalisés dans l'objet `Person`.

## Exemple de code 2

Utilisation du builder :

```
Person person = new Person.Builder()  
    .name("Alice")  
    .age(30)  
    .build();
```

Cette approche simplifie grandement le processus de création d'objets complexes, améliorant ainsi la clarté du code.

## Exercices

1. **Exercice 1 :** Créez une classe `Car` utilisant le pattern builder immuable.

Elle doit inclure au moins les champs `make`, `model`, `year`, et `isElectric`.

- Critère : Les objets `Car` doivent être immuables et permettre le chaînage des méthodes.

2. **Exercice 2 :** Modifiez la classe `Car` pour ajouter une méthode `fuelEfficiency` tout en maintenant l'immuabilité.

- Critère : Expliquez le modèle de sécurité thread-safe que vous choisissez.

## Synthèse et ouverture

Le pattern builder immuable offre une méthode à la fois robuste et flexible pour la création d'objets en Java.

**Essentiel lorsque l'immuabilité est requise, il ouvre également la voie vers des solutions thread-safe et des API lisibles. À l'avenir, vous pourriez explorer d'autres patterns de conception qui combinent l'immuabilité avec des pratiques orientées objet modernes, comme les records introduits dans les versions récentes de Java.**

# Garbage Collection tuning

## Introduction

Le tuning du Garbage Collector (GC) est essentiel pour optimiser la performance des applications Java.

Le GC gère automatiquement la mémoire, mais une configuration inadaptée peut entraîner des interruptions prolongées et des performances dégradées.

Ce module vous aidera à comprendre comment ajuster le GC pour améliorer l'efficacité et la réactivité de votre application.

## Objectifs pédagogiques

- Comprendre le fonctionnement du Garbage Collector en Java
- Analyser l'impact du GC sur la performance
- Apprendre à configurer différents types de GC pour optimiser l'application
- Évaluer et ajuster les paramètres du GC selon les besoins de l'application

## Fonctionnement du Garbage Collector

Java utilise le Garbage Collector pour automatiser la gestion de la mémoire.

Le GC libère de la mémoire en nettoyant les objets non référencés.

Le processus de GC se divise en plusieurs phases :

- **Marking** : identifier les objets accessibles
- **Deletion ou compaction** : libérer la mémoire des objets inutiles
- **Collection** : récupération de la mémoire

Il est crucial de comprendre ces phases pour ajuster le GC.

## Types de Garbage Collectors

Java propose plusieurs types de GC, chacun adapté à des scénarios spécifiques :

- **Serial GC** : Convient pour les applications à faible mémoire et d'entrée de gamme.
- **Parallel GC** : Utile dans des environnements multi-thread.
- **Concurrent Mark-Sweep (CMS) GC** : Réduit les temps de pause en effectuant le marquage de manière concurrente.

- **G1 GC (Garbage-First)** : Conçu pour une faible latence, forme un compromis entre les temps de pause et la conception concurrente.

Il est important de choisir le bon GC pour vos besoins spécifiques.

## Configuration du GC

La configuration efficace du GC nécessite l'ajustement de divers paramètres en fonction des exigences de l'application :

- **Heap Size (-Xms, -Xmx)** : Définit la taille initiale et maximale du tas.
- **Survivor Ratio (-XX:SurvivorRatio)** : Contrôle la proportion de la mémoire jeune.
- **G1 GC Options** : G1 est souvent réglé pour optimiser la latence avec des paramètres comme `-XX:MaxGCPauseMillis`.

Ces ajustements peuvent réduire les interruptions du GC et améliorer le débit.

## Exemples de Code

### Configuration de base pour Serial GC

```
java -XX:+UseSerialGC -Xms512m -Xmx1024m -jar MonApp.jar
```

**Ce paramètre utilise le Serial Garbage Collector avec une taille du tas initiale de 512 Mo et maximale de 1024 Mo.**

### Configuration pour G1 GC avec pause contrôlée

```
java -XX:+UseG1GC -Xms2g -Xmx2g -XX:MaxGCPauseMillis=200 -jar MonApp.jar
```

Ici, G1 GC est utilisé, visant à maintenir les pauses du GC sous 200 ms, avec une taille de tas fixe de 2 Go.

## Exercices Pratiques

### 1. Analyse des Logs GC :

- Exécutez une application Java avec Serial GC et analysez les logs pour identifier les phases du GC.
- Critères : Identifier les périodes de pause et l'efficacité du nettoyage.

### 2. Comparaison des Collectors :

- Configurez l'application avec G1 GC et CMS GC séparément.

Comparez les performances en termes de temps de réponse et de débit.

- Critères : Documenter les différences de performance entre les deux configurations.

### 3. Tuning en situation réelle :

- Utilisez VisualVM pour monitorer l'activité du GC en temps réel sur une application lourde.
- Critères : Apporter des ajustements pour réduire les temps de pause identifiés.

## Synthèse

Le tuning du Garbage Collector peut améliorer significativement la performance des applications Java.

Comprendre les différents types de GC et savoir configurer les paramètres clés est essentiel.

Il est recommandé de tester plusieurs configurations pour identifier la solution optimale, en tenant compte des besoins spécifiques de l'application et de ses contraintes de performance.

## Introduction à JVM Tuning

L'optimisation de la JVM (Java Virtual Machine) est cruciale pour améliorer la performance d'applications Java.

Lorsque vous ajustez les paramètres de la JVM, vous pouvez influencer de manière significative le comportement du temps d'exécution et l'utilisation des ressources.

Cela permet aux applications de fonctionner de manière plus efficace et d'améliorer l'expérience utilisateur.

## Objectifs Pédagogiques

- Identifier les paramètres de performance principaux de la JVM.
- Expérimenter les techniques de tuning de la mémoire pour réduire les temps de pause.
- Analyser l'impact des paramètres JVM sur les performances applicatives.
- Mettre en œuvre des ajustements pour optimiser la gestion du garbage collection.

## Comprendre le Fonctionnement de la JVM

La JVM est une machine virtuelle qui exécute le code Java.

Elle gère la conversion du bytecode en instructions machine, la mémoire et le garbage collection.

Le tuning de la JVM inclut l'ajustement des paramètres de mémoire comme la taille de la heap, les pools permgen/metaspaces, et la surveillance des threads pour améliorer les performances et la réactivité de l'application.

## Paramètres Clés de la JVM

Les paramètres de la JVM influencent le comportement et la performance d'une application Java.

Les plus courants incluent :

- **Xms/Xmx** : Définissent la taille initiale et maximale de la heap.
- **Xmn** : Taille de la Young Generation.
- **XX:PermSize/Metaspacesize** : Taille initiale de la zone PermGen ou Metaspaces.
- **GC Collectors** : Choix entre Serial, Parallel, CMS ou G1.

# Techniques de Tuning de la Mémoire

Optimiser l'utilisation de la mémoire est essentiel pour éviter les temps de pause.

Voici quelques techniques :

- Ajustez la taille de la heap avec `-Xms` et `-Xmx` pour éviter les re-dimensionnements fréquents.
- Utilisez le Garbage Collector G1 pour applications à faible latence.
- Surveillez les ratios Eden/Survivor dans la Young Generation pour un équilibrage optimisé.

## Exemple de Configurations JVM

Considérez cet exemple de configuration pour une application avec de faibles exigences en latence :

```
-Xms1024m -Xmx4096m -XX:+UseG1GC -XX:MaxGCPauseMillis=200
```

Ce paramétrage initialise la heap à 1GB, avec un maximum de 4GB, utilise le Garbage Collector G1, et cible des pauses de ramassage des ordures de 200ms.

## Exemples Pratiques

Voici des exemples de tuning pour divers scénarios :

### 1. Petite Application :

```
-Xms512m -Xmx1024m -XX:+UseSerialGC
```

### 2. Service de Backend avec Haut Débit :

```
-Xms2g -Xmx8g -XX:+UseParallelGC -XX:GCTimeRatio=4
```

### 3. Application à Faible Latence :

```
-Xms4g -Xmx16g -XX:+UseG1GC -XX:MaxGCPauseMillis=100
```

## Exercices Pratiques

### 1. Tuning Basique :

Configurez une JVM pour une application ayant une heap maximale de 2GB, avec attention particulière au temps de latence.

### 2. Analyse de GC :

Utilisez un outil de monitoring pour identifier les pauses de GC dans une application et proposez des ajustements.

### 3. Optimisation de la Mémoire :

Ajustez les paramètres de heap et surveillez les performances avec des workloads variés.

## Synthèse

Le tuning de la JVM est une compétence essentielle pour les développeurs souhaitant optimiser les performances de leurs applications Java.

En ajustant les paramètres de mémoire et en sélectionnant le bon Garbage Collector, vous pouvez améliorer la réactivité et l'efficacité des applications. explorons des outils de monitoring et des métriques pour un meilleur tuning à l'avenir.

# Exercice : profiler et améliorer la performance d'un code lent

## Introduction

Comprendre et optimiser les performances d'une application est crucial en programmation Java, surtout lorsqu'un code est lent.

Dans cet exercice, nous allons apprendre à "profiler" un code, identifier les goulets d'étranglement et appliquer des stratégies pour améliorer ses performances.

## Objectifs pédagogiques

- Identifier les outils de profiling disponibles en Java
- Analyser et interpréter les résultats d'un profiling
- Optimiser un code avec des techniques éprouvées
- Appliquer les bonnes pratiques de performance en Java

## Contenu détaillé

Le "profiling" est un processus qui vous aide à analyser les performances d'une application pour identifier où le code passe le plus de temps ou utilise le plus de ressources.

- **Outils de Profiling :** Plusieurs outils existent pour Java, comme JProfiler, VisualVM ou le plugin IntelliJ Profiler.
- **Analyse des Résultats :** Déterminer les parties du code qui consomment le plus de temps de CPU ou de mémoire.
- **Optimisation :** Techniques comme l'amélioration des algorithmes, la réduction de la complexité de boucle, ou l'optimisation des opérations d'I/O.
- **Bonnes Pratiques :** Utiliser des structures de données adaptées, minimiser la synchronisation dans les threads, et éviter les allocations inutiles.

## Exemples pratiques

### Exemple 1 : VisualVM

1. **Lancer VisualVM :** Téléchargez et connectez-le à votre JVM.
2. **Profiler une Application :** Parcourez les onglets CPU et Memory pour identifier les charges les plus lourdes.

### Exemple 2 : Optimisation d'une Boucle

Considérez le code suivant avant optimisation :

```

int sum = 0;
for (int i = 0; i < numbers.size(); i++) {
    if (numbers.get(i) % 2 == 0) {
        sum += numbers.get(i);
    }
}

```

Après optimisation :

```

int sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(Integer::intValue)
    .sum();

```

Ce changement utilise le Stream API pour rendre le code plus lisible et potentiellement plus performant grâce au parallélisme intégré.

## Exercices pratiques

### Exercice 1 : Utilisation de VisualVM

- **But :** Profiler un programme Java pour identifier ses points faibles.
- **Instructions :**
  - Téléchargez un outil de profiling comme VisualVM.
  - Profitez de différents scénarios et notez les observations sur les temps CPU et les consommations mémoire.

### Exercice 2 : Optimiser des Algorithmes

- **But :** Récrire un algorithme lent avec de meilleures performances.
- **Instructions :**
  - Choisissez un algorithme de type recherche ou tri.
  - Comparez initialement le temps d'exécution.
  - Optimisez en améliorant l'algorithme, puis comparez les résultats.

## Synthèse

Savoir profiler et améliorer la performance d'un code est une compétence clé pour tout développeur Java.

De l'utilisation d'outils de profiling aux techniques d'optimisation, chaque étape contribue à créer des applications plus efficaces. À l'avenir, explorez davantage d'outils et de techniques pour perfectionner ces compétences, et surveillez les mises à jour autour des JVM et JDK pour rester à jour avec les dernières innovations en matière d'optimisation.

## Introduction

La programmation Java avancée intègre de nouvelles fonctionnalités et des bonnes pratiques essentielles pour créer des applications robustes et performantes.

Parmi les principaux axes, nous explorerons divers concepts modernes tels que les records avancés, les interfaces scellées et l'immutabilité.

Comprendre ces notions est crucial pour manipuler efficacement le langage.

# Vue d'ensemble des sous-notions

1. **Record avancé** - Simplifie la définition de classes immuables.
2. **Sealed interface** - Restriction des implémentations possibles.
3. **Immutabilité** - Garantit l'état constant d'un objet.
4. **Thread-safety** - Assure la sécurité des opérations concurrentes.
5. **Pattern builder immutable** - Aide à construire des objets complexes.
6. **Garbage Collection tuning** - Optimise la gestion de la mémoire.
7. **JVM tuning** - Améliore les performances globales de l'application.
8. **Exercice pratique** - Profiler et améliorer la performance d'un code lent.

Ces éléments se complètent pour favoriser des bonnes pratiques de développement en Java.

## Introduction

Le découpage en modules indépendants est une approche clé dans la programmation Java moderne.

Elle permet de structurer le code en unités bien définies, chacune ayant une responsabilité claire.

Ce modèle architectural améliore la maintenabilité, facilite le développement parallèle et renforce la réutilisabilité du code.

## Objectifs pédagogiques

- Comprendre le concept de module en Java.
- Apprendre à structurer une application avec des modules indépendants.
- Savoir déclarer et utiliser des modules à l'aide de `module-info.java`.
- Évaluer les avantages du découpage modulaire pour la maintenance et l'extensibilité.

## Concepts de Modules

En Java, un module est une collection de packages avec un fichier de déclaration `module-info.java` à sa racine.

Il définit de manière explicite les dépendances vers d'autres modules et expose les packages exploitables par d'autres modules.

Ce paradigme vise à encapsuler le code et réduire les effets de bord.

## Créer un Module

Pour créer un module :

1. Créez un dossier pour votre module.
2. Placez-y vos packages et classes Java.
3. Ajoutez un fichier `module-info.java` où vous déclarez le nom du module et ses dépendances.

```
module com.example.module {  
    exports com.example.module.api;  
    requires another.module;  
}
```

## Avantages du Découpage Modulaire

- **Maintenance Facilités** : Chaque module pouvant être testé et mis à jour indépendamment.
- **Réutilisation** : Modules bien définis et documentés sont réutilisables dans divers projets.
- **Encapsulation** : Cachant les détails d'implémentation, encourageant une interface bien définie.

## Exemple d'implémentation

Considérons un projet d'application e-commerce structuré en trois modules :

- **module core** : Gère la logique métier.
- **module api** : Expose des services via API.
- **module service** : Implémente des fonctionnalités particulières comme le paiement.

## Exemple de `module-info.java`

Pour le module `api` :

```
module com.ecommerce.api {  
    exports com.ecommerce.api.services;  
    requires com.ecommerce.core;  
}
```

Ceci montre comment exposer certains services internes tout en limitant les dépendances.

## Exercice Pratique

1. **Créez un module core gérant les produits.**
  - Inclure des classes pour les produits avec des méthodes CRUD.
2. **Ajoutez un module service pour la gestion de la commande.**
  - Implémentez des services pour le traitement des commandes.
3. **Intégrez un module api pour exposer en REST.**
  - Utilisez un framework léger pour mettre à disposition POJOs comme ressources.

**Critères d'évaluation :** Clarté du découpage modulaire, utilisation appropriée des directives `exports` et `requires`.

## Synthèse

Le découpage en modules indépendants offre une organisation claire et efficace du code Java, favorisant la réutilisation et la maintenance.

La structure explicite fournie par `module-info.java` assure une interface bien définie entre les différentes parties de l'application. À l'avenir, explorez des outils et pratiques modernes comme JPMS pour tirer le meilleur parti de l'architecture modulaire.

## Introduction

Dans cette section sur le "Module core", nous allons explorer un des éléments fondamentaux de l'architecture modulaire en Java.

Le module core est crucial puisqu'il comprend les composants principaux qui assurent le fonctionnement de base de toute application modulaire Java.

Sa bonne définition et sa structuration sont essentielles pour garantir la modularité et la scalabilité du système.

## Objectifs pédagogiques

- Comprendre le rôle et l'importance du module core dans une architecture modulaire Java.
- Apprendre à structurer efficacement un module core.
- Savoir implémenter et configurer le fichier `module-info.java` pour le module core.
- Identifier et gérer les dépendances internes et externes du module core.

## Contenu détaillé

Le module core contient les éléments de base ou de noyau de votre application.

Il définit :

- Les classes principales qui assurent les fonctionnalités essentielles.
- Les interfaces qui établissent des contrats pour d'autres modules.
- Les méthodes utilitaires utilisées à travers différents modules.

Pour structurer un module core :

- Identifiez les classes et interfaces vraiment cruciales.
- Évitez une surcharge de fonctionnalités : le module core doit rester focalisé.
- Pensez à la maintenabilité et à l'évolutivité dès la conception.

## Fichier `module-info.java`

Chaque module Java doit contenir un fichier `module-info.java`.

Pour un module core, il spécifie les exports nécessaires :

- Modifiez `module-info.java` pour ne divulguer que les packages essentiels.
- Utilisez l'instruction `exports` pour chaque package à rendre accessible.
- Utilisez l'instruction `requires` pour indiquer les dépendances vers d'autres modules.

Exemple :

```
module com.example.core {  
    exports com.example.core.services;  
    requires java.logging;  
}
```

# Exemples de code

## Exemple 1 : Structure de base

```
module com.myapp.core {  
    exports com.myapp.core.utility;  
    requires java.sql;  
}
```

Cette structure permet l'exportation du package `com.myapp.core.utility` tout en demandant l'accès à l'API Java SQL.

## Exemple 2 : Interface et Classe

```
// Interface in module core  
package com.myapp.core.services;  
public interface CoreService {  
    void execute();  
}  
  
// Class implementing the interface  
package com.myapp.core.impl;  
public class CoreServiceImpl implements CoreService {  
    public void execute() {  
        // Implementation details  
    }  
}
```

# Exercices pratiques

## Exercice 1 : Crédit d'un Module core

1. Créez un package `com.example.core.base`.
2. Ajoutez une classe `MainService` avec une méthode `run()`.
3. Configurez `module-info.java` pour exporter ce package.
4. Testez l'accès à votre module depuis un autre module.

## Critères d'évaluation

- La `module-info.java` doit être correctement configurée.
- La classe `MainService` doit être accessible depuis un autre module.
- Les fonctionnalités de base doivent être clairement définies et documentées.

# Synthèse

Le module core constitue la base de votre application modulaire Java.

Une structure bien pensée facilite l'évolutivité et l'intégration de nouveaux modules.

Nous avons appris à configurer le fichier `module-info.java` et à gérer efficacement les dépendances.

Poursuivre avec les autres types de modules, tels que les modules service et API, enrichira notre compréhension globale de l'architecture modulaire Java.

## Introduction

Dans le cadre d'une architecture modulaire en Java, le **Module Service** joue un rôle crucial.

Il fournit des fonctionnalités spécifiques pouvant être utilisées par d'autres modules.

Ce module peut intégrer des services grâce à l'utilisation de `ServiceLoader`, permettant ainsi une extensibilité et une flexibilité accrues.

Nous allons explorer comment concevoir, implémenter et utiliser efficacement un module de service.

## Objectifs pédagogiques

- Comprendre le rôle du module service dans une architecture modulaire.
- Apprendre à définir et déclarer un module service en Java.
- Découvrir comment utiliser `ServiceLoader` pour charger dynamiquement des services.
- Mettre en œuvre un module service fonctionnel avec des exemples pratiques.

## Architecture modulaire en Java

Une architecture modulaire divise une application Java en plusieurs modules distincts, chacun encapsulant un ensemble de fonctionnalités.

Le module service s'intègre typiquement dans ce cadre pour offrir des services réutilisables et évolutifs.

Cela encourage une meilleure organisation du code, facilite la maintenance et améliore la portée des applications Java.

## Définition d'un Module Service

Un module service en Java est essentiellement un conteneur de logique métier ou de fonctionnalités spécifiques.

Un module est défini par un fichier `module-info.java`, où il expose des interfaces ou des services qu'il fournit aux autres modules.

Les autres modules peuvent alors consommer ces services sans dépendre des détails d'implémentation.

## Utilisation de ServiceLoader

`ServiceLoader` est un utilitaire Java qui permet de découvrir et de charger des services dynamiquement à partir d'un module service.

Cela réduit le couplage entre les modules consommateurs et les modules fournisseurs de services.

Un module service définit les services qu'il exporte, et `ServiceLoader` se charge de charger ces services à la demande.

# Exemple de Module Service

## 1. Définir une Interface de Service

Créez une interface dans votre module service.

Par exemple :

```
public interface PaymentService {  
    void processPayment(double amount);  
}
```

## 2. Implémenter le Service

Créez des classes qui implémentent votre interface :

```
public class PayPalService implements PaymentService {  
    public void processPayment(double amount) {  
        // Logic to process payment via PayPal  
    }  
}
```

## 3. Déclaration dans module-info.java

Exportez le service dans votre module :

```
module com.example.payment {  
    exports com.example.payment;  
    provides com.example.payment.PaymentService with com.example.payment.PayPalService;  
}
```

# Exercices pratiques

## 1. Crédit d'un Module Service

Créez un module service qui propose une fonctionnalité d'authentification.

Définissez une interface `AuthenticationService` et implémentez deux versions : `BasicAuthService` et `TokenAuthService`.

## 2. Utilisation de ServiceLoader

Écrivez un petit programme Java qui utilise `ServiceLoader` pour charger et utiliser dynamiquement les implémentations de `AuthenticationService` définies dans l'exercice précédent.

## 3. Evaluation

Vérifiez que votre programme peut passer d'une implémentation de service à une autre sans modification du code de l'application principale.

# Synthèse

Le module service est une composante essentielle de l'architecture modulaire Java, permettant une organisation flexible et évolutive du code.

En utilisant `ServiceLoader`, nous pouvons charger et modifier les services de manière dynamique, offrant ainsi une extensibilité accrue.

La maîtrise de ces concepts renforce la robustesse et la maintenabilité des applications Java modernes. À l'avenir, l'exploration du couplage avec des outils tels que Spring pourrait enrichir davantage cette approche modulaire.

# Introduction

Le Module API en Java est crucial pour structurer et organiser les interactions entre différents modules.

Ce concept, introduit avec Java 9, améliore la modularité et maintenabilité des projets.

Apprendre à concevoir des API modulaires permet de développer des applications flexibles et évolutives.

## Objectifs pédagogiques

- Comprendre le concept du Module API en Java.
- Savoir comment définir et utiliser les API dans une architecture modulaire.
- Maîtriser l'exportation des packages au sein d'un module.
- Être capable de concevoir une application modulaire incluant des APIs.

## Définition du Module API

Un Module API en Java est un module dont l'objectif est de définir et exposer une interface publique que d'autres modules peuvent consommer.

Contrairement aux modules internes, les modules API se concentrent sur l'interaction inter-modules en exposant uniquement les fonctionnalités nécessaires.

- Les API sont définies via le fichier `module-info.java`.
- Les modules API permettent de masquer l'implémentation tout en exposant les services nécessaires à d'autres modules.

## Utilisation du `module-info.java`

Le fichier `module-info.java` est le point d'entrée de la modularité en Java.

Pour une API, il déclare quels packages sont exportés :

```
module com.example.api {  
    exports com.example.api.service;  
}
```

- Le mot-clé `exports` indique que le package est accessible aux autres modules.
- Les modules utilisant cette API devront la déclarer dans leur propre `module-info.java`.

## Avantages des Modules API

Les modules API offrent de nombreux avantages, notamment :

- **Encapsulation** : seuls les packages nécessaires sont exposés, le reste est caché.
- **Réutilisabilité** : une API bien conçue peut être utilisée par plusieurs modules offrant une flexibilité accrue.
- **Séparation des préoccupations** : améliore la clarté et la maintenabilité du projet.

# Exemple d'API Modulaire

Considérons un module API fournissant des services de paiement :

## module-info.java

```
module payment.api {  
    exports com.payment.api.service;  
}
```

## Classe Service

```
package com.payment.api.service;  
public interface PaymentService {  
    void processPayment(double amount);  
}
```

Cet exemple montre comment définir une API qui expose un service de paiement.

## Exercices pratiques

1. **Création d'un Module API** : Créez un module API pour un service de notification.

Définissez une interface exposant un service d'envoi de messages.

- **Critères** : Utilisation de `module-info.java` pour exposer votre service.

2. **Consommation d'une API** : Développez un second module consommant l'API de notification créée.

Implementez une classe réalisant l'interface définie dans l'API.

3. **Test de l'application** : Intégrez et testez votre application modulaire.

Assurez-vous que le module consommateur peut accéder au service du module API.

## Synthèse et ouverture

La gestion des API en architecture modulaire est une compétence essentielle pour concevoir des applications complexes.

En appliquant les principes de modularité et d'encapsulation, les développeurs peuvent créer des systèmes évolutifs et résilients.

La prochaine étape pourrait explorer l'intégration de services dynamiques avec le ServiceLoader pour augmenter la flexibilité des API.

## Introduction

Dans ce module, nous allons explorer le concept du `ServiceLoader` en Java.

Le `ServiceLoader` facilite la découverte et le chargement dynamiques de services implémentés à l'aide du mécanisme de service SPI (Service Provider Interface) en Java.

L'usage judicieux du `ServiceLoader` permet de construire des applications modulaires où les dépendances entre composants peuvent être résolues dynamiquement à l'exécution, ajoutant ainsi flexibilité et extensibilité à votre code.

## Objectifs pédagogiques

- Comprendre le concept de SPI en Java.
- Utiliser le `ServiceLoader` pour découvrir et charger des services.
- Intégrer le `ServiceLoader` dans une architecture modulaire Java.
- Résoudre des problèmes de dépendances dynamiques à l'aide de `ServiceLoader`.

## Utilisation du ServiceLoader

Le `ServiceLoader` est une classe centrale pour implémenter le pattern de l'interface fournisseur de services en Java.

Ce modèle repose sur l'enregistrement des implémentations de services, qui peuvent être découverts et chargés à l'exécution sans modifier le code source consommateur.

1. **Définition de l'interface de service** : Cette interface spécifie le contrat que toutes les implémentations doivent suivre.
2. **Création des implémentations** : Les classes spécifiques qui implémentent cette interface fournissent le comportement réel.
3. **Déclaration du fournisseur de service** : Un fichier de configuration placé dans `META-INF/services` pour indiquer quelles classes implémentent l'interface fournie.

## Déclaration du ServiceLoader

Pour utiliser `ServiceLoader`, vous devez créer un fichier dans `META-INF/services` nommé d'après l'interface de votre service.

Ce fichier contient le nom des classes qui fournissent l'implémentation de l'interface du service.

Cela permet à `ServiceLoader` de découvrir automatiquement les services à l'exécution.

- **Exemple de fichier** : `META-INF/services/com.example.MyService`
  - Contenu :

```
com.example.MyServiceImpl1  
com.example.MyServiceImpl2
```

## Exemples de Code avec ServiceLoader

Observons comment intégrer le `ServiceLoader` dans un projet Java.

Voici un exemple de base démontrant l'utilisation du `ServiceLoader` :

```
public interface MyService {  
    void execute();  
}
```

Implementation Example:

```

public class MyServiceImpl implements MyService {
    public void execute() {
        System.out.println("Service Execution");
    }
}

```

Chargement du service :

```

ServiceLoader<MyService> loader = ServiceLoader.load(MyService.class);
for (MyService service : loader) {
    service.execute();
}

```

Dans cet exemple, `ServiceLoader` est utilisé pour charger toutes les implémentations de `MyService` enregistrées dans le fichier `META-INF/services`.

## Exercices Pratiques

### 1. Exercice 1 : Création de service

- Définissez une interface de service.
- Implémentez au moins deux classes fournissant ce service.
- Utilisez `ServiceLoader` pour les charger et exécuter une méthode de service.

### 2. Exercice 2 : Extension dynamique

- Ajoutez une nouvelle implémentation de service sans modifier le code existant du client.
- Testez que `ServiceLoader` détecte et charge automatiquement cette nouvelle implémentation.

### 3. Exercice 3 : Projet modulaire

- Créez un projet Java modulaire en utilisant `module-info.java`.
- Assurez-vous que votre module héberge votre interface de service et utilise `ServiceLoader`.

Critères d'évaluation : Capacité à intégrer et étendre dynamiquement les services sans modification directe du code source du client.

## Synthèse

Le `ServiceLoader` est un outil puissant dans l'arsenal de Java pour la construction d'applications modulaires et extensibles.

Il permet la découverte dynamique et la gestion de services en utilisant le pattern SPI, simplifiant ainsi la gestion des dépendances et améliorant la flexibilité.

Cette approche est particulièrement utile pour développer des systèmes qui nécessitent extensibilité et adaptabilité, tels que les plug-ins ou les frameworks.

En maîtrisant `ServiceLoader`, vous êtes bien préparés à tirer parti des fonctionnalités avancées de l'architecture modulaire de Java.

## Introduction

L'injection de dépendances est une pratique essentielle en programmation modulaire qui permet de créer des applications flexibles et maintenables.

Le `ServiceLoader` en Java facilite cette approche en permettant de dynamiquement découvrir et instancier des services définis dans des modules distincts.

Grâce au `ServiceLoader`, vous pouvez remplacer des implémentations à la volée sans modifier le code client, améliorant ainsi l'extensibilité et la modularité de votre application.

# Objectifs pédagogiques

- Comprendre le concept de l'injection de dépendances.
- Appliquer le mécanisme de `ServiceLoader` en Java.
- Découvrir comment définir et consommer des services modulaires.
- Pratiquer l'implémentation de services interchangeables.

## Contexte et concepts clés

L'injection de dépendances permet aux objets de déclarer leurs dépendances plutôt que de les créer.

En Java, le `ServiceLoader` est une classe utilitaire qui recherche des implémentations de services déclarés dans des fichiers spécifiques sous `META-INF/services`.

- **Services** : Interfaces ou classes abstraites définissant certaines fonctionnalités.
- **Providers** : Implémentations concrètes de ces services.
- **ServiceLoader** : Moteur de chargement et d'instanciation des services.

## Utiliser ServiceLoader

Java utilise le fichier `module-info.java` pour déclarer les services fournis et consommés.

1. **Déclaration du service** : Définissez une interface ou une classe abstraite.
2. **Implémentation du provider** : Fournissez une ou plusieurs classes implémentant le service.
3. **Configuration** : Créez un fichier nommé après votre interface dans `META-INF/services`, listant les implémentations.

## Exemples de code

### Déclaration de service :

```
public interface PaymentService {  
    void processPayment(double amount);  
}
```

### Provider :

```
public class CreditCardPayment implements PaymentService {  
    public void processPayment(double amount) {  
        // Traitement du paiement par carte de crédit  
    }  
}
```

### Module configuration :

Dans `module-info.java` :

```
module payment.module {  
    provides PaymentService with CreditCardPayment;  
}
```

## Chargement des services

Pour charger et utiliser les services, vous pouvez utiliser le `ServiceLoader` dans votre code client :

```
ServiceLoader<PaymentService> loader = ServiceLoader.load(PaymentService.class);  
for (PaymentService service : loader) {  
    service.processPayment(100.00);  
}
```

Cela parcourra toutes les implémentations disponibles, en traitant un paiement de 100.00 pour chaque implémentation.

## Exercices pratiques

1. **Création d'un service** : Définissez un service `EmailService` avec des méthodes d'envoi de mails simples.
2. **Implémentation de providers** : Créez deux implémentations, l'une pour envoyer des mails via SMTP et l'autre via un service web.
3. **Utilisation de ServiceLoader** : Chargez et testez vos services dans une application console. Évaluez quelle implémentation est utilisée.

Critères d'évaluation : Le programme doit être modulaire, avec le bon découpage en modules, et permettre d'interchanger les implémentations sans modifier le code d'appel.

## Synthèse et ouverture

L'injection de dépendances via `ServiceLoader` en Java est un puissant moyen d'améliorer la modularité de vos applications.

Cette approche facilite la gestion des implémentations de services sans modification directe du code, ce qui est essentiel pour les systèmes évolutifs.

En explorant des technologies comme CDI (Contexts and Dependency Injection) pour des besoins plus complexes, vous pouvez étendre les capacités de l'injection de dépendances dans vos projets Java.

## Introduction

Dans cette section, vous découvrirez comment créer un projet Java modulaire complet.

L'architecture modulaire permet de structurer un programme en unités indépendantes appelées modules, facilitant la maintenance et l'évolution du code.

Nous aborderons chaque étape de cette création, de la conception à l'implémentation.

## Objectifs pédagogiques

- Concevoir un projet Java en modules distincts
- Créer et configurer des fichiers `module-info.java`
- Utiliser `ServiceLoader` pour l'injection de dépendances

- Planter un projet modulaire complet et fonctionnel

## Conception modulaire

Le premier pas dans la création d'un projet modulaire est de définir l'architecture souhaitée.

Un projet modulaire se compose généralement de plusieurs modules, chacun remplissant un rôle spécifique et communiquant avec les autres via des interfaces bien définies.

- **Module Core** : Contient la logique centrale et les fonctionnalités de base.
- **Module API** : Définit les interfaces que les autres modules implémenteront.
- **Module Service** : Fournit les implémentations spécifiques des interfaces définies dans le module API.

## Configuration des Modules

Chaque module inclut un fichier `module-info.java` qui déclare les dépendances :

- `module com.example.core { requires com.example.api; }`
- `module com.example.api { exports com.example.api.interfaces; }`
- `module com.example.service { requires com.example.api; provides com.example.api.interfaces.Service with com.example.service.Service; }`

Cette configuration permet de contrôler l'accès entre les modules et de partager les API nécessaires.

## Utilisation de ServiceLoader

ServiceLoader permet de découvrir et d'utiliser des implémentations fournies par des modules de services dynamiques.

Ceci peut être configuré dans le module API, qui expose des interfaces, et dans les modules de services qui fournissent les implémentations.

- **Declaring a Service:** Dans `module-info.java`, utilisez `provides` et `with` pour lier l'interface aux implémentations.
- **Consommer un Service:** Utilisez `ServiceLoader.load(Service.class)` pour obtenir une instance du service.

## Exemples de Code

Voici une illustration simple de configuration modulaire :

```
// module-info.java in API module
module com.example.api {
    exports com.example.api.interfaces;
}

// Service interface
package com.example.api.interfaces;
public interface GreetingService {
    String greet(String name);
}
```

```

// module-info.java in Service module
module com.example.service {
    requires com.example.api;
    provides com.example.api.interfaces.GreetingService with com.example.service.BasicGreetingService;
}

// Service implementation
package com.example.service;
import com.example.api.interfaces.GreetingService;
public class BasicGreetingService implements GreetingService {
    public String greet(String name) {
        return "Hello, " + name + "!";
    }
}

```

## Exercice Progressif

1. **Conception Modulaires** : Créez une architecture de base avec au moins trois modules : Core, API, et Service.

2. **Implémentation des Modules** : Ajoutez des classes et interfaces de base dans chaque module.

Assurez-vous que les services peuvent être découverts et utilisés via `ServiceLoader`.

3. **Tests de Fonctionnalité** : Implémentez un cas d'utilisation simple où le module Core utilise un service du module Service via l'API.

## Synthèse et Ouverture

Vous avez maintenant une vue d'ensemble sur la création d'un projet Java modulaire complet, en partant de la conception des modules jusqu'à leur intégration à l'aide de `ServiceLoader`.

Ce modèle modulaire est essentiel pour la création d'applications Java évolutives et maintenables.

Vous pouvez explorer des projets plus complexes et considérer des outils comme la gestion de version et le CI/CD pour aller plus loin.

## Introduction à l'Architecture modulaire

L'architecture modulaire en Java est une approche qui permet de structurer les applications en unités fonctionnelles distinctes appelées modules.

Cela favorise la clarté du code, la réutilisation et la facilité de maintenance.

Ce cours couvre les éléments essentiels de l'architecture modulaire, notamment :

- `module-info.java`: Déclaration de modules.
- Découpage en modules indépendants.
- Modules `core`, `service`, et `API`.
- Utilisation de `ServiceLoader`.
- Injection de dépendances via `ServiceLoader`.
- Exercice pratique sur la création d'un projet modulaire complet.

## Vue d'ensemble de l'Architecture modulaire

Le découpage en modules permet de séparer les préoccupations fonctionnelles et d'améliorer la modularité du code.

Le fichier `module-info.java` est central pour la déclaration, et chaque module a un rôle spécifique : le module `core` contient le cœur de l'application, les modules `service` offrent des fonctionnalités additionnelles, et le module `API` définit les interfaces.

L'utilisation de `ServiceLoader` et l'injection de dépendances facilitent l'extensibilité.

Enfin, un exercice pratique consolidera les acquis sur la création d'un projet modulaire complet.

## Introduction à Java Avancé

La programmation Java avancée couvre des concepts clés pour développer des applications robustes et performantes.

Ce cours se concentrera sur trois grands thèmes :

- **Généricité et collections** : Apprenez à utiliser les types génériques et les collections de manière efficiente pour créer des applications flexibles.
- **Fonctionnalités modernes et bonnes pratiques** : Découvrez les nouvelles fonctionnalités de Java et les méthodes pour améliorer la sécurité et les performances du code.
- **Architecture modulaire** : Comprenez comment structurer un projet Java en modules pour une meilleure maintenabilité.

Chaque sous-notion sera approfondie dans les sections suivantes.

## Vue d'ensemble des Sous-notions

La programmation Java avancée s'articule autour de trois axes :

- **Généricité et collections** : Axé sur l'utilisation de types génériques et d'outils de collection pour gérer efficacement des données.
- **Fonctionnalités modernes et bonnes pratiques** : Couvre des concepts comme les records avancés et l'immutabilité pour créer un code durable.
- **Architecture modulaire** : Se concentre sur l'organisation de projets en modules indépendants et intégration avec `ServiceLoader`.

Ces approches visent à développer des compétences essentielles en Java avancé.

## CompletableFuture

### Introduction

Le `CompletableFuture` en Java est une classe puissante qui simplifie la gestion des tâches asynchrones.

Introduite avec Java 8, elle permet de construire des chaînes de calculs asynchrones de manière fluide et lisible.

L'usage du `CompletableFuture` favorise l'exécution concurrente moderne et rend le code plus réactif et adaptable aux tâches exécutées en arrière-plan sans bloquer le fil d'exécution principal.

## Objectifs Pédagogiques

- Comprendre le concept de programmation asynchrone avec `CompletableFuture`.
- Apprendre à créer et gérer des tâches asynchrones.

- Maîtriser la composition et la combinaison de CompletableFutures.
- Implémenter des traitements asynchrones réactifs en Java.

## Contenu Détailé (1)

### Comprendre CompletableFuture

Un CompletableFuture représente une opération asynchrone qui peut être complétée à une date ultérieure.

Contrairement aux Future, CompletableFuture fournit une API plus riche qui permet de chaîner facilement plusieurs opérations asynchrones.

- **Création d'un CompletableFuture:** peut être fait via des méthodes statiques telles que supplyAsync() pour lancer des tâches asynchrones.
- **Chaînage de tâches:** Permet de chaîner plusieurs opérations grâce à des méthodes comme thenApply, thenAccept, et thenRun .

## Contenu Détailé (2)

### Gestion de l'asynchronisme

- **Compléter manuellement:** Utilisez la méthode complete() pour définir manuellement la valeur du futur.
- **Gestion des exceptions:** CompletableFuture fournit exceptionally(), handle(), et whenComplete() pour gérer les exceptions pendant l'exécution des tâches.

Exemple de code :

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) throw new RuntimeException("Oups!");
    return "Succès";
});

future.exceptionally(ex -> "Erreur: " + ex.getMessage())
    .thenAccept(System.out::println);
```

## Exemples de Code (1)

### Exemple de création et chaînage

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 5)
    .thenApply(n -> n * n)
    .thenApply(String::valueOf);

future.thenAccept(System.out::println); // Affiche "25"
```

Ce code démontre comment créer un CompletableFuture qui calcule le carré de 5 et convertit le résultat en chaîne de caractères.

## Exemples de Code (2)

### Gestion des tâches asynchrones

```
CompletableFuture<Void> futureTask =  
    CompletableFuture.runAsync(() -> {  
        try {  
            Thread.sleep(1000);  
            System.out.println("Tâche Asynchrone Complétée!");  
        } catch (InterruptedException e) {  
            throw new IllegalStateException(e);  
        }  
    });  
  
futureTask.join(); // Attend la fin de la tâche
```

Cet exemple lance une tâche qui dort pendant une seconde avant d'afficher un message, illustrant comment gérer l'exécution asynchrone.

### Exercices Pratiques

1. **Exercice 1:** Créez un `CompletableFuture` qui télécharge le contenu d'une URL en arrière-plan et traite la réponse.
  - Critères : Le code doit gérer les exceptions correctement et afficher le contenu après téléchargement.
2. **Exercice 2:** Utilisez `CompletableFuture` pour lire des tâches depuis une liste, les exécuter en parallèle, et calculer la somme des résultats.
  - Critères : Assurez-vous que le code est non-bloquant et bien synchronisé.

### Synthèse

`CompletableFuture` est un outil essentiel pour gérer la concurrence de façon moderne en Java.

En facilitant la création et la coordination des tâches asynchrones, il contribue à rendre le code plus flexible et réactif.

Maîtriser ses fonctionnalités permet de créer des applications capables de gérer efficacement les opérations en coulisse.

Les prochaines étapes pourraient inclure l'intégration avec les flux réactifs pour des systèmes encore plus dynamiques.

## Introduction

Les chaînes d'exécution asynchrone en Java permettent de gérer des tâches de manière non bloquante, optimisant ainsi l'utilisation des ressources et améliorant la réactivité des applications.

En se libérant des contraintes des traitements synchrones, ou synchronisés, les développeurs peuvent concevoir des applications réactives, capables de gérer plusieurs opérations simultanément sans interférences.

### Objectifs pédagogiques

- Comprendre le concept d'exécution asynchrone.
- Mettre en œuvre des chaînes d'exécution asynchrones en Java.
- Analyser et débugger des flux asynchrones.

- Construire des applications réactives.

## Détails de l'exécution asynchrone

L'exécution asynchrone repose sur la capacité à lancer des tâches sans attendre leur achèvement immédiat pour passer à d'autres opérations.

En Java, cela s'effectue principalement avec l'API `CompletableFuture`, qui permet de composer des actions qui s'exécuteront de manière non bloquante.

Cette approche est particulièrement efficace pour les opérations d'I/O ou les calculs longs, où une attente active serait inefficace.

## Complétion et gestion d'erreurs

Avec `CompletableFuture`, on peut chaîner plusieurs opérations asynchrones et définir des actions en cas de réussite (`thenApply`, `thenAccept`) ou d'échec (`exceptionally`).

Ceci permet une gestion fine et réactive des erreurs, de même qu'une optimisation du flux de travail.

Cela augmente aussi la résilience des applications face aux erreurs non prévues.

## Composition de tâches

`CompletableFuture` permet de combiner différentes futures, soit en les exécutant indépendamment (`allOf`), soit en attendant la fin de l'une d'entre elles (`anyOf`).

Cette flexibilité est cruciale pour répondre à des cas d'usage où la nature des opérations dépend de la disponibilité des ressources ou de la réponse des services externes.

## Exemple 1 : Traitement simple

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    return "Hello, World!";
});

future.thenAccept(result -> System.out.println(result));
```

Ce code initialise une tâche qui retourne le message "Hello, World!" et l'affiche une fois complétée, sans bloquer le fil principal.

## Exemple 2 : Gestion des erreurs

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (true) throw new RuntimeException("Erreur !");
    return "Bonjour";
}).exceptionally(ex -> "Erreur capturée : " + ex.getMessage());

future.thenAccept(System.out::println);
```

Ici, une exception est intentionnellement lancée pour montrer comment `exceptionally` peut capturer et traiter l'erreur.

## Exercice 1 : Invoquer des APIs

Créez un programme qui effectue des appels à plusieurs services web en parallèle.

Utilisez `CompletableFuture` pour récupérer et afficher les résultats dès qu'ils sont disponibles.

Critères d'évaluation :

- Bon usage de `CompletableFuture`.
- Gestion des erreurs.
- Lisibilité du code.

## Exercice 2 : Pipeline de calcul

Implémentez un pipeline qui enchaîne plusieurs opérations de transformations de données (ex. : conversion de chaînes, calculs numériques) de manière asynchrone.

Assurez-vous que chaque étape dépend du résultat de la précédente.

Critères d'évaluation :

- Bon enchaînement des tâches.
- Performance (mesurer le temps d'exécution).
- Gestion des erreurs et des exceptions.

## Synthèse

Les chaînes d'exécution asynchrone en Java, via `CompletableFuture`, offrent une puissante façon de construire des applications réactives, capables d'opérer sous de lourdes charges sans sacrifier la performance.

La maîtrise des flux asynchrones permet de créer des logiciels plus robustes, réactifs et adaptés aux environnements modernes, où la conciliation efficacité et réactivité est critique.

Explorons ensuite les paradigmes réactifs complets pour enrichir davantage vos applications.

# Introduction

La classe `ForkJoinPool` en Java est un cadre puissant pour le traitement parallèle.

Utilisée pour diviser et régner sur les tâches, elle simplifie la programmation concurrente.

Comprendre `ForkJoinPool` permet de mieux exploiter les architectures multi-cœurs modernes, optimisant la performance des applications.

## Objectifs pédagogiques

- Comprendre le fonctionnement de `ForkJoinPool`.
- Apprendre à diviser et fusionner des tâches concurrentes.
- Utiliser efficacement `ForkJoinPool` pour améliorer les performances.
- Évaluer et optimiser l'utilisation de `ForkJoinPool` dans des applications Java.

## Fonctionnement de ForkJoinPool

`ForkJoinPool` repose sur le principe du *divide and conquer*.

Les tâches sont divisées en sous-tâches (`fork`), gérées individuellement, puis leurs résultats sont combinés (`join`).

Ce modèle exploite les threads de façon dynamique, allouant et exécutant efficacement des tâches indépendantes.

## Structure de ForkJoinPool

Une `ForkJoinPool` est constituée de plusieurs workers threads.

Ces threads gèrent les tâches déposées dans une queue de tâches.

Contrairement à `ExecutorService`, `ForkJoinPool` optimise l'utilisation CPU en adaptant automatiquement le nombre de threads actifs.

## Utilisation Pratique

Pour utiliser `ForkJoinPool`, créez des tâches en étendant `RecursiveTask` ou `RecursiveAction`. `RecursiveTask` est utilisé lorsque la tâche renvoie un résultat, tandis que `RecursiveAction` est pour une tâche ne renvoyant rien.

Implémentez la méthode `compute` pour définir le comportement.

# Exemple : Calcul de la somme

```
import java.util.concurrent.RecursiveTask;

public class ForkJoinSum extends RecursiveTask<Integer> {
    private final int[] array;
    private final int start, end;

    public ForkJoinSum(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= 10) { // seuil
            int sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            int mid = (start + end) / 2;
            ForkJoinSum leftTask = new ForkJoinSum(array, start, mid);
            ForkJoinSum rightTask = new ForkJoinSum(array, mid, end);
            leftTask.fork(); // exécute en parallèle
            return rightTask.compute() + leftTask.join();
        }
    }
}
```

# Exemple : Utilisation

```
import java.util.concurrent.ForkJoinPool;

public class Main {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
        ForkJoinSum task = new ForkJoinSum(array, 0, array.length);
        int result = pool.invoke(task);
        System.out.println("Sum: " + result); // Affiche la somme totale
    }
}
```

# Exercices Pratiques

- 1. Basic Implementation:** Créez une classe `ForkJoinMax` pour trouver la valeur maximale d'un tableau d'entiers.
- 2. Threshold Experimentation:** Modifiez la taille de seuil de division pour observer les impacts sur les performances.
- 3. Complex Task:** Implémentez une tâche qui combine plusieurs calculs (moyenne, min, max) et analysez l'utilisation des ressources.

# Synthèse et Ouverture

`ForkJoinPool` est une technologie qui module efficacement la charge de calcul en utilisant tous les processeurs disponibles.

En maîtrisant cet outil, les développeurs peuvent concevoir des applications Java plus réactives et performantes.

Pour aller plus loin, explorez les optimisations et comparaisons avec d'autres modèles de concurrence en Java.

Il semble que vous avez envoyé un message vide.

Si vous souhaitez que je développe la partie sur `ExecutorService` en Java, veuillez confirmer, et je vais créer un ensemble de slides pédagogiques pour cette notion finale.

## Introduction à `synchronized`

La programmation concurrente en Java nécessite un contrôle précis des accès aux ressources partagées.

Le mot-clé `synchronized` est essentiel pour assurer l'exclusivité d'accès aux sections critiques de votre code.

En garantissant qu'un seul fil d'exécution accède à une ressource à la fois, il aide à éviter les problèmes tels que les conditions de course, assurant ainsi la cohérence des données.

## Objectifs pédagogiques

- Comprendre le rôle de `synchronized` dans la gestion de la concurrence.
- Être capable d'identifier et de protéger les sections critiques du code.
- Appliquer `synchronized` pour éviter les conditions de course.
- Évaluer les impacts de `synchronized` sur la performance de l'application.

## Concepts de base de `synchronized`

En Java, une méthode ou un bloc de code peut être marqué comme `synchronized` pour contrôler l'accès aux ressources partagées.

Quand un fil d'exécution entre dans une méthode ou un bloc `synchronized`, il acquiert un verrou qui empêche les autres fils d'y accéder jusqu'à ce que le verrou soit libéré.

- **Méthode synchronized** : Se verrouille sur l'objet appelant.
- **Bloc synchronized** : Peut se verrouiller sur un objet spécifique.

Cette approche garantit que seules les opérations unitaires atomiques sont exécutées dans des environnements multithread.

## Utilisation de `synchronized`

L'utilisation de `synchronized` est essentielle pour assurer que les threads n'entrent pas dans une section critique du code en même temps.

Dans une méthode synchronisée, le verrou est implicitement acquis par l'objet sur lequel la méthode est appelée.

Pour les blocs synchronisés, vous pouvez spécifier explicitement l'objet à verrouiller.

```
public synchronized void updateSharedResource() {  
    // Opération sur une ressource partagée  
}  
  
public void updateSharedResourceWithBlock() {  
    synchronized(this) {  
        // Opération sur une ressource partagée  
    }  
}
```

## Exemples pratiques

Voyons maintenant plusieurs exemples concrets de l'utilisation de `synchronized` :

### Exemple 1 : Synchronisation de méthode

```
public synchronized void increment() {  
    counter++;  
}
```

Dans cet exemple, `increment()` s'assure que seul un thread à la fois peut modifier `counter`.

### Exemple 2 : Bloc synchronisé

```
public void incrementWithBlock() {  
    synchronized(this) {  
        counter++;  
    }  
}
```

Ce bloc synchronisé verrouille l'instance actuelle (`this`), limitant l'accès concurrent à `counter`.

## Exemples avancés

### Exemple 3 : Bloc synchronisé sur un objet mutable

```
private final Object lock = new Object();  
  
public void complexOperation() {  
    synchronized(lock) {  
        // Opérations complexes sur des ressources partagées  
    }  
}
```

**Dans ce scénario, lock est utilisé pour la synchronisation de différentes méthodes reliant des opérations complexes sur des ressources.**

## Exemple 4 : Synchronisation de code critique

```
public class BankAccount {  
    private int balance;  
  
    public void deposit(int amount) {  
        synchronized(this) {  
            balance += amount;  
        }  
    }  
}
```

Cette méthode garantit que `deposit` est effectué de manière atomique, prévenant les conditions de course.

## Exercices pratiques

### Exercice 1 : Synchronisation simple

Implémentez une classe Java qui simule un guichet bancaire.

Utilisez `synchronized` pour protéger les opérations de dépôt et de retrait.

Critères d'évaluation :

- Utilisation correcte de `synchronized` pour éviter les conditions de course.

### Exercice 2 : Bloc synchronized

Créez une classe de compteur partagé entre plusieurs threads.

Assurez-vous que les incréments et décréments sont thread-safe.

Critères d'évaluation :

- Bloc `synchronized` appliqué adéquatement pour protéger l'accès concurrent.

## Synthèse

Le mot-clé `synchronized` est un outil simple mais puissant pour la gestion de la concurrence en Java.

Il offre un moyen efficace de prévenir les conditions de course en contrôlant l'accès simultané aux ressources partagées.

En maîtrisant `synchronized`, vous pouvez améliorer la fiabilité et la robustesse de vos applications concurrentes.

Les prochaines étapes pourraient inclure l'exploration des alternatives comme `ReentrantLock` pour une flexibilité accrue.

# Introduction

Le **ReentrantLock** est une conception d'arrêt avancé de verrouillage introduit en Java 5 pour une gestion flexible et efficace des threads dans la programmation concurrente.

Il est conçu pour surpasser les limitations des verrous intrinsèques (`synchronized`) en offrant plus de possibilités comme l'équité entre threads et la possibilité de personnaliser le comportement de verrouillage.

## Objectifs pédagogiques

- Comprendre l'usage du ReentrantLock dans la gestion des threads.
- Différencier ReentrantLock et synchronized pour une meilleure prise de décision.
- Maîtriser les méthodes et propriétés spécifiques du ReentrantLock.
- Appliquer des ReentrantLock pour résoudre des problèmes de concurrence.

## Comprendre ReentrantLock

Le ReentrantLock implémente l'interface Lock du package `java.util.concurrent.locks`.

Contrairement à `synchronized`, il donne plus de contrôle sur l'acquisition et la libération du verrou.

- **Acquisition et libération explicite :** Contrairement au bloc `synchronized` où le verrou est acquis automatiquement, avec ReentrantLock, cela doit être fait manuellement avec les méthodes `lock()` et `unlock()`.
- **Équité des threads :** Vous pouvez définir un constructeur avec un flag d'équité pour garantir qu'au moins les threads attendent le verrou dans l'ordre de leur demande.

## Avantages du ReentrantLock

Par rapport à `synchronized`, ReentrantLock offre :

1. **Équité :** Vous pouvez éviter la famine en donnant un accès équitable aux threads.
2. **Verrouillage et déverrouillage manuels :** Permet de libérer le verrou à des endroits différents du code, donnant plus de flexibilité.
3. **Tentative de verrouillage :** Utilisez `tryLock()` pour ne pas bloquer un thread indéfiniment s'il ne peut acquérir le verrou.
4. **Verrouillage interruptible :** Avec `lockInterruptibly()`, un thread peut être interrompu lors de l'attente du verrou.

## Méthodes clés en ReentrantLock

- `lock()` : Acquiert le verrou.
- `unlock()` : Libère le verrou.
- `tryLock()` : Retente d'acquérir le verrou, et retourne immédiatement si le verrou est inaccessible.
- `lockInterruptibly()` : Acquiert le verrou sauf si le thread est interrompu.

Il est essentiel d'utiliser la méthode `unlock()` dans un bloc `finally` pour garantir que le verrou est toujours relâché.

# Exemples pratiques

```
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        lock.lock();
        try {
            return count;
        } finally {
            lock.unlock();
        }
    }
}
```

Dans cet exemple, le verrouillage est acquis manuellement pour garantir que les opérations d'incrémentation et de lecture s'exécutent de manière atomique et sûre en termes de threads.

# Exercices pratiques

1. **Exercice 1:** Implementez un simple système bancaire avec plusieurs threads où les comptes utilisent ReentrantLock pour les opérations de dépôt et de retrait.
  - **Critères d'évaluation :** Fonctionnement correct du système de verrouillage et absence de conditions de concurrence.
2. **Exercice 2:** Modifiez une application existante pour remplacer tous les blocs synchronized par des ReentrantLock en maintenant le même comportement.
  - **Critères d'évaluation :** Comparaison des performances et comportement correct du programme.

# Synthèse

Le ReentrantLock est un outil flexible et puissant pour la gestion de verrouillage dans les applications Java concurrentes.

Il permet un contrôle plus fin sur l'acquisition de verrou et favorise des solutions robustes dans des environnements intensément multi-threadés.

Bien que similaire à synchronized, il offre des fonctionnalités supplémentaires qui le rendent préférable dans de nombreux scénarios.

L'apprentissage et la mise en pratique de ReentrantLock ouvrent la voie à des solutions de concurrence plus sophistiquées et nuancées en Java.

# Introduction aux Atomic Variables

Les atomic variables jouent un rôle crucial dans la programmation concurrente en fournissant une manipulation thread-safe des données sans avoir recours à des verrouillages explicites.

Elles permettent d'effectuer des opérations atomiques de manière efficace, évitant ainsi les problèmes de synchronisation traditionnels.

## Objectifs pédagogiques

- Comprendre le concept d'atomicité dans la programmation concurrente
- Identifier les types d'atomic variables disponibles en Java
- Appliquer l'utilisation des atomic variables pour résoudre des problèmes de concurrence
- Comparer les atomic variables avec d'autres mécanismes de synchronisation

## Atomicité et Concurrence

L'atomicité est un concept clé dans la programmation concurrente, désignant des opérations qui s'exécutent de manière indivisible, sans interruption.

Les atomic variables offrent cette garantie en assurant une mise à jour sûre dans des environnements multithread.

## Types d'Atomic Variables

Java offre plusieurs types d'atomic variables dans le package `java.util.concurrent.atomic` :

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

Chacune de ces classes permet d'effectuer des opérations atomiques courantes telles que l'incrémentation et la décrémentation.

## Usage des Atomic Variables

Les atomic variables sont particulièrement utiles dans les situations où de simples opérations arithmétiques ou de mise à jour de référence sont nécessaires, mais doivent être thread-safe.

Elles évitent l'utilisation de `synchronized`, améliorant ainsi la performance sous certaines charges.

## Exemples d'Usage

Considérons une variable de type `AtomicInteger` pour un compteur partagé entre plusieurs threads :

```

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {
    private final AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();
    }

    public int getValue() {
        return counter.get();
    }
}

```

Ici, `incrementAndGet()` est une opération atomique, assurant qu'aucun autre thread ne peut modifier `counter` entre l'incrémentation et l'obtention de sa valeur.

## Comparaison avec Synchronized

Contrairement aux blocs synchronisés qui peuvent introduire des problèmes de contention, les atomic variables fournissent une alternative légère.

Les verrous explicites entraînent souvent de l'attente si un thread détient déjà le verrou. À l'inverse, les atomic variables minimisent ce besoin en utilisant des opérations sur le CPU, comme les comparaisons et les échanges atomiques.

## Exercices Pratiques

### 1. Manipulation de Compteurs :

- Implémentez un compteur utilisant `AtomicLong`.

Testez-le avec plusieurs threads qui modifient la valeur simultanément.

### 2. Démo de Performance :

- Comparez les performances entre un compteur implémenté avec `AtomicInteger` et un autre avec `synchronized`.

### 3. Évaluation d'`AtomicReference` :

- Créez un exemple où une `AtomicReference` est utilisée pour gérer des objets partagés.

## Synthèse et Ouverture

En conclusion, les atomic variables en Java fournissent un moyen rapide et thread-safe pour gérer des opérations sur les données partagées.

Elles sont idéales pour des scénarios nécessitant hautes performances sans les complications des mécanismes de verrou classiques. À l'avenir, l'exploration des alternatives comme `VarHandle` ou des structures de données immuables pourrait enrichir votre boîte à outils pour la concurrence.

## Introduction

Dans cet exercice, nous allons explorer un aspect fondamental de la concurrence moderne en Java : l'implémentation d'un pipeline parallèle de calcul.

En programmant un pipeline, vous pourrez diviser une tâche en plusieurs étapes indépendantes, améliorant ainsi les performances via la parallélisation, tout en appliquant les concepts vus précédemment comme les `CompletableFuture` et les chaînes d'exécution.

# Objectifs pédagogiques

- Appliquer la programmation concurrente pour optimiser les performances.
- Utiliser `CompletableFuture` pour structurer un pipeline de tâches asynchrones.
- Mettre en œuvre des concepts de concurrence moderne via des exemples pratiques.
- Concevoir et évaluer l'efficacité d'un système de calcul parallèle.

## Pipeline parallèle : Concepts

Un pipeline parallèle de calcul est une série de tâches exécutées simultanément, où le résultat d'une tâche est passé à la suivante.

Ceci permet :

- L'amélioration des performances grâce à l'application simultanée des étapes.
- Une utilisation efficace des ressources par la répartition de la charge de travail.
- La modularité, facilitant la maintenance et l'évolution du code.

## CompletableFuture en Java

`CompletableFuture` est un composant Java utilisé pour des calculs asynchrones :

- Il permet de définir des traitements qui s'exécutent sans bloquer le thread principal.
- Supporte la composition de tâches via des méthodes comme `thenApply`, `thenCompose` et `thenCombine`.

Un pipeline est créé en enchaînant ces appels, qui définissent chaque étape du pipeline.

## Exécution parallèle avec ForkJoinPool

Pour gérer l'exécution parallèle :

- `ForkJoinPool` est un framework de gestion de threads permettant la division de tâches en sous-tâches.
- Exploite plusieurs cœurs du processeur en effectuant le traitement de nombreuses petites tâches en parallèle.

# Exemple de code : Pipeline

```
import java.util.concurrent.*;  
  
public class PipelineExample {  
    public static void main(String[] args) {  
        ExecutorService executor = Executors.newFixedThreadPool(4);  
  
        CompletableFuture.supplyAsync(() -> {  
            return computeInitialStep();  
        }, executor).thenApplyAsync(result -> {  
            return processStepOne(result);  
        }, executor).thenApplyAsync(result -> {  
            return processStepTwo(result);  
        }, executor).thenAccept(result -> {  
            System.out.println("Final Result: " + result);  
        });  
  
        executor.shutdown();  
    }  
  
    private static int computeInitialStep() {  
        // Simuler une opération initiale  
        return 2;  
    }  
  
    private static int processStepOne(int input) {  
        // Simuler une première transformation  
        return input + 3;  
    }  
  
    private static int processStepTwo(int input) {  
        // Simuler une deuxième transformation  
        return input * 5;  
    }  
}
```

## Exercices pratiques

1. **Simple Pipeline:** Créez un pipeline avec `CompletableFuture` pour réaliser un calcul simple et afficher le résultat final.
2. **Pipeline Complexe:** Étendez le pipeline existant avec des étapes supplémentaires, comme le filtrage ou l'agrégation de données.
3. **Optimisation:** Évaluez et optimisez le pipeline pour réduire le temps d'exécution, en ajustant le nombre de threads dans le `ForkJoinPool`.

## Critères d'évaluation

- **Pertinence :** Le pipeline doit être efficace et correct, respectant la logique des étapes.
- **Performance :** Utilisation optimale des ressources avec un temps d'exécution réduit.
- **Robustesse :** Gestion des exceptions et des erreurs potentielles dans le pipeline.

## Synthèse et Ouverture

Nous avons mis en œuvre un pipeline parallèle de calcul, appliquant la concurrence moderne grâce à `CompletableFuture`.

Ce module vous permet d'augmenter l'efficacité de vos applications Java en distribuant les tâches.

**En prolongement, on pourrait explorer la programmation réactive, qui élargit les possibilités de traitement asynchrone à des systèmes distribués.**

## Introduction à la Concurrence Moderne

La "Concurrence moderne" est une facette clé de la programmation en Java, essentielle pour tirer parti des capacités multicœurs des processeurs modernes.

Elle s'articule autour de plusieurs concepts et outils puissants visant à simplifier et optimiser l'exécution parallèle et asynchrone des tâches.

## Vue d'ensemble des sous-notions

Dans ce module, nous explorerons :

- **CompletableFuture** : Pour des opérations asynchrones flexibles.
- **Chaînes d'exécution asynchrone** : Pour structurer l'asynchronisme.
- **ForkJoinPool** : Pour diviser le travail efficacement.
- **ExecutorService** : Pour gérer des threads de manière simplifiée.
- **synchronized et ReentrantLock** : Pour contrôler l'accès aux ressources partagées.
- **Atomic variables** : Pour les opérations atomiques et sécurisées en multithreading.
- **Exercice** : Un défi pour implémenter un pipeline parallèle de calcul.

Ces éléments vous équiperont pour développer des applications plus réactives et performantes.

## Introduction

Le concept de **Reactive Streams** est au cœur de la programmation réactive moderne.

Il s'agit d'une initiative pour définir un standard de traitement asynchrone de flux de données avec une gestion de backpressure.

Ce modèle est crucial pour créer des applications capables de traiter de grandes quantités de données de manière efficace et non bloquante.

## Objectifs pédagogiques

- Comprendre le rôle et les bénéfices des Reactive Streams.
- Maîtriser les composants clés de Reactive Streams.
- Apprendre à manipuler des flux de données asynchrones.
- Appliquer le modèle de backpressure dans des applications Java.

# Concept et fonctionnement

Les **Reactive Streams** sont conçus pour gérer les flux de données asynchrones.

Ce modèle repose sur quatre interfaces principales :

1. **Publisher** : Émet des éléments aux abonnés enregistrés.
2. **Subscriber** : Consomme des éléments fournis par le Publisher.
3. **Subscription** : Lien entre Publisher et Subscriber, gère la demande d'éléments.
4. **Processor** : Transforme les données en agissant à la fois comme un Subscriber et un Publisher.

## Importance de Backpressure

Backpressure est un concept crucial pour éviter la surcharge de mémoire quand les consommateurs ne peuvent pas suivre le rythme des producteurs.

Cette fonctionnalité permet de :

- Demander explicitement le nombre d'éléments à recevoir.
- Suspendre la réception jusqu'à nouvelle demande, ce qui prévient la saturation.
- Maintenir une application stable, même lors d'un traitement de gros volumes de données.

## Exemples de code

Exemple basique de création d'un Publisher :

```
import org.reactivestreams.Publisher;
import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

public class SimplePublisher implements Publisher<String> {
    @Override
    public void subscribe(Subscriber<? super String> subscriber) {
        // Manage subscription logic here
        subscriber.onSubscribe(new SimpleSubscription(subscriber));
    }
}
```

À explorer plus en détail : l'implémentation de `SimpleSubscription`.

## Exemple illustré : Subscriber

Implémentation d'un simple Subscriber :

```

public class SimpleSubscriber implements Subscriber<String> {
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1); // Demand one item at start
    }

    @Override
    public void onNext(String item) {
        System.out.println("Received: " + item);
        subscription.request(1); // Request next item
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Complete!");
    }
}

```

## Exercices pratiques

- Exercice 1 :** Implémentez un Processor qui transforme un flux de chaînes en leur version majuscules.
  - Critères : Assurez-vous que le backpressure est géré correctement.
- Exercice 2 :** Créez un système simple de publication et abonnement où un Publisher envoie des nombres entiers, et un Subscriber calcule leur somme.
  - Critères : Mettre en œuvre la gestion du backpressure de façon que le Subscriber puisse contrôler le flux des données.

## Synthèse et ouverture

Les **Reactive Streams** offrent un modèle puissant pour traiter efficacement de grandes quantités de données en mode asynchrone.

La gestion du backpressure est un atout essentiel pour maintenir la robustesse de l'application face à des volumes de données variables.

En suivant cette approche, vous êtes préparés à aborder des frameworks comme Project Reactor et RxJava pour construire des systèmes réactifs évolutifs.

En poursuivant votre apprentissage, explorez comment intégrer ces concepts dans des architectures microservices pour maximiser la performance et la réactivité.

## Introduction à Backpressure

La Backpressure est un concept clé en programmation réactive qui s'attaque aux problèmes liés à la gestion du flux de données entre fournisseurs et consommateurs.

Dans les systèmes réactifs, il est essentiel de contrôler la pression exercée par le flux de données pour éviter les débordements de mémoire et garantir des performances optimales.

Cette section apportera une compréhension approfondie de la façon dont la Backpressure améliore la résilience des systèmes réactifs.

## Objectifs pédagogiques

- Comprendre le concept de Backpressure.
- Appliquer la Backpressure dans des systèmes réactifs.
- Identifier et résoudre les problèmes de flux de données avec la Backpressure.
- Implémenter des stratégies efficaces pour gérer le flux de données.

## Comprendre la Backpressure

La Backpressure se réfère à la capacité du système à gérer les situations où le rythme des données émises dépasse la capacité de traitement du consommateur.

Cela est crucial pour éviter les exceptions liées à la mémoire ou aux ressources, assurant que les consommateurs ne soient pas submergés.

## Mécanismes de Backpressure

La Backpressure repose sur plusieurs mécanismes pour moduler le flux de données :

- **Buffering** : Stocker temporairement les données en attente de traitement.
- **Drop** : Ignorer certaines données en cas de surcharge du consommateur.
- **Throttling** : Réguler la fréquence des données transmises.

Ces stratégies permettent d'équilibrer la charge de travail et de garantir la stabilité du système.

## Exemples de Backpressure

```
import io.reactivex.Flowable;
import io.reactivex.schedulers.Schedulers;

// Exemple d'un émetteur rapide
Flowable<Integer> fastEmitter = Flowable.range(1, 1000)
    .onBackpressureBuffer(100); // Ajoute un tampon de 100 éléments

fastEmitter
    .observeOn(Schedulers.computation())
    .subscribe(data -> {
        // Simule un traitement lent
        Thread.sleep(10);
        System.out.println("Élément traité: " + data);
    });
}
```

Ce code montre comment appliquer un tampon pour gérer un flux rapide.

# Stratégies avancées

Pour des cas plus complexes, utiliser des stratégies comme `onBackpressureDrop()` ou `onBackpressureLatest()` permet de gérer intelligemment les éléments de flux.

- **onBackpressureDrop** : Abandonne les éléments excédentaires.
- **onBackpressureLatest** : Ne conserve que les éléments les plus récents.

Ces options sont utiles lorsque la latence et la fraîcheur des données sont critiques.

## Exercices pratiques

1. **Exercice 1 :** Implémentez un système avec une fréquence d'émission de données élevée, en utilisant une stratégie de tamponnage.
  - Critère : Assurez-vous que le consommateur ne soit pas saturé.
2. **Exercice 2 :** Expérimitez avec `onBackpressureDrop` pour comprendre son comportement dans une application Java réactive.
  - Critère : Notez les différences de performances.

## Synthèse et ouverture

En résumé, la Backpressure joue un rôle indispensable dans l'ingénierie des systèmes réactifs, garantissant que les fournisseurs ne satureront pas les consommateurs.

Les capacités à implémenter efficacement des stratégies de Backpressure sont essentielles pour les développeurs cherchant à construire des applications fiables et réactives.

Dans les sections futures, nous explorerons comment combiner la Backpressure avec d'autres concepts réactifs pour améliorer encore les performances de nos systèmes.

## Introduction

Le Project Reactor est une bibliothèque de programmation réactive pour Java, conçue pour faciliter la construction d'applications concurrentes et évolutives.

Elle repose sur les spécifications Reactive Streams et offre des abstractions puissantes pour traiter des flux asynchrones de données.

Ce module explore comment utiliser Project Reactor pour créer des applications réactives efficaces.

## Objectifs pédagogiques

- Comprendre les concepts clés de Project Reactor.
- Manipuler les types principaux de Project Reactor : Mono et Flux.
- Implémenter un modèle de backpressure.
- Développer des applications réactives résilientes et performantes.

# Concepts clés

Project Reactor est basé sur les principes de la programmation réactive.

Les deux types principaux sont :

- **Mono** : Représente 0 ou 1 résultat.

Il est idéal pour les tâches qui produisent un résultat unique ou aucune sortie.

- **Flux** : Équivalent à un observable dans RxJava, il peut représenter une séquence de 0 à N éléments.

Avec Project Reactor, vous pouvez composer des chaînes de traitement de manière déclarative.

## Types principaux

### Mono

- **Description** : Modélise une opération asynchrone qui aboutira à un résultat unique ou à une erreur.
- **Usage** : Adapter pour les appels REST, requêtes de base de données uniques, etc.

### Flux

- **Description** : Gère des séquences de flux pouvant avoir zéro à plusieurs émissions.
- **Usage** : Correspond aux streamings, tickers ou tout autre flux de données continu ou répété.

## Opérations communes

Avec Mono et Flux, vous pouvez réaliser différentes opérations réactives :

- **map()** : Transforme les éléments émis.
- **flatMap()** : Aplatit des Flux ou Mono émis en un seul Flux.
- **filter()** : Filtre les émissions basées sur des conditions définies.

Ces opérations sont la base pour composer des flux de traitement complexes.

## Exemples de code

### Usage de Mono

```
Mono<String> mono = Mono.just("Hello World");
mono.subscribe(System.out::println);
```

- **Explication** : Crée un Mono émettant "Hello World" immédiatement.

## Usage de Flux

```
Flux<Integer> flux = Flux.range(1, 5);
flux.map(i -> i * 2)
    .subscribe(System.out::println);
```

- **Explication** : Génère un Flux de nombres de 1 à 5, les double et les imprime.

## Projet : Backpressure

Pour gérer le backpressure avec Flux :

1. **Stratégies** : Utiliser les opérateurs `onBackpressureBuffer` , `onBackpressureDrop` , etc.

2. **Exemple** :

```
Flux.range(1, 100)
    .onBackpressureBuffer()
    .subscribe(System.out::println,
              System.err::println,
              () -> System.out.println("Done"));
```

- **Explication** : Gestion basique du backpressure en utilisant un buffer.

## Exercices progressifs

1. **Créer un Flux** :

- Créez un Flux qui émet les chiffres de 1 à 10, appliquez une transformation et affichez le résultat.

2. **Gestion d'erreurs** :

- Implémentez une gestion d'erreurs soignée avec `onErrorResume` .

3. **Backpressure avancé** :

- Expérimitez avec `onBackpressureDrop` pour observer l'impact sur des flux à haute fréquence.

## Synthèse

Project Reactor améliore la capacité à gérer des opérations asynchrones en favorisant une approche fonctionnelle et réactive.

En maîtrisant Mono et Flux, vous pouvez construire des applications résilientes et performantes.

Poursuivez avec des cas d'utilisation réels pour perfectionner votre pratique.

## Introduction

RxJava est une implémentation de la Programmation Réactive pour Java.

Elle permet de composer des programmes asynchrones et événementiels avec des flux de données observables.

Ce paradigme améliore la gestion des flux de données dans des applications concurrentes, simplifiant la manipulation de la concurrence et de l'asynchronisme.

# Objectifs Pédagogiques

- Écrire des programmes asynchrones avec RxJava.
- Manipuler les Observables et comprendre leur rôle dans RxJava.
- Implémenter et utiliser les opérateurs courants comme `map`, `flatMap`, `filter`, et `merge`.
- Comprendre le concept de backpressure et comment le gérer en RxJava.

## Concepts de Base

### Observables

Les Observables en RxJava sont la pierre angulaire, permettant de représenter et de gérer des flux de données.

Ils émettent des items sur la base d'une souscription.

**L'Observable va notifier l'observateur par des événements `onNext` , `onCompleted` , ou `onError` .**

### Observateurs

Un observateur souscrit à un Observable pour recevoir ces événements.

**Il réagit à chaque émission de valeurs, à la complétion du flux ou à des erreurs.**

### Opérateurs

Les opérateurs modifient, filtrent, et combinent les flux de données.

Ils permettent de transformer les données ainsi que de contrôler le flux, comme `map` pour transformer chaque élément, `filter` pour filtrer certaines valeurs, ou `flatMap` pour décomposer les flux.

## Opérateurs Courants

### map

L'opérateur `map` transforme chaque item émis.

Par exemple, pour doubler chaque nombre dans un flux:

```
Observable.just(1, 2, 3, 4)
    .map(x -> x * 2)
    .subscribe(System.out::println);
```

## flatMap

`flatMap` transforme chaque item en un Observable, puis combine les résultats.

Idéal pour les transformations asynchrones.

```
Observable.just(1, 2, 3, 4)
    .flatMap(x -> Observable.just(x * 2))
    .subscribe(System.out::println);
```

## filter

Cet opérateur supprime les items du flux en fonction d'un prédictat:

```
Observable.just(1, 2, 3, 4)
    .filter(x -> x % 2 == 0)
    .subscribe(System.out::println);
```

## merge

`merge` combine plusieurs Observables en un seul flux simultané.

```
Observable.merge(
    Observable.just(1, 2),
    Observable.just(3, 4)
).subscribe(System.out::println);
```

# Backpressure et Gestion

Le backpressure survient quand une source d'Observable émet des items plus rapidement que ce que le flux récepteur peut traiter.

RxJava offre des stratégies comme `onBackpressureBuffer` pour gérer ces situations.

# Exemples Pratiques

## Exemple 1 : Système de Notification

Imaginez un système de notifications pour envoyer des messages asynchrones:

```
Observable<String> notifications = Observable.just("Email", "SMS", "Push");
notifications.subscribe(System.out::println);
```

## Exemple 2 : Traitement Parallèle

Utiliser `flatMap` pour exécuter des tâches en parallèle:

```
Observable.range(1, 10)
    .flatMap(x -> Observable.just(x).subscribeOn(Schedulers.computation()))
    .subscribe(System.out::println);
```

## Exercices Pratiques

### Exercice 1 : Créer un Observables

- Créez un Observable qui émet les nombres de 1 à 10 et appliquez `map` pour doubler chaque nombre.
- Critères d'évaluation: le code doit être clair, et correctement commenté.

### Exercice 2 : Gestion de Backpressure

- Implémentez un flux qui utilise `onBackpressureBuffer` pour gérer un flux rapide d'items.

## Synthèse et Ouverture

RxJava simplifie grandement la gestion de la programmation asynchrone.

En maîtrisant ses Observables et opérateurs, vous pouvez construire des applications souples et réactives.

En poursuivant, explorez Project Reactor pour plus de fonctionnalités avancées et une interopérabilité accrue avec d'autres systèmes.

## Introduction

Dans cette section, nous allons explorer la notion d'Observable, un concept clé dans la programmation réactive.

L'Observable permet de gérer des flux de données de manière asynchrone et événementielle.

Il est essentiel pour développer des systèmes réactifs performants capables de réagir aux changements d'état en temps réel.

## Objectifs pédagogiques

- Comprendre le rôle de l'Observable dans la programmation réactive.
- Apprendre à créer et utiliser des Observables en Java.
- Manipuler les flux de données avec des opérateurs réactifs.

## Notion d'Observable

Un Observable est une entité qui émet des éléments auxquels les observateurs peuvent s'abonner.

Cette capacité à gérer les flux asynchrones est au cœur des systèmes réactifs.

- Un Observable émet trois types d'événements : valeurs de données, erreurs et notification de fin.
- Les Observateurs (ou abonnées) consomment ces événements lorsqu'ils surviennent.
- Cela rend le modèle Observable idéal pour manipuler des flux continus de données.

## Création d'un Observable

La création d'un Observable est simple.

Il s'agit de définir de quelle manière les valeurs seront émises.

```
Observable<String> observable = Observable.create(emitter -> {
    emitter.onNext("Message 1");
    emitter.onNext("Message 2");
    emitter.onComplete();
});
```

- `onNext()` émet des valeurs.
- `onComplete()` signale la fin des émissions.
- Les Observables peuvent également émettre des erreurs avec `onError()`.

## Utilisation d'Observable

L'Observateur s'abonne à l'Observable pour recevoir les données émises.

```
observable.subscribe(
    item -> System.out.println("Reçu : " + item),
    error -> System.err.println("Erreur : " + error),
    () -> System.out.println("Flux terminé")
);
```

- L'observateur reçoit les éléments via les méthodes `onNext()`, `onError()`, et `onComplete()`.

## Exemples de code

### Exemple 1 : Émission de valeurs simples

```
Observable.just("Red", "Green", "Blue")
    .subscribe(System.out::println);
```

## Exemple 2 : Gestion des erreurs

```
Observable<String> observableWithError = Observable.create(emitter -> {
    try {
        emitter.onNext("Before Error");
        throw new Exception("Erreur simulée");
    } catch (Exception e) {
        emitter.onError(e);
    }
});
observableWithError.subscribe(
    System.out::println,
    error -> System.err.println("Erreur reçue : " + error)
);
```

## Exercices

- Créer un Observable** : Créez un Observable qui émet les nombres de 1 à 10, et laissez l'observateur afficher chaque nombre.
- Gérer les erreurs** : Modifiez l'Observable pour provoquer une erreur après le 5ème nombre, puis gérez cette erreur dans l'observateur.
- Combiner des Observables** : Créez deux Observables distincts et fusionnez-les pour qu'un seul observateur affiche leurs valeurs combinées.

Critères d'évaluation :

- Correctitude des valeurs émises.
- Gestion appropriée des erreurs.
- Utilisation correcte des opérateurs réactifs.

## Synthèse

Les Observables offrent un moyen puissant et flexible de gérer les flux asynchrones dans les applications réactives.

Ils permettent une gestion fluide des données, des erreurs et du cycle de vie des événements. À travers des exercices progressifs, vous pouvez renforcer votre compréhension et optimiser vos applications Java réactives.

Les prochaines étapes incluent l'exploration des opérateurs réactifs avancés pour une manipulation plus fine des flux.

## Introduction

La fonction `map` en programmation réactive est un opérateur fondamental permettant la transformation des éléments émis par un flux (stream).

Dans ce cours, nous découvrirons comment utiliser cet opérateur pour manipuler les données de manière fluide et intuitive.

La compréhension du `map` est essentielle pour écrire du code réactif efficace et propre.

## Objectifs pédagogiques

- Comprendre le rôle de l'opérateur `map` en programmation réactive
- Savoir appliquer `map` pour transformer des données au sein d'un flux
- Être capable d'expliquer les avantages de l'utilisation de `map`

- Mettre en œuvre des transformations de flux concrètes avec des exemples en Java
- Maîtriser l'enchaînement de transformations multiples avec `map`

## Fonctionnement de `map`

L'opérateur `map` est utilisé pour transformer chaque élément émis par un flux en un nouvel élément via une fonction fournie.

Il s'agit d'une opération de transformation simple mais puissante.

Contrairement à `flatMap`, `map` ne change pas la nature du flux, mais modifie seulement les données.

## Utilisation de `map` en Java

Pour utiliser `map` dans un contexte de flux réactif, vous devez avoir un flux source, tel qu'un `Flux` ou un `Observable`.

Vous lui appliquez ensuite `map` avec une fonction lambda qui décrit comment les éléments doivent être transformés.

```
Flux<String> flux = Flux.just("apple", "banana", "cherry");
flux.map(String::toUpperCase).subscribe(System.out::println);
```

## Transformation de données réactives

Le `map` est idéal pour les cas où chaque élément du flux nécessitera une transformation identique.

Par exemple, convertir toutes les chaînes en majuscules ou ajouter un préfixe commun.

Exemples pratiques :

- Calculer le double de chaque nombre dans un flux d'entiers
- Convertir des objets en un format spécifique en les mappant à d'autres objets

## Exemple de code : Conversion

Voyons un exemple pour convertir un flux d'entiers représentant des températures en Fahrenheit vers Celsius.

```
Flux<Integer> fahrenheitTemps = Flux.just(32, 212, 98);
fahrenheitTemps.map(f -> (f - 32) * 5 / 9)
    .subscribe(temp -> System.out.println(temp + " °C"));
```

Ce code transforme chaque température de Fahrenheit en Celsius et les émet.

## Exemple de code : Complexité

Un autre exemple consiste à mapper des objets `Person` vers leurs noms en majuscules.

```

public class Person {
    private String name;

    // Constructeur, getter et setter
}

Flux<Person> peopleFlux = Flux.just(new Person("Alice"), new Person("Bob"));
peopleFlux.map(person -> person.getName().toUpperCase())
    .subscribe(System.out::println);

```

## Exercice : Transformation simple

- **Objectif :** Transformer un flux de mots pour en sortir leur longueur.
- **Énoncé :** À partir d'un `Flux` de mots, utilisez `map` pour créer un nouveau flux contenant la longueur de chaque mot.

Affichez les longueurs.

- **Critères d'évaluation :** Code fonctionnel, conformité à la consigne, utilisation correcte de `map`.

## Exercice : Transformation complexe

- **Objectif :** Transformez un flux d'objets complexes.
- **Énoncé :** Avec un flux de `Person` objets, utilisez `map` pour créer un flux de `String` contenant le prénom suivi de l'âge.

Affichez le résultat avec une phrase structurée.

- **Critères d'évaluation :** Utilisation correcte de `map`, transformations réussies, bonne structure de phrase.

## Synthèse et ouverture

L'opérateur `map` est un outil puissant pour transformer les données émises par un flux réactif.

En maîtrisant son utilisation, vous pouvez écrire des transformations données de manière claire et concise.

En guise de prochaine étape, vous pourriez explorer l'opérateur `flatMap` pour des scénarios de transformation plus complexes, comme la manipulation de flux imbriqués.

## flatMap

### Introduction

La méthode `flatMap` est un composant clé de la programmation réactive qui permet de transformer des éléments émis par un flux en d'autres flux, puis de les aplatis en un seul flux.

Cela est essentiel pour gérer des flux de données complexes et imbriqués en les simplifiant.

En apprenant `flatMap`, vous enrichirez vos compétences pour créer des applications réactives performantes.

## Objectifs pédagogiques

- Comprendre le fonctionnement de `flatMap` dans le contexte de la programmation réactive
- Appliquer `flatMap` pour transformer et aplatis des flux complexes
- Intégrer `flatMap` dans des flux réactifs en Java avec RxJava ou Project Reactor
- Résoudre des problèmes concrets de transformation de données grâce à `flatMap`

## Contenu détaillé

La méthode `flatMap` est utilisée pour prendre chaque élément d'un flux et le transformer en un autre flux.

Ce processus permet d'imbriquer des flux, puis de les combiner en un seul flux de données linéaire.

**Ce modèle est idéal pour traiter des collections de données où chaque élément peut générer plusieurs nouvelles données.**

### Utilisation de `flatMap`

Dans le contexte de la programmation réactive en Java, `flatMap` est souvent utilisé avec des bibliothèques comme RxJava ou Project Reactor.

**Il permet d'émettre des événements, de les transformer, et de les aplatis, simplifiant ainsi les traitements asynchrones.**

### Comparaison avec `map`

Contrairement à `map` qui transforme un élément en un autre élément, `flatMap` prend un élément et le transforme en un flux, déployant et combinant les résultats en un flux unique.

## Exemples de code

### Exemple simple

Utilisons `flatMap` pour transformer une liste de chaînes de caractères en leurs caractères individuels :

```
import io.reactivex.Observable;

Observable<String> strings = Observable.just("Hello", "World");
strings
    .flatMap(s -> Observable.fromArray(s.split ""))
    .subscribe(System.out::println);
```

### Exemple d'application pratique

Supposons que vous avez une application de gestion de fichiers, et que vous souhaitez lister tous les fichiers d'un ensemble de répertoires :

```
import io.reactivex.Observable;
import java.io.File;

Observable<File> directories = Observable.just(new File("dir1"), new File("dir2"));
directories
    .flatMap(dir -> Observable.fromArray(dir.listFiles()))
    .subscribe(file -> System.out.println(file.getName()));
```

## Exercices pratiques

### Exercice 1 : Transformation de données

Écrivez un programme qui utilise `flatMap` pour transformer une liste de phrases en un flux de mots.

**Utilisez RxJava pour cette tâche.**

### Exercice 2 : Traiter des flux imbriqués

Créez un programme qui prend une liste d'URLs et utilise `flatMap` pour télécharger le contenu de chaque page.

Analysez et affichez les titres des pages.

**Critères d'évaluation :**

- Usage correct de `flatMap` pour aplatiser les flux
- Gestion correcte des erreurs asynchrones
- Optimisation des performances et lisibilité du code

## Synthèse et ouverture

La méthode `flatMap` est un outil puissant dans la programmation réactive, offrant des capacités uniques pour transformer et aplatiser des flux de données.

En comprenant son fonctionnement, vous êtes prêt à aborder des applications complexes et à optimiser le traitement des données réactives.

**Dans la suite, explorez comment intégrer `flatMap` dans un système de notifications réactif complet.**

## Filter

## Introduction

Dans le paradigme de la programmation réactive, le filtre (`filter`) est une opération essentielle.

Il permet de sélectionner les éléments d'une séquence qui répondent à un critère spécifique.

En Java, cela est souvent appliqué aux flux réactifs, permettant une gestion plus fine et plus efficace des données circulant dans votre application.

Intégrer un `filter` dans vos pipelines réactifs peut réduire la charge de traitement en ne traitant que les éléments pertinents pour votre logique métier.

## Objectifs pédagogiques

- Comprendre l'utilité de la méthode `filter`.
- Appliquer `filter` sur un flux de données avec des critères spécifiques.
- Intégrer `filter` dans un pipeline réactif.
- Démontrer comment `filter` peut optimiser les flux de traitement des données.

## Fonctionnement du Filter

La méthode `filter` est utilisée pour sélectionner des éléments d'un flux réactif répondant à une condition.

Elle prend en argument un prédicat, une fonction logique qui renvoie un booléen.

- **Signature :** `filter(Predicate<? super T> predicate)`
- **Type de retour :** Un flux ne contenant que les éléments validé par le prédicat.

Par exemple, pour n'obtenir que les nombres pairs dans un flux, un prédicat testant la divisibilité par deux est utilisé.

## Utilisation avec Streams

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

Stream<Integer> numberStream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> evenNumberStream = numberStream.filter(n -> n % 2 == 0);

// Résultat : [2, 4]
```

Dans cet exemple, le `filter` isole les nombres pairs du flux initial.

Chaque élément est testé contre le prédicat `(n -> n % 2 == 0)`.

## Filter en Programmation Réactive

Dans des bibliothèques réactives comme RxJava ou Project Reactor, `filter` opère de manière similaire, mais sur des flux de données asynchrones.

Par exemple, avec Project Reactor :

```
import reactor.core.publisher.Flux;

Flux<Integer> positiveNumbers = Flux.just(-1, 2, -3, 4, 5)
    .filter(n -> n > 0);

// Résultat : seulement les nombres positifs [2, 4, 5]
```

Ici, seuls les nombres positifs passent le filtre, réduisant ainsi la charge de traitement des éléments non pertinents.

## Exemple complet avec RxJava

```
import io.reactivex.rxjava3.core.Observable;

Observable<Integer> observable = Observable.just(5, 13, 2, 8, 21);

// Filtre les nombres supérieurs à 10
observable.filter(i -> i > 10)
    .subscribe(System.out::println); // Affiche 13, 21
```

Dans cet exemple, un `filter` sur un `Observable` extrait les nombres supérieurs à 10 et les imprime.

## Exercice Pratique

- Objectif :** Créer un flux d'entiers de 1 à 100 et utilisez `filter` pour n'extraire que les multiples de 5.
- Consigne :** Implémentez cette logique en utilisant Project Reactor ou RxJava.

Vérifiez le résultat en affichant chaque élément filtré.

- Critères de réussite :** Un flux qui affiche uniquement les nombres 5, 10, ..., jusqu'à 100.

## Synthèse

La méthode `filter` est cruciale pour contrôler le flux de données dans la programmation réactive.

En sélectionnant seulement ce qui est pertinent, elle optimise le traitement et rend vos applications plus efficaces.

En maîtrisant `filter`, vous pouvez concevoir des pipelines de traitement de données à la fois réactifs et performants. À mesure que vous progressez, explorez l'intégration de plusieurs `filter` et autres transformations pour enrichir vos compétences en programmation réactive.

## Introduction

Dans cette session, nous allons explorer la notion de "merge" dans le contexte de la programmation réactive en Java.

Le concept de `merge` est crucial pour combiner plusieurs flux de données en un seul flux.

Cela permet de gérer et synchroniser les données provenant de sources multiples de manière réactive, augmentant ainsi l'efficacité et la réactivité de votre application.

## Objectifs pédagogiques

- Comprendre le concept de `merge` en programmation réactive.
- Apprendre à combiner plusieurs flux de données en un seul.
- Être capable d'implémenter et d'utiliser `merge` dans un contexte applicatif Java.
- Appliquer `merge` pour la synchronisation de flux de données hétérogènes.

# Concept de Merge

`merge` est utilisé en programmation réactive pour combiner plusieurs sources de données asynchrones en un flux unique sans respecter l'ordre des événements.

- **Non-bloquant** : Permet la combinaison sans attente bloquante.
- **Indépendant de l'ordre** : Les émissions des flux sourcés sont émises au fur et à mesure qu'elles se produisent.
- **Utilisation courante** : L'agrégation de résultats de plusieurs appels de services asynchrones.

## Implémentation en Java

**En Java, particulièrement avec des bibliothèques comme RxJava et Project Reactor, `merge` est utilisé pour combiner les flux (Observables ou Flux).**

### RxJava Exemple

```
Observable<String> source1 = Observable.just("A", "B", "C");
Observable<String> source2 = Observable.just("1", "2", "3");

Observable<String> merged = Observable.merge(source1, source2);
merged.subscribe(System.out::println);
```

- **Description** : Les flux `source1` et `source2` sont fusionnés en un seul flux avec `Observable.merge`.

## Exemples de Code

### Flux Réactif en Project Reactor

```
Flux<String> flux1 = Flux.just("Apple", "Orange");
Flux<String> flux2 = Flux.just("Circle", "Square");

Flux<String> mergedFlux = Flux.merge(flux1, flux2);
mergedFlux.subscribe(System.out::println);
```

- **Explication** : `flux1` et `flux2` sont fusionnés.

Les événements de chaque flux sont émis dès qu'ils se produisent.

Cet exemple reflète la nature réactive où les types de données peuvent être différents mais partagent le même flux de traitement.

## Exercice Pratique

1. **Objectif** : Créer une application Java utilisant RxJava pour fusionner les flux de données provenant de deux sources différentes (par exemple, des capteurs de température et d'humidité).
2. **Étapes** :
  - Créez deux `Observable` émettant respectivement les températures et les pourcentages d'humidité.
  - Utilisez `Observable.merge` pour combiner ces flux.

- Affichez chaque événement émis par le flux fusionné.

### 3. Critères d'évaluation :

- Utilisation correcte de la méthode `merge`.
- Gestion de l'émission et de la consommation des flux fusionnés sans blocage.
- Clarté et efficacité du code.

## Synthèse et Ouverture

Grâce à la notion de `merge`, nous avons appris à combiner plusieurs flux de façon réactive en Java, ce qui permet de gérer efficacement les flux de données asynchrones.

En maîtrisant `merge`, vous pouvez développer des applications plus réactives et résilientes.

Pour aller plus loin, explorez d'autres opérations réactives comme `combineLatest` ou `zip` pour synchroniser plusieurs flux avec des contraintes spécifiques.

## Introduction

Dans cet exercice, nous allons créer un système de notifications réactif utilisant la programmation réactive en Java.

Ce projet pratique vous permettra d'appliquer les concepts théoriques de la programmation réactive pour gérer des flux de données de manière non-bloquante et responsive.

## Objectifs pédagogiques

- Comprendre les principes de la programmation réactive.
- Créer et gérer des flux de données réactifs.
- Implémenter un système de notifications efficace.
- Utiliser les opérateurs de transformation pour manipuler des flux de données.
- Mettre en œuvre la gestion des erreurs et le contrôle de flux.

## Contenu détaillé: Programmation réactive

La programmation réactive est un paradigme qui permet de traiter des flux d'événements asynchrones.

Elle repose sur la gestion des flux de données en temps réel, les changements dans les données déclenchant automatiquement des opérations définies.

- **Flux de données :** Les flux peuvent être considérés comme des séquences temporelles d'éléments.
- **Réactivité :** Gérer les événements au fur et à mesure de leur apparition, assurant une interaction fluide.
- **Asynchronisme :** Utilisation d'opérations non-bloquantes pour éviter les délais.

## Contenu détaillé: Outils réactifs

Pour ce projet, nous utiliserons des bibliothèques Java telles que Project Reactor ou RxJava qui facilitent la création et la gestion de flux réactifs.

- **Reactive Streams :** Norme pour traiter les flux asynchrones avec backpressure.

- **Project Reactor** : Fournit des types réactifs comme Flux et Mono pour manipuler les flux de données.
- **Opérateurs** : Fonctions pour transformer et combiner les flux.

## Exemples de code: Flux de données

```
Flux<String> notifications = Flux.just("Email reçu", "Message instantané", "Nouveau commentaire");
notifications.subscribe(System.out::println);
```

Cet exemple de base montre comment créer un flux de notifications et y souscrire pour suivre les événements.

## Exemples de code: Transformation et manipulation

```
Flux<String> processedNotifications = notifications
    .map(String::toUpperCase)
    .filter(notif -> notif.contains("EMAIL"));

processedNotifications.subscribe(System.out::println);
```

Cet exemple démontre l'utilisation d'opérateurs `map` et `filter` pour transformer et filtrer les notifications.

## Exemples de code: Gestion des erreurs

```
Flux<String> resilientNotifications = notifications
    .onErrorResume(e -> Flux.just("Erreur : Notification introuvable"))
    .subscribe(System.out::println);
```

Utilisez `onErrorResume` pour gérer les erreurs et fournir un flux alternatif en cas de problème.

## Exercices pratiques

1. Créez un flux réactif émettant différents types d'événements (emails, messages SMS, alertes système).
2. Appliquez des transformations pour identifier et taguer les événements critiques.
3. Implémentez la gestion des erreurs pour assurer la continuité du flux en cas d'échec.

**Critères d'évaluation :** Fonctionnalité réactive intégrée, gestion des erreurs efficace, utilisation correcte des opérateurs.

## Synthèse et ouverture

En complétant cet exercice, vous avez exploré les fondements de la programmation réactive et sa mise en application pour créer un système de notifications réactif.

En approfondissant cette compétence, vous pourrez développer des applications robustes et scalables qui répondent efficacement aux demandes changeantes des utilisateurs.

**Cette maîtrise ouvre la voie à une exploration plus poussée des architectures réactives dans des environnements à grande échelle.**

# Programmation réactive

## Introduction

La programmation réactive est un paradigme axé sur l'architecture d'applications répondant efficacement à des événements en temps réel.

Elle permet de gérer des flux de données de manière asynchrone.

Nous aborderons plusieurs composants clés de ce paradigme :

- **Reactive Streams** pour la manipulation de flux.
- **Backpressure** pour gérer la pression des données.
- **Project Reactor** et **RxJava**, deux bibliothèques populaires pour implémenter des flux réactifs.
- Concepts comme **Observable**, **map**, **flatMap**, **filter**, et **merge** utilisés pour transformer et combiner les flux de données.
- Un exercice pratique : créer un système de notifications réactif.

## Vue d'ensemble

Les sous-notions de la programmation réactive s'articulent autour des flux de données et de leur manipulation :

- **Reactive Streams** et **Backpressure** offrent une structure de base pour traiter les données.
- **Project Reactor** et **RxJava** fournissent des outils pour implémenter ces concepts.
- Opérateurs comme **map**, **flatMap**, **filter**, et **merge** permettent de transformer et combiner les flux.
- L'exercice final mettra ces notions en pratique avec un système de notifications réactif.

## Introduction à VisualVM

VisualVM est un outil performant utilisé pour le monitorage, le profiling, et le debugging des applications Java.

Il attire l'attention des développeurs car il fournit une interface unifiée pour examiner les performances des applications Java, analyser les dumps de heap, et inspecter des threads entre autres fonctionnalités, favorisant ainsi une optimisation efficace.

## Objectifs pédagogiques

- Comprendre les fonctionnalités clés de VisualVM.
- Utiliser VisualVM pour profiler une application Java.
- Analyser et interpréter les résultats du profiling pour une optimisation efficace.
- Déetecter et résoudre les problèmes de performance dans une application Java.

## Découverte de VisualVM

VisualVM offre une interface graphique facilitant l'intégration et le suivi des performances applicatives.

- **Interface unifiée** : Combinaison de plusieurs outils JDK.
- **Fonctionnalités principales** : Profilage, Analyse de heap dump, Surveillance des threads, et Monitoring en temps réel.

Ces fonctionnalités permettent une gestion fine de l'application pour des ajustements précis.

## Démarrage avec VisualVM

Pour utiliser VisualVM, suivez ces étapes :

- **Installation** : Inclus dans le JDK ou peut être téléchargé séparément.
- **Lancement** : Exécuter via la ligne de commande avec `jvisualvm` ou à travers une interface utilisateur.

Vérifiez que votre application est prête à être profilée ou analysée avant de commencer.

## Profilage d'une Application

Le profilage consiste à analyser en profondeur l'exécution d'une application pour identifier les parties du code les plus coûteuses.

- **CPU Profiling** : Identifie les méthodes qui consomment le plus de temps.
- **Memory Profiling** : Suivi de l'affectation et de la libération de la mémoire.

Un bon profilage peut révéler des goulets d'étranglement cachés.

## Cas d'Usage : CPU Profiling

Avec VisualVM, vous pouvez identifier les méthodes les plus consommantes en utilisant le CPU Profiling.

- **Démarrer un Profiling** : Cliquez sur 'Profil' > 'CPU Profiling'.
- **Résultats** : Obtenez des statistiques détaillées sur l'utilisation du processeur par méthode.

Utilisez ces résultats pour optimiser les méthodes critiques de votre application.

## Exemples de Code

```
public class Example {  
    public static void main(String[] args) {  
        for (int i = 0; i < 1000; i++) {  
            performTask();  
        }  
    }  
  
    private static void performTask() {  
        // Une tâche qui pourrait être optimisée  
    }  
}
```

Le profilage avec VisualVM peut montrer que `performTask()` est la plus coûteuse en temps et mérite une optimisation.

# Exercices Pratiques

1. **Installation de VisualVM** : Installez et configurez VisualVM dans votre environnement de développement.
2. **Profiling Pratique** : Profiler une application simple et identifier les goulets d'étranglement.
3. **Heap Analysis** : Réaliser un dump de heap et analyser l'usage mémoire.

Critères : Présentation des résultats, suggestions d'optimisation basées sur les résultats obtenus.

## Synthèse

VisualVM est un outil essentiel pour tout développeur Java cherchant à optimiser les performances applicatives.

En maîtrisant le profilage CPU et l'analyse de mémoire, on peut significativement améliorer l'efficacité d'une application.

Pour aller plus loin, explorez d'autres outils de debugging comme JProfiler ou Java Flight Recorder.

## Introduction

JProfiler est un outil puissant et polyvalent utilisé pour optimiser et déboguer les applications Java.

Il permet d'analyser en profondeur l'utilisation de la mémoire, la performance des threads, et l'efficacité des procédures.

En utilisant JProfiler, vous pouvez identifier les goulets d'étranglement dans votre code et comprendre comment optimiser l'allocation des ressources, ce qui est essentiel pour les applications concurrentes et réactives.

## Objectifs pédagogiques

- Apprendre à installer et configurer JProfiler pour une application Java.
- Analyser la consommation de mémoire et l'utilisation des threads avec JProfiler.
- Identifier et résoudre les goulets d'étranglement en utilisant les fonctionnalités de profilage de JProfiler.
- Optimiser les performances d'une application Java à l'aide de diagnostics précis.

## Installation et configuration

Pour utiliser JProfiler, commencez par télécharger le logiciel depuis le site officiel et suivez les instructions d'installation.

Intégrez-le à votre IDE, tel qu'Eclipse ou IntelliJ IDEA, via des plugins spécifiques.

Pour configurer JProfiler, créez une nouvelle session et associez-la à votre application Java, en définissant les paramètres de l'environnement d'exécution.

## Profilage de la mémoire

JProfiler vous permet de surveiller la consommation de mémoire de votre application.

Il fournit des vues détaillées des allocations d'objets et des fuites de mémoire possibles.

Par exemple, en analysant les 'heap dumps', vous pouvez identifier quelles classes consomment le plus de mémoire, et ajuster votre code pour réduire l'usage excessif.

## Analyse des threads

JProfiler offre une vue complète de tous les threads en cours dans une application.

Vous pouvez surveiller l'état des threads (actif, bloqué, en attente), la durée et les appels associés.

Cette fonctionnalité est cruciale pour comprendre le comportement multitâche et résoudre les problèmes de concurrence, comme les blocages inattendus ou la synchronisation inefficace.

## Optimisation des performances

Utilisez JProfiler pour identifier les principaux goulets d'étranglement en analysant le 'CPU profiling'.

Examinez les méthodes les plus coûteuses en termes de temps d'exécution.

Ce processus vous permet de concentrer vos efforts d'optimisation sur les parties du code qui en bénéficieraient le plus, améliorant ainsi globalement la performance de l'application.

## Exemple : Analyse de mémoire

```
public class MemoryIntensive {  
    public static void main(String[] args) {  
        List<Object> objects = new ArrayList<>();  
        for (int i = 0; i < 10000; i++) {  
            objects.add(new Object());  
        }  
        System.out.println("Object list created.");  
    }  
}
```

Dans cet exemple, nous utilisons JProfiler pour analyser la création de 10 000 objets.

Cela nous aidera à visualiser les allocations et potentiellement identifier un usage inefficace de la mémoire.

# Exemple : Analyse de threads

```
public class ThreadExample {  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(() -> {  
            while (true) {  
                System.out.println("Running");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
        thread1.start();  
    }  
}
```

Ce code crée un thread qui imprime "Running" chaque seconde.

En utilisant JProfiler, nous pouvons examiner l'état et l'impact de ce thread sur les ressources système.

## Exercice : Profilage d'une application

1. Configurez JProfiler pour une application Java de votre choix.
2. Effectuez un profilage de la mémoire et identifiez les classes utilisant le plus de mémoire.
3. Analysez les threads actifs et déterminez s'il existe des blocages.
4. Proposez des optimisations possibles basées sur vos observations.

Critères d'évaluation :

- Compréhension des profils de mémoire et de threads.
- Capacité à identifier et résoudre les problèmes de performance.
- Justification des optimisations suggérées.

## Synthèse

JProfiler est un outil essentiel pour le développement Java, fournissant des informations détaillées sur la performance et l'utilisation des ressources des applications.

Son utilisation compétente permet d'identifier les goulots d'étranglement et d'optimiser le code pour une meilleure performance et une plus grande efficacité. À l'avenir, vous pourriez explorer l'intégration continue de profilage pour une optimisation régulière des applications.

## Introduction

`jcmd` est un outil de diagnostic puissant inclus dans le JDK de Java.

Il permet d'interagir dynamiquement avec les processus Java pour effectuer diverses tâches de gestion, de diagnostic et de contrôle de performance.

Ce cours vous guidera à travers l'utilisation de `jcmd`, vous permettant d'améliorer l'efficacité de vos processus Java.

# Objectifs pédagogiques

- Explorer les fonctionnalités principales de `jcmd`.
- Utiliser `jcmd` pour diagnostiquer des problèmes dans des applications Java.
- Optimiser des applications Java en identifiant des goulets d'étranglement de performance.
- Apprendre à exécuter des commandes pour la collecte de données.

## Qu'est-ce que jcmand ?

`jcmd` est un utilitaire de ligne de commande utilisé pour communiquer avec une JVM active.

Contrairement à d'autres outils de diagnostic, `jcmd` offre une interface unifiée pour émettre des commandes de diagnostic parmi un large éventail.

## Les fonctionnalités de jcmand

`jcmd` fournit un accès à plusieurs types de commandes, telles que :

- L'affichage d'informations sur la configuration de la JVM.
- La collection de profils de pile (thread).
- L'exécution de garbage collection.
- La génération de snapshots de heap.

## Utilisation basique de jcmand

Pour utiliser `jcmd`, commencez par lister tous les processus Java actifs :

```
jcmd
```

Chaque processus a un ID qui peut être utilisé pour envoyer des commandes spécifiques :

```
jcmd <PID> help
```

Cette commande affiche toutes les commandes disponibles pour le processus spécifié.

## Exemple : Lister les threads

Pour lister les threads d'un processus spécifique, utilisez :

```
jcmd <PID> Thread.print
```

Cela vous montrera une liste des threads actifs dans le processus, avec des détails sur chaque pile.

# Exemple : Provoquer Garbage Collection

Voici comment vous pouvez demander à la JVM d'effectuer une collecte des ordures :

```
jcmd <PID> GC.run
```

Cela peut être utile pour libérer de la mémoire immédiatement dans un environnement de test ou de développement.

## Exercice 1 : Utilisation de base

1. Lister les processus Java actifs avec `jcmd`.
2. Choisir un processus (ID).
3. Utiliser `jcmd <PID> help` pour afficher les commandes disponibles.

Critères d'évaluation : Capacité à identifier les processus Java actifs et à explorer les commandes disponibles pour une instance spécifique.

## Exercice 2 : Diagnostic de Thread

1. Utilisez `jcmd` pour lister les threads d'une application Java en cours d'exécution.
2. Identifiez au moins un thread problématique en observant l'état et la pile d'appels.

Critères d'évaluation : Identifier les threads et interpréter les informations obtenues.

## Synthèse

`jcmd` est un outil polyvalent pour gérer et diagnostiquer des applications Java en temps réel.

En maîtrisant `jcmd`, les développeurs peuvent surveiller efficacement les performances, diagnostiquer les goulets d'étranglement et optimiser le comportement de la JVM.

Pour aller plus loin, explorez des cas d'utilisation avancés tels que l'analyse de heap dumps et la détection de deadlocks.

## Introduction

JConsole est un outil de surveillance Java qui permet de superviser et de gérer les performances des applications exécutées sur la machine virtuelle Java (JVM).

Il fournit une interface graphique pour observer divers paramètres comme l'utilisation de la mémoire, le chargement des classes, et l'activité des threads.

JConsole est principalement utilisé pour diagnostiquer les problèmes de performance et pour optimiser les applications Java en temps réel.

## Objectifs pédagogiques

- Comprendre le fonctionnement de JConsole et son utilisation.
- Apprendre à surveiller une application Java avec JConsole.
- Analyser les métriques fournies pour diagnostiquer des problèmes de performance.

- Optimiser l'application à partir des données observées.

## Fonctionnement de JConsole

JConsole est inclus dans le JDK et offre une interface graphique pour la surveillance des applications Java.

Il utilise le protocole JMX (Java Management Extensions) pour collecter des informations sur l'application.

- **Connexion à la JVM** : JConsole peut se connecter à n'importe quelle JVM en cours d'exécution locale ou distante via JMX.
- **Tableaux de bord** : Affiche des graphiques en temps réel pour la surveillance de l'utilisation CPU, de la mémoire, du chargement de classes et de l'activité des threads.
- **Outils de gestion** : Permet de gérer les threads et d'examiner les deadlocks potentiels.

## Surveillance avec JConsole

- **Utilisation mémoire** : Suivi des différents pools de mémoire, y compris l'empreinte du tas et de la pile.
- **Performances des threads** : Affichage de l'état des threads, du nombre de threads actifs et bloqués.
- **Chargement des classes** : Analyse du nombre de classes chargées et déchargées en temps réel.

## Exemples de surveillance

### Exemple 1 : Connexion à une JVM

**Pour se connecter à une JVM locale, démarrez JConsole et sélectionnez l'application cible dans la liste.**

### Exemple 2 : Monitoring de la mémoire

**Ouvrez l'onglet "Memory" pour observer l'historique de l'utilisation du tas et des collectes de déchets.**

### Exemple 3 : Analyse des threads

Dans l'onglet "Threads", vérifiez l'activité et identifiez les threads bloqués pour détecter les deadlocks potentiels.

## Exercices pratiques

### 1. Exercice 1 : Connexion et Surveillance

- Connectez-vous à une application Java locale avec JConsole.
- Identifiez les principales métriques de performance disponibles.

### 2. Exercice 2 : Analyse des performances

- Surveillez l'utilisation de la mémoire pour une application Java causant un niveau élevé de Garbage Collection.
- Proposez des améliorations.

### 3. Exercice 3 : Diagnostics avancés

- Identifiez un thread bloqué et proposez une solution pour résoudre un éventuel deadlock.

Critères d'évaluation :

- Capacité à se connecter correctement à une JVM.
- Aptitude à analyser et interpréter les données de performance.
- Propositions d'optimisation pertinentes.

## Synthèse et ouverture

JConsole est un outil puissant pour le diagnostic et l'optimisation des applications Java.

En surveillant activement les métriques de performance, vous pouvez identifier et résoudre efficacement les goulets d'étranglement.

**Une compréhension approfondie de ces mesures prépare le terrain pour l'utilisation d'outils plus avancés comme Java Flight Recorder pour une analyse plus détaillée des performances.**

## Java Flight Recorder

### Introduction

Java Flight Recorder (JFR) est un outil de profilage intégré dans la JVM qui permet de collecter des données de performance et d'exécution.

JFR offre des capacités avancées pour comprendre le comportement des applications Java en contexte de production avec une faible surcharge.

L'objectif est d'optimiser les performances en identifiant les goulets d'étranglement.

### Objectifs pédagogiques

- Comprendre le fonctionnement de Java Flight Recorder.
- Savoir configurer et démarrer un enregistrement JFR.
- Analyser les enregistrements JFR pour identifier les problèmes de performance.
- Pratiquer l'optimisation basée sur les données collectées.

### Fonctionnement et Architecture

Java Flight Recorder fonctionne en s'intégrant étroitement avec la JVM, étant capable de récupérer une quantité riche de données sur l'exécution et la performance de l'application.

Il utilise une approche d'enregistrement avec des événements qui sont collectés et peuvent être analysés ultérieurement.

- **Événements de la JVM** : Inclut le suivi des threads, l'optimisation du compilateur, les exceptions et l'utilisation de la mémoire.
- **Événements de l'application** : Permet de collecter des informations spécifiques à l'application, telles que le temps passé dans un bloc de code.
- **Surcharge minimale** : Conçu pour fonctionner en production avec un impact minimal sur la performance.

## Configuration et Usage de JFR

Pour utiliser JFR, vous devez généralement passer des options spéciales à la JVM lors du démarrage de votre application.

Cela inclut l'utilisation de différents paramètres pour ajuster les détails capturés par l'enregistrement.

- **Activation de JFR :** `java -XX:StartFlightRecording=name=Profiling,settings=profile,duration=60s -jar app.jar`
- **Paramètres d'enregistrement :** Peut être configuré pour obtenir des détails fins ou une vue d'ensemble selon le besoin.

Avec ces commandes, on peut contrôler la durée, la fréquence d'échantillonnage et bien d'autres aspects de l'enregistrement.

## Analyse des Enregistrements

Une fois le profil capturé, les fichiers JFR peuvent être chargés dans un outil d'analyse comme JMC (Java Mission Control) pour inspecter les données collectées.

Cela permet de visualiser les comportements de l'application et d'identifier les zones nécessitant une attention ou une optimisation.

- **Utilisation de JMC :** Chargez le fichier .jfr pour explorer les threads, la consommation CPU, et les délais de réponse.
- **Détection de goulets d'étranglement :** Repérer facilement où le temps est passé dans votre application et quelles fonctions consomment trop de temps.

## Exemples Pratiques

### 1. Activation Simple :

```
// Démarrage de JFR en ligne de commande  
java -XX:StartFlightRecording=duration=1m,filename=my_recording.jfr -jar MyApp.jar
```

Commence l'enregistrement pendant 1 minute et sauvegarde les données dans `my_recording.jfr`.

### 2. Exemple de Conférence :

Utilisez un profil pour réduire la consommation CPU pendant la collecte en modifiant les paramètres d'échantillonnage.

## Exercices Progressifs

1. **Basique :** Configurez une application simple pour utiliser JFR et capturez un profil de 1 minute.
2. **Analyse Avancée :** Sur un projet de taille moyenne, activez JFR, collectez des données et utilisez JMC pour identifier au moins un potentiel d'optimisation.
3. **Optimisation :** Après l'analyse, effectuez une modification suggérée par les données de JFR et mesurez les améliorations.

Critères de réussite : Identification correcte des goulets d'étranglement et mise en œuvre d'une optimisation mesurable.

## Synthèse

Java Flight Recorder est un outil puissant pour le profilage Java en production, offrant des insights précieux avec un minimum d'impact sur les performances.

L'analyse de ces données peut conduire à des optimisations significatives dans votre application.

Pour aller plus loin, explorez les customisations avancées pour des profils sur mesure et plongez plus profondément dans les analyses offertes par JMC.

# Introduction

Un "heap dump" est un instantané de la mémoire d'un programme Java, capturant l'état de tous les objets présents dans le tas à un moment donné.

Cette opération est essentielle pour analyser les fuites de mémoire, optimiser l'utilisation de la mémoire, et comprendre la structure des objets en cours d'exécution.

Elle est surtout utilisée dans les contextes de debugging et d'optimisation où la gestion de la mémoire est cruciale.

## Objectifs pédagogiques

- Comprendre ce qu'est un heap dump en Java.
- Identifier les outils pour générer et analyser un heap dump.
- Analyser un heap dump pour déceler les fuites de mémoire.
- Optimiser le code Java basé sur les analyses de heap dump.

## Contenu détaillé

Un heap dump est une technique qui permet de capturer une représentation complète de la mémoire d'une application Java à un instant T.

**Il collecte toutes les instances d'objets du heap, leurs références et tailles.**

## Utilisation des Heap Dumps

- **Détection de Fuites de Mémoire** : Identifier des objets orphelins qui ne sont plus nécessaires mais toujours référencés.
- **Optimisation de la Mémoire**: Vérifier l'efficacité de l'utilisation mémoire, améliorer les algorithmes ou structures de données.
- **Debugging**: Isoler et comprendre des anomalies de comportement en cours d'exécution.

## Outils et Techniques

Pour générer un heap dump en Java, plusieurs outils peuvent être utilisés :

- **jcmd** : Permet de créer un heap dump en exécutant `jcmd <PID> GC.heap_dump filename`.
- **jmap** : Utilisé pour générer un dump avec la commande `jmap -dump:format=b,file=<file> <PID>`.
- **VisualVM** : Offre une interface graphique pour générer et analyser les heap dumps.

Chaque outil présente des avantages selon le contexte d'exécution (environnement de production, développement, etc.).

## Exemples de code

### Exemple : Utilisation de jcmd

Pour capturer un heap dump avec jcmd, utilisez la ligne de commande suivante :

```
jcmd 12345 GC.heap_dump /path/to/heapdump.hprof
```

Ici, 12345 est le PID de l'application Java.

**Le fichier .hprof peut ensuite être ouvert avec des outils comme VisualVM pour analyse.**

## Exemple : VisualVM

1. Installer VisualVM.
2. Lancer et connecter à la JVM en cours d'exécution.
3. Générer un heap dump via l'interface graphique pour une analyse intuitive.

# Exercices et Travaux Pratiques

## Exercice 1

**Sujet :** Générer un heap dump d'une application Java en cours d'exécution.

**Instructions :**

- Identifiez un processus Java actif sur votre machine.
- Utilisez jmap pour créer un heap dump.
- Ouvrez le dump dans VisualVM et identifiez les plus gros consommateurs de mémoire.

**Critères d'évaluation :**

- Capacité à générer un heap dump avec succès.
- Analyse correcte de la mémoire pour des fuites potentielles.

## Exercice 2

**Sujet :** Analyse avancée de heap dump.

**Instructions :**

- Chargez un heap dump dans VisualVM.
- Identifiez tous les objets non collectés malgré une absence de référence.

**Critères d'évaluation :**

- Identification précise des objets inappropriés.
- Propositions d'optimisation pertinentes.

# Synthèse

Un heap dump est un outil puissant pour l'analyse mémoire en Java.

Il permet de déceler les fuites de mémoire et de comprendre le comportement de la mémoire d'une application.

**Maîtriser les outils comme jcmd, jmap, et VisualVM est crucial pour optimiser l'usage mémoire. À l'avenir, nous explorerons comment intégrer ces pratiques dans un flux de travail DevOps pour une supervision continue.**

## Thread dump

### Introduction

Un **thread dump** est un outil crucial pour analyser les performances et les problèmes de concurrence dans une application Java.

Il capture l'état actuel de tous les threads de la JVM, ce qui permet de diagnostiquer des blocages ou des performances dégradées.

Cette technique est particulièrement utile dans les scénarios de debugging et d'optimisation des applications concurrentes.

### Objectifs pédagogiques

- Comprendre le concept de thread dump.
- Identifier les situations justifiant un thread dump.
- Analyser un thread dump pour détecter des problèmes.
- Utiliser des outils pour générer et analyser un thread dump.

### Concept de Thread Dump

Un thread dump est une photographie instantanée des threads de la JVM, montrant leur état d'exécution.

Cela inclut les threads en cours, suspendus, ou bloqués.

Il fournit des informations essentielles telles que :

- Le nom des threads.
- L'état (RUNNABLE, BLOCKED, WAITING, etc.).
- La pile d'appel pour chaque thread.

Un thread dump est souvent utilisé pour diagnostiquer :

- Les blocages (deadlocks).
- L'utilisation anormale du CPU.
- Le thread starvation ou les lenteurs.

### Quand Utiliser un Thread Dump

On génère un thread dump dans plusieurs situations clés :

- Lorsque l'application ne répond pas.
- Pour analyser une utilisation excessive du CPU.
- Lors de soupçons de deadlock ou de contention excessive.
- Pour comprendre la répartition du travail entre différents threads.

## Générer un Thread Dump

Avec jcmd :

```
jcmd <pid> Thread.print
```

Avec jstack :

```
jstack <pid>
```

Ces commandes affichent l'état actuel de tous les threads d'une application Java en cours d'exécution. `pid` fait référence à l'identifiant du processus Java.

## Analyser un Thread Dump

Lors de l'analyse d'un thread dump :

1. Identifier les threads avec un long temps de blocage.
2. Surveiller les threads souvent en état BLOCKED ou WAITING.
3. Rechercher des cycles de dépendance entre threads (indique un deadlock possible).
4. Vérifier les threads consommatifs en cycles CPU (indiqués par RUNNABLE).

## Exemple 1 : Détection de Deadlock

Imaginons deux threads, A et B, essayant d'accéder à deux ressources dans un ordre inverse :

- **Thread A** tente de verrouiller la ressource 1 puis 2.
- **Thread B** fait l'inverse.

L'analyse du thread dump révélera une impasse, où chaque thread attend une ressource détenue par l'autre.

## Exemple 2 : Usage Intensif du CPU

Un thread RUNNABLE mais consomme beaucoup de CPU peut causer des problèmes de performance.

- **Thread Dump :** Observons le thread "Worker-Thread-1" :

```
"Worker-Thread-1" #17 prio=5 os_prio=0 tid=0x00007f6df0010800 nid=0x3abf runnable [0x00007f6df8d1a000]
    java.lang.Thread.State: RUNNABLE
        at java.util.HashMap.getNode(HashMap.java:580)
        at java.util.HashMap.get(HashMap.java:556)
```

Cela peut indiquer une boucle interne inefficace.

## Exercices Pratiques

### 1. Capture d'un Thread Dump

- Générez un thread dump de votre application Java en utilisant `jcmd`.

Notez les états des threads RUNNABLE et BLOCKED.

### 2. Analyse de Contention

- Sur un code Java multithreadé fourni, modifiez les sections de code pour éviter la contention de ressources détectée via un thread dump.

### 3. Deadlock Simulation

- Créez un simple programme multithreadé qui introduit un deadlock.

Capturez un thread dump et identifiez les threads et leurs dépendances.

#### 4. Optimisation de Performance

- Modifiez un programme Java dont le thread dump indique un usage élevé des ressources CPU, en ajustant la logique de boucles ou la gestion des ressources.

## Synthèse et Ouverture

Le thread dump est un composant indispensable du debugging et de l'optimisation d'applications Java concurrentes.

En identifiant et résolvant les problèmes de thread, on peut significativement améliorer les performances. À l'avenir, explorez comment les thread dumps s'intègrent avec d'autres outils de surveillance pour créer une image complète de la santé de votre application.

## Introduction

La détection de deadlocks est cruciale en programmation concurrente.

Un deadlock se produit lorsque deux ou plusieurs threads attendent indéfiniment des ressources verrouillées par chacun.

Comprendre comment identifier et résoudre un deadlock améliore la fiabilité et la performance des applications Java multithreadées.

## Objectifs pédagogiques

- Comprendre les causes courantes des deadlocks.
- Savoir utiliser des outils pour détecter les deadlocks.
- Acquérir des compétences pratiques pour éviter et résoudre les deadlocks.
- Appliquer des solutions de conception pour prévenir les deadlocks.

## Deadlocks en Java

Un deadlock se produit lorsqu'un thread attend indéfiniment une ressource verrouillée par un autre, qui attend à son tour une ressource verrouillée par le premier.

Les deadlocks peuvent survenir dans des systèmes multithreadés où des ressources limitées sont partagées entre plusieurs threads.

- **Causes** : ordonnancement non déterministe, ressources partagées, verrouillages circulaires.
- **Conséquence** : Les threads arrêtent de progresser, bloquant l'exécution du programme.

## Détection des deadlocks

- **Analyse de l'état des threads** : Utilisez des outils Java pour vérifier les états des threads en cours d'exécution.
- **Utilisation de thread dump** : Capture un snapshot des états des threads et des ressources pour détecter les cycles ou l'attente de ressources.

Les outils comme `jconsole`, `VisualVM` et `jcmt` peuvent être utilisés pour analyser et visualiser les deadlocks.

# Java Thread Dump

Un thread dump peut être généré en utilisant l'outil `jstack` qui fournit des informations sur chaque thread de la JVM.

Le dump identifie les threads en attente de ressources qui ne se libèrent pas, indiquant potentiellement la présence d'un deadlock.

Exécuter `jstack` :

```
jstack <process_id>
```

Cherchez les états comme `BLOCKED` et vérifiez les blocages circulaires.

## Exemple de Deadlock Java

Considérons le code suivant mettant en scène deux threads accédant à deux ressources.

```
class Resource {
    void metA(Resource b) {
        synchronized(this) {
            b.metB();
        }
    }

    synchronized void metB() {
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        final Resource res1 = new Resource();
        final Resource res2 = new Resource();

        Thread t1 = new Thread(() -> res1.metA(res2));
        Thread t2 = new Thread(() -> res2.metA(res1));

        t1.start();
        t2.start();
    }
}
```

Dans ce scénario, `metA` et `metB` provoquent un deadlock lorsque `t2` verrouille `res1` et attend `res2`, pendant que `t1` verrouille `res2` et attend `res1`.

## Exercices pratiques

### 1. Exercice de détection de deadlock :

- En utilisant le code précédemment présenté, introduisez un mécanisme pour détecter le deadlock avec `jstack`.

### 2. Exercice de résolution de deadlock :

- Modifiez l'ordre de verrouillage dans le code pour prévenir le deadlock. Évaluez votre solution avec des tests.

### Critères d'évaluation :

- Capacité à générer et analyser les threads dumps.
- Conception d'une solution sans deadlock.

# Synthèse

Comprendre les deadlocks et savoir les détecter est essentiel pour garantir la stabilité des applications Java.

En implémentant des solutions comme l'évitement de verrouillages circulaires et en utilisant des outils de surveillance, vous pouvez réduire la probabilité de deadlocks.

Continuons à explorer d'autres techniques avancées de debugging et d'optimisation pour améliorer les performances de nos systèmes.

## Introduction

L'exercice "Identifier et Corriger un Deadlock Multithreadé" vous plongera dans les complexités de la programmation concurrente en Java.

Un deadlock est une situation où deux threads ou plus se bloquent éternellement, chacun attendant que l'autre libère des ressources.

Comprendre et résoudre ces deadlocks est crucial pour garantir que les applications concurrentes fonctionnent correctement et efficacement.

## Objectifs pédagogiques

- Analyser du code multithreadé pour détecter des deadlocks
- Appliquer des techniques de debugging pour identifier les causes des deadlocks
- Implémenter des solutions pour prévenir et résoudre les deadlocks
- Évaluer le fonctionnement correct du code multithreadé après correction

## Comprendre les Deadlocks

Un deadlock survient lorsque deux threads ou plus tentent de verrouiller des ressources dans un ordre opposé, provoquant un blocage perpétuel.

En Java, cela implique généralement les objets `synchronized`.

Trouver et résoudre ces boucles d'attente est essentiel pour fiabiliser le code.

## Approches pour la Détection

1. **Thread Dump Analysis** : Utilisation d'outils comme `jconsole` ou `VisualVM` pour analyser les dumps de thread et identifier les deadlocks.
2. **Logs** : Ajouter des journaux pour suivre l'acquisition et la libération des verrous.
3. **Java Concurrency Utilities** : Utiliser des outils de diagnostic intégrés dans Java.

## Prévention et Résolution

- **Ordonnancement Consistant des Verrous** : Structurer le code pour acquérir les verrous dans un ordre cohérent.
- **Time-out sur les Verrous** : Utiliser `tryLock` avec un délai pour éviter l'attente infinie.
- **Thread Hierarchy** : Implémenter une hiérarchie stricte pour l'acquisition des ressources.

# Exemple de Code Annoté

```
public class DeadlockExample {  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
  
    public void method1() {  
        synchronized (lock1) {  
            // Simulating work  
            Thread.sleep(50);  
            synchronized (lock2) {  
                // Critical section  
            }  
        }  
    }  
  
    public void method2() {  
        synchronized (lock2) {  
            // Simulating work  
            Thread.sleep(50);  
            synchronized (lock1) {  
                // Critical section  
            }  
        }  
    }  
}
```

Les méthodes ci-dessus démontrent un potentiel deadlock dû à l'ordre opposé des verrous.

# Solution Proposée

```
public class DeadlockResolved {  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
  
    public void safeMethod() {  
        synchronized (lock1) {  
            synchronized (lock2) {  
                // Critical section  
            }  
        }  
    }  
}
```

En modifiant l'ordre de verrouillage, ici consistant, nous évitons le deadlock.

# Exercices Pratiques

1. **Identifier le deadlock** : Prenez l'exemple initial et simulez-le avec plusieurs threads.

Utilisez jconsole pour identifier le deadlock.

2. **Résoudre le deadlock** : Modifiez le code pour corriger le deadlock, assurez-vous de tester l'absence de blocages avec des tests multithreadés.
3. **Refactoring** : Refactorez le code pour implémenter des ReentrantLock avec timeout.

# Synthèse

Les deadlocks constituent un défi courant dans la programmation concurrente, mais avec des techniques de diagnostic appropriées et des bonnes pratiques de code, ils peuvent être résolus efficacement.

Porter attention aux ordres de verrouillage et adopter des techniques de prévention comme l'utilisation de `tryLock` avec délais peut drastiquement réduire les risques de deadlocks dans vos applications Java.

## Ouverture

Explorez des outils avancés de debugging et de profilage, tels que Java Flight Recorder ou JProfiler, pour détecter des deadlocks et autres problèmes de performance.

Ces compétences étendront vos capacités à maintenir et optimiser des applications Java concurrentes à grande échelle.

## Introduction à Debugging et optimisation

Dans le cadre de la programmation concurrente et réactive, le debugging et l'optimisation jouent un rôle crucial pour garantir la performance et la fiabilité des applications Java.

Ce module explorera divers outils et techniques essentiels pour identifier et résoudre les problèmes de performance et de concurrence.

## Vue d'ensemble des sous-notions

Ce cours couvrira des outils tels que VisualVM, JProfiler, jcmd, et jconsole, ainsi que des méthodes comme les Heap et Thread dumps et Java Flight Recorder.

**Nous aborderons également la détection de deadlocks et pratiquerons l'identification et la correction de deadlocks multithreadés.**

## Introduction

La programmation concurrente et réactive est un concept essentiel en informatique, particulièrement en Java.

Ce domaine permet l'exécution simultanée de plusieurs tâches, améliorant ainsi la réactivité et l'efficacité des applications.

Ce cours introduira trois sous-notions clés : la Concurrence moderne, la Programmation réactive, et le Debugging et optimisation, chacune étant fondamentale pour le développement d'applications robustes et performantes.

# Vue d'ensemble

1. **Concurrence moderne** : Explorez CompletableFuture, les chaînages asynchrones, et les outils de gestion des threads comme ForkJoinPool et ExecutorService.

Apprenez à utiliser synchronized, ReentrantLock, et les variables atomiques.

2. **Programmation réactive** : Découvrez les Reactive Streams, la gestion du backpressure, et les bibliothèques comme Project Reactor et RxJava pour créer des systèmes réactifs.

3. **Debugging et optimisation** : Maîtrisez des outils comme VisualVM, JProfiler, et Java Flight Recorder pour optimiser et débugger des applications multithreadées, et apprenez à détecter et corriger les problèmes comme les deadlocks.

## Introduction

La syntaxe de Java sert de fondement à la compréhension de la programmation dans ce langage.

En maîtrisant la syntaxe de base, vous pourrez écrire, lire et déboguer du code Java efficacement.

Ce module introduit des éléments syntaxiques clés qui facilitent le développement de programmes structurés et orientés objet.

## Objectifs pédagogiques

- Identifier les structures syntaxiques de base en Java
- Écrire du code Java simple et structuré
- Comprendre l'organisation d'un programme Java
- Appliquer les conventions de nommage standard

## Bases de la syntaxe Java

La syntaxe Java repose sur quelques principes fondamentaux :

- Chaque programme commence par une classe avec une méthode `main`.
- Les instructions se terminent par un point-virgule `;`.
- Les blocs de code sont délimités par des accolades `{}`.

Un programme simple suit généralement cette structure.

Voici l'exemple d'un programme Java basique.

## Exemple de programme simple

Premier programme en Java :

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

- `public class HelloWorld` : Définit une classe publique nommée `HelloWorld`.
- `public static void main(String[] args)` : Point d'entrée du programme.
- `System.out.println()` :.Invoke une méthode de sortie pour afficher du texte.

## Variables et Types de Données

Java est un langage fortement typé.

Les variables doivent être déclarées avant utilisation.

- Types primitifs : `int`, `double`, `char`, `boolean`
- Types de référence : `String`, `Array`, `Object`

Déclaration d'une variable :

```
int age = 25;
```

Ce code crée une variable entière `age` initialisée à 25.

## Structures de Contrôle

Java utilise des structures de contrôle pour orienter le flux d'exécution :

- Conditionnelles : `if`, `else`, `switch`
- Boucles : `for`, `while`, `do-while`

Exemple d'une structure conditionnelle :

```
if (age >= 18) {  
    System.out.println("Adult");  
} else {  
    System.out.println("Minor");  
}
```

Le bloc `if` exécute une instruction basée sur la condition.

## Exemples de code

Manipuler les structures de contrôle basiques :

Exemple de boucle `for` :

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

Exemple de `switch` :

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    default:
        System.out.println("Other day");
}
```

## Exercices Pratiques

- Déclarer et utiliser des variables :** Écrire un programme qui déclare trois variables de types différents et manipule leurs valeurs.
- Implémenter des structures conditionnelles :** Écrire un programme qui utilise une instruction `if-else` pour vérifier si un nombre est pair ou impair.
- Boucles et itérations :** Créer un programme qui génère et imprime les 10 premiers nombres de Fibonacci.

**Critères d'évaluation :** Code fonctionnel et lisible, utilisation correcte des structures syntaxiques.

## Synthèse et Ouverture

La maîtrise de la syntaxe Java est cruciale pour progresser dans la programmation orientée objet.

Ces bases vous permettent de structurer efficacement vos programmes.

En avançant, nous explorerons des concepts plus avancés tels que les classes et les objets, améliorant ainsi la modularité et la réutilisabilité du code.

## Introduction

Dans cette section, nous allons explorer le concept de "Classes" en Java, qui est un élément fondamental de la programmation orientée objet.

Les classes constituent le modèle à partir duquel les objets sont créés.

Elles définissent les propriétés et les comportements d'un objet.

## Objectifs pédagogiques

- Comprendre le rôle des classes dans la programmation Java
- Savoir créer et utiliser des classes en Java
- Apprendre à définir des attributs et méthodes au sein d'une classe
- Maîtriser les concepts de constructeur et d'instanciation

# Les classes en Java

Les classes en Java constituent la pierre angulaire de la programmation orientée objet.

Elles permettent de créer des types de données personnalisés qui encapsulent des données et des méthodes.

- **Définition** : Une classe est un modèle définissant les propriétés (attributs) et comportements (méthodes) d'un objet.
- **Structure** : Une classe se compose généralement de variables d'instance, de méthodes et d'un ou plusieurs constructeurs.

Un exemple simple de classe :

```
public class Voiture {  
    String couleur;  
    String marque;  
  
    public void démarrer() {  
        System.out.println("La voiture démarre.");  
    }  
}
```

## Attributs d'une classe

Les attributs, ou variables d'instance, dans une classe définissent l'état des objets créés à partir de cette classe.

- **Déclaration** : Les attributs sont déclarés à l'intérieur de la classe, mais en dehors de toute méthode.
- **Types** : Les attributs peuvent être de n'importe quel type de données supporté par Java (int, String, boolean, etc.).

Exemple d'attributs pour la classe Voiture :

```
String couleur;  
String marque;
```

## Méthodes d'une classe

Les méthodes définissent le comportement d'une classe.

Elles permettent d'effectuer des actions en utilisant les attributs de la classe ou de fournir un service à d'autres classes.

- **Syntaxe** : Les méthodes ont une syntaxe similaire aux fonctions, comprenant un type de retour, un nom, et des paramètres optionnels.
- **Visibilité** : Une méthode peut être publique, privée, ou protégée, affectant son accessibilité depuis d'autres classes.

Exemple de méthode pour la classe Voiture :

```
public void démarrer() {  
    System.out.println("La voiture démarre.");  
}
```

## Constructeurs en Java

Les constructeurs sont des méthodes spéciales utilisées pour initialiser les objets.

Ils portent le même nom que la classe et n'ont pas de type de retour.

- **Par défaut** : Java fournit un constructeur par défaut si aucun n'est explicitement défini.
- **Personnalisé** : Un constructeur peut être personnalisé pour initialiser des attributs avec des valeurs spécifiques.

Exemple de constructeur :

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

## Exemple de classe complète

Voyons un exemple de classe complète et fonctionnelle, avec attributs, méthodes et constructeur :

```
public class Voiture {  
    String couleur;  
    String marque;  
  
    // Constructeur  
    public Voiture(String couleur, String marque) {  
        this.couleur = couleur;  
        this.marque = marque;  
    }  
  
    // Méthode pour démarrer la voiture  
    public void démarrer() {  
        System.out.println("La voiture de couleur " + couleur + " démarre.");  
    }  
}
```

## Exercices pratiques

1. **Créer une classe** : Créez une classe `Animal` avec des attributs `nom` et `age`, une méthode `manger()`, et un constructeur pour initialiser les attributs.
2. **Instancier des objets** : Utilisez la classe `Animal` pour créer plusieurs objets avec différents noms et âges.

Appelez les méthodes pour démontrer leur fonctionnement.

3. **Étendre la classe** : Ajoutez une méthode `dormir()` à la classe `Animal` et testez son intégration.

Évaluation : Vérifiez si les objets sont correctement créés et si les méthodes produisent l'output attendu.

## Synthèse

Les classes en Java sont essentielles pour structurer et organiser le code dans un style orienté objet.

Elles favorisent la réutilisation et l'extensibilité du code via l'encapsulation de données et de comportements spécifiques.

En maîtrisant les classes, vous acquérez une compétence clé pour développer des applications robustes et bien organisées en Java.

**Dans les prochaines leçons, nous explorerons comment les classes interagissent avec d'autres concepts tels que l'héritage et le polymorphisme.**

# Objets

## Introduction

Dans le paradigme de programmation orientée objet (POO), les objets sont au cœur de la modularité et de la réutilisabilité du code.

En Java, un objet est une instance d'une classe et encapsule des états et comportements.

Cette section explore en détail comment manipuler et utiliser des objets pour structurer efficacement vos programmes.

## Objectifs pédagogiques

- Décrire le concept d'objet en POO.
- Créer et manipuler des objets en Java.
- Utiliser des méthodes d'instance pour interagir avec les objets.
- Comprendre la différence entre une classe et un objet.

## Les objets en Java

Les objets en Java sont créés à partir de classes, qui définissent les propriétés (attributs) et actions possibles (méthodes) de ces objets.

Un objet représente une instance concrète de ces définitions, permettant l'exécution de comportement spécifique à cette instance.

### 1. Création d'objets : Utilisation du mot-clé new .

- Associez une classe à une variable pour créer un objet.
- Exemple : `MonObjet obj = new MonObjet();`

### 2. Attributs et méthodes :

- Les attributs définissent l'état de l'objet.
- Les méthodes définissent les comportements.

## Exemple de classe et d'objet

Considérons une classe Voiture :

```
public class Voiture {  
    String marque;  
    int vitesse;  
  
    public Voiture(String marque) {  
        this.marque = marque;  
        this.vitesse = 0;  
    }  
  
    public void accelerer(int increment) {  
        this.vitesse += increment;  
    }  
}
```

Créons un objet de cette classe :

```
Voiture maVoiture = new Voiture("Toyota");
maVoiture.accelerer(50);
```

- L'objet `maVoiture` est une instance de `Voiture` avec un état unique.
- La méthode `accelerer` modifie cet état.

## Exercices pratiques

### 1. Création et manipulation :

- Créez une classe `Animal` avec des attributs `nom` et `age`.
- Instanciez un objet de cette classe et imprimez ses attributs.

### 2. Interagir avec des méthodes :

- Ajoutez une méthode `vieillir` à la classe `Animal` qui incrémente l'`age`.
- Testez cette méthode sur votre instance.

### Critères d'évaluation :

- Vérification de la création correcte des objets,
- Utilisation appropriée des méthodes pour modifier l'état des objets.

## Synthèse

Les objets sont essentiels pour structurer la logique en programmation Java.

En encapsulant l'état et le comportement, ils simplifient la gestion et la manipulation des données.

Maîtriser leur création et utilisation est crucial pour développer des applications Java robustes.

La prochaine étape pourrait inclure l'exploration des patrons de conception pour améliorer encore davantage la réutilisation du code et l'organisation des objets.

## Introduction

Les interfaces en Java sont des ensembles de méthodes abstraites qui définissent un comportement que les classes peuvent implémenter.

Elles jouent un rôle clé dans l'abstraction et la programmation orientée objet en aidant à définir des contrats sans implémentation détaillée, promouvant ainsi la flexibilité et la réutilisabilité du code.

## Objectifs pédagogiques

- Comprendre le concept et l'usage des interfaces en Java
- Distinguer entre interfaces et classes abstraites
- Implémenter des interfaces dans les classes
- Concevoir des architectures modulaires utilisant des interfaces

# Qu'est-ce qu'une Interface ?

Une interface en Java est un type de référence similaire à une classe, mais qui ne peut contenir que des variables finales (constantes) et des méthodes abstraites.

Depuis Java 8, les interfaces peuvent également contenir des méthodes par défaut et statiques, qui fournissent une implémentation que les classes peuvent utiliser.

## Rôle des Interfaces

Les interfaces permettent d'atteindre l'abstraction et la séparation des contrats d'implémentation.

Elles définissent un ensemble de méthodes qu'une classe doit implémenter, ce qui permet à différentes classes d'implémenter la même interface tout en fournissant leurs propres implémentations des méthodes.

## Syntaxe d'une Interface

Une interface est déclarée avec le mot clé `interface` :

```
public interface Animal {  
    void makeSound();  
    void move();  
}
```

Dans cet exemple, toute classe implémentant `Animal` doit fournir des implémentations concrètes pour `makeSound` et `move`.

## Implémentation d'une Interface

Pour qu'une classe implémente une interface, elle utilise le mot clé `implements` :

```
public class Dog implements Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof");  
    }  
  
    @Override  
    public void move() {  
        System.out.println("Runs");  
    }  
}
```

La classe `Dog` implémente l'interface `Animal` et fournit des implémentations concrètes des deux méthodes.

## Interfaces vs Classes Abstraites

Contrairement aux interfaces, les classes abstraites peuvent avoir des méthodes avec implémentation et des variables d'instance.

Cependant, elles ne permettent pas la "multiple inheritance" qu'une interface peut simuler en permettant la classe à implémenter plusieurs interfaces.

# Exemples d'utilisation

Les interfaces sont souvent utilisées dans les API pour fournir une flexibilité maximale.

Par exemple, `List`, `Map`, et `Set` dans le package `java.util` sont des interfaces, laissant aux classes comme `ArrayList`, `HashMap`, et `HashSet` l'implémentation des détails internes.

## Exemples de Code Annotés

Imaginez que vous concevez une application pour un zoo :

```
interface Flyer {  
    void fly();  
}  
  
class Bird implements Flyer {  
    @Override  
    public void fly() {  
        System.out.println("Flies in the sky");  
    }  
}  
  
class Plane implements Flyer {  
    @Override  
    public void fly() {  
        System.out.println("Flies using engines");  
    }  
}
```

Ici, `Bird` et `Plane` implémentent tous deux l'interface `Flyer`, illustrant l'abstraction du vol.

## Exercices Pratiques

**1. Crédit d'Interfaces :** Créez une interface `Vehicle` avec des méthodes `startEngine` et `stopEngine`.

Implémentez cette interface dans deux classes : `Car` et `Boat`.

**2. Interface Avancée :** Ajoutez une méthode par défaut à `Vehicle` qui imprime `Checking vehicle status`.

Implémentez-la dans vos classes précédemment créées.

**Critères d'évaluation :** Votre solution doit démontrer une compréhension de l'abstraction, la réutilisation de code, et l'utilisation correcte des interfaces et méthodes par défaut.

## Synthèse et Ouverture

Les interfaces jouent un rôle majeur dans la conception de Java, fournissant une manière de garantir qu'un ensemble de classes implémente certaines méthodes.

**Cela permet de concevoir des systèmes modulaires et extensibles. À l'avenir, explorez comment les interfaces fonctionnent en conjonction avec les lambdas et les streams en Java 8 pour tirer parti davantage des interfaces fonctionnelles.**

## Introduction

Dans cette section dédiée au langage Java, nous allons revoir les fondations essentielles pour bien comprendre la programmation orientée objet.

Les sous-notions abordées incluront la syntaxe de base, la structure de classes et d'objets, ainsi que les concepts clés comme les interfaces, l'héritage et le polymorphisme.

Nous explorerons également des principes de programmation importants tels que les principes SOLID, DRY et KISS pour écrire un code propre et maintenable.

## Vue d'ensemble des sous-notions

Voici comment les différentes sous-notions s'articulent :

- **Syntaxe, Classes, Objets** : Les bases de la programmation orientée objet.
- **Interfaces, Héritage, Polymorphisme** : Constructions avancées pour la réutilisation et l'extensibilité du code.
- **Encapsulation, Surcharge et redéfinition, Portée des variables** : Contrôle de l'accès et des opérations sur les données.
- **Principes SOLID, DRY, KISS** : Bonnes pratiques pour la qualité du code.
- **Exercice pratique** pour mettre en application les principes SOLID.

## Fondations et rappels

### Introduction aux Fondations

Dans ce module, nous explorerons les fondations essentielles du développement en Java.

Nous aborderons les bases du langage avec ses structures et principes clés.

Nous examinerons également les APIs standards qui sont indispensables au développement Java moderne.

Enfin, nous découvrirons les principes de la programmation fonctionnelle introduits avec Java 8.

## Vue d'ensemble des sous-notions

1. **Rappel des bases du langage** : Comprendre comment les concepts fondamentaux de Java comme les classes, objets et interfaces s'articulent.
2. **APIs standards** : Étudier les collections et les APIs I/O pour une gestion efficace des données.
3. **Programmation fonctionnelle (Java 8)** : Explorer les lambdas, Streams et autres innovations apportées par Java 8.

# Modularité avec Java 9

## Introduction

Avec Java 9, le projet Jigsaw a introduit un système de modularité révolutionnaire.

Ce système permet d'organiser le code Java en modules distincts, améliorant la gestion et l'évolutivité des applications.

La modularité offre des avantages comme une encapsulation stricte et une meilleure gestion des dépendances.

## Objectifs pédagogiques

- Comprendre les concepts de base de la modularité en Java 9.
- Apprendre à créer et gérer des modules Java.
- Savoir identifier et résoudre les problèmes courants de dépendance.
- Maîtriser les outils et fonctionnalités fournis par Java pour la modularité.

## Concept de Modularité

La modularité permet de diviser une application en unités distinctes nommées *modules*.

Chaque module contient des *packages* et des *ressources*, et définit de manière explicite quels autres modules il utilise et quels packages il exporte.

- **Module** : Conteneur logique de packages.
- **Module Descriptor (`module-info.java`)** : Fichier de déclaration des modules.
- **Exports** : Déclaration des packages accessibles aux autres modules.
- **Requires** : Spécifie les modules dont le module actuel dépend.

## Création d'un Module

Pour créer un module, il faut un fichier `module-info.java` à la racine du répertoire du module.

```
module com.example.myapp {  
    requires java.base;  
    exports com.example.myapp.api;  
}
```

- `module com.example.myapp` : Définit le nom unique du module.
- `requires java.base` : Import implicite du module de base Java.
- `exports com.example.myapp.api` : Rend le package accessible aux autres modules.

## Avantages de la Modularité

- **Encapsulation stricte** : Seuls les packages explicitement exportés sont accessibles.
- **Clarté des dépendances** : Les dépendances entre modules sont clairement spécifiées.
- **Chargement dynamique** : Permet à l'application de ne charger que les modules nécessaires à l'exécution.

Ces avantages permettent de construire des applications plus robustes et maintenables, avec un démarrage plus rapide, car seuls les modules nécessaires sont chargés en mémoire.

## Exemple 1 : Module Basique

Voyons un exemple simple où un module `com.example.app` utilise un autre module `com.example.utils`.

- `com.example.app/module-info.java`

```
module com.example.app {  
    requires com.example.utils;  
}
```

- `com.example.utils/module-info.java`

```
module com.example.utils {  
    exports com.example.utils;  
}
```

Ce setup montre comment un module client déclare ses dépendances et comment un module fournisseur expose ses fonctionnalités.

## Exemple 2 : Résolution de Conflits

Supposons que `com.example.utils` et `com.example.logging` exposent chacun un utilitaire nommé `Logger`, menant à un conflit.

- Utilisez `requires` avec `as` pour résoudre le conflit :

```
module com.example.app {  
    requires com.example.utils;  
    requires com.example.logging as log;  
}
```

En spécifiant `as`, il est possible de différencier et d'utiliser les deux modules en conflit.

## Exercices Pratiques

1. **Créer un module simple** : Mettez en place un module `com.example.library` qui exporte une classe `Book`.
  - Vérifiez qu'une classe `BookApp` dans `com.example.app` puisse utiliser `Book`.
2. **Gestion des dépendances** : Ajoutez un module `com.example.network` dépendant de `com.example.library`.
  - Modifiez le module de la bibliothèque pour qu'il nécessite le module `java.net.http`.

Critères de succès : Les modules doivent compiler et s'exécuter correctement, en assurant une séparation des packages et une gestion des dépendances nette.

## Synthèse

Le système de modularité introduit avec Java 9 permet une organisation claire et optimisée du code Java.

En utilisant les modules efficacement, nous bénéficiions d'une meilleure encapsulation et gestion des dépendances.

La maîtrise de ces concepts ouvre la voie à la création d'applications plus robustes et évolutives, préparant le terrain pour les versions futures de Java.

# Introduction

L'inférence de type local introduite avec Java 10 permet de simplifier le code en remplaçant explicitement le type d'une variable par le mot-clé `var`.

Cela favorise la lisibilité et la concision du code, tout en améliorant la capacité du développeur à se concentrer sur la logique plutôt que sur la syntaxe de type.

## Objectifs pédagogiques

- Comprendre l'usage de l'inférence de type local avec `var`.
- Appliquer `var` pour simplifier le code Java existant.
- Identifier les situations adaptées et inadaptées pour l'utilisation de `var`.
- Évaluer les impacts de l'inférence de type sur la lisibilité et la maintenance du code Java.

## Qu'est-ce que l'inférence de type ?

L'inférence de type en Java 10 introduit `var` pour permettre au compilateur de déterminer le type de la variable.

Par conséquent, vous n'avez pas besoin de déclarer explicitement le type, ce qui simplifie le code.

Bien que `var` ne soit utilisable que dans des contextes locaux, il conserve la sécurité de type fournie par le compilateur.

## Cas d'utilisation appropriés

`var` est idéal lorsque le type est évident à partir du contexte, par exemple :

- Lorsque le type est dupliqué dans l'initialisation (`Map<String, List<Integer>> map = new HashMap<>()`).
- Avec des déclarations locales de `for` améliorées.
- Initialisation avec des méthodes dont le type de retour est évident.

## Limitations et précautions

Même si `var` simplifie le code, il peut nuire à la lisibilité quand le type n'est pas évident.

Il convient de l'éviter :

- Lorsque le type n'est pas évident.
- En cas de déclaration non initialisée.
- Pour des déclarations de champs ou de méthodes, `var` étant limité aux variables locales.

## Exemples pratiques

Déclaration explicite :

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

Avec var :

```
var scores = new HashMap<String, Integer>();
```

Autre exemple :

```
var list = List.of("Java", "Kotlin", "Scala");
```

Comme vous le voyez, var simplifie significativement la déclaration surtout lorsque le type est complexe.

## Exercices pratiques

### 1. Simplification de code :

Transformez les déclarations explicites en déclarations avec var :

```
ArrayList<String> names = new ArrayList<>();
HashMap<Integer, String> idMap = new HashMap<>();
```

### 2. Discernement :

Identifiez le cas où l'utilisation de var n'est pas appropriée dans le code suivant et justifiez votre choix :

```
var calculator = getCalculator();
var value; // Déclaration inappropriée
```

## Synthèse et ouverture

L'inférence de type local avec var est un ajout puissant de Java 10 qui augmente la lisibilité et la concision du code tout en conservant une forte sécurité de type.

En avançant, explorez comment d'autres qualités améliorées du langage dans les versions ultérieures peuvent construire sur ces bases pour rendre Java encore plus expressif et convivial.

## Introduction

Les expressions switch introduites dans Java 12 représentent une évolution significative par rapport au switch traditionnel.

Elles offrent une syntaxe plus concise et expressive, améliorant la lisibilité et réduisant le risque d'erreurs, telles que l'oubli des break.

Cette nouvelle fonctionnalité contribue à rendre le code Java plus moderne et fonctionnel.

## Objectifs pédagogiques

- Comprendre et utiliser les expressions switch.
- Différencier les expressions switch des instructions switch traditionnelles.
- Implémenter des expressions switch pour simplifier le code.
- Analyser les avantages en termes de lisibilité et sécurité.

# Syntaxe des expressions switch

Les expressions switch en Java 12 se présentent avec une syntaxe compacte.

Contrairement aux switch classiques, elles peuvent retourner une valeur.

L'utilisation de "case" et des flèches "->" permet de définir les actions.

```
int numLetters = switch (day) {  
    case "MONDAY", "FRIDAY", "SUNDAY" -> 6;  
    case "TUESDAY" -> 7;  
    case "THURSDAY", "SATURDAY" -> 8;  
    case "WEDNESDAY" -> 9;  
    default -> day.length();  
};
```

## Différences avec le switch classique

Les expressions switch éliminent le besoin de break.

Elles permettent d'attribuer le résultat directement à une variable.

Cela réduit le risque d'oublier un break, prévenant ainsi les "fall-through" non désirés.

## Avantages des expressions switch

1. **Lisibilité accrue** : Syntaxe plus claire, surtout pour les cas nombreux.
2. **Sécurité accrue** : Moins de risque d'erreur d'exécution liée au manque de break.
3. **Expressivité** : Possibilité de retourner une valeur, facilitant l'affectation.

## Exemples pratiques

### Exemple 1 : Calcul de jours

```
String dayType = switch (day) {  
    case "SATURDAY", "SUNDAY" -> "Weekend";  
    default -> {  
        System.out.println("It's a weekday");  
        yield "Weekday";  
    };  
};
```

## Exemple 2 : Calcul basé sur l'âge

```
int category = switch (age) {  
    case 0 -> "Infant";  
    case 1, 2 -> "Toddler";  
    default -> "Older";  
};
```

## Exercices pratiques

### Exercice 1 : Catégorisation

**Écrivez une expression switch pour catégoriser les mois en "Hiver", "Printemps", "Été", "Automne".**

### Exercice 2 : Jour de la semaine

**Créez une expression switch qui détermine si un jour est en semaine ou le weekend.**

#### Critères d'évaluation

- Utiliser la nouvelle syntaxe avec flèches.
- Retourner une valeur basé sur les cases.

## Synthèse

Les expressions switch modernisent Java en offrant une syntaxe simplifiée et sécurisée.

Elles favorisent un code plus propre et expressif.

L'apprentissage de cette fonctionnalité est crucial pour écrire du code Java moderne et efficace, préparant le terrain pour un usage avancé dans les prochaines éditions de Java.

## Introduction

Le Garbage Collector G1 (Garbage-First) est introduit pour améliorer la gestion de la mémoire dans les applications Java.

Conçu pour remplacer le traditionnel CMS (Concurrent Mark-Sweep), G1 cible les pauses courtes et prévisibles, une valeur essentielle pour les applications nécessitant une haute performance et une gestion efficace de la mémoire.

## Objectifs pédagogiques

- Comprendre le rôle et le fonctionnement du Garbage Collector G1.

- Identifier les avantages et limites du GC G1 par rapport à d'autres collecteurs.
- Configurer G1 dans une application Java.
- Analyser et optimiser la performance avec le GC G1.

## Fonctionnement du GC G1

Le GC G1 fonctionne en divisant la mémoire en plusieurs régions, par opposition à un format contigu.

Lors du processus de collection, G1 cible d'abord les régions avec le plus de déchets (récupération de mémoire mort), d'où son nom "Garbage-First".

Cela permet une gestion plus efficace et moins intrusive de la mémoire.

Avantages :

- Régions indépendantes permettent des pauses plus courtes.
- Priorisation des régions maximise l'efficacité de la collection.
- Offre des options de tuning pour adapter la performance aux besoins spécifiques des applications.

## Configurer le GC G1

Pour activer le GC G1, ajoutez l'option suivante lors de l'exécution de votre application Java :

```
java -XX:+UseG1GC -jar MonApplication.jar
```

Principaux paramètres de configuration :

- `-XX:MaxGCPauseMillis` : Objectif de durée maximale pour les pauses de garbage collection.
- `-XX:InitiatingHeapOccupancyPercent` : Seuil de déclenchement de la collection basée sur le pourcentage d'occupation du tas.
- `-XX:G1HeapRegionSize` : Taille des régions dans le tas, réglable selon les besoins de l'application.

## Exemples de configuration

Pour un ajustement spécifique, considérer les besoins de production :

```
java -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:InitiatingHeapOccupancyPercent=45 -XX:G1HeapRegionSize=8M -jar MonApplication.jar
```

Cet exemple configure G1 pour des pauses maximales de 200ms, avec un déclenchement à 45% d'occupation du tas et une taille de région de 8Mo pour équilibrer performances et utilisation mémoire.

## Exercices pratiques

1. **Configuration Basique:** Activez le GC G1 dans une application Java et observez les résultats.
2. **Optimisation Avancée:** Ajustez les paramètres de `MaxGCPauseMillis` et `InitiatingHeapOccupancyPercent`, et évaluez l'impact sur la performance.

Pour chaque exercice, documentez les configurations appliquées et les résultats observés en termes de réduction de pauses et de performance globale.

# Synthèse

Le Garbage Collector G1 est une avancée significative pour gérer les besoins de mémoires intenses et les pauses minimales. À travers sa gestion des régions et son approche priorisée, il répond aux attentes des applications modernes. À l'avenir, explorez des techniques avancées de tuning pour perfectionner encore la performance de votre application Java.

## Introduction

L'évolution de Java entre les versions 8 et 12 a introduit des améliorations significatives.

Ces versions ont favorisé la modularité du code grâce à Project Jigsaw (Java 9), simplifié la syntaxe avec l'inférence de type local (var, Java 10), et amélioré le contrôle du flux avec les expressions switch (Java 12).

## Vue d'ensemble

Les avancées de Java 8 à 12 incluent l'intégration de nouveaux Garbage Collectors (G1 et ZGC), et invitent à explorer la modularité via un exercice pratique de création d'un mini-module Java avec dépendances internes.

## Introduction

L'évolution du langage Java de la version 8 à la 21 marque des avancées significatives en syntaxe, modularité et gestion des ressources.

Ce parcours sera exploré à travers trois phases clés : Java 8 à 12, Java 13 à 17, et Java 18 à 21. Chaque segment introduit des fonctionnalités transformatrices façonnant le développement moderne en Java.

## Vue d'ensemble des évolutions

- **Java 8 à 12** : Introduction de la modularité avec Project Jigsaw et de nouvelles expressions avec `switch`.
- **Java 13 à 17** : Amélioration de la manipulation de texte avec Text Blocks et introduction de `Records` pour simplifier les données immuables.
- **Java 18 à 21** : Focus sur la performance et la concurrence avec les Virtual Threads et de nouveaux patterns.

Ces étapes démontrent les efforts continus de Java pour répondre aux besoins changeants des développeurs.

## Introduction

Les types génériques bornés en Java sont essentiels pour créer des classes et des méthodes flexibles tout en garantissant la sécurité.

Ils permettent de restreindre les types pouvant être utilisés avec un paramètre générique, grâce aux mots-clés `extends` et `super`.

Cette fonctionnalité est cruciale pour écrire du code réutilisable et robuste, tout en évitant les erreurs au moment de la compilation.

# Objectifs pédagogiques

- Comprendre et utiliser les bornes supérieures et inférieures en Java.
- Appliquer les concepts de `extends` et `super` dans des classes et méthodes génériques.
- Maîtriser l'écriture de code générique réutilisable tout en assurant la sécurité de type.

## Bornes supérieures avec `extends`

L'utilisation de `extends` dans les types génériques permet de définir une "borne supérieure".

Cela limite les types pouvant être utilisés à une classe spécifique ou à ses sous-classes.

- Syntaxe : `<T extends ClassName>`.
- Utile pour implémenter des interfaces ou des classes abstraites.

Exemple : Les collections qui acceptent des objets de types homogènes.

## Bornes inférieures avec `super`

Le mot-clé `super` est utilisé pour définir une "borne inférieure".

Cela contraint l'utilisation d'un certain type ou de ses superclasses.

- Syntaxe : `<T super ClassName>`.
- Pratique pour consommer des instances d'un type ou de ses sous-types dans les méthodes.

Exemple : Algorithme de sorting qui fonctionne avec un `Comparator`.

## Exemples de `extends`

```
class Box<T extends Number> {  
    private T t;  
  
    public void set(T t) {  
        this.t = t;  
    }  
  
    public T get() {  
        return t;  
    }  
}  
  
Box<Integer> integerBox = new Box<>();  
Box<Double> doubleBox = new Box<>();
```

- Ici, `Box` ne peut accepter que des types qui sont ou dérivent de `Number`.

# Exemples de super

```
public static void addNumbers(List<? super Integer> list) {  
    list.add(new Integer(42));  
}
```

- Cette méthode accepte des listes dont le type de base est `Integer` ou toute superclasse de `Integer`, permettant l'ajout d'entiers.

# Travaux pratiques

1. Créer une classe `ShapeBox<T extends Shape>` pouvant stocker des types dérivés de `Shape`.
  - Implémentez une méthode pour calculer l'aire totale des formes contenues.
2. Écrire une méthode générique `public static void copy(List<? super T> dest, List<T> src)` pour copier tous les éléments de `src` à `dest`.

Consignes : Utiliser les bornes génériques pour manager la sécurité des types.

# Synthèse

Les types génériques bornés permettent de restreindre les paramètres tout en augmentant la flexibilité. `extends` offre des bornes supérieures, restreignant les types à une hiérarchie spécifique, tandis que `super` facilite l'opération sur des hiérarchies de superclasses.

Ces concepts sont vitaux pour l'écriture de code générique performant et maintenable.

Pour aller plus loin, explorez les motifs de conception qui tirent parti des génériques en combinaison avec les collections.

# Introduction aux Wildcards

Les wildcards en Java sont une fonctionnalité puissante qui permet de rendre vos programmes plus flexibles lorsqu'il s'agit de travailler avec des collections génériques.

Ils servent à définir des limites de compatibilité dans les types paramétrisés.

En comprenant et utilisant les wildcards, vous pouvez manipuler des collections plus diversifiées tout en maintenant la sécurité de type.

# Objectifs pédagogiques

- Comprendre le concept de wildcards en Java
- Savoir utiliser les wildcards avec `? extends` et `? super`
- Identifier quand utiliser les wildcards pour la flexibilité du code
- Manipuler des collections génériques avec des wildcards

# Concept de Wildcards

Les wildcards en Java sont représentés par le symbole `?`.

Ils permettent de réaliser des opérations polymorphiques sur des collections.

Les wildcards sont utiles pour traiter des objets d'un type inconnu dans un contexte de typage paramétré et consistent en deux principaux types :

- **Upper Bounded Wildcards ( ? extends T )** : Permet de lire des objets de la collection en garantissant qu'ils sont d'un certain type ou sous-type.
- **Lower Bounded Wildcards ( ? super T )** : Permet d'écrire des objets dans la collection tout en garantissant qu'ils sont d'un certain type ou supertype.

## Upper Bounded Wildcards ( ? extends T )

Avec `? extends T`, vous pouvez lire de la collection, mais ne pas y écrire.

Cela signifie que vous pouvez effectuer des opérations qui nécessitent la lecture d'éléments, en vous assurant que l'élément est un type T ou un sous-type de T.

Par exemple, cela peut être utilisé pour calculer la somme d'une liste de nombres où la liste peut être composée d'Integer, Double, ou n'importe quelle sous-classe de Number.

## Lower Bounded Wildcards ( ? super T )

À l'inverse, `? super T` permet d'ajouter des éléments à la collection, garantissant seulement qu'ils sont d'un type T ou d'un supertype de T.

Cette contrainte est utile pour implémenter des algorithmes de consommation de données sur une collection.

Par exemple, si vous voulez ajouter des instances de Integer à une collection de Numbers, vous pouvez déclarer la collection avec `? super Integer`.

## Exemple avec Upper Bounded Wildcards

```
public static double sumOfList(List<? extends Number> list) {  
    double sum = 0.0;  
    for (Number n : list) {  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

Dans cet exemple, `List<? extends Number>` autorise la méthode à accepter des List de Integer, Double, etc., mais vous ne pouvez pas ajouter dans cette liste.

## Exemple avec Lower Bounded Wildcards

```
public static void addElements(List<? super Integer> list) {  
    list.add(1);  
    list.add(2);  
    list.add(3);  
}
```

Cet exemple montre `List<? super Integer>`, qui permet d'ajouter des entiers à la collection.

Les wildcards garantissent que le type dans la liste est Integer ou un supertype, comme Number ou Object.

# Exercice Pratique

1. **Exercice 1 :** Écrivez une méthode qui prend en paramètre une `List<? extends Animal>` et affiche le son spécifique de chaque animal.
  - **Indication :** Utilisez la classe de base `Animal` avec une méthode `makeSound()`.
2. **Exercice 2 :** Créez une méthode capable d'ajouter plusieurs types d'éléments (par exemple, des `Integer` et des `Double`) dans une `List<? super Number>`.
  - **Critères d'évaluation :** Vérifiez que la méthode fonctionne avec différents types qui sont des sous-classes de `Number`.

## Synthèse et Ouverture

Les wildcards améliorent la flexibilité et l'interopérabilité des méthodes génériques en Java.

En maîtrisant `? extends` et `? super`, vous pouvez écrire des fonctions plus dynamiques et compatibilité-tolérantes.

La prochaine étape pour approfondir vos compétences avancées en Java consiste à explorer comment combiner différentes structures de données ou utiliser des patterns de design modernes avec la puissance de la généréricité.

## Introduction

La notion de `Comparable` en Java joue un rôle crucial dans la gestion des collections triées.

Elle permet à un objet de se comparer à d'autres objets pour définir un ordre naturel, facilitant ainsi des opérations comme le tri et la recherche.

Comprendre `Comparable` offre une meilleure prise en main des collections Java.

## Objectifs pédagogiques

- Comprendre le rôle de l'interface `Comparable`.
- Implémenter `Comparable` dans ses classes Java.
- Utiliser le tri naturel dans les collections Java.
- Analyser les différences entre `Comparable` et `Comparator`.

## Contenu détaillé

### Comprendre Comparable

L'interface `Comparable` est une interface générique de Java utilisée pour imposer un ordre naturel sur les objets d'une classe.

Elle contient une méthode unique : `compareTo`.

Cette méthode compare l'objet courant à un autre objet du même type.

## Méthode compareTo

La signature de la méthode compareTo est `int compareTo(T o)`.

Elle doit retourner :

- Un nombre négatif si l'objet courant est inférieur à l'objet spécifié.
- Zéro si l'objet courant est égal à l'objet spécifié.
- Un nombre positif si l'objet courant est supérieur à l'objet spécifié.

## Implémentation pratique

### Exemple simple

Supposons une classe `Person` avec un attribut `name`.

Nous souhaitons trier des objets `Person` par nom :

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

    public Person(String name) {
        this.name = name;
    }

    @Override
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }

    // Getter, Setter et autres méthodes
}
```

### Exemple avec tri

Une liste de personnes peut être triée facilement grâce à `Comparable` :

```
List<Person> people = Arrays.asList(new Person("Alice"), new Person("Bob"), new Person("Charlie"));
Collections.sort(people);
// La liste est maintenant triée par ordre alphabétique
```

## Comparaison avec Comparator

Le `Comparator` est une interface fonctionnelle utilisée pour définir plusieurs façons de comparer deux objets.

Contrairement à `Comparable`, `Comparator` peut être implémenté séparément de la classe des objets à comparer.

Il est souvent utilisé lorsque l'ordre d'un objet ne peut pas être défini dans la classe elle-même ou si plusieurs ordres sont nécessaires.

# Exercices pratiques

## Exercice 1 : Implémenter Comparable

Créez une classe `Book` avec des attributs `title` et `author`.

**Implémentez Comparable pour trier par title .**

### Critères d'évaluation

- La méthode `compareTo` doit comparer correctement les `title`.
- Vérifiez le bon tri avec une collection de `Book`.

## Exercice 2 : Comparaison multiple

**Associer un Comparator pour trier d'abord par author , puis par title si les auteurs sont identiques.**

### Critères d'évaluation

- Implémentation correcte de `Comparator`.
- Vérifiez le tri avec une collection appropriée.

## Synthèse

La maîtrise de `Comparable` et `Comparator` permet une flexibilité et une efficacité accrues dans la manipulation des collections triées en Java.

Tandis que `Comparable` définit un ordre naturel, `Comparator` offre la possibilité de multiples classements.

**Pour aller plus loin, explorez l'intégration de ces concepts avec les nouvelles fonctionnalités des Streams pour obtenir un code toujours plus performant et élégant.**

## Comparator

### Introduction

Le `Comparator` en Java est une interface fonctionnelle qui permet de définir un ordre de tri personnalisé pour des collections d'objets.

Contrairement à `Comparable`, qui impose un ordre naturel, `Comparator` offre une flexibilité pour trier les objets différemment selon les besoins.

Cette fonctionnalité est essentielle pour manipuler efficacement des collections avec des critères de tri spécifiques.

## Objectifs pédagogiques

- Comprendre l'interface Comparator.
- Appliquer Comparator pour trier des collections.
- Différencier Comparator de Comparable.
- Implémenter des tris personnalisés en Java.
- Utiliser des méthodes de référence pour simplifier le code.

## Contenu détaillé

### Interface Comparator :

- Comparator est une interface fonctionnelle introduite dans Java 1.2.
- Utilisée pour définir l'ordre relatif de deux objets.
- Contient une méthode unique `compare(T o1, T o2)`.

### Déférence avec Comparable :

- Comparable impose un ordre naturel (implémenté par l'objet).
- Comparator permet des ordres multiples (implémenté indépendamment).

### Méthodes clés :

- `compare` : Compare deux objets pour définir leur ordre.
- `reversed` : Inverse l'ordre du tri.
- `thenComparing` : Chaine plusieurs critères de tri.

## Contenu détaillé (suite)

### Avantages de Comparator :

- Flexibilité pour multiples critères de tri sans modifier les objets.
- Facilité de réutilisation pour différents types de tri.

### Implémentation :

- Peut être implémenté en tant que classe anonyme ou via expression lambda.
- Compatible avec les méthodes de tri des collections (`sort`, `sorted`).

## Exemples de code

```
// Exercice de tri des chaînes par longueur
Comparator<String> lengthComparator = (s1, s2) -> Integer.compare(s1.length(), s2.length());

// Exercice de tri des personnes par âge
Comparator<Person> ageComparator = Comparator.comparingInt(Person::getAge);

List<Person> people = Arrays.asList(new Person("Alice", 30), new Person("Bob", 25));
Collections.sort(people, ageComparator);
```

## Exemple annoté

```
// Exemple avec classe anonyme
Comparator<String> cmp = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
};

// Utilisation avec lambda
Comparator<String> cmpLambda = (s1, s2) -> s1.compareTo(s2);
```

- L'utilisation de lambdas réduit le code accidentel.
- `compareTo` est utilisé pour comparer les objets String eux-mêmes.

## Exercices pratiques

### Exercice 1 :

- Implémentez un Comparator pour trier une liste de produits par prix.

### Exercice 2 :

- Tri des étudiants par notes, puis par ordre alphabétique croissant.

### Critères d'évaluation :

- Fonctionnalité de tri respectée.
- Utilisation adéquate de Comparator.
- Simplicité et lisibilité du code.

## Synthèse

Le Comparator offre une solution puissante pour le tri sur mesure des collections Java.

Sa flexibilité facilite la gestion de cas complexes où plusieurs critères sont envisagés.

La compréhension et l'utilisation efficace des Comparators permettent de manipuler les données de manière précise et intégrée dans l'écosystème Java.

**Pour les prochaines étapes, nous explorerons les autres fonctionnalités avancées des collections Java, telles que Streams et Collectors.**

## Tri personnalisé

## Introduction

Dans ce module, nous explorerons le tri personnalisé en Java, une fonction clé lorsque l'ordre naturel d'une collection d'objets ne suffit pas.

En utilisant les interfaces `Comparable` et `Comparator`, nous pouvons définir des critères de tri personnalisés pour répondre aux besoins spécifiques de nos applications Java.

# Objectifs pédagogiques

- Comprendre et appliquer l'interface `Comparable`.
- Utiliser l'interface `Comparator` pour des tris personnalisés.
- Maîtriser le tri de collections avec des critères multiples.
- Implémenter des solutions de tri pratiques et optimales en Java.

## Usage de Comparable

L'interface `Comparable` est utilisée pour définir l'ordre naturel d'une classe donnée.

En implémentant la méthode `compareTo()`, un objet peut être comparé à un autre objet du même type.

Cela est utile pour les tris simples où une seule logique de comparaison est nécessaire.

## Utilisation de Comparator

Le `Comparator` est une interface fonctionnelle qui offre plus de flexibilité qu'une implémentation `Comparable`.

Vous pouvez créer plusieurs comparateurs pour une seule classe, permettant des critères de tri variés selon les besoins spécifiques. `Comparator` est idéal pour des conditions complexes ou multiples.

## Définitions de Comparateurs

`Comparator` permet de définir des comparaisons par fonctionnalités lambda ou par des classes anonymes.

Cela permet de créer un comparateur à la volée, augmentant la flexibilité et réduisant le besoin de code API supplémentaire.

Une approche commune est d'utiliser les méthodes statiques comme `Comparator.comparing()`.

## Exemple de code : Comparable

Considérons une classe `Person` :

```
public class Person implements Comparable<Person> {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    @Override  
    public int compareTo(Person other) {  
        return Integer.compare(this.age, other.age);  
    }  
}
```

Ce code trie les `Person` par âge.

## Exemple de code : Comparator

Pour une approche différente, imaginons un tri par nom :

```
Comparator<Person> nameComparator = new Comparator<Person>() {  
    @Override  
    public int compare(Person p1, Person p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
};
```

Ou avec une expression lambda pour la même tâche :

```
Comparator<Person> nameComparatorLambda =  
    Comparator.comparing(Person::getName);
```

## Exemple Combiné

Nous pouvons combiner plusieurs `Comparators` pour comparer par nom d'abord, puis par âge :

```
Comparator<Person> combinedComparator =  
    Comparator.comparing(Person::getName)  
        .thenComparing(Person::getAge);
```

Cette combinaison est puissante pour des tris multicritères.

## Exercices pratiques

### Exercice 1 : Tri simple

- Créez une liste de personnes et triez-la en utilisant `Comparable`.

### Exercice 2 : Tri multiple

- Implémentez un `Comparator` pour trier la même liste d'abord par âge, puis par nom.

## Critères d'évaluation

- Respect du contrat des interfaces.
- Correctitude et lisibilité du code.

## Synthèse

Nous avons vu comment personnaliser le tri en Java grâce aux interfaces `Comparable` et `Comparator`.

Ces outils permettent de répondre à des besoins de tri spécifiques et complexes, offrant ainsi une grande flexibilité dans la gestion des collections.

Dans les modules futurs, nous aborderons d'autres aspects avancés des collections et la programmation fonctionnelle avec Java Streams.

## Introduction

La notion de Stream parallèle en Java est essentielle pour exploiter la puissance des processeurs multi-cœurs.

Les Streams parallèles permettent de traiter les collections de données de manière concurrente, améliorant ainsi les performances des opérations lourdes.

Cette approche est particulièrement efficace pour les tâches de calcul intensif.

## Objectifs pédagogiques

- Comprendre le concept de Stream parallèle en Java.
- Apprendre à créer et manipuler des Streams parallèles.
- Identifier les avantages et les inconvénients des Streams parallèles.
- Écrire et exécuter des programmes Java exploitant les Streams parallèles de manière efficace.

## Concept de Stream parallèle

Les Streams en Java, introduits avec Java 8, facilitent le traitement des collections.

Les Streams parallèles partitionnent automatiquement le flux en sous-flux traités en parallèle.

- Utilisation du framework Fork/Join pour le parallélisme.
- Bénéficie immédiatement des architectures multi-cœurs.
- Favorise un code concis et lisible pour les opérations en parallèle.

## Avantages et Inconvénients

Les Streams parallèles accélèrent les traitements mais peuvent introduire de la complexité :

- **Avantages** : Traitement plus rapide grâce au parallélisme.
- **Inconvénients** : Overhead de la gestion de threads, complexité du débogage.

L'usage inapproprié des Streams parallèles peut causer des problèmes de performance.

## Créer un Stream Parallèle

Pour créer un Stream parallèle, on utilise la méthode `parallelStream()` sur une collection ou `stream().parallel()`.

Voici comment convertir un Stream séquentiel en parallèle :

```
List<String> list = Arrays.asList("a", "b", "c");
list.parallelStream().forEach(System.out::println);
```

L'exécution des éléments se fait en parallèle, l'ordre de sortie n'est pas garanti.

# Exemples de code

Considérons le calcul de la somme des carrés des nombres d'une liste en utilisant un Stream parallèle :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sumOfSquares = numbers.parallelStream()
    .mapToInt(n -> n * n)
    .sum();
System.out.println("Sum of squares: " + sumOfSquares);
```

Cet exemple calcule la somme des carrés de manière efficace grâce au parallélisme.

## Exercices pratiques

1. **Exercice 1 :** Utilisez un Stream parallèle pour filtrer et compter les mots d'un fichier texte ayant plus de 5 lettres.
  - Critère : Utiliser les méthodes `filter()` et `count()`.
  - Critères d'évaluation : Exactitude du résultat, bonne utilisation du parallélisme.
2. **Exercice 2 :** Écrire un programme pour multiplier simultanément les éléments d'un tableau d'entiers par un facteur donné en utilisant Streams parallèles.
  - Critères d'évaluation : Efficacité du programme, code clair et concise.

## Synthèse et ouverture

Les Streams parallèles offrent une manière puissante de traiter les données de manière concurrente, tirant parti des architectures multi-coeurs.

Cependant, il est crucial de comprendre où et comment les utiliser pour éviter les pièges potentiels liés à la gestion des threads.

En poursuivant l'exploration des fonctionnalités modernes de Java, intégrer les Collectors pour regrouper et modifier les résultats pourrait être la prochaine étape dans l'apprentissage approfondi des Streams.

## Introduction

Le `Collectors.groupingBy` en Java est une fonctionnalité puissante du Stream API qui permet de regrouper des éléments en fonction d'une clé donnée.

Cela offre une flexibilité considérable pour manipuler et analyser des collections de données, rendant le traitement des données plus clair et efficace.

## Objectifs pédagogiques

- Comprendre le concept de regroupement avec `groupingBy`.
- Appliquer `Collectors.groupingBy` à des collections de données en Java.
- Utiliser des Collectors complémentaires pour des agrégations avancées.

# Contenu détaillé

Le `Collectors.groupingBy` est utilisé pour collecter les éléments d'un Stream en une Map, où les clés sont les résultats de l'application d'une fonction de classification et les valeurs sont des listes d'objets de l'élément de Stream.

- `groupingBy` est utile pour créer des sous-groupes au sein d'une collection.
- Utilisation typique : transformation d'un Stream d'objets en une Map structurée.

Le `Collectors.groupingBy` supporte plusieurs formes :

1. `Collectors.groupingBy(classifier)` .
2. `Collectors.groupingBy(classifier, downstream)` .
3. `Collectors.groupingBy(classifier, supplier, downstream)` .

Chaque forme augmente en complexité et en fonctionnalités.

## Contenu détaillé (suite)

1. **Basic Grouping** : Regroupement basique avec une clé simple.
2. **Complex Grouping** : Utilisation de `downstream collector` pour effectuer des opérations supplémentaires sur les groupes.
3. **Performance Tuning** : Choix d'une `supplier` pour un Map personnalisé afin de contrôler l'implémentation sous-jacente.

Les classes fréquemment utilisées incluent `Collectors`, `Stream` et `Function`.

Le concept de lambdas simplifie souvent l'utilisation.

## Exemples de code

### Exemple de Grouping Basique

```
List<String> items = Arrays.asList("apple", "banana", "orange", "watermelon");
Map<Integer, List<String>> lengthMap = items.stream()
    .collect(Collectors.groupingBy(String::length));
System.out.println(lengthMap);
```

Cet exemple regroupe des chaînes de caractères par la longueur.

### Exemple de Grouping Complexé

```
Map<Integer, Set<String>> lengthMap = items.stream()
    .collect(Collectors.groupingBy(String::length, Collectors.toSet()));
System.out.println(lengthMap);
```

Ici, nous utilisons `toSet` pour éviter les doublons au sein des groupes.

## Exemple pratique

## Regroupement avec une Opération de Comptage

```
List<String> items = Arrays.asList("apple", "banana", "orange", "apple", "orange", "watermelon");
Map<String, Long> countMap = items.stream()
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
System.out.println(countMap);
```

Ce code compte la fréquence des éléments dans la liste initiale, regroupant par identité.

## Exercices pratiques

1. Implémentez un regroupement de personnes par longueur de nom à partir d'une liste d'objets `Person`.
2. Modifiez pour utiliser un `Collector` supplémentaire qui concatène les noms.
3. Évaluez la performance en utilisant un `supplier` pour une Map dédiée de haute performance.

Critères d'évaluation : Utilisation correcte des méthodes, efficacité du code, et clarté de la solution.

## Synthèse et ouverture

En conclusion, `Collectors.groupingBy` est un outil essentiel pour organiser des données complexes en sous-groupes logiques.

Il améliore la lisibilité et l'efficacité du code Java moderne.

Pour aller plus loin, explorez comment `groupingBy` peut être combiné avec d'autres opérations de Stream pour réaliser des analyses de données encore plus poussées.

## Collectors.mapping

## Introduction

Dans le cadre de l'utilisation des streams en Java, `Collectors.mapping` est une fonction puissante qui permet de transformer les éléments d'un Stream lors de la phase de collecte.

Elle est particulièrement utile pour appliquer une fonction de mapping sur les éléments collectés, facilitant ainsi l'extraction et la transformation des données dans les collections.

## Objectifs pédagogiques

- Comprendre le rôle de `Collectors.mapping` dans les flux de données.
- Appliquer `Collectors.mapping` pour transformer et collecter des données.
- Mettre en œuvre des cas concrets d'utilisation de `Collectors.mapping`.
- Comparer `Collectors.mapping` avec d'autres méthodes de collecte.

# Utilisation de Collectors.mapping

Collectors.mapping est utilisé avec collect pour transformer un stream.

Le mapping applique une fonction sur chaque élément avant de collecter les résultats.

Par exemple, vous pouvez extraire et collecter uniquement certains attributs d'objets.

- **Syntaxe principale:** Utilise Collectors.mapping conjointement avec une clé de regroupement pour transformer des données dans un Map .
- **Pourquoi l'utiliser ?** Simplifie la transformation et la collecte de données en une seule opération fluide.
- **Nature des transformations:** Permet des opérations comme la conversion de types, l'extraction de propriétés ou la transformation en chaînes.

## Exemples de code

### Exemple de base

Supposons que nous ayons une liste d'objets Person et que nous souhaitons collecter tous les noms en majuscules :

```
List<Person> people = ...;
Map<String, List<String>> namesByCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity,
        Collectors.mapping(person -> person.getName().toUpperCase(),
        Collectors.toList())));
```

Ici, Collectors.mapping transforme les noms en majuscules avant de les collecter dans une liste par ville.

### Comparaison avec d'autres méthodes

#### Sans Collectors.mapping

```
List<String> names = people.stream()
    .map(person -> person.getName().toUpperCase())
    .collect(Collectors.toList());
```

#### Avec Collectors.mapping

Permet une intégration directe dans des collectes plus complexes, comme le regroupement.

## Exercices pratiques

## Exercice 1

**Objectif:** Utiliser `Collectors.mapping` pour extraire et formater des données.

- Créez une liste d'objets `Book` avec des attributs `title` et `author`.
- Utilisez `Collectors.mapping` pour collecter les titres de tous les livres en minuscules, regroupés par année de publication.

## Exercice 2

**Objectif:** Intégrer `Collectors.mapping` dans un processus de transformation.

- Modifiez l'exercice précédent pour inclure à la fois la transformation des titres et l'ajout d'une longueur de chaque titre après le nom de l'auteur.

**Critères d'évaluation:**

- Correction des résultats : les titres doivent être en minuscules et groupés correctement.
- Utilisation efficace de `Collectors.mapping`.

## Synthèse et perspectives

Le `Collectors.mapping` élargit les possibilités de transformation lors de la collecte des streams en Java, apportant flexibilité et précision aux opérations de traitement des données.

Il s'intègre bien dans des chaînes de traitement plus complexes, surtout lors de l'utilisation conjointe avec d'autres collecteurs.

En pratique, maîtriser cette fonctionnalité permet d'écrire un code plus concis et performant.

Dans la suite de vos apprentissages, explorez comment `Collectors.mapping` s'intègre avec d'autres opérations liées aux streams pour augmenter l'efficacité de vos manipulations de données.

## Introduction

Le `Collectors.teeing` est une fonctionnalité avancée introduite dans Java 12, qui enrichit les capacités de traitement des collections avec les Streams.

Elle permet de combiner les résultats de deux collectors distincts en un résultat final, apportant une grande flexibilité et puissance dans le traitement des données.

Cette fonctionnalité est particulièrement utile lorsque vous devez effectuer deux opérations distinctes sur un même Stream et réunir les résultats.

## Objectifs pédagogiques

- Comprendre le concept de `Collectors.teeing`.
- Apprendre à créer des pipelines de Stream complexes.
- Savoir combiner les résultats de plusieurs opérations sur un même Stream.
- Pouvoir implémenter du code Java efficace avec `Collectors.teeing`.

# Fonctionnement du Collectors.teeing

Le `Collectors.teeing` prend trois arguments : deux Collectors et une fonction de combinaison.

Chaque Collector traite les éléments du Stream et la fonction combine leurs résultats.

Cela permet d'extraire des données de manière flexible et efficace.

Il nécessite, cependant, d'avoir une bonne compréhension des Collectors et de la manière dont les Streams fonctionnent dans Java.

## Pédagogie et Utilité

Utiliser `Collectors.teeing` évite le traitement multiple de données en permettant de réaliser deux opérations en parallèle sur le même Stream.

Cela conduit non seulement à un code plus lisible et maintenable mais aussi potentiellement à des gains de performance.

## Exemple : Somme et Moyenne

Considérons un Stream d'entiers; nous souhaitons calculer la somme et la moyenne des nombres en une seule passe.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
double result = numbers.stream()
    .collect(Collectors.teeing(
        Collectors.summingInt(Integer::intValue),
        Collectors.averagingInt(Integer::intValue),
        (sum, avg) -> sum + avg));
```

Dans cet exemple, `Collectors.summingInt` calcule la somme et `Collectors.averagingInt` calcule la moyenne.

La fonction de combinaison associe les deux résultats.

## Exemple : Min et Max

Vous pouvez également utiliser `Collectors.teeing` pour obtenir simultanément le minimum et le maximum d'un ensemble de données.

```
List<Integer> numbers = List.of(23, 4, 15, 42, 8);
String result = numbers.stream()
    .collect(Collectors.teeing(
        Collectors.minBy(Integer::compareTo),
        Collectors.maxBy(Integer::compareTo),
        (min, max) -> "Min: " + min.orElseThrow() + ", Max: " + max.orElseThrow()));
```

Ici, `Collectors.minBy` et `Collectors.maxBy` identifient respectivement le minimum et le maximum, que la fonction de combinaison réunit.

## Exercice Pratique

1. **Objectif :** Utiliser `Collectors.teeing` pour déterminer le nombre total de mots et la longueur moyenne des mots dans un texte.

2. **Consignes :**

- Créez un Stream à partir d'une liste de mots.

- Utilisez `Collectors.teeing` pour obtenir le total de mots et leur longueur moyenne.
- Affichez les résultats combinés dans une phrase bien structurée.

**Critères d'évaluation :** Vérification de l'utilisation correcte de `Collectors.teeing`, et de l'exactitude des résultats.

## Synthèse

Le `Collectors.teeing` est un outil puissant dans Java pour traiter des streams en parallèle et combiner les résultats sans devoir retraverser les données.

Grâce à sa flexibilité et ses capacités de combinaison, `Collectors.teeing` aide à simplifier le code tout en améliorant potentiellement la performance.

Cela ouvre la voie à des pratiques de traitement de données plus efficaces dans des applications Java avancées.

## Exercice : Trier et Grouper un Dataset Complexé

### Introduction

Dans cet exercice, nous explorerons les techniques avancées de Java pour manipuler un dataset complexe en utilisant des structures de données modernes et efficaces.

Nous nous concentrerons sur la capacité à trier et regrouper des données grâce à l'utilisation de comparateurs et des API de flux de Java.

Cette compétence est cruciale pour travailler avec de grandes quantités de données de manière performante et structurée.

### Objectifs Pédagogiques

- Comprendre et utiliser `Comparator` pour le tri personnalisé.
- Appliquer les opérations de regroupement avec `Collectors.groupingBy`.
- Manipuler les données avec les Streams Java pour améliorer la lisibilité et la performance du code.
- Concevoir des solutions modulaires permettant une extension facile des comportements de tri et de regroupement.

### Contenu Détailé

Dans cet exercice, nous allons d'abord revoir comment utiliser des `Comparators` en Java.

Ces objets permettent de définir des ordres personnalisés pour n'importe quel type d'objet.

En parallèle, vous apprendrez à utiliser des collecteurs permettant un regroupement efficace des données.

Nous appliquerons ces concepts afin de résoudre des cas concrets de manipulation de données.

# Comparators et Trie

Un `Comparator` permet de définir un ordre personnalisé de tri.

En Java, vous pouvez implémenter l'interface `Comparator` et la méthode `compare`.

Considérez cet exemple où nous trions une liste de personnes par nom et âge :

```
List<Person> people = ...;
people.sort(Comparator.comparing(Person::getName).thenComparing(Person::getAge));
```

Dans cet exemple, nous utilisons `Comparator.comparing` avec une méthode de référence pour trier d'abord par nom, puis par âge.

# Utilisation des Streams et GroupingBy

Les Streams en Java permettent un traitement fluide et efficace des collections de données. `Collectors.groupingBy` est particulièrement utile pour regrouper des objets sous certaines propriétés.

Par exemple, pour regrouper une liste de transactions par type :

```
Map<String, List<Transaction>> transactionsByType =
    transactions.stream()
        .collect(Collectors.groupingBy(Transaction::getType));
```

Ce code regroupe toutes les transactions par leur type en utilisant un seul appel fluide de méthode.

# Exemples Pratiques

Essayons un exemple simple pour solidifier la compréhension.

Considérez une classe `Produit` avec des attributs pour le nom et le prix.

Nous trierons une liste de produits par prix de manière décroissante et les regrouperons par catégorie :

```
List<Produit> produits = ...;
Map<String, List<Produit>> produitsParCategorie = produits.stream()
    .sorted(Comparator.comparing(Produit:: getPrix).reversed())
    .collect(Collectors.groupingBy(Produit:: getCategorie));
```

Ce code montre comment combiner tri et regroupement pour organiser des produits par prix décroissant par catégorie.

# Exercices Progressifs

1. **Tri de Personnes** : Donnez une liste d'objets `Personne`, triez la liste par nom puis par âge.
2. **Regroupement de Commandes** : Pour des objets `Commande`, groupez-les par statut (e.g., livré, en attente) et après, triez chaque groupe par date de commande.
3. **Analyse de Données** : En partant d'un dataset de `Ventes`, regroupez-les par région et triez chaque groupe de manière décroissante par le montant de la vente.

Critères d'évaluation : efficacité du code et clarté des solutions proposées.

# Synthèse

Cet exercice a permis de découvrir comment trier et regrouper efficacement des datasets complexes en Java grâce aux `Comparators` et à l'API Stream.

Vous devriez maintenant être à l'aise pour appliquer ces techniques à vos propres projets, améliorant ainsi la performance et la lisibilité de votre code.

**Pour aller plus loin, explorez les opérations additionnelles proposées par les Streams et investiguez des cas d'usage réels de gestion de données volumineuses.**

## Introduction

Dans le cadre de la programmation Java avancée, l'utilisation de la généricité et des collections joue un rôle crucial.

Ce module explore diverses techniques et concepts pour manipuler les structures de données de manière flexible et efficace.

Nous approfondirons des notions telles que les types génériques bornés, les wildcards, et les interfaces `Comparable` et `Comparator`.

De plus, nous aborderons les techniques de tri personnalisé et l'utilisation de flux parallèles pour optimiser les traitements de données.

## Vue d'ensemble

Les sous-notions se décomposent en plusieurs aspects :

- **Types génériques bornés** : Restriction des types pour assurer la sécurité de type.
- **Wildcards** : Introduction de flexibilité dans les méthodes génériques.
- **Comparable et Comparator** : Interfaces cruciales pour le tri d'objets.
- **Tri personnalisé** : Personnalisation des critères de tri.
- **Stream parallèle** : Amélioration des performances de traitement.
- **Collectors** : Utilisation avancée pour regrouper et transformer les données (groupingBy, mapping, teeing).
- **Exercice pratique** : Applications réelles pour structurer et manipuler de vastes datasets de manière efficace.

## Introduction

Les Records en Java, introduits dans la version 14, simplifient la création de classes immuables.

Ils réduisent significativement le code boilerplate en générant automatiquement les méthodes comme `equals()`, `hashCode()`, et `toString()`.

Dans cette section avancée, nous explorerons comment optimiser l'utilisation des Records et les bonnes pratiques pour tirer le meilleur parti de ces fonctionnalités.

## Objectifs pédagogiques

- Comprendre l'utilité des Records en Java
- Maîtriser la syntaxe et les bonnes pratiques d'utilisation des Records

- Différencier les Records des classes traditionnelles
- Implémenter des Records dans des scénarios avancés
- Appliquer les Records pour améliorer la concision et la clarté du code

## Records en Java

Un Record en Java est une classe immuable qui représente de façon concise des données simples.

Lorsqu'un Record est déclaré, Java génère automatiquement des implémentations pour plusieurs méthodes.

Cela permet d'économiser du temps et d'éviter des erreurs classiques lors de la réécriture de ces méthodes.

- Les Records sont définis avec le mot-clé `record`.
- Ils ne peuvent pas répondre au même besoin qu'une classe si des comportements spécifiques sont demandés.
- Ils sont immuables par nature.

## Création et structure

Déclarer un Record est simple.

Voici sa structure de base:

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

Ce code définit un Record appelé `Point` avec deux composants `x` et `y`.

- `x` et `y` sont les champs immuables du Record.
- Les méthodes `equals()`, `hashCode()`, et `toString()` sont générées automatiquement.
- Les composants peuvent également avoir des annotations et peuvent préciser des comportements avec des méthodes personnalisées.

## Avantages des Records

L'utilisation des Records offre plusieurs avantages:

- **Concision:** Réduction significative du code nécessaire pour créer des objets simples.
- **Immuabilité:** Promotion de la sécurité des threads grâce à l'immutabilité.
- **Lisibilité:** Structure claire qui met en avant les données sans ajouter de complexité.

Ces caractéristiques font des Records un bon choix pour modéliser des données claires et standardisées, telles que des données de configuration ou de transfert.

## Différences avec les Classes

Les Records ne doivent pas être utilisés pour tout par défaut.

Leur immuabilité peut être une contrainte dans certains cas.

- Les Records ne peuvent pas étendre d'autres classes (mais peuvent implémenter des interfaces).
- Ils sont principalement utilisés pour des données contenantes et pas pour des logiques complexes.
- Comparativement, les classes traditionnelles offrent plus de flexibilité sur la gestion des comportements.

# Exemples de Code

Utilisation de Record pour un objet simple:

```
public record Product(String name, double price) {  
    public Product {  
        if(price < 0) {  
            throw new IllegalArgumentException("Price cannot be negative.");  
        }  
    }  
}
```

Ce Record `Product` impose une validation sur le prix pour garantir qu'il est positif.

La validation est effectuée dans le constructeur compact du Record.

# Exemples de Scénarios

- **Transfert de données** : En simplifiant les échanges au sein de systèmes distribués par leur clarté et leur concision.
- **Configurations appliquées** : En encapsulant les configurations immuables utilisées dans les applications.

Les Records permettent de facilement réutiliser ces données avec peu de possibilités d'erreur.

# Exercices pratiques

1. **Créer un Record simple**: Écrire un Record pour un objet `Rectangle` avec des validations pour s'assurer que la longueur et la largeur sont positives.
2. **Étendre un Record** : Implémentez une interface existante avec un Record.
3. **Refactoriser le code existant** : Trouvez une classe immuable existante et réécrivez-la en tant que Record pour observer la réduction de code.

Évaluez chaque exercice en vous assurant que les règles d'immutabilité et de validation sont correctement implémentées.

# Synthèse et Ouverture

Les Records en Java simplifient la gestion des données immuables dans les applications.

En réduisant le code indispensable pour gérer les objets simples, ils augmentent la lisibilité et la sécurité. À l'avenir, les fonctionnalités des Records peuvent être étendues pour fournir des modèles plus flexibles, rendant Java encore plus efficace pour le développement moderne.

Anticipez l'apprentissage des Sealed Interfaces pour explorer plus de capacités de Java 17 et au-delà.

# Sealed interface

## Introduction

Les `sealed interfaces`, introduites en Java 17, permettent de restreindre les classes qui peuvent les implémenter.

Elles fournissent un contrôle granulé sur l'héritage, augmentant ainsi la sécurité et la clarté du code.

En utilisant les `sealed interfaces`, les développeurs peuvent définir un ensemble limité de sous-types autorisés, améliorant ainsi l'intégrité de la hiérarchie des types.

## Objectifs pédagogiques

- Comprendre le concept de `sealed interfaces` en Java.
- Savoir comment les implémenter dans un projet.
- Apprendre à définir des sous-types autorisés pour des interfaces scellées.
- Analyser les avantages en termes de sécurité et de maintenance du code.

## Contenu détaillé

Le concept de `sealed interfaces` vise à offrir un contrôle précis sur quelle classe peut les implémenter.

Une `sealed interface` peut spécifier quels sont les sous-types autorisés en utilisant le mot-clé `permits`.

Cela empêche les extensions imprévues et renforce l'invariant de subclassing.

Les sous-types d'une `sealed interface` doivent être marqués comme `final`, `non-sealed` ou rester `sealed`.

- `final` : Empêche tout sous-typage.
- `non-sealed` : Permet un classement descendant non restreint.
- `sealed` : Maintient des restrictions pour les sous-classes.

## Avantages des Sealed Interfaces

Les `sealed interfaces` offrent plusieurs avantages :

- **Sécurité renforcée** : Protègent l'intégrité de la hiérarchie des classes.
- **Lisibilité améliorée** : Rendent explicites les relations d'héritage.
- **Maintenance facilitée** : Réduisent les risques d'extensions non intentionnelles du code.

Leurs implémentations permettent un modèle d'héritage plus contrôlé et prévisible, particulièrement utile dans de grands projets où la cohérence est cruciale.

## Exemples de code

```
// Définition d'une sealed interface
public sealed interface Shape
    permits Circle, Square { }

// Implémentation finale d'un sous-type
public final class Circle implements Shape {
    // Implementation spécifique à Circle
}

// Implémentation d'un sous-type non-sealed
public non-sealed class Square implements Shape {
    // Implementation spécifique à Square
    // D'autres sous-types peuvent dériver de Square
}
```

Cet exemple montre comment la `sealed interface Shape` limite ses implémentations à `Circle` et `Square`, garantissant que pas d'autres classes ne peuvent implémenter `Shape` sans être déclarées.

## Exercices pratiques

- Définir une Sealed Interface :** Créez une interface scellée `Transport` permettant uniquement les classes `Car` et `Bike` comme sous-types.
- Ajout de sous-types et exploration :** Ajoutez un nouveau sous-type non-sealed `Skateboard` qui étend `Transport` et implémentez des méthodes spécifiques pour ce sous-type.
- Analyse Comparative :** Réalisez une analyse des avantages d'utiliser les `sealed interfaces` par rapport aux interfaces classiques dans un court essai.

Critères d'évaluation : Exactitude du code, compréhension des concepts, qualités des réflexions et des analyses produites.

## Synthèse

Les `sealed interfaces` enrichissent significativement Java en offrant une nouvelle manière de structurer les hiérarchies de classes.

Ces interfaces fournissent sécurité et maintenance, tout en garantissant que seul un ensemble prédéfini de sous-types peut les implémenter.

Pour les développeurs cherchant à contrôler le design et l'intégrité de leurs systèmes, les `sealed interfaces` constituent un ajout précieux et puissant. À l'avenir, explorez comment elles peuvent interagir avec d'autres fonctionnalités modernes de Java comme les records pour construire des applications robustes.

## Introduction

L'immutabilité est un concept clé en programmation qui garantit que les objets ne peuvent pas être modifiés après leur création.

En Java, cela assure non seulement la sécurité des données contre les changements accidentels mais aide aussi à simplifier la gestion des états dans les applications multithread.

## Objectifs pédagogiques

- Comprendre le concept d'immutabilité et son importance
- Distinguer les objets immuables des objets muables
- Mettre en œuvre l'immutabilité dans les classes Java

- Utiliser l'immédiateté pour améliorer la sécurité et la performance des applications

## Concept de l'Immutabilité

En Java, un objet immuable est un objet dont l'état ne peut pas être modifié après qu'il a été créé.

Pour qu'une classe soit immuable, il faut :

- Déclarer toutes les variables d'instance comme `final`.
- Ne fournir que des méthodes "getter", pas de "setter".
- Initialiser les variables d'instance dans le constructeur, sans exposer de références modifiables.

## Avantages de l'Immutabilité

Les objets immuables offrent plusieurs avantages :

- **Plus sûr en multithreading** : Pas de souci avec les conditions de concurrence.
- **Simplicité** : Plus facile à comprendre car aucun état ne change.
- **Optimisation** : Possibilité de les mettre en cache pour éviter des créations répétées.

## Créer une Classe Immuable

Pour créer une classe immuable en Java, suivez les principes :

- Rendez la classe `final` pour empêcher d'autres classes de la sous-classer.
- Rendez les champs `final` et `private`.
- Fourni un constructeur paramétré pour initialiser les attributs.
- Aucun "setter" : ne modifiez jamais l'état après la création.
- Si l'objet contient des références à des objets modifiables, assurez-vous de copier chaque objet à l'intérieur de votre classe.

## Exemple de Classe Immuable

```
public final class ImmutablePoint {  
    private final int x;  
    private final int y;  
  
    public ImmutablePoint(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}
```

Cette classe `ImmutablePoint` offre un bon exemple d'immortalité.

## Utiliser les Collections Immuables

Java 9 a introduit des méthodes pour créer des collections immuables :

```
List<String> list = List.of("A", "B", "C");
Set<String> set = Set.of("X", "Y", "Z");
Map<String, String> map = Map.of("key1", "value1", "key2", "value2");
```

Ces collections ne peuvent pas être modifiées après leur création, rendant leur état sûr et stable.

## Exercices Pratiques

1. **Créer une classe immuable** : Conception d'une classe `ImmutableUser` avec des champs `name` et `email`.
2. **Collections immuables** : Convertir une Liste modifiable existante en Liste immuable.
3. **Discussion** : Quels sont les scénarios où l'immortalité peut limiter la flexibilité ?

## Synthèse

L'immortalité en Java joue un rôle crucial dans le développement sécurisé et efficace d'applications modernes.

Elle contribue à réduire les erreurs liées à la concurrence et à assurer une gestion plus simple des états.

En tant que pratique de programmation, cela renforce la robustesse et la maintenabilité de vos codes, préfigurant des concepts explorés plus avant dans des domaines comme le design de software et la manipulation des données massives.

## Introduction

La thread-safety en Java est essentielle dans les applications multi-thread où plusieurs threads peuvent accéder simultanément aux mêmes données.

Assurer la sécurité des threads permet d'éviter des incohérences et des erreurs d'accès concurrentiel, garantissant ainsi une exécution prévisible et fiable de votre programme.

C'est crucial pour développer des applications performantes et robustes.

## Objectifs pédagogiques

- Comprendre le concept de thread-safety en Java.
- Identifier les problèmes courants de concurrence.
- Appliquer des techniques pour développer des applications thread-safe.
- Utiliser les classes et API de Java pour la synchronisation des threads.

# Concepts de Thread-safety

La thread-safety signifie que plusieurs threads peuvent accéder à une ressource partagée sans entraîner d'état incohérent.

Un objet est dit thread-safe si son état est toujours cohérent, même lorsque plusieurs threads le modifient simultanément.

Les problèmes courants incluent les conditions de course, où l'ordre des opérations conduit à un comportement erroné.

## Problèmes de Concurrence

Les problèmes de concurrence surviennent souvent lorsque des threads tentent de mettre à jour ou de lire des données à partir de ressources partagées.

Les conditions de course et les interblocages sont parmi les erreurs classiques.

Il est crucial d'encadrer l'accès à ces ressources pour éviter les corruptions de données et les comportements imprévisibles.

## Techniques de Synchronisation

Java propose différentes techniques pour assurer la sécurité des threads, incluant :

- Les blocs synchronisés avec le mot-clé `synchronized`.
- L'utilisation de verrous explicites (`java.util.concurrent.locks.Lock`).
- Les objets `ThreadLocal` pour des variables locales de thread.
- Les collections concurrentes fournies par Java pour manipuler en sécurité des collections partagées.

## Exemples de Code (Synchronized)

```
public class Counter {  
    private int count = 0;  
  
    public synchronized void increment() {  
        count++;  
    }  
  
    public synchronized int getCount() {  
        return count;  
    }  
}
```

Cet exemple montre un compteur simple où les méthodes sont synchronisées pour éviter les incohérences lorsque plusieurs threads accèdent simultanément à l'objet Counter.

# Exemples de Code (Lock)

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private int count = 0;
    private Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        return count;
    }
}
```

Dans cet exemple, `ReentrantLock` est utilisé pour un contrôle explicite du verrouillage, offrant plus de flexibilité que le mot-clé `synchronized`.

## Exercice 1 : Bloc Synchronized

1. Créez une classe Java avec un compteur partageable entre threads.
2. Utilisez le mot-clé `synchronized` pour protéger la méthode d'incrémentation.
3. Comparez le comportement du compteur avec et sans synchronisation en créant plusieurs threads.

Critères d'évaluation : Cohérence des valeurs de compteur dans des environnements multi-thread.

## Exercice 2 : Utilisation de Locks

1. Imitez l'exercice précédent en remplaçant `synchronized` par `ReentrantLock`.
2. Testez le code avec plusieurs threads et commentez les résultats.

Critères d'évaluation : Compréhension des bénéfices de l'utilisation de locks par rapport à `synchronized`.

## Synthèse

Pour assurer la sécurité des threads en Java, il est essentiel de comprendre et d'utiliser correctement les techniques de synchronisation.

En choisissant judicieusement entre `synchronized`, locks et collections concurrentes, les développeurs peuvent prévenir les problèmes de concurrence et garantir le bon fonctionnement de leurs applications.

**Les prochaines étapes incluront l'exploration avancée des outils Java pour optimiser les performances multithread.**

## Pattern builder immuable

### Introduction

Le pattern builder immuable est une construction avancée en Java permettant de créer des objets de manière flexible et sécurisée tout en garantissant leur immutabilité.

Dans un monde où le parallélisme et la sécurité des données sont cruciaux, ce pattern offre une solution élégante pour éviter les effets de bord imprévus et les bugs liés à la mutabilité des objets.

### Objectifs pédagogiques

- Comprendre le concept de pattern builder immuable
- Construire des objets complexes en garantissant leur immutabilité
- Utiliser les builders pour une construction fluide et lisible
- Appliquer des techniques pour rendre le code thread-safe

### Contenu détaillé

Le pattern builder immuable vise à combiner la fluidité et la lisibilité avec l'immuabilité.

Cette pratique permet de créer des objets dont l'état ne peut pas être modifié après leur création.

L'immuabilité élimine une classe d'erreurs liées aux états changeants.

Le pattern est particulièrement utile lors de la construction d'objets ayant de nombreuses options de configuration.

### Création d'un Builder

Pour créer un builder immuable en Java, il faut d'abord définir une classe statique interne statique nommée `Builder` dans la classe cible.

Cette classe contiendra les mêmes champs que la classe cible.

Ensuite, une méthode `build()` retourne une nouvelle instance de l'objet cible.

Elle initialisera les valeurs à partir du builder.

Chaque option de construction sera définie via une méthode qui retourne `this`, permettant le chaînage de méthodes.

### Avantages du Pattern

1. **Immuabilité et sécurité:** Une fois construit, l'objet ne peut être modifié.
2. **Lisibilité accrue:** Construction intuitive grâce aux méthodes enchaînées.
3. **Facilité d'ajout de nouvelles options:** Ajouter de nouvelles options de construction sans affecter le code existant.

## Exemple de code 1

Prenons l'exemple d'une classe `Person` utilisant un builder immuable :

```
public final class Person {  
    private final String name;  
    private final int age;  
  
    private Person(Builder builder) {  
        this.name = builder.name;  
        this.age = builder.age;  
    }  
  
    public static class Builder {  
        private String name;  
        private int age;  
  
        public Builder name(String name) {  
            this.name = name;  
            return this;  
        }  
  
        public Builder age(int age) {  
            this.age = age;  
            return this;  
        }  
  
        public Person build() {  
            return new Person(this);  
        }  
    }  
}
```

Ici, le nom et l'âge sont définis dans le builder et finalisés dans l'objet `Person`.

## Exemple de code 2

Utilisation du builder :

```
Person person = new Person.Builder()  
    .name("Alice")  
    .age(30)  
    .build();
```

Cette approche simplifie grandement le processus de création d'objets complexes, améliorant ainsi la clarté du code.

## Exercices

1. **Exercice 1 :** Créez une classe `Car` utilisant le pattern builder immuable.

Elle doit inclure au moins les champs `make`, `model`, `year`, et `isElectric`.

- Critère : Les objets `Car` doivent être immuables et permettre le chaînage des méthodes.

2. **Exercice 2 :** Modifiez la classe `Car` pour ajouter une méthode `fuelEfficiency` tout en maintenant l'immuabilité.

- Critère : Expliquez le modèle de sécurité thread-safe que vous choisissez.

## Synthèse et ouverture

Le pattern builder immuable offre une méthode à la fois robuste et flexible pour la création d'objets en Java.

**Essentiel lorsque l'immuabilité est requise, il ouvre également la voie vers des solutions thread-safe et des API lisibles. À l'avenir, vous pourriez explorer d'autres patterns de conception qui combinent l'immuabilité avec des pratiques orientées objet modernes, comme les records introduits dans les versions récentes de Java.**

# Garbage Collection tuning

## Introduction

Le tuning du Garbage Collector (GC) est essentiel pour optimiser la performance des applications Java.

Le GC gère automatiquement la mémoire, mais une configuration inadaptée peut entraîner des interruptions prolongées et des performances dégradées.

Ce module vous aidera à comprendre comment ajuster le GC pour améliorer l'efficacité et la réactivité de votre application.

## Objectifs pédagogiques

- Comprendre le fonctionnement du Garbage Collector en Java
- Analyser l'impact du GC sur la performance
- Apprendre à configurer différents types de GC pour optimiser l'application
- Évaluer et ajuster les paramètres du GC selon les besoins de l'application

## Fonctionnement du Garbage Collector

Java utilise le Garbage Collector pour automatiser la gestion de la mémoire.

Le GC libère de la mémoire en nettoyant les objets non référencés.

Le processus de GC se divise en plusieurs phases :

- **Marking** : identifier les objets accessibles
- **Deletion ou compaction** : libérer la mémoire des objets inutiles
- **Collection** : récupération de la mémoire

Il est crucial de comprendre ces phases pour ajuster le GC.

## Types de Garbage Collectors

Java propose plusieurs types de GC, chacun adapté à des scénarios spécifiques :

- **Serial GC** : Convient pour les applications à faible mémoire et d'entrée de gamme.
- **Parallel GC** : Utile dans des environnements multi-thread.
- **Concurrent Mark-Sweep (CMS) GC** : Réduit les temps de pause en effectuant le marquage de manière concurrente.

- **G1 GC (Garbage-First)** : Conçu pour une faible latence, forme un compromis entre les temps de pause et la conception concurrente.

Il est important de choisir le bon GC pour vos besoins spécifiques.

## Configuration du GC

La configuration efficace du GC nécessite l'ajustement de divers paramètres en fonction des exigences de l'application :

- **Heap Size (-Xms, -Xmx)** : Définit la taille initiale et maximale du tas.
- **Survivor Ratio (-XX:SurvivorRatio)** : Contrôle la proportion de la mémoire jeune.
- **G1 GC Options** : G1 est souvent réglé pour optimiser la latence avec des paramètres comme `-XX:MaxGCPauseMillis`.

Ces ajustements peuvent réduire les interruptions du GC et améliorer le débit.

## Exemples de Code

### Configuration de base pour Serial GC

```
java -XX:+UseSerialGC -Xms512m -Xmx1024m -jar MonApp.jar
```

**Ce paramètre utilise le Serial Garbage Collector avec une taille du tas initiale de 512 Mo et maximale de 1024 Mo.**

### Configuration pour G1 GC avec pause contrôlée

```
java -XX:+UseG1GC -Xms2g -Xmx2g -XX:MaxGCPauseMillis=200 -jar MonApp.jar
```

Ici, G1 GC est utilisé, visant à maintenir les pauses du GC sous 200 ms, avec une taille de tas fixe de 2 Go.

## Exercices Pratiques

### 1. Analyse des Logs GC :

- Exécutez une application Java avec Serial GC et analysez les logs pour identifier les phases du GC.
- Critères : Identifier les périodes de pause et l'efficacité du nettoyage.

### 2. Comparaison des Collectors :

- Configurez l'application avec G1 GC et CMS GC séparément.

Comparez les performances en termes de temps de réponse et de débit.

- Critères : Documenter les différences de performance entre les deux configurations.

### 3. Tuning en situation réelle :

- Utilisez VisualVM pour monitorer l'activité du GC en temps réel sur une application lourde.
- Critères : Apporter des ajustements pour réduire les temps de pause identifiés.

## Synthèse

Le tuning du Garbage Collector peut améliorer significativement la performance des applications Java.

Comprendre les différents types de GC et savoir configurer les paramètres clés est essentiel.

Il est recommandé de tester plusieurs configurations pour identifier la solution optimale, en tenant compte des besoins spécifiques de l'application et de ses contraintes de performance.

## Introduction à JVM Tuning

L'optimisation de la JVM (Java Virtual Machine) est cruciale pour améliorer la performance d'applications Java.

Lorsque vous ajustez les paramètres de la JVM, vous pouvez influencer de manière significative le comportement du temps d'exécution et l'utilisation des ressources.

Cela permet aux applications de fonctionner de manière plus efficace et d'améliorer l'expérience utilisateur.

## Objectifs Pédagogiques

- Identifier les paramètres de performance principaux de la JVM.
- Expérimenter les techniques de tuning de la mémoire pour réduire les temps de pause.
- Analyser l'impact des paramètres JVM sur les performances applicatives.
- Mettre en œuvre des ajustements pour optimiser la gestion du garbage collection.

## Comprendre le Fonctionnement de la JVM

La JVM est une machine virtuelle qui exécute le code Java.

Elle gère la conversion du bytecode en instructions machine, la mémoire et le garbage collection.

Le tuning de la JVM inclut l'ajustement des paramètres de mémoire comme la taille de la heap, les pools permgen/metaspaces, et la surveillance des threads pour améliorer les performances et la réactivité de l'application.

## Paramètres Clés de la JVM

Les paramètres de la JVM influencent le comportement et la performance d'une application Java.

Les plus courants incluent :

- **Xms/Xmx** : Définissent la taille initiale et maximale de la heap.
- **Xmn** : Taille de la Young Generation.
- **XX:PermSize/Metaspacesize** : Taille initiale de la zone PermGen ou Metaspaces.
- **GC Collectors** : Choix entre Serial, Parallel, CMS ou G1.

# Techniques de Tuning de la Mémoire

Optimiser l'utilisation de la mémoire est essentiel pour éviter les temps de pause.

Voici quelques techniques :

- Ajustez la taille de la heap avec `-Xms` et `-Xmx` pour éviter les re-dimensionnements fréquents.
- Utilisez le Garbage Collector G1 pour applications à faible latence.
- Surveillez les ratios Eden/Survivor dans la Young Generation pour un équilibrage optimisé.

## Exemple de Configurations JVM

Considérez cet exemple de configuration pour une application avec de faibles exigences en latence :

```
-Xms1024m -Xmx4096m -XX:+UseG1GC -XX:MaxGCPauseMillis=200
```

Ce paramétrage initialise la heap à 1GB, avec un maximum de 4GB, utilise le Garbage Collector G1, et cible des pauses de ramassage des ordures de 200ms.

## Exemples Pratiques

Voici des exemples de tuning pour divers scénarios :

### 1. Petite Application :

```
-Xms512m -Xmx1024m -XX:+UseSerialGC
```

### 2. Service de Backend avec Haut Débit :

```
-Xms2g -Xmx8g -XX:+UseParallelGC -XX:GCTimeRatio=4
```

### 3. Application à Faible Latence :

```
-Xms4g -Xmx16g -XX:+UseG1GC -XX:MaxGCPauseMillis=100
```

## Exercices Pratiques

### 1. Tuning Basique :

Configurez une JVM pour une application ayant une heap maximale de 2GB, avec attention particulière au temps de latence.

### 2. Analyse de GC :

Utilisez un outil de monitoring pour identifier les pauses de GC dans une application et proposez des ajustements.

### 3. Optimisation de la Mémoire :

Ajustez les paramètres de heap et surveillez les performances avec des workloads variés.

## Synthèse

Le tuning de la JVM est une compétence essentielle pour les développeurs souhaitant optimiser les performances de leurs applications Java.

En ajustant les paramètres de mémoire et en sélectionnant le bon Garbage Collector, vous pouvez améliorer la réactivité et l'efficacité des applications. explorons des outils de monitoring et des métriques pour un meilleur tuning à l'avenir.

# Exercice : profiler et améliorer la performance d'un code lent

## Introduction

Comprendre et optimiser les performances d'une application est crucial en programmation Java, surtout lorsqu'un code est lent.

Dans cet exercice, nous allons apprendre à "profiler" un code, identifier les goulets d'étranglement et appliquer des stratégies pour améliorer ses performances.

## Objectifs pédagogiques

- Identifier les outils de profiling disponibles en Java
- Analyser et interpréter les résultats d'un profiling
- Optimiser un code avec des techniques éprouvées
- Appliquer les bonnes pratiques de performance en Java

## Contenu détaillé

Le "profiling" est un processus qui vous aide à analyser les performances d'une application pour identifier où le code passe le plus de temps ou utilise le plus de ressources.

- **Outils de Profiling :** Plusieurs outils existent pour Java, comme JProfiler, VisualVM ou le plugin IntelliJ Profiler.
- **Analyse des Résultats :** Déterminer les parties du code qui consomment le plus de temps de CPU ou de mémoire.
- **Optimisation :** Techniques comme l'amélioration des algorithmes, la réduction de la complexité de boucle, ou l'optimisation des opérations d'I/O.
- **Bonnes Pratiques :** Utiliser des structures de données adaptées, minimiser la synchronisation dans les threads, et éviter les allocations inutiles.

## Exemples pratiques

### Exemple 1 : VisualVM

1. **Lancer VisualVM :** Téléchargez et connectez-le à votre JVM.
2. **Profiler une Application :** Parcourez les onglets CPU et Memory pour identifier les charges les plus lourdes.

### Exemple 2 : Optimisation d'une Boucle

Considérez le code suivant avant optimisation :

```

int sum = 0;
for (int i = 0; i < numbers.size(); i++) {
    if (numbers.get(i) % 2 == 0) {
        sum += numbers.get(i);
    }
}

```

Après optimisation :

```

int sum = numbers.stream()
    .filter(n -> n % 2 == 0)
    .mapToInt(Integer::intValue)
    .sum();

```

Ce changement utilise le Stream API pour rendre le code plus lisible et potentiellement plus performant grâce au parallélisme intégré.

## Exercices pratiques

### Exercice 1 : Utilisation de VisualVM

- **But :** Profiler un programme Java pour identifier ses points faibles.
- **Instructions :**
  - Téléchargez un outil de profiling comme VisualVM.
  - Profitez de différents scénarios et notez les observations sur les temps CPU et les consommations mémoire.

### Exercice 2 : Optimiser des Algorithmes

- **But :** Récrire un algorithme lent avec de meilleures performances.
- **Instructions :**
  - Choisissez un algorithme de type recherche ou tri.
  - Comparez initialement le temps d'exécution.
  - Optimisez en améliorant l'algorithme, puis comparez les résultats.

## Synthèse

Savoir profiler et améliorer la performance d'un code est une compétence clé pour tout développeur Java.

De l'utilisation d'outils de profiling aux techniques d'optimisation, chaque étape contribue à créer des applications plus efficaces. À l'avenir, explorez davantage d'outils et de techniques pour perfectionner ces compétences, et surveillez les mises à jour autour des JVM et JDK pour rester à jour avec les dernières innovations en matière d'optimisation.

## Introduction

La programmation Java avancée intègre de nouvelles fonctionnalités et des bonnes pratiques essentielles pour créer des applications robustes et performantes.

Parmi les principaux axes, nous explorerons divers concepts modernes tels que les records avancés, les interfaces scellées et l'immutabilité.

Comprendre ces notions est crucial pour manipuler efficacement le langage.

# Vue d'ensemble des sous-notions

1. **Record avancé** - Simplifie la définition de classes immuables.
2. **Sealed interface** - Restriction des implémentations possibles.
3. **Immutabilité** - Garantit l'état constant d'un objet.
4. **Thread-safety** - Assure la sécurité des opérations concurrentes.
5. **Pattern builder immutable** - Aide à construire des objets complexes.
6. **Garbage Collection tuning** - Optimise la gestion de la mémoire.
7. **JVM tuning** - Améliore les performances globales de l'application.
8. **Exercice pratique** - Profiler et améliorer la performance d'un code lent.

Ces éléments se complètent pour favoriser des bonnes pratiques de développement en Java.

## Introduction

Le découpage en modules indépendants est une approche clé dans la programmation Java moderne.

Elle permet de structurer le code en unités bien définies, chacune ayant une responsabilité claire.

Ce modèle architectural améliore la maintenabilité, facilite le développement parallèle et renforce la réutilisabilité du code.

## Objectifs pédagogiques

- Comprendre le concept de module en Java.
- Apprendre à structurer une application avec des modules indépendants.
- Savoir déclarer et utiliser des modules à l'aide de `module-info.java`.
- Évaluer les avantages du découpage modulaire pour la maintenance et l'extensibilité.

## Concepts de Modules

En Java, un module est une collection de packages avec un fichier de déclaration `module-info.java` à sa racine.

Il définit de manière explicite les dépendances vers d'autres modules et expose les packages exploitables par d'autres modules.

Ce paradigme vise à encapsuler le code et réduire les effets de bord.

## Créer un Module

Pour créer un module :

1. Créez un dossier pour votre module.
2. Placez-y vos packages et classes Java.
3. Ajoutez un fichier `module-info.java` où vous déclarez le nom du module et ses dépendances.

```
module com.example.module {  
    exports com.example.module.api;  
    requires another.module;  
}
```

## Avantages du Découpage Modulaire

- **Maintenance Facilités** : Chaque module pouvant être testé et mis à jour indépendamment.
- **Réutilisation** : Modules bien définis et documentés sont réutilisables dans divers projets.
- **Encapsulation** : Cachant les détails d'implémentation, encourageant une interface bien définie.

## Exemple d'implémentation

Considérons un projet d'application e-commerce structuré en trois modules :

- `module core` : Gère la logique métier.
- `module api` : Expose des services via API.
- `module service` : Implémente des fonctionnalités particulières comme le paiement.

## Exemple de `module-info.java`

Pour le module `api` :

```
module com.ecommerce.api {  
    exports com.ecommerce.api.services;  
    requires com.ecommerce.core;  
}
```

Ceci montre comment exposer certains services internes tout en limitant les dépendances.

## Exercice Pratique

1. **Créez un module core gérant les produits.**
  - Inclure des classes pour les produits avec des méthodes CRUD.
2. **Ajoutez un module service pour la gestion de la commande.**
  - Implémentez des services pour le traitement des commandes.
3. **Intégrez un module api pour exposer en REST.**
  - Utilisez un framework léger pour mettre à disposition POJOs comme ressources.

**Critères d'évaluation :** Clarté du découpage modulaire, utilisation appropriée des directives `exports` et `requires`.

## Synthèse

Le découpage en modules indépendants offre une organisation claire et efficace du code Java, favorisant la réutilisation et la maintenance.

La structure explicite fournie par `module-info.java` assure une interface bien définie entre les différentes parties de l'application. À l'avenir, explorez des outils et pratiques modernes comme JPMS pour tirer le meilleur parti de l'architecture modulaire.

## Introduction

Dans cette section sur le "Module core", nous allons explorer un des éléments fondamentaux de l'architecture modulaire en Java.

Le module core est crucial puisqu'il comprend les composants principaux qui assurent le fonctionnement de base de toute application modulaire Java.

Sa bonne définition et sa structuration sont essentielles pour garantir la modularité et la scalabilité du système.

## Objectifs pédagogiques

- Comprendre le rôle et l'importance du module core dans une architecture modulaire Java.
- Apprendre à structurer efficacement un module core.
- Savoir implémenter et configurer le fichier `module-info.java` pour le module core.
- Identifier et gérer les dépendances internes et externes du module core.

## Contenu détaillé

Le module core contient les éléments de base ou de noyau de votre application.

Il définit :

- Les classes principales qui assurent les fonctionnalités essentielles.
- Les interfaces qui établissent des contrats pour d'autres modules.
- Les méthodes utilitaires utilisées à travers différents modules.

Pour structurer un module core :

- Identifiez les classes et interfaces vraiment cruciales.
- Évitez une surcharge de fonctionnalités : le module core doit rester focalisé.
- Pensez à la maintenabilité et à l'évolutivité dès la conception.

## Fichier `module-info.java`

Chaque module Java doit contenir un fichier `module-info.java`.

Pour un module core, il spécifie les exports nécessaires :

- Modifiez `module-info.java` pour ne divulguer que les packages essentiels.
- Utilisez l'instruction `exports` pour chaque package à rendre accessible.
- Utilisez l'instruction `requires` pour indiquer les dépendances vers d'autres modules.

Exemple :

```
module com.example.core {  
    exports com.example.core.services;  
    requires java.logging;  
}
```

# Exemples de code

## Exemple 1 : Structure de base

```
module com.myapp.core {  
    exports com.myapp.core.utility;  
    requires java.sql;  
}
```

Cette structure permet l'exportation du package `com.myapp.core.utility` tout en demandant l'accès à l'API Java SQL.

## Exemple 2 : Interface et Classe

```
// Interface in module core  
package com.myapp.core.services;  
public interface CoreService {  
    void execute();  
}  
  
// Class implementing the interface  
package com.myapp.core.impl;  
public class CoreServiceImpl implements CoreService {  
    public void execute() {  
        // Implementation details  
    }  
}
```

# Exercices pratiques

## Exercice 1 : Crédit d'un Module core

1. Créez un package `com.example.core.base`.
2. Ajoutez une classe `MainService` avec une méthode `run()`.
3. Configurez `module-info.java` pour exporter ce package.
4. Testez l'accès à votre module depuis un autre module.

## Critères d'évaluation

- La `module-info.java` doit être correctement configurée.
- La classe `MainService` doit être accessible depuis un autre module.
- Les fonctionnalités de base doivent être clairement définies et documentées.

# Synthèse

Le module core constitue la base de votre application modulaire Java.

Une structure bien pensée facilite l'évolutivité et l'intégration de nouveaux modules.

Nous avons appris à configurer le fichier `module-info.java` et à gérer efficacement les dépendances.

Poursuivre avec les autres types de modules, tels que les modules service et API, enrichira notre compréhension globale de l'architecture modulaire Java.

## Introduction

Dans le cadre d'une architecture modulaire en Java, le **Module Service** joue un rôle crucial.

Il fournit des fonctionnalités spécifiques pouvant être utilisées par d'autres modules.

Ce module peut intégrer des services grâce à l'utilisation de `ServiceLoader`, permettant ainsi une extensibilité et une flexibilité accrues.

Nous allons explorer comment concevoir, implémenter et utiliser efficacement un module de service.

## Objectifs pédagogiques

- Comprendre le rôle du module service dans une architecture modulaire.
- Apprendre à définir et déclarer un module service en Java.
- Découvrir comment utiliser `ServiceLoader` pour charger dynamiquement des services.
- Mettre en œuvre un module service fonctionnel avec des exemples pratiques.

## Architecture modulaire en Java

Une architecture modulaire divise une application Java en plusieurs modules distincts, chacun encapsulant un ensemble de fonctionnalités.

Le module service s'intègre typiquement dans ce cadre pour offrir des services réutilisables et évolutifs.

Cela encourage une meilleure organisation du code, facilite la maintenance et améliore la portée des applications Java.

## Définition d'un Module Service

Un module service en Java est essentiellement un conteneur de logique métier ou de fonctionnalités spécifiques.

Un module est défini par un fichier `module-info.java`, où il expose des interfaces ou des services qu'il fournit aux autres modules.

Les autres modules peuvent alors consommer ces services sans dépendre des détails d'implémentation.

## Utilisation de ServiceLoader

`ServiceLoader` est un utilitaire Java qui permet de découvrir et de charger des services dynamiquement à partir d'un module service.

Cela réduit le couplage entre les modules consommateurs et les modules fournisseurs de services.

Un module service définit les services qu'il exporte, et `ServiceLoader` se charge de charger ces services à la demande.

## Exemple de Module Service

### 1. Définir une Interface de Service

Créez une interface dans votre module service.

Par exemple :

```
public interface PaymentService {  
    void processPayment(double amount);  
}
```

### 2. Implémenter le Service

Créez des classes qui implémentent votre interface :

```
public class PayPalService implements PaymentService {  
    public void processPayment(double amount) {  
        // Logic to process payment via PayPal  
    }  
}
```

### 3. Déclaration dans module-info.java

Exportez le service dans votre module :

```
module com.example.payment {  
    exports com.example.payment;  
    provides com.example.payment.PaymentService with com.example.payment.PayPalService;  
}
```

## Exercices pratiques

### 1. Crédit d'un Module Service

Créez un module service qui propose une fonctionnalité d'authentification.

Définissez une interface `AuthenticationService` et implémentez deux versions : `BasicAuthService` et `TokenAuthService`.

### 2. Utilisation de ServiceLoader

Écrivez un petit programme Java qui utilise `ServiceLoader` pour charger et utiliser dynamiquement les implémentations de `AuthenticationService` définies dans l'exercice précédent.

### 3. Evaluation

Vérifiez que votre programme peut passer d'une implémentation de service à une autre sans modification du code de l'application principale.

## Synthèse

Le module service est une composante essentielle de l'architecture modulaire Java, permettant une organisation flexible et évolutive du code.

En utilisant `ServiceLoader`, nous pouvons charger et modifier les services de manière dynamique, offrant ainsi une extensibilité accrue.

La maîtrise de ces concepts renforce la robustesse et la maintenabilité des applications Java modernes. À l'avenir, l'exploration du couplage avec des outils tels que Spring pourrait enrichir davantage cette approche modulaire.

# Introduction

Le Module API en Java est crucial pour structurer et organiser les interactions entre différents modules.

Ce concept, introduit avec Java 9, améliore la modularité et maintenabilité des projets.

Apprendre à concevoir des API modulaires permet de développer des applications flexibles et évolutives.

## Objectifs pédagogiques

- Comprendre le concept du Module API en Java.
- Savoir comment définir et utiliser les API dans une architecture modulaire.
- Maîtriser l'exportation des packages au sein d'un module.
- Être capable de concevoir une application modulaire incluant des APIs.

## Définition du Module API

Un Module API en Java est un module dont l'objectif est de définir et exposer une interface publique que d'autres modules peuvent consommer.

Contrairement aux modules internes, les modules API se concentrent sur l'interaction inter-modules en exposant uniquement les fonctionnalités nécessaires.

- Les API sont définies via le fichier `module-info.java`.
- Les modules API permettent de masquer l'implémentation tout en exposant les services nécessaires à d'autres modules.

## Utilisation du `module-info.java`

Le fichier `module-info.java` est le point d'entrée de la modularité en Java.

Pour une API, il déclare quels packages sont exportés :

```
module com.example.api {  
    exports com.example.api.service;  
}
```

- Le mot-clé `exports` indique que le package est accessible aux autres modules.
- Les modules utilisant cette API devront la déclarer dans leur propre `module-info.java`.

## Avantages des Modules API

Les modules API offrent de nombreux avantages, notamment :

- **Encapsulation** : seuls les packages nécessaires sont exposés, le reste est caché.
- **Réutilisabilité** : une API bien conçue peut être utilisée par plusieurs modules offrant une flexibilité accrue.
- **Séparation des préoccupations** : améliore la clarté et la maintenabilité du projet.

# Exemple d'API Modulaire

Considérons un module API fournissant des services de paiement :

## module-info.java

```
module payment.api {  
    exports com.payment.api.service;  
}
```

## Classe Service

```
package com.payment.api.service;  
public interface PaymentService {  
    void processPayment(double amount);  
}
```

Cet exemple montre comment définir une API qui expose un service de paiement.

## Exercices pratiques

1. **Création d'un Module API** : Créez un module API pour un service de notification.

Définissez une interface exposant un service d'envoi de messages.

- **Critères** : Utilisation de `module-info.java` pour exposer votre service.

2. **Consommation d'une API** : Développez un second module consommant l'API de notification créée.

Implementez une classe réalisant l'interface définie dans l'API.

3. **Test de l'application** : Intégrez et testez votre application modulaire.

Assurez-vous que le module consommateur peut accéder au service du module API.

## Synthèse et ouverture

La gestion des API en architecture modulaire est une compétence essentielle pour concevoir des applications complexes.

En appliquant les principes de modularité et d'encapsulation, les développeurs peuvent créer des systèmes évolutifs et résilients.

La prochaine étape pourrait explorer l'intégration de services dynamiques avec le ServiceLoader pour augmenter la flexibilité des API.

## Introduction

Dans ce module, nous allons explorer le concept du `ServiceLoader` en Java.

Le `ServiceLoader` facilite la découverte et le chargement dynamiques de services implémentés à l'aide du mécanisme de service SPI (Service Provider Interface) en Java.

L'usage judicieux du `ServiceLoader` permet de construire des applications modulaires où les dépendances entre composants peuvent être résolues dynamiquement à l'exécution, ajoutant ainsi flexibilité et extensibilité à votre code.

## Objectifs pédagogiques

- Comprendre le concept de SPI en Java.
- Utiliser le `ServiceLoader` pour découvrir et charger des services.
- Intégrer le `ServiceLoader` dans une architecture modulaire Java.
- Résoudre des problèmes de dépendances dynamiques à l'aide de `ServiceLoader`.

## Utilisation du ServiceLoader

Le `ServiceLoader` est une classe centrale pour implémenter le pattern de l'interface fournisseur de services en Java.

Ce modèle repose sur l'enregistrement des implémentations de services, qui peuvent être découverts et chargés à l'exécution sans modifier le code source consommateur.

1. **Définition de l'interface de service** : Cette interface spécifie le contrat que toutes les implémentations doivent suivre.
2. **Création des implémentations** : Les classes spécifiques qui implémentent cette interface fournissent le comportement réel.
3. **Déclaration du fournisseur de service** : Un fichier de configuration placé dans `META-INF/services` pour indiquer quelles classes implémentent l'interface fournie.

## Déclaration du ServiceLoader

Pour utiliser `ServiceLoader`, vous devez créer un fichier dans `META-INF/services` nommé d'après l'interface de votre service.

Ce fichier contient le nom des classes qui fournissent l'implémentation de l'interface du service.

Cela permet à `ServiceLoader` de découvrir automatiquement les services à l'exécution.

- **Exemple de fichier** : `META-INF/services/com.example.MyService`
  - Contenu :

```
com.example.MyServiceImpl1  
com.example.MyServiceImpl2
```

## Exemples de Code avec ServiceLoader

Observons comment intégrer le `ServiceLoader` dans un projet Java.

Voici un exemple de base démontrant l'utilisation du `ServiceLoader` :

```
public interface MyService {  
    void execute();  
}
```

Implementation Example:

```
public class MyServiceImpl implements MyService {  
    public void execute() {  
        System.out.println("Service Execution");  
    }  
}
```

Chargement du service :

```
ServiceLoader<MyService> loader = ServiceLoader.load(MyService.class);  
for (MyService service : loader) {  
    service.execute();  
}
```

Dans cet exemple, `ServiceLoader` est utilisé pour charger toutes les implémentations de `MyService` enregistrées dans le fichier `META-INF/services`.

## Exercices Pratiques

### 1. Exercice 1 : Création de service

- Définissez une interface de service.
- Implémentez au moins deux classes fournissant ce service.
- Utilisez `ServiceLoader` pour les charger et exécuter une méthode de service.

### 2. Exercice 2 : Extension dynamique

- Ajoutez une nouvelle implémentation de service sans modifier le code existant du client.
- Testez que `ServiceLoader` détecte et charge automatiquement cette nouvelle implémentation.

### 3. Exercice 3 : Projet modulaire

- Créez un projet Java modulaire en utilisant `module-info.java`.
- Assurez-vous que votre module héberge votre interface de service et utilise `ServiceLoader`.

Critères d'évaluation : Capacité à intégrer et étendre dynamiquement les services sans modification directe du code source du client.

## Synthèse

Le `ServiceLoader` est un outil puissant dans l'arsenal de Java pour la construction d'applications modulaires et extensibles.

Il permet la découverte dynamique et la gestion de services en utilisant le pattern SPI, simplifiant ainsi la gestion des dépendances et améliorant la flexibilité.

Cette approche est particulièrement utile pour développer des systèmes qui nécessitent extensibilité et adaptabilité, tels que les plug-ins ou les frameworks.

En maîtrisant `ServiceLoader`, vous êtes bien préparés à tirer parti des fonctionnalités avancées de l'architecture modulaire de Java.

## Introduction

L'injection de dépendances est une pratique essentielle en programmation modulaire qui permet de créer des applications flexibles et maintenables.

Le `ServiceLoader` en Java facilite cette approche en permettant de dynamiquement découvrir et instancier des services définis dans des modules distincts.

Grâce au `ServiceLoader`, vous pouvez remplacer des implémentations à la volée sans modifier le code client, améliorant ainsi l'extensibilité et la modularité de votre application.

# Objectifs pédagogiques

- Comprendre le concept de l'injection de dépendances.
- Appliquer le mécanisme de `ServiceLoader` en Java.
- Découvrir comment définir et consommer des services modulaires.
- Pratiquer l'implémentation de services interchangeables.

## Contexte et concepts clés

L'injection de dépendances permet aux objets de déclarer leurs dépendances plutôt que de les créer.

En Java, le `ServiceLoader` est une classe utilitaire qui recherche des implémentations de services déclarés dans des fichiers spécifiques sous `META-INF/services`.

- **Services** : Interfaces ou classes abstraites définissant certaines fonctionnalités.
- **Providers** : Implémentations concrètes de ces services.
- **ServiceLoader** : Moteur de chargement et d'instanciation des services.

## Utiliser ServiceLoader

Java utilise le fichier `module-info.java` pour déclarer les services fournis et consommés.

1. **Déclaration du service** : Définissez une interface ou une classe abstraite.
2. **Implémentation du provider** : Fournissez une ou plusieurs classes implémentant le service.
3. **Configuration** : Créez un fichier nommé après votre interface dans `META-INF/services`, listant les implémentations.

## Exemples de code

### Déclaration de service :

```
public interface PaymentService {  
    void processPayment(double amount);  
}
```

### Provider :

```
public class CreditCardPayment implements PaymentService {  
    public void processPayment(double amount) {  
        // Traitement du paiement par carte de crédit  
    }  
}
```

### Module configuration :

Dans `module-info.java` :

```
module payment.module {  
    provides PaymentService with CreditCardPayment;  
}
```

## Chargement des services

Pour charger et utiliser les services, vous pouvez utiliser le `ServiceLoader` dans votre code client :

```
ServiceLoader<PaymentService> loader = ServiceLoader.load(PaymentService.class);  
for (PaymentService service : loader) {  
    service.processPayment(100.00);  
}
```

Cela parcourra toutes les implémentations disponibles, en traitant un paiement de 100.00 pour chaque implémentation.

## Exercices pratiques

1. **Création d'un service** : Définissez un service `EmailService` avec des méthodes d'envoi de mails simples.
2. **Implémentation de providers** : Créez deux implémentations, l'une pour envoyer des mails via SMTP et l'autre via un service web.
3. **Utilisation de ServiceLoader** : Chargez et testez vos services dans une application console. Évaluez quelle implémentation est utilisée.

Critères d'évaluation : Le programme doit être modulaire, avec le bon découpage en modules, et permettre d'interchanger les implémentations sans modifier le code d'appel.

## Synthèse et ouverture

L'injection de dépendances via `ServiceLoader` en Java est un puissant moyen d'améliorer la modularité de vos applications.

Cette approche facilite la gestion des implémentations de services sans modification directe du code, ce qui est essentiel pour les systèmes évolutifs.

En explorant des technologies comme CDI (Contexts and Dependency Injection) pour des besoins plus complexes, vous pouvez étendre les capacités de l'injection de dépendances dans vos projets Java.

## Introduction

Dans cette section, vous découvrirez comment créer un projet Java modulaire complet.

L'architecture modulaire permet de structurer un programme en unités indépendantes appelées modules, facilitant la maintenance et l'évolution du code.

Nous aborderons chaque étape de cette création, de la conception à l'implémentation.

## Objectifs pédagogiques

- Concevoir un projet Java en modules distincts
- Créer et configurer des fichiers `module-info.java`
- Utiliser `ServiceLoader` pour l'injection de dépendances

- Planifier un projet modulaire complet et fonctionnel

## Conception modulaire

Le premier pas dans la création d'un projet modulaire est de définir l'architecture souhaitée.

Un projet modulaire se compose généralement de plusieurs modules, chacun remplissant un rôle spécifique et communiquant avec les autres via des interfaces bien définies.

- **Module Core** : Contient la logique centrale et les fonctionnalités de base.
- **Module API** : Définit les interfaces que les autres modules implémenteront.
- **Module Service** : Fournit les implementations spécifiques des interfaces définies dans le module API.

## Configuration des Modules

Chaque module inclut un fichier `module-info.java` qui déclare les dépendances :

- ```
• module com.example.core { requires com.example.api; }
• module com.example.api { exports com.example.api.interfaces; }
• module com.example.service { requires com.example.api; provides com.example.api.interfaces.Service with com.example.service.Service;
```

Cette configuration permet de contrôler l'accès entre les modules et de partager les API nécessaires.

## Utilisation de ServiceLoader

ServiceLoader permet de découvrir et d'utiliser des implementations fournies par des modules de services dynamiques.

Ceci peut être configuré dans le module API, qui expose des interfaces, et dans les modules de services qui fournissent les implementations.

- **Déclarer un Service**: Dans `module-info.java`, utilisez `provides` et `with` pour lier l'interface aux implementations.
- **Consommer un Service**: Utilisez `ServiceLoader.load(Service.class)` pour obtenir une instance du service.

## Exemples de Code

Voici une illustration simple de configuration modulaire :

```
// module-info.java in API module
module com.example.api {
    exports com.example.api.interfaces;
}

// Service interface
package com.example.api.interfaces;
public interface GreetingService {
    String greet(String name);
}
```

```

// module-info.java in Service module
module com.example.service {
    requires com.example.api;
    provides com.example.api.interfaces.GreetingService with com.example.service.BasicGreetingService;
}

// Service implementation
package com.example.service;
import com.example.api.interfaces.GreetingService;
public class BasicGreetingService implements GreetingService {
    public String greet(String name) {
        return "Hello, " + name + "!";
    }
}

```

## Exercice Progressif

- Conception Modulaires** : Créez une architecture de base avec au moins trois modules : Core, API, et Service.
  - Implémentation des Modules** : Ajoutez des classes et interfaces de base dans chaque module.
- Assurez-vous que les services peuvent être découverts et utilisés via `ServiceLoader`.
- Tests de Fonctionnalité** : Implémentez un cas d'utilisation simple où le module Core utilise un service du module Service via l'API.

## Synthèse et Ouverture

Vous avez maintenant une vue d'ensemble sur la création d'un projet Java modulaire complet, en partant de la conception des modules jusqu'à leur intégration à l'aide de `ServiceLoader`.

Ce modèle modulaire est essentiel pour la création d'applications Java évolutives et maintenables.

Vous pouvez explorer des projets plus complexes et considérer des outils comme la gestion de version et le CI/CD pour aller plus loin.

## Introduction à l'Architecture modulaire

L'architecture modulaire en Java est une approche qui permet de structurer les applications en unités fonctionnelles distinctes appelées modules.

Cela favorise la clarté du code, la réutilisation et la facilité de maintenance.

Ce cours couvre les éléments essentiels de l'architecture modulaire, notamment :

- `module-info.java`: Déclaration de modules.
- Découpage en modules indépendants.
- Modules *core*, *service*, et *API*.
- Utilisation de `ServiceLoader`.
- Injection de dépendances via `ServiceLoader`.
- Exercice pratique sur la création d'un projet modulaire complet.

## Vue d'ensemble de l'Architecture modulaire

Le découpage en modules permet de séparer les préoccupations fonctionnelles et d'améliorer la modularité du code.

Le fichier `module-info.java` est central pour la déclaration, et chaque module a un rôle spécifique : le module `core` contient le cœur de l'application, les modules `service` offrent des fonctionnalités additionnelles, et le module `API` définit les interfaces.

L'utilisation de `ServiceLoader` et l'injection de dépendances facilitent l'extensibilité.

Enfin, un exercice pratique consolidera les acquis sur la création d'un projet modulaire complet.

## Introduction à Java Avancé

La programmation Java avancée couvre des concepts clés pour développer des applications robustes et performantes.

Ce cours se concentrera sur trois grands thèmes :

- **Généricité et collections** : Apprenez à utiliser les types génériques et les collections de manière efficiente pour créer des applications flexibles.
- **Fonctionnalités modernes et bonnes pratiques** : Découvrez les nouvelles fonctionnalités de Java et les méthodes pour améliorer la sécurité et les performances du code.
- **Architecture modulaire** : Comprenez comment structurer un projet Java en modules pour une meilleure maintenabilité.

Chaque sous-notion sera approfondie dans les sections suivantes.

## Vue d'ensemble des Sous-notions

La programmation Java avancée s'articule autour de trois axes :

- **Généricité et collections** : Axé sur l'utilisation de types génériques et d'outils de collection pour gérer efficacement des données.
- **Fonctionnalités modernes et bonnes pratiques** : Couvre des concepts comme les records avancés et l'immutabilité pour créer un code durable.
- **Architecture modulaire** : Se concentre sur l'organisation de projets en modules indépendants et intégration avec `ServiceLoader`.

Ces approches visent à développer des compétences essentielles en Java avancé.

## CompletableFuture

### Introduction

Le `CompletableFuture` en Java est une classe puissante qui simplifie la gestion des tâches asynchrones.

Introduite avec Java 8, elle permet de construire des chaînes de calculs asynchrones de manière fluide et lisible.

L'usage du `CompletableFuture` favorise l'exécution concurrente moderne et rend le code plus réactif et adaptable aux tâches exécutées en arrière-plan sans bloquer le fil d'exécution principal.

## Objectifs Pédagogiques

- Comprendre le concept de programmation asynchrone avec `CompletableFuture`.
- Apprendre à créer et gérer des tâches asynchrones.

- Maîtriser la composition et la combinaison de CompletableFutures.
- Implémenter des traitements asynchrones réactifs en Java.

## Contenu Détailé (1)

### Comprendre CompletableFuture

Un CompletableFuture représente une opération asynchrone qui peut être complétée à une date ultérieure.

Contrairement aux Future, CompletableFuture fournit une API plus riche qui permet de chaîner facilement plusieurs opérations asynchrones.

- **Création d'un CompletableFuture:** peut être fait via des méthodes statiques telles que supplyAsync() pour lancer des tâches asynchrones.
- **Chaînage de tâches:** Permet de chaîner plusieurs opérations grâce à des méthodes comme thenApply, thenAccept, et thenRun .

## Contenu Détailé (2)

### Gestion de l'asynchronisme

- **Compléter manuellement:** Utilisez la méthode complete() pour définir manuellement la valeur du futur.
- **Gestion des exceptions:** CompletableFuture fournit exceptionally(), handle(), et whenComplete() pour gérer les exceptions pendant l'exécution des tâches.

Exemple de code :

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) throw new RuntimeException("Oups!");
    return "Succès";
});

future.exceptionally(ex -> "Erreur: " + ex.getMessage())
    .thenAccept(System.out::println);
```

## Exemples de Code (1)

### Exemple de création et chaînage

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 5)
    .thenApply(n -> n * n)
    .thenApply(String::valueOf);

future.thenAccept(System.out::println); // Affiche "25"
```

Ce code démontre comment créer un CompletableFuture qui calcule le carré de 5 et convertit le résultat en chaîne de caractères.

## Exemples de Code (2)

### Gestion des tâches asynchrones

```
CompletableFuture<Void> futureTask =  
    CompletableFuture.runAsync(() -> {  
        try {  
            Thread.sleep(1000);  
            System.out.println("Tâche Asynchrone Complétée!");  
        } catch (InterruptedException e) {  
            throw new IllegalStateException(e);  
        }  
    });  
  
futureTask.join(); // Attend la fin de la tâche
```

Cet exemple lance une tâche qui dort pendant une seconde avant d'afficher un message, illustrant comment gérer l'exécution asynchrone.

### Exercices Pratiques

1. **Exercice 1:** Créez un `CompletableFuture` qui télécharge le contenu d'une URL en arrière-plan et traite la réponse.
  - Critères : Le code doit gérer les exceptions correctement et afficher le contenu après téléchargement.
2. **Exercice 2:** Utilisez `CompletableFuture` pour lire des tâches depuis une liste, les exécuter en parallèle, et calculer la somme des résultats.
  - Critères : Assurez-vous que le code est non-bloquant et bien synchronisé.

### Synthèse

`CompletableFuture` est un outil essentiel pour gérer la concurrence de façon moderne en Java.

En facilitant la création et la coordination des tâches asynchrones, il contribue à rendre le code plus flexible et réactif.

Maîtriser ses fonctionnalités permet de créer des applications capables de gérer efficacement les opérations en coulisse.

Les prochaines étapes pourraient inclure l'intégration avec les flux réactifs pour des systèmes encore plus dynamiques.

### Introduction

Les chaînes d'exécution asynchrone en Java permettent de gérer des tâches de manière non bloquante, optimisant ainsi l'utilisation des ressources et améliorant la réactivité des applications.

En se libérant des contraintes des traitements synchrones, ou synchronisés, les développeurs peuvent concevoir des applications réactives, capables de gérer plusieurs opérations simultanément sans interférences.

### Objectifs pédagogiques

- Comprendre le concept d'exécution asynchrone.
- Mettre en œuvre des chaînes d'exécution asynchrones en Java.
- Analyser et débugger des flux asynchrones.

- Construire des applications réactives.

## Détails de l'exécution asynchrone

L'exécution asynchrone repose sur la capacité à lancer des tâches sans attendre leur achèvement immédiat pour passer à d'autres opérations.

En Java, cela s'effectue principalement avec l'API `CompletableFuture`, qui permet de composer des actions qui s'exécuteront de manière non bloquante.

Cette approche est particulièrement efficace pour les opérations d'I/O ou les calculs longs, où une attente active serait inefficace.

## Complétion et gestion d'erreurs

Avec `CompletableFuture`, on peut chaîner plusieurs opérations asynchrones et définir des actions en cas de réussite (`thenApply`, `thenAccept`) ou d'échec (`exceptionally`).

Ceci permet une gestion fine et réactive des erreurs, de même qu'une optimisation du flux de travail.

Cela augmente aussi la résilience des applications face aux erreurs non prévues.

## Composition de tâches

`CompletableFuture` permet de combiner différentes futures, soit en les exécutant indépendamment (`allOf`), soit en attendant la fin de l'une d'entre elles (`anyOf`).

Cette flexibilité est cruciale pour répondre à des cas d'usage où la nature des opérations dépend de la disponibilité des ressources ou de la réponse des services externes.

## Exemple 1 : Traitement simple

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    return "Hello, World!";
});

future.thenAccept(result -> System.out.println(result));
```

Ce code initialise une tâche qui retourne le message "Hello, World!" et l'affiche une fois complétée, sans bloquer le fil principal.

## Exemple 2 : Gestion des erreurs

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (true) throw new RuntimeException("Erreur !");
    return "Bonjour";
}).exceptionally(ex -> "Erreur capturée : " + ex.getMessage());

future.thenAccept(System.out::println);
```

Ici, une exception est intentionnellement lancée pour montrer comment `exceptionally` peut capturer et traiter l'erreur.

## Exercice 1 : Invoquer des APIs

Créez un programme qui effectue des appels à plusieurs services web en parallèle.

Utilisez `CompletableFuture` pour récupérer et afficher les résultats dès qu'ils sont disponibles.

Critères d'évaluation :

- Bon usage de `CompletableFuture`.
- Gestion des erreurs.
- Lisibilité du code.

## Exercice 2 : Pipeline de calcul

Implémentez un pipeline qui enchaîne plusieurs opérations de transformations de données (ex. : conversion de chaînes, calculs numériques) de manière asynchrone.

Assurez-vous que chaque étape dépend du résultat de la précédente.

Critères d'évaluation :

- Bon enchaînement des tâches.
- Performance (mesurer le temps d'exécution).
- Gestion des erreurs et des exceptions.

## Synthèse

Les chaînes d'exécution asynchrone en Java, via `CompletableFuture`, offrent une puissante façon de construire des applications réactives, capables d'opérer sous de lourdes charges sans sacrifier la performance.

La maîtrise des flux asynchrones permet de créer des logiciels plus robustes, réactifs et adaptés aux environnements modernes, où la conciliation efficacité et réactivité est critique.

Explorons ensuite les paradigmes réactifs complets pour enrichir davantage vos applications.

# Introduction

La classe `ForkJoinPool` en Java est un cadre puissant pour le traitement parallèle.

Utilisée pour diviser et régner sur les tâches, elle simplifie la programmation concurrente.

Comprendre `ForkJoinPool` permet de mieux exploiter les architectures multi-cœurs modernes, optimisant la performance des applications.

## Objectifs pédagogiques

- Comprendre le fonctionnement de `ForkJoinPool`.
- Apprendre à diviser et fusionner des tâches concurrentes.
- Utiliser efficacement `ForkJoinPool` pour améliorer les performances.
- Évaluer et optimiser l'utilisation de `ForkJoinPool` dans des applications Java.

## Fonctionnement de ForkJoinPool

`ForkJoinPool` repose sur le principe du *divide and conquer*.

Les tâches sont divisées en sous-tâches (`fork`), gérées individuellement, puis leurs résultats sont combinés (`join`).

Ce modèle exploite les threads de façon dynamique, allouant et exécutant efficacement des tâches indépendantes.

## Structure de ForkJoinPool

Une `ForkJoinPool` est constituée de plusieurs workers threads.

Ces threads gèrent les tâches déposées dans une queue de tâches.

Contrairement à `ExecutorService`, `ForkJoinPool` optimise l'utilisation CPU en adaptant automatiquement le nombre de threads actifs.

## Utilisation Pratique

Pour utiliser `ForkJoinPool`, créez des tâches en étendant `RecursiveTask` ou `RecursiveAction`. `RecursiveTask` est utilisé lorsque la tâche renvoie un résultat, tandis que `RecursiveAction` est pour une tâche ne renvoyant rien.

Implémentez la méthode `compute` pour définir le comportement.

# Exemple : Calcul de la somme

```
import java.util.concurrent.RecursiveTask;

public class ForkJoinSum extends RecursiveTask<Integer> {
    private final int[] array;
    private final int start, end;

    public ForkJoinSum(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= 10) { // seuil
            int sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            int mid = (start + end) / 2;
            ForkJoinSum leftTask = new ForkJoinSum(array, start, mid);
            ForkJoinSum rightTask = new ForkJoinSum(array, mid, end);
            leftTask.fork(); // exécute en parallèle
            return rightTask.compute() + leftTask.join();
        }
    }
}
```

# Exemple : Utilisation

```
import java.util.concurrent.ForkJoinPool;

public class Main {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
        ForkJoinSum task = new ForkJoinSum(array, 0, array.length);
        int result = pool.invoke(task);
        System.out.println("Sum: " + result); // Affiche la somme totale
    }
}
```

# Exercices Pratiques

- 1. Basic Implementation:** Créez une classe `ForkJoinMax` pour trouver la valeur maximale d'un tableau d'entiers.
- 2. Threshold Experimentation:** Modifiez la taille de seuil de division pour observer les impacts sur les performances.
- 3. Complex Task:** Implémentez une tâche qui combine plusieurs calculs (moyenne, min, max) et analysez l'utilisation des ressources.

# Synthèse et Ouverture

`ForkJoinPool` est une technologie qui module efficacement la charge de calcul en utilisant tous les processeurs disponibles.

En maîtrisant cet outil, les développeurs peuvent concevoir des applications Java plus réactives et performantes.

Pour aller plus loin, explorez les optimisations et comparaisons avec d'autres modèles de concurrence en Java.

Il semble que vous avez envoyé un message vide.

Si vous souhaitez que je développe la partie sur `ExecutorService` en Java, veuillez confirmer, et je vais créer un ensemble de slides pédagogiques pour cette notion finale.

## Introduction à `synchronized`

La programmation concurrente en Java nécessite un contrôle précis des accès aux ressources partagées.

Le mot-clé `synchronized` est essentiel pour assurer l'exclusivité d'accès aux sections critiques de votre code.

En garantissant qu'un seul fil d'exécution accède à une ressource à la fois, il aide à éviter les problèmes tels que les conditions de course, assurant ainsi la cohérence des données.

## Objectifs pédagogiques

- Comprendre le rôle de `synchronized` dans la gestion de la concurrence.
- Être capable d'identifier et de protéger les sections critiques du code.
- Appliquer `synchronized` pour éviter les conditions de course.
- Évaluer les impacts de `synchronized` sur la performance de l'application.

## Concepts de base de `synchronized`

En Java, une méthode ou un bloc de code peut être marqué comme `synchronized` pour contrôler l'accès aux ressources partagées.

Quand un fil d'exécution entre dans une méthode ou un bloc `synchronized`, il acquiert un verrou qui empêche les autres fils d'y accéder jusqu'à ce que le verrou soit libéré.

- **Méthode synchronized** : Se verrouille sur l'objet appelant.
- **Bloc synchronized** : Peut se verrouiller sur un objet spécifique.

Cette approche garantit que seules les opérations unitaires atomiques sont exécutées dans des environnements multithread.

## Utilisation de `synchronized`

L'utilisation de `synchronized` est essentielle pour assurer que les threads n'entrent pas dans une section critique du code en même temps.

Dans une méthode synchronisée, le verrou est implicitement acquis par l'objet sur lequel la méthode est appelée.

Pour les blocs synchronisés, vous pouvez spécifier explicitement l'objet à verrouiller.

```
public synchronized void updateSharedResource() {  
    // Opération sur une ressource partagée  
}  
  
public void updateSharedResourceWithBlock() {  
    synchronized(this) {  
        // Opération sur une ressource partagée  
    }  
}
```

## Exemples pratiques

Voyons maintenant plusieurs exemples concrets de l'utilisation de `synchronized` :

### Exemple 1 : Synchronisation de méthode

```
public synchronized void increment() {  
    counter++;  
}
```

Dans cet exemple, `increment()` s'assure que seul un thread à la fois peut modifier `counter`.

### Exemple 2 : Bloc synchronisé

```
public void incrementWithBlock() {  
    synchronized(this) {  
        counter++;  
    }  
}
```

Ce bloc synchronisé verrouille l'instance actuelle (`this`), limitant l'accès concurrent à `counter`.

## Exemples avancés

### Exemple 3 : Bloc synchronisé sur un objet mutable

```
private final Object lock = new Object();  
  
public void complexOperation() {  
    synchronized(lock) {  
        // Opérations complexes sur des ressources partagées  
    }  
}
```

**Dans ce scénario, lock est utilisé pour la synchronisation de différentes méthodes reliant des opérations complexes sur des ressources.**

## Exemple 4 : Synchronisation de code critique

```
public class BankAccount {  
    private int balance;  
  
    public void deposit(int amount) {  
        synchronized(this) {  
            balance += amount;  
        }  
    }  
}
```

Cette méthode garantit que `deposit` est effectué de manière atomique, prévenant les conditions de course.

## Exercices pratiques

### Exercice 1 : Synchronisation simple

Implémentez une classe Java qui simule un guichet bancaire.

Utilisez `synchronized` pour protéger les opérations de dépôt et de retrait.

Critères d'évaluation :

- Utilisation correcte de `synchronized` pour éviter les conditions de course.

### Exercice 2 : Bloc synchronized

Créez une classe de compteur partagé entre plusieurs threads.

Assurez-vous que les incréments et décréments sont thread-safe.

Critères d'évaluation :

- Bloc `synchronized` appliqué adéquatement pour protéger l'accès concurrent.

## Synthèse

Le mot-clé `synchronized` est un outil simple mais puissant pour la gestion de la concurrence en Java.

Il offre un moyen efficace de prévenir les conditions de course en contrôlant l'accès simultané aux ressources partagées.

En maîtrisant `synchronized`, vous pouvez améliorer la fiabilité et la robustesse de vos applications concurrentes.

Les prochaines étapes pourraient inclure l'exploration des alternatives comme `ReentrantLock` pour une flexibilité accrue.

# Introduction

Le **ReentrantLock** est une conception d'arrêt avancé de verrouillage introduit en Java 5 pour une gestion flexible et efficace des threads dans la programmation concurrente.

Il est conçu pour surpasser les limitations des verrous intrinsèques (`synchronized`) en offrant plus de possibilités comme l'équité entre threads et la possibilité de personnaliser le comportement de verrouillage.

## Objectifs pédagogiques

- Comprendre l'usage du ReentrantLock dans la gestion des threads.
- Différencier ReentrantLock et synchronized pour une meilleure prise de décision.
- Maîtriser les méthodes et propriétés spécifiques du ReentrantLock.
- Appliquer des ReentrantLock pour résoudre des problèmes de concurrence.

## Comprendre ReentrantLock

Le ReentrantLock implémente l'interface Lock du package `java.util.concurrent.locks`.

Contrairement à `synchronized`, il donne plus de contrôle sur l'acquisition et la libération du verrou.

- **Acquisition et libération explicite :** Contrairement au bloc `synchronized` où le verrou est acquis automatiquement, avec ReentrantLock, cela doit être fait manuellement avec les méthodes `lock()` et `unlock()`.
- **Équité des threads :** Vous pouvez définir un constructeur avec un flag d'équité pour garantir qu'au moins les threads attendent le verrou dans l'ordre de leur demande.

## Avantages du ReentrantLock

Par rapport à `synchronized`, ReentrantLock offre :

1. **Équité :** Vous pouvez éviter la famine en donnant un accès équitable aux threads.
2. **Verrouillage et déverrouillage manuels :** Permet de libérer le verrou à des endroits différents du code, donnant plus de flexibilité.
3. **Tentative de verrouillage :** Utilisez `tryLock()` pour ne pas bloquer un thread indéfiniment s'il ne peut acquérir le verrou.
4. **Verrouillage interruptible :** Avec `lockInterruptibly()`, un thread peut être interrompu lors de l'attente du verrou.

## Méthodes clés en ReentrantLock

- `lock()` : Acquiert le verrou.
- `unlock()` : Libère le verrou.
- `tryLock()` : Retente d'acquérir le verrou, et retourne immédiatement si le verrou est inaccessible.
- `lockInterruptibly()` : Acquiert le verrou sauf si le thread est interrompu.

Il est essentiel d'utiliser la méthode `unlock()` dans un bloc `finally` pour garantir que le verrou est toujours relâché.

# Exemples pratiques

```
import java.util.concurrent.locks.ReentrantLock;

public class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

    public int getCount() {
        lock.lock();
        try {
            return count;
        } finally {
            lock.unlock();
        }
    }
}
```

Dans cet exemple, le verrouillage est acquis manuellement pour garantir que les opérations d'incrémentation et de lecture s'exécutent de manière atomique et sûre en termes de threads.

## Exercices pratiques

- Exercice 1:** Implementez un simple système bancaire avec plusieurs threads où les comptes utilisent ReentrantLock pour les opérations de dépôt et de retrait.
  - Critères d'évaluation :** Fonctionnement correct du système de verrouillage et absence de conditions de concurrence.
- Exercice 2:** Modifiez une application existante pour remplacer tous les blocs synchronized par des ReentrantLock en maintenant le même comportement.
  - Critères d'évaluation :** Comparaison des performances et comportement correct du programme.

## Synthèse

Le ReentrantLock est un outil flexible et puissant pour la gestion de verrouillage dans les applications Java concurrentes.

Il permet un contrôle plus fin sur l'acquisition de verrou et favorise des solutions robustes dans des environnements intensément multi-threadés.

Bien que similaire à synchronized, il offre des fonctionnalités supplémentaires qui le rendent préférable dans de nombreux scénarios.

L'apprentissage et la mise en pratique de ReentrantLock ouvrent la voie à des solutions de concurrence plus sophistiquées et nuancées en Java.

# Introduction aux Atomic Variables

Les atomic variables jouent un rôle crucial dans la programmation concurrente en fournissant une manipulation thread-safe des données sans avoir recours à des verrouillages explicites.

Elles permettent d'effectuer des opérations atomiques de manière efficace, évitant ainsi les problèmes de synchronisation traditionnels.

## Objectifs pédagogiques

- Comprendre le concept d'atomicité dans la programmation concurrente
- Identifier les types d'atomic variables disponibles en Java
- Appliquer l'utilisation des atomic variables pour résoudre des problèmes de concurrence
- Comparer les atomic variables avec d'autres mécanismes de synchronisation

## Atomicité et Concurrence

L'atomicité est un concept clé dans la programmation concurrente, désignant des opérations qui s'exécutent de manière indivisible, sans interruption.

Les atomic variables offrent cette garantie en assurant une mise à jour sûre dans des environnements multithread.

## Types d'Atomic Variables

Java offre plusieurs types d'atomic variables dans le package `java.util.concurrent.atomic` :

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

Chacune de ces classes permet d'effectuer des opérations atomiques courantes telles que l'incrémentation et la décrémentation.

## Usage des Atomic Variables

Les atomic variables sont particulièrement utiles dans les situations où de simples opérations arithmétiques ou de mise à jour de référence sont nécessaires, mais doivent être thread-safe.

Elles évitent l'utilisation de `synchronized`, améliorant ainsi la performance sous certaines charges.

## Exemples d'Usage

Considérons une variable de type `AtomicInteger` pour un compteur partagé entre plusieurs threads :

```

import java.util.concurrent.atomic.AtomicInteger;

public class AtomicCounter {
    private final AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();
    }

    public int getValue() {
        return counter.get();
    }
}

```

Ici, `incrementAndGet()` est une opération atomique, assurant qu'aucun autre thread ne peut modifier `counter` entre l'incrémentation et l'obtention de sa valeur.

## Comparaison avec Synchronized

Contrairement aux blocs synchronisés qui peuvent introduire des problèmes de contention, les atomic variables fournissent une alternative légère.

Les verrous explicites entraînent souvent de l'attente si un thread détient déjà le verrou. À l'inverse, les atomic variables minimisent ce besoin en utilisant des opérations sur le CPU, comme les comparaisons et les échanges atomiques.

## Exercices Pratiques

### 1. Manipulation de Compteurs :

- Implémentez un compteur utilisant `AtomicLong`.

Testez-le avec plusieurs threads qui modifient la valeur simultanément.

### 2. Démo de Performance :

- Comparez les performances entre un compteur implémenté avec `AtomicInteger` et un autre avec `synchronized`.

### 3. Évaluation d'`AtomicReference` :

- Créez un exemple où une `AtomicReference` est utilisée pour gérer des objets partagés.

## Synthèse et Ouverture

En conclusion, les atomic variables en Java fournissent un moyen rapide et thread-safe pour gérer des opérations sur les données partagées.

Elles sont idéales pour des scénarios nécessitant hautes performances sans les complications des mécanismes de verrou classiques. À l'avenir, l'exploration des alternatives comme `VarHandle` ou des structures de données immuables pourrait enrichir votre boîte à outils pour la concurrence.

## Introduction

Dans cet exercice, nous allons explorer un aspect fondamental de la concurrence moderne en Java : l'implémentation d'un pipeline parallèle de calcul.

En programmant un pipeline, vous pourrez diviser une tâche en plusieurs étapes indépendantes, améliorant ainsi les performances via la parallélisation, tout en appliquant les concepts vus précédemment comme les `CompletableFuture` et les chaînes d'exécution.

# Objectifs pédagogiques

- Appliquer la programmation concurrente pour optimiser les performances.
- Utiliser `CompletableFuture` pour structurer un pipeline de tâches asynchrones.
- Mettre en œuvre des concepts de concurrence moderne via des exemples pratiques.
- Concevoir et évaluer l'efficacité d'un système de calcul parallèle.

## Pipeline parallèle : Concepts

Un pipeline parallèle de calcul est une série de tâches exécutées simultanément, où le résultat d'une tâche est passé à la suivante.

Ceci permet :

- L'amélioration des performances grâce à l'application simultanée des étapes.
- Une utilisation efficace des ressources par la répartition de la charge de travail.
- La modularité, facilitant la maintenance et l'évolution du code.

## CompletableFuture en Java

`CompletableFuture` est un composant Java utilisé pour des calculs asynchrones :

- Il permet de définir des traitements qui s'exécutent sans bloquer le thread principal.
- Supporte la composition de tâches via des méthodes comme `thenApply`, `thenCompose` et `thenCombine`.

Un pipeline est créé en enchaînant ces appels, qui définissent chaque étape du pipeline.

## Exécution parallèle avec ForkJoinPool

Pour gérer l'exécution parallèle :

- `ForkJoinPool` est un framework de gestion de threads permettant la division de tâches en sous-tâches.
- Exploite plusieurs cœurs du processeur en effectuant le traitement de nombreuses petites tâches en parallèle.

# Exemple de code : Pipeline

```
import java.util.concurrent.*;  
  
public class PipelineExample {  
    public static void main(String[] args) {  
        ExecutorService executor = Executors.newFixedThreadPool(4);  
  
        CompletableFuture.supplyAsync(() -> {  
            return computeInitialStep();  
        }, executor).thenApplyAsync(result -> {  
            return processStepOne(result);  
        }, executor).thenApplyAsync(result -> {  
            return processStepTwo(result);  
        }, executor).thenAccept(result -> {  
            System.out.println("Final Result: " + result);  
        });  
  
        executor.shutdown();  
    }  
  
    private static int computeInitialStep() {  
        // Simuler une opération initiale  
        return 2;  
    }  
  
    private static int processStepOne(int input) {  
        // Simuler une première transformation  
        return input + 3;  
    }  
  
    private static int processStepTwo(int input) {  
        // Simuler une deuxième transformation  
        return input * 5;  
    }  
}
```

## Exercices pratiques

1. **Simple Pipeline:** Créez un pipeline avec `CompletableFuture` pour réaliser un calcul simple et afficher le résultat final.
2. **Pipeline Complexe:** Étendez le pipeline existant avec des étapes supplémentaires, comme le filtrage ou l'agrégation de données.
3. **Optimisation:** Évaluez et optimisez le pipeline pour réduire le temps d'exécution, en ajustant le nombre de threads dans le `ForkJoinPool`.

## Critères d'évaluation

- **Pertinence :** Le pipeline doit être efficace et correct, respectant la logique des étapes.
- **Performance :** Utilisation optimale des ressources avec un temps d'exécution réduit.
- **Robustesse :** Gestion des exceptions et des erreurs potentielles dans le pipeline.

## Synthèse et Ouverture

Nous avons mis en œuvre un pipeline parallèle de calcul, appliquant la concurrence moderne grâce à `CompletableFuture`.

Ce module vous permet d'augmenter l'efficacité de vos applications Java en distribuant les tâches.

**En prolongement, on pourrait explorer la programmation réactive, qui élargit les possibilités de traitement asynchrone à des systèmes distribués.**

## Introduction à la Concurrence Moderne

La "Concurrence moderne" est une facette clé de la programmation en Java, essentielle pour tirer parti des capacités multicœurs des processeurs modernes.

Elle s'articule autour de plusieurs concepts et outils puissants visant à simplifier et optimiser l'exécution parallèle et asynchrone des tâches.

## Vue d'ensemble des sous-notions

Dans ce module, nous explorerons :

- **CompletableFuture** : Pour des opérations asynchrones flexibles.
- **Chaînes d'exécution asynchrone** : Pour structurer l'asynchronisme.
- **ForkJoinPool** : Pour diviser le travail efficacement.
- **ExecutorService** : Pour gérer des threads de manière simplifiée.
- **synchronized et ReentrantLock** : Pour contrôler l'accès aux ressources partagées.
- **Atomic variables** : Pour les opérations atomiques et sécurisées en multithreading.
- **Exercice** : Un défi pour implémenter un pipeline parallèle de calcul.

Ces éléments vous équiperont pour développer des applications plus réactives et performantes.

## Introduction

Le concept de **Reactive Streams** est au cœur de la programmation réactive moderne.

Il s'agit d'une initiative pour définir un standard de traitement asynchrone de flux de données avec une gestion de backpressure.

Ce modèle est crucial pour créer des applications capables de traiter de grandes quantités de données de manière efficace et non bloquante.

## Objectifs pédagogiques

- Comprendre le rôle et les bénéfices des Reactive Streams.
- Maîtriser les composants clés de Reactive Streams.
- Apprendre à manipuler des flux de données asynchrones.
- Appliquer le modèle de backpressure dans des applications Java.

# Concept et fonctionnement

Les **Reactive Streams** sont conçus pour gérer les flux de données asynchrones.

Ce modèle repose sur quatre interfaces principales :

1. **Publisher** : Émet des éléments aux abonnés enregistrés.
2. **Subscriber** : Consomme des éléments fournis par le Publisher.
3. **Subscription** : Lien entre Publisher et Subscriber, gère la demande d'éléments.
4. **Processor** : Transforme les données en agissant à la fois comme un Subscriber et un Publisher.

## Importance de Backpressure

Backpressure est un concept crucial pour éviter la surcharge de mémoire quand les consommateurs ne peuvent pas suivre le rythme des producteurs.

Cette fonctionnalité permet de :

- Demander explicitement le nombre d'éléments à recevoir.
- Suspendre la réception jusqu'à nouvelle demande, ce qui prévient la saturation.
- Maintenir une application stable, même lors d'un traitement de gros volumes de données.

## Exemples de code

Exemple basique de création d'un Publisher :

```
import org.reactivestreams.Publisher;
import org.reactivestreams.Subscriber;
import org.reactivestreams.Subscription;

public class SimplePublisher implements Publisher<String> {
    @Override
    public void subscribe(Subscriber<? super String> subscriber) {
        // Manage subscription logic here
        subscriber.onSubscribe(new SimpleSubscription(subscriber));
    }
}
```

À explorer plus en détail : l'implémentation de `SimpleSubscription`.

## Exemple illustré : Subscriber

Implémentation d'un simple Subscriber :

```

public class SimpleSubscriber implements Subscriber<String> {
    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1); // Demand one item at start
    }

    @Override
    public void onNext(String item) {
        System.out.println("Received: " + item);
        subscription.request(1); // Request next item
    }

    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Complete!");
    }
}

```

## Exercices pratiques

- Exercice 1 :** Implémentez un Processor qui transforme un flux de chaînes en leur version majuscules.
  - Critères : Assurez-vous que le backpressure est géré correctement.
- Exercice 2 :** Créez un système simple de publication et abonnement où un Publisher envoie des nombres entiers, et un Subscriber calcule leur somme.
  - Critères : Mettre en œuvre la gestion du backpressure de façon que le Subscriber puisse contrôler le flux des données.

## Synthèse et ouverture

Les **Reactive Streams** offrent un modèle puissant pour traiter efficacement de grandes quantités de données en mode asynchrone.

La gestion du backpressure est un atout essentiel pour maintenir la robustesse de l'application face à des volumes de données variables.

En suivant cette approche, vous êtes préparés à aborder des frameworks comme Project Reactor et RxJava pour construire des systèmes réactifs évolutifs.

En poursuivant votre apprentissage, explorez comment intégrer ces concepts dans des architectures microservices pour maximiser la performance et la réactivité.

## Introduction à Backpressure

La Backpressure est un concept clé en programmation réactive qui s'attaque aux problèmes liés à la gestion du flux de données entre fournisseurs et consommateurs.

Dans les systèmes réactifs, il est essentiel de contrôler la pression exercée par le flux de données pour éviter les débordements de mémoire et garantir des performances optimales.

Cette section apportera une compréhension approfondie de la façon dont la Backpressure améliore la résilience des systèmes réactifs.

## Objectifs pédagogiques

- Comprendre le concept de Backpressure.
- Appliquer la Backpressure dans des systèmes réactifs.
- Identifier et résoudre les problèmes de flux de données avec la Backpressure.
- Implémenter des stratégies efficaces pour gérer le flux de données.

## Comprendre la Backpressure

La Backpressure se réfère à la capacité du système à gérer les situations où le rythme des données émises dépasse la capacité de traitement du consommateur.

Cela est crucial pour éviter les exceptions liées à la mémoire ou aux ressources, assurant que les consommateurs ne soient pas submergés.

## Mécanismes de Backpressure

La Backpressure repose sur plusieurs mécanismes pour moduler le flux de données :

- **Buffering** : Stocker temporairement les données en attente de traitement.
- **Drop** : Ignorer certaines données en cas de surcharge du consommateur.
- **Throttling** : Réguler la fréquence des données transmises.

Ces stratégies permettent d'équilibrer la charge de travail et de garantir la stabilité du système.

## Exemples de Backpressure

```
import io.reactivex.Flowable;
import io.reactivex.schedulers.Schedulers;

// Exemple d'un émetteur rapide
Flowable<Integer> fastEmitter = Flowable.range(1, 1000)
    .onBackpressureBuffer(100); // Ajoute un tampon de 100 éléments

fastEmitter
    .observeOn(Schedulers.computation())
    .subscribe(data -> {
        // Simule un traitement lent
        Thread.sleep(10);
        System.out.println("Élément traité: " + data);
    });
}
```

Ce code montre comment appliquer un tampon pour gérer un flux rapide.

# Stratégies avancées

Pour des cas plus complexes, utiliser des stratégies comme `onBackpressureDrop()` ou `onBackpressureLatest()` permet de gérer intelligemment les éléments de flux.

- **onBackpressureDrop** : Abandonne les éléments excédentaires.
- **onBackpressureLatest** : Ne conserve que les éléments les plus récents.

Ces options sont utiles lorsque la latence et la fraîcheur des données sont critiques.

## Exercices pratiques

1. **Exercice 1 :** Implémentez un système avec une fréquence d'émission de données élevée, en utilisant une stratégie de tamponnage.
  - Critère : Assurez-vous que le consommateur ne soit pas saturé.
2. **Exercice 2 :** Expérimitez avec `onBackpressureDrop` pour comprendre son comportement dans une application Java réactive.
  - Critère : Notez les différences de performances.

## Synthèse et ouverture

En résumé, la Backpressure joue un rôle indispensable dans l'ingénierie des systèmes réactifs, garantissant que les fournisseurs ne satureront pas les consommateurs.

Les capacités à implémenter efficacement des stratégies de Backpressure sont essentielles pour les développeurs cherchant à construire des applications fiables et réactives.

Dans les sections futures, nous explorerons comment combiner la Backpressure avec d'autres concepts réactifs pour améliorer encore les performances de nos systèmes.

## Introduction

Le Project Reactor est une bibliothèque de programmation réactive pour Java, conçue pour faciliter la construction d'applications concurrentes et évolutives.

Elle repose sur les spécifications Reactive Streams et offre des abstractions puissantes pour traiter des flux asynchrones de données.

Ce module explore comment utiliser Project Reactor pour créer des applications réactives efficaces.

## Objectifs pédagogiques

- Comprendre les concepts clés de Project Reactor.
- Manipuler les types principaux de Project Reactor : Mono et Flux.
- Implémenter un modèle de backpressure.
- Développer des applications réactives résilientes et performantes.

# Concepts clés

Project Reactor est basé sur les principes de la programmation réactive.

Les deux types principaux sont :

- **Mono** : Représente 0 ou 1 résultat.

Il est idéal pour les tâches qui produisent un résultat unique ou aucune sortie.

- **Flux** : Équivalent à un observable dans RxJava, il peut représenter une séquence de 0 à N éléments.

Avec Project Reactor, vous pouvez composer des chaînes de traitement de manière déclarative.

## Types principaux

### Mono

- **Description** : Modélise une opération asynchrone qui aboutira à un résultat unique ou à une erreur.
- **Usage** : Adapter pour les appels REST, requêtes de base de données uniques, etc.

### Flux

- **Description** : Gère des séquences de flux pouvant avoir zéro à plusieurs émissions.
- **Usage** : Correspond aux streamings, tickers ou tout autre flux de données continu ou répété.

## Opérations communes

Avec Mono et Flux, vous pouvez réaliser différentes opérations réactives :

- **map()** : Transforme les éléments émis.
- **flatMap()** : Aplatit des Flux ou Mono émis en un seul Flux.
- **filter()** : Filtre les émissions basées sur des conditions définies.

Ces opérations sont la base pour composer des flux de traitement complexes.

## Exemples de code

### Usage de Mono

```
Mono<String> mono = Mono.just("Hello World");
mono.subscribe(System.out::println);
```

- **Explication** : Crée un Mono émettant "Hello World" immédiatement.

## Usage de Flux

```
Flux<Integer> flux = Flux.range(1, 5);
flux.map(i -> i * 2)
    .subscribe(System.out::println);
```

- **Explication** : Génère un Flux de nombres de 1 à 5, les double et les imprime.

## Projet : Backpressure

Pour gérer le backpressure avec Flux :

1. **Stratégies** : Utiliser les opérateurs `onBackpressureBuffer` , `onBackpressureDrop` , etc.

2. **Exemple** :

```
Flux.range(1, 100)
    .onBackpressureBuffer()
    .subscribe(System.out::println,
              System.err::println,
              () -> System.out.println("Done"));
```

- **Explication** : Gestion basique du backpressure en utilisant un buffer.

## Exercices progressifs

1. **Créer un Flux** :

- Créez un Flux qui émet les chiffres de 1 à 10, appliquez une transformation et affichez le résultat.

2. **Gestion d'erreurs** :

- Implémentez une gestion d'erreurs soignée avec `onErrorResume` .

3. **Backpressure avancé** :

- Expérimitez avec `onBackpressureDrop` pour observer l'impact sur des flux à haute fréquence.

## Synthèse

Project Reactor améliore la capacité à gérer des opérations asynchrones en favorisant une approche fonctionnelle et réactive.

En maîtrisant Mono et Flux, vous pouvez construire des applications résilientes et performantes.

Poursuivez avec des cas d'utilisation réels pour perfectionner votre pratique.

## Introduction

RxJava est une implémentation de la Programmation Réactive pour Java.

Elle permet de composer des programmes asynchrones et événementiels avec des flux de données observables.

Ce paradigme améliore la gestion des flux de données dans des applications concurrentes, simplifiant la manipulation de la concurrence et de l'asynchronisme.

# Objectifs Pédagogiques

- Écrire des programmes asynchrones avec RxJava.
- Manipuler les Observables et comprendre leur rôle dans RxJava.
- Implémenter et utiliser les opérateurs courants comme `map`, `flatMap`, `filter`, et `merge`.
- Comprendre le concept de backpressure et comment le gérer en RxJava.

## Concepts de Base

### Observables

Les Observables en RxJava sont la pierre angulaire, permettant de représenter et de gérer des flux de données.

Ils émettent des items sur la base d'une souscription.

**L'Observable va notifier l'observateur par des événements `onNext` , `onCompleted` , ou `onError` .**

### Observateurs

Un observateur souscrit à un Observable pour recevoir ces événements.

**Il réagit à chaque émission de valeurs, à la complétion du flux ou à des erreurs.**

### Opérateurs

Les opérateurs modifient, filtrent, et combinent les flux de données.

Ils permettent de transformer les données ainsi que de contrôler le flux, comme `map` pour transformer chaque élément, `filter` pour filtrer certaines valeurs, ou `flatMap` pour décomposer les flux.

## Opérateurs Courants

### map

L'opérateur `map` transforme chaque item émis.

Par exemple, pour doubler chaque nombre dans un flux:

```
Observable.just(1, 2, 3, 4)
    .map(x -> x * 2)
    .subscribe(System.out::println);
```

## flatMap

`flatMap` transforme chaque item en un Observable, puis combine les résultats.

Idéal pour les transformations asynchrones.

```
Observable.just(1, 2, 3, 4)
    .flatMap(x -> Observable.just(x * 2))
    .subscribe(System.out::println);
```

## filter

Cet opérateur supprime les items du flux en fonction d'un prédictat:

```
Observable.just(1, 2, 3, 4)
    .filter(x -> x % 2 == 0)
    .subscribe(System.out::println);
```

## merge

`merge` combine plusieurs Observables en un seul flux simultané.

```
Observable.merge(
    Observable.just(1, 2),
    Observable.just(3, 4)
).subscribe(System.out::println);
```

# Backpressure et Gestion

Le backpressure survient quand une source d'Observable émet des items plus rapidement que ce que le flux récepteur peut traiter.

RxJava offre des stratégies comme `onBackpressureBuffer` pour gérer ces situations.

# Exemples Pratiques

## Exemple 1 : Système de Notification

Imaginez un système de notifications pour envoyer des messages asynchrones:

```
Observable<String> notifications = Observable.just("Email", "SMS", "Push");
notifications.subscribe(System.out::println);
```

## Exemple 2 : Traitement Parallèle

Utiliser `flatMap` pour exécuter des tâches en parallèle:

```
Observable.range(1, 10)
    .flatMap(x -> Observable.just(x).subscribeOn(Schedulers.computation()))
    .subscribe(System.out::println);
```

## Exercices Pratiques

### Exercice 1 : Créer un Observables

- Créez un Observable qui émet les nombres de 1 à 10 et appliquez `map` pour doubler chaque nombre.
- Critères d'évaluation: le code doit être clair, et correctement commenté.

### Exercice 2 : Gestion de Backpressure

- Implémentez un flux qui utilise `onBackpressureBuffer` pour gérer un flux rapide d'items.

## Synthèse et Ouverture

RxJava simplifie grandement la gestion de la programmation asynchrone.

En maîtrisant ses Observables et opérateurs, vous pouvez construire des applications souples et réactives.

En poursuivant, explorez Project Reactor pour plus de fonctionnalités avancées et une interopérabilité accrue avec d'autres systèmes.

## Introduction

Dans cette section, nous allons explorer la notion d'Observable, un concept clé dans la programmation réactive.

L'Observable permet de gérer des flux de données de manière asynchrone et événementielle.

Il est essentiel pour développer des systèmes réactifs performants capables de réagir aux changements d'état en temps réel.

## Objectifs pédagogiques

- Comprendre le rôle de l'Observable dans la programmation réactive.
- Apprendre à créer et utiliser des Observables en Java.
- Manipuler les flux de données avec des opérateurs réactifs.

## Notion d'Observable

Un Observable est une entité qui émet des éléments auxquels les observateurs peuvent s'abonner.

Cette capacité à gérer les flux asynchrones est au cœur des systèmes réactifs.

- Un Observable émet trois types d'événements : valeurs de données, erreurs et notification de fin.
- Les Observateurs (ou abonnées) consomment ces événements lorsqu'ils surviennent.
- Cela rend le modèle Observable idéal pour manipuler des flux continus de données.

## Création d'un Observable

La création d'un Observable est simple.

Il s'agit de définir de quelle manière les valeurs seront émises.

```
Observable<String> observable = Observable.create(emitter -> {
    emitter.onNext("Message 1");
    emitter.onNext("Message 2");
    emitter.onComplete();
});
```

- `onNext()` émet des valeurs.
- `onComplete()` signale la fin des émissions.
- Les Observables peuvent également émettre des erreurs avec `onError()`.

## Utilisation d'Observable

L'Observateur s'abonne à l'Observable pour recevoir les données émises.

```
observable.subscribe(
    item -> System.out.println("Reçu : " + item),
    error -> System.err.println("Erreur : " + error),
    () -> System.out.println("Flux terminé")
);
```

- L'observateur reçoit les éléments via les méthodes `onNext()`, `onError()`, et `onComplete()`.

## Exemples de code

### Exemple 1 : Émission de valeurs simples

```
Observable.just("Red", "Green", "Blue")
    .subscribe(System.out::println);
```

## Exemple 2 : Gestion des erreurs

```
Observable<String> observableWithError = Observable.create(emitter -> {
    try {
        emitter.onNext("Before Error");
        throw new Exception("Erreur simulée");
    } catch (Exception e) {
        emitter.onError(e);
    }
});
observableWithError.subscribe(
    System.out::println,
    error -> System.err.println("Erreur reçue : " + error)
);
```

## Exercices

- Créer un Observable** : Créez un Observable qui émet les nombres de 1 à 10, et laissez l'observateur afficher chaque nombre.
- Gérer les erreurs** : Modifiez l'Observable pour provoquer une erreur après le 5ème nombre, puis gérez cette erreur dans l'observateur.
- Combiner des Observables** : Créez deux Observables distincts et fusionnez-les pour qu'un seul observateur affiche leurs valeurs combinées.

Critères d'évaluation :

- Correctitude des valeurs émises.
- Gestion appropriée des erreurs.
- Utilisation correcte des opérateurs réactifs.

## Synthèse

Les Observables offrent un moyen puissant et flexible de gérer les flux asynchrones dans les applications réactives.

Ils permettent une gestion fluide des données, des erreurs et du cycle de vie des événements. À travers des exercices progressifs, vous pouvez renforcer votre compréhension et optimiser vos applications Java réactives.

Les prochaines étapes incluent l'exploration des opérateurs réactifs avancés pour une manipulation plus fine des flux.

## Introduction

La fonction `map` en programmation réactive est un opérateur fondamental permettant la transformation des éléments émis par un flux (stream).

Dans ce cours, nous découvrirons comment utiliser cet opérateur pour manipuler les données de manière fluide et intuitive.

La compréhension du `map` est essentielle pour écrire du code réactif efficace et propre.

## Objectifs pédagogiques

- Comprendre le rôle de l'opérateur `map` en programmation réactive
- Savoir appliquer `map` pour transformer des données au sein d'un flux
- Être capable d'expliquer les avantages de l'utilisation de `map`

- Mettre en œuvre des transformations de flux concrètes avec des exemples en Java
- Maîtriser l'enchaînement de transformations multiples avec `map`

## Fonctionnement de `map`

L'opérateur `map` est utilisé pour transformer chaque élément émis par un flux en un nouvel élément via une fonction fournie.

Il s'agit d'une opération de transformation simple mais puissante.

Contrairement à `flatMap`, `map` ne change pas la nature du flux, mais modifie seulement les données.

## Utilisation de `map` en Java

Pour utiliser `map` dans un contexte de flux réactif, vous devez avoir un flux source, tel qu'un `Flux` ou un `Observable`.

Vous lui appliquez ensuite `map` avec une fonction lambda qui décrit comment les éléments doivent être transformés.

```
Flux<String> flux = Flux.just("apple", "banana", "cherry");
flux.map(String::toUpperCase).subscribe(System.out::println);
```

## Transformation de données réactives

Le `map` est idéal pour les cas où chaque élément du flux nécessitera une transformation identique.

Par exemple, convertir toutes les chaînes en majuscules ou ajouter un préfixe commun.

Exemples pratiques :

- Calculer le double de chaque nombre dans un flux d'entiers
- Convertir des objets en un format spécifique en les mappant à d'autres objets

## Exemple de code : Conversion

Voyons un exemple pour convertir un flux d'entiers représentant des températures en Fahrenheit vers Celsius.

```
Flux<Integer> fahrenheitTemps = Flux.just(32, 212, 98);
fahrenheitTemps.map(f -> (f - 32) * 5 / 9)
    .subscribe(temp -> System.out.println(temp + " °C"));
```

Ce code transforme chaque température de Fahrenheit en Celsius et les émet.

## Exemple de code : Complexité

Un autre exemple consiste à mapper des objets `Person` vers leurs noms en majuscules.

```

public class Person {
    private String name;

    // Constructeur, getter et setter
}

Flux<Person> peopleFlux = Flux.just(new Person("Alice"), new Person("Bob"));
peopleFlux.map(person -> person.getName().toUpperCase())
    .subscribe(System.out::println);

```

## Exercice : Transformation simple

- **Objectif :** Transformer un flux de mots pour en sortir leur longueur.
- **Énoncé :** À partir d'un `Flux` de mots, utilisez `map` pour créer un nouveau flux contenant la longueur de chaque mot.

Affichez les longueurs.

- **Critères d'évaluation :** Code fonctionnel, conformité à la consigne, utilisation correcte de `map`.

## Exercice : Transformation complexe

- **Objectif :** Transformez un flux d'objets complexes.
- **Énoncé :** Avec un flux de `Person` objets, utilisez `map` pour créer un flux de `String` contenant le prénom suivi de l'âge.

Affichez le résultat avec une phrase structurée.

- **Critères d'évaluation :** Utilisation correcte de `map`, transformations réussies, bonne structure de phrase.

## Synthèse et ouverture

L'opérateur `map` est un outil puissant pour transformer les données émises par un flux réactif.

En maîtrisant son utilisation, vous pouvez écrire des transformations données de manière claire et concise.

En guise de prochaine étape, vous pourriez explorer l'opérateur `flatMap` pour des scénarios de transformation plus complexes, comme la manipulation de flux imbriqués.

## flatMap

### Introduction

La méthode `flatMap` est un composant clé de la programmation réactive qui permet de transformer des éléments émis par un flux en d'autres flux, puis de les aplatis en un seul flux.

Cela est essentiel pour gérer des flux de données complexes et imbriqués en les simplifiant.

En apprenant `flatMap`, vous enrichirez vos compétences pour créer des applications réactives performantes.

# Objectifs pédagogiques

- Comprendre le fonctionnement de `flatMap` dans le contexte de la programmation réactive
- Appliquer `flatMap` pour transformer et aplatis des flux complexes
- Intégrer `flatMap` dans des flux réactifs en Java avec RxJava ou Project Reactor
- Résoudre des problèmes concrets de transformation de données grâce à `flatMap`

## Contenu détaillé

La méthode `flatMap` est utilisée pour prendre chaque élément d'un flux et le transformer en un autre flux.

Ce processus permet d'imbriquer des flux, puis de les combiner en un seul flux de données linéaire.

**Ce modèle est idéal pour traiter des collections de données où chaque élément peut générer plusieurs nouvelles données.**

## Utilisation de `flatMap`

Dans le contexte de la programmation réactive en Java, `flatMap` est souvent utilisé avec des bibliothèques comme RxJava ou Project Reactor.

**Il permet d'émettre des événements, de les transformer, et de les aplatis, simplifiant ainsi les traitements asynchrones.**

## Comparaison avec `map`

Contrairement à `map` qui transforme un élément en un autre élément, `flatMap` prend un élément et le transforme en un flux, déployant et combinant les résultats en un flux unique.

## Exemples de code

### Exemple simple

Utilisons `flatMap` pour transformer une liste de chaînes de caractères en leurs caractères individuels :

```
import io.reactivex.Observable;

Observable<String> strings = Observable.just("Hello", "World");
strings
    .flatMap(s -> Observable.fromArray(s.split ""))
    .subscribe(System.out::println);
```

### Exemple d'application pratique

Supposons que vous avez une application de gestion de fichiers, et que vous souhaitez lister tous les fichiers d'un ensemble de répertoires :

```
import io.reactivex.Observable;
import java.io.File;

Observable<File> directories = Observable.just(new File("dir1"), new File("dir2"));
directories
    .flatMap(dir -> Observable.fromArray(dir.listFiles()))
    .subscribe(file -> System.out.println(file.getName()));
```

## Exercices pratiques

### Exercice 1 : Transformation de données

Écrivez un programme qui utilise `flatMap` pour transformer une liste de phrases en un flux de mots.

**Utilisez RxJava pour cette tâche.**

### Exercice 2 : Traiter des flux imbriqués

Créez un programme qui prend une liste d'URLs et utilise `flatMap` pour télécharger le contenu de chaque page.

Analysez et affichez les titres des pages.

**Critères d'évaluation :**

- Usage correct de `flatMap` pour aplatiser les flux
- Gestion correcte des erreurs asynchrones
- Optimisation des performances et lisibilité du code

## Synthèse et ouverture

La méthode `flatMap` est un outil puissant dans la programmation réactive, offrant des capacités uniques pour transformer et aplatiser des flux de données.

En comprenant son fonctionnement, vous êtes prêt à aborder des applications complexes et à optimiser le traitement des données réactives.

**Dans la suite, explorez comment intégrer `flatMap` dans un système de notifications réactif complet.**

## Filter

### Introduction

Dans le paradigme de la programmation réactive, le filtre (`filter`) est une opération essentielle.

Il permet de sélectionner les éléments d'une séquence qui répondent à un critère spécifique.

En Java, cela est souvent appliqué aux flux réactifs, permettant une gestion plus fine et plus efficace des données circulant dans votre application.

Intégrer un `filter` dans vos pipelines réactifs peut réduire la charge de traitement en ne traitant que les éléments pertinents pour votre logique métier.

## Objectifs pédagogiques

- Comprendre l'utilité de la méthode `filter`.
- Appliquer `filter` sur un flux de données avec des critères spécifiques.
- Intégrer `filter` dans un pipeline réactif.
- Démontrer comment `filter` peut optimiser les flux de traitement des données.

## Fonctionnement du Filter

La méthode `filter` est utilisée pour sélectionner des éléments d'un flux réactif répondant à une condition.

Elle prend en argument un prédicat, une fonction logique qui renvoie un booléen.

- **Signature :** `filter(Predicate<? super T> predicate)`
- **Type de retour :** Un flux ne contenant que les éléments validé par le prédicat.

Par exemple, pour n'obtenir que les nombres pairs dans un flux, un prédicat testant la divisibilité par deux est utilisé.

## Utilisation avec Streams

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

Stream<Integer> numberStream = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> evenNumberStream = numberStream.filter(n -> n % 2 == 0);

// Résultat : [2, 4]
```

Dans cet exemple, le `filter` isole les nombres pairs du flux initial.

Chaque élément est testé contre le prédicat `(n -> n % 2 == 0)`.

## Filter en Programmation Réactive

Dans des bibliothèques réactives comme RxJava ou Project Reactor, `filter` opère de manière similaire, mais sur des flux de données asynchrones.

Par exemple, avec Project Reactor :

```
import reactor.core.publisher.Flux;

Flux<Integer> positiveNumbers = Flux.just(-1, 2, -3, 4, 5)
    .filter(n -> n > 0);

// Résultat : seulement les nombres positifs [2, 4, 5]
```

Ici, seuls les nombres positifs passent le filtre, réduisant ainsi la charge de traitement des éléments non pertinents.

## Exemple complet avec RxJava

```
import io.reactivex.rxjava3.core.Observable;

Observable<Integer> observable = Observable.just(5, 13, 2, 8, 21);

// Filtre les nombres supérieurs à 10
observable.filter(i -> i > 10)
    .subscribe(System.out::println); // Affiche 13, 21
```

Dans cet exemple, un `filter` sur un `Observable` extrait les nombres supérieurs à 10 et les imprime.

## Exercice Pratique

- Objectif :** Créer un flux d'entiers de 1 à 100 et utilisez `filter` pour n'extraire que les multiples de 5.
- Consigne :** Implémentez cette logique en utilisant Project Reactor ou RxJava.

Vérifiez le résultat en affichant chaque élément filtré.

- Critères de réussite :** Un flux qui affiche uniquement les nombres 5, 10, ..., jusqu'à 100.

## Synthèse

La méthode `filter` est cruciale pour contrôler le flux de données dans la programmation réactive.

En sélectionnant seulement ce qui est pertinent, elle optimise le traitement et rend vos applications plus efficaces.

En maîtrisant `filter`, vous pouvez concevoir des pipelines de traitement de données à la fois réactifs et performants. À mesure que vous progressez, explorez l'intégration de plusieurs `filter` et autres transformations pour enrichir vos compétences en programmation réactive.

## Introduction

Dans cette session, nous allons explorer la notion de "merge" dans le contexte de la programmation réactive en Java.

Le concept de `merge` est crucial pour combiner plusieurs flux de données en un seul flux.

Cela permet de gérer et synchroniser les données provenant de sources multiples de manière réactive, augmentant ainsi l'efficacité et la réactivité de votre application.

## Objectifs pédagogiques

- Comprendre le concept de `merge` en programmation réactive.
- Apprendre à combiner plusieurs flux de données en un seul.
- Être capable d'implémenter et d'utiliser `merge` dans un contexte applicatif Java.
- Appliquer `merge` pour la synchronisation de flux de données hétérogènes.

# Concept de Merge

`merge` est utilisé en programmation réactive pour combiner plusieurs sources de données asynchrones en un flux unique sans respecter l'ordre des événements.

- **Non-bloquant** : Permet la combinaison sans attente bloquante.
- **Indépendant de l'ordre** : Les émissions des flux sourcés sont émises au fur et à mesure qu'elles se produisent.
- **Utilisation courante** : L'agrégation de résultats de plusieurs appels de services asynchrones.

## Implémentation en Java

**En Java, particulièrement avec des bibliothèques comme RxJava et Project Reactor, `merge` est utilisé pour combiner les flux (Observables ou Flux).**

### RxJava Exemple

```
Observable<String> source1 = Observable.just("A", "B", "C");
Observable<String> source2 = Observable.just("1", "2", "3");

Observable<String> merged = Observable.merge(source1, source2);
merged.subscribe(System.out::println);
```

- **Description** : Les flux `source1` et `source2` sont fusionnés en un seul flux avec `Observable.merge`.

## Exemples de Code

### Flux Réactif en Project Reactor

```
Flux<String> flux1 = Flux.just("Apple", "Orange");
Flux<String> flux2 = Flux.just("Circle", "Square");

Flux<String> mergedFlux = Flux.merge(flux1, flux2);
mergedFlux.subscribe(System.out::println);
```

- **Explication** : `flux1` et `flux2` sont fusionnés.

Les événements de chaque flux sont émis dès qu'ils se produisent.

Cet exemple reflète la nature réactive où les types de données peuvent être différents mais partagent le même flux de traitement.

## Exercice Pratique

1. **Objectif** : Créer une application Java utilisant RxJava pour fusionner les flux de données provenant de deux sources différentes (par exemple, des capteurs de température et d'humidité).
2. **Étapes** :
  - Créez deux `Observable` émettant respectivement les températures et les pourcentages d'humidité.
  - Utilisez `Observable.merge` pour combiner ces flux.

- Affichez chaque événement émis par le flux fusionné.

### 3. Critères d'évaluation :

- Utilisation correcte de la méthode `merge`.
- Gestion de l'émission et de la consommation des flux fusionnés sans blocage.
- Clarté et efficacité du code.

## Synthèse et Ouverture

Grâce à la notion de `merge`, nous avons appris à combiner plusieurs flux de façon réactive en Java, ce qui permet de gérer efficacement les flux de données asynchrones.

En maîtrisant `merge`, vous pouvez développer des applications plus réactives et résilientes.

Pour aller plus loin, explorez d'autres opérations réactives comme `combineLatest` ou `zip` pour synchroniser plusieurs flux avec des contraintes spécifiques.

## Introduction

Dans cet exercice, nous allons créer un système de notifications réactif utilisant la programmation réactive en Java.

Ce projet pratique vous permettra d'appliquer les concepts théoriques de la programmation réactive pour gérer des flux de données de manière non-bloquante et responsive.

## Objectifs pédagogiques

- Comprendre les principes de la programmation réactive.
- Créer et gérer des flux de données réactifs.
- Implémenter un système de notifications efficace.
- Utiliser les opérateurs de transformation pour manipuler des flux de données.
- Mettre en œuvre la gestion des erreurs et le contrôle de flux.

## Contenu détaillé: Programmation réactive

La programmation réactive est un paradigme qui permet de traiter des flux d'événements asynchrones.

Elle repose sur la gestion des flux de données en temps réel, les changements dans les données déclenchant automatiquement des opérations définies.

- **Flux de données :** Les flux peuvent être considérés comme des séquences temporelles d'éléments.
- **Réactivité :** Gérer les événements au fur et à mesure de leur apparition, assurant une interaction fluide.
- **Asynchronisme :** Utilisation d'opérations non-bloquantes pour éviter les délais.

## Contenu détaillé: Outils réactifs

Pour ce projet, nous utiliserons des bibliothèques Java telles que Project Reactor ou RxJava qui facilitent la création et la gestion de flux réactifs.

- **Reactive Streams :** Norme pour traiter les flux asynchrones avec backpressure.

- **Project Reactor** : Fournit des types réactifs comme Flux et Mono pour manipuler les flux de données.
- **Opérateurs** : Fonctions pour transformer et combiner les flux.

## Exemples de code: Flux de données

```
Flux<String> notifications = Flux.just("Email reçu", "Message instantané", "Nouveau commentaire");
notifications.subscribe(System.out::println);
```

Cet exemple de base montre comment créer un flux de notifications et y souscrire pour suivre les événements.

## Exemples de code: Transformation et manipulation

```
Flux<String> processedNotifications = notifications
    .map(String::toUpperCase)
    .filter(notif -> notif.contains("EMAIL"));

processedNotifications.subscribe(System.out::println);
```

Cet exemple démontre l'utilisation d'opérateurs `map` et `filter` pour transformer et filtrer les notifications.

## Exemples de code: Gestion des erreurs

```
Flux<String> resilientNotifications = notifications
    .onErrorResume(e -> Flux.just("Erreur : Notification introuvable"))
    .subscribe(System.out::println);
```

Utilisez `onErrorResume` pour gérer les erreurs et fournir un flux alternatif en cas de problème.

## Exercices pratiques

1. Créez un flux réactif émettant différents types d'événements (emails, messages SMS, alertes système).
2. Appliquez des transformations pour identifier et taguer les événements critiques.
3. Implémentez la gestion des erreurs pour assurer la continuité du flux en cas d'échec.

**Critères d'évaluation :** Fonctionnalité réactive intégrée, gestion des erreurs efficace, utilisation correcte des opérateurs.

## Synthèse et ouverture

En complétant cet exercice, vous avez exploré les fondements de la programmation réactive et sa mise en application pour créer un système de notifications réactif.

En approfondissant cette compétence, vous pourrez développer des applications robustes et scalables qui répondent efficacement aux demandes changeantes des utilisateurs.

**Cette maîtrise ouvre la voie à une exploration plus poussée des architectures réactives dans des environnements à grande échelle.**

# Programmation réactive

## Introduction

La programmation réactive est un paradigme axé sur l'architecture d'applications répondant efficacement à des événements en temps réel.

Elle permet de gérer des flux de données de manière asynchrone.

Nous aborderons plusieurs composants clés de ce paradigme :

- **Reactive Streams** pour la manipulation de flux.
- **Backpressure** pour gérer la pression des données.
- **Project Reactor** et **RxJava**, deux bibliothèques populaires pour implémenter des flux réactifs.
- Concepts comme **Observable**, **map**, **flatMap**, **filter**, et **merge** utilisés pour transformer et combiner les flux de données.
- Un exercice pratique : créer un système de notifications réactif.

## Vue d'ensemble

Les sous-notions de la programmation réactive s'articulent autour des flux de données et de leur manipulation :

- **Reactive Streams** et **Backpressure** offrent une structure de base pour traiter les données.
- **Project Reactor** et **RxJava** fournissent des outils pour implémenter ces concepts.
- Opérateurs comme **map**, **flatMap**, **filter**, et **merge** permettent de transformer et combiner les flux.
- L'exercice final mettra ces notions en pratique avec un système de notifications réactif.

## Introduction à VisualVM

VisualVM est un outil performant utilisé pour le monitorage, le profiling, et le debugging des applications Java.

Il attire l'attention des développeurs car il fournit une interface unifiée pour examiner les performances des applications Java, analyser les dumps de heap, et inspecter des threads entre autres fonctionnalités, favorisant ainsi une optimisation efficace.

## Objectifs pédagogiques

- Comprendre les fonctionnalités clés de VisualVM.
- Utiliser VisualVM pour profiler une application Java.
- Analyser et interpréter les résultats du profiling pour une optimisation efficace.
- Déetecter et résoudre les problèmes de performance dans une application Java.

## Découverte de VisualVM

VisualVM offre une interface graphique facilitant l'intégration et le suivi des performances applicatives.

- **Interface unifiée** : Combinaison de plusieurs outils JDK.
- **Fonctionnalités principales** : Profilage, Analyse de heap dump, Surveillance des threads, et Monitoring en temps réel.

Ces fonctionnalités permettent une gestion fine de l'application pour des ajustements précis.

## Démarrage avec VisualVM

Pour utiliser VisualVM, suivez ces étapes :

- **Installation** : Inclus dans le JDK ou peut être téléchargé séparément.
- **Lancement** : Exécuter via la ligne de commande avec `jvisualvm` ou à travers une interface utilisateur.

Vérifiez que votre application est prête à être profilée ou analysée avant de commencer.

## Profilage d'une Application

Le profilage consiste à analyser en profondeur l'exécution d'une application pour identifier les parties du code les plus coûteuses.

- **CPU Profiling** : Identifie les méthodes qui consomment le plus de temps.
- **Memory Profiling** : Suivi de l'affectation et de la libération de la mémoire.

Un bon profilage peut révéler des goulets d'étranglement cachés.

## Cas d'Usage : CPU Profiling

Avec VisualVM, vous pouvez identifier les méthodes les plus consommantes en utilisant le CPU Profiling.

- **Démarrer un Profiling** : Cliquez sur 'Profil' > 'CPU Profiling'.
- **Résultats** : Obtenez des statistiques détaillées sur l'utilisation du processeur par méthode.

Utilisez ces résultats pour optimiser les méthodes critiques de votre application.

## Exemples de Code

```
public class Example {  
    public static void main(String[] args) {  
        for (int i = 0; i < 1000; i++) {  
            performTask();  
        }  
    }  
  
    private static void performTask() {  
        // Une tâche qui pourrait être optimisée  
    }  
}
```

Le profilage avec VisualVM peut montrer que `performTask()` est la plus coûteuse en temps et mérite une optimisation.

# Exercices Pratiques

1. **Installation de VisualVM** : Installez et configurez VisualVM dans votre environnement de développement.
2. **Profiling Pratique** : Profiler une application simple et identifier les goulets d'étranglement.
3. **Heap Analysis** : Réaliser un dump de heap et analyser l'usage mémoire.

Critères : Présentation des résultats, suggestions d'optimisation basées sur les résultats obtenus.

## Synthèse

VisualVM est un outil essentiel pour tout développeur Java cherchant à optimiser les performances applicatives.

En maîtrisant le profilage CPU et l'analyse de mémoire, on peut significativement améliorer l'efficacité d'une application.

Pour aller plus loin, explorez d'autres outils de debugging comme JProfiler ou Java Flight Recorder.

## Introduction

JProfiler est un outil puissant et polyvalent utilisé pour optimiser et déboguer les applications Java.

Il permet d'analyser en profondeur l'utilisation de la mémoire, la performance des threads, et l'efficacité des procédures.

En utilisant JProfiler, vous pouvez identifier les goulets d'étranglement dans votre code et comprendre comment optimiser l'allocation des ressources, ce qui est essentiel pour les applications concurrentes et réactives.

## Objectifs pédagogiques

- Apprendre à installer et configurer JProfiler pour une application Java.
- Analyser la consommation de mémoire et l'utilisation des threads avec JProfiler.
- Identifier et résoudre les goulets d'étranglement en utilisant les fonctionnalités de profilage de JProfiler.
- Optimiser les performances d'une application Java à l'aide de diagnostics précis.

## Installation et configuration

Pour utiliser JProfiler, commencez par télécharger le logiciel depuis le site officiel et suivez les instructions d'installation.

Intégrez-le à votre IDE, tel qu'Eclipse ou IntelliJ IDEA, via des plugins spécifiques.

Pour configurer JProfiler, créez une nouvelle session et associez-la à votre application Java, en définissant les paramètres de l'environnement d'exécution.

## Profilage de la mémoire

JProfiler vous permet de surveiller la consommation de mémoire de votre application.

Il fournit des vues détaillées des allocations d'objets et des fuites de mémoire possibles.

Par exemple, en analysant les 'heap dumps', vous pouvez identifier quelles classes consomment le plus de mémoire, et ajuster votre code pour réduire l'usage excessif.

## Analyse des threads

JProfiler offre une vue complète de tous les threads en cours dans une application.

Vous pouvez surveiller l'état des threads (actif, bloqué, en attente), la durée et les appels associés.

Cette fonctionnalité est cruciale pour comprendre le comportement multitâche et résoudre les problèmes de concurrence, comme les blocages inattendus ou la synchronisation inefficace.

## Optimisation des performances

Utilisez JProfiler pour identifier les principaux goulets d'étranglement en analysant le 'CPU profiling'.

Examinez les méthodes les plus coûteuses en termes de temps d'exécution.

Ce processus vous permet de concentrer vos efforts d'optimisation sur les parties du code qui en bénéficieraient le plus, améliorant ainsi globalement la performance de l'application.

## Exemple : Analyse de mémoire

```
public class MemoryIntensive {  
    public static void main(String[] args) {  
        List<Object> objects = new ArrayList<>();  
        for (int i = 0; i < 10000; i++) {  
            objects.add(new Object());  
        }  
        System.out.println("Object list created.");  
    }  
}
```

Dans cet exemple, nous utilisons JProfiler pour analyser la création de 10 000 objets.

Cela nous aidera à visualiser les allocations et potentiellement identifier un usage inefficace de la mémoire.

# Exemple : Analyse de threads

```
public class ThreadExample {  
    public static void main(String[] args) {  
        Thread thread1 = new Thread(() -> {  
            while (true) {  
                System.out.println("Running");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
        thread1.start();  
    }  
}
```

Ce code crée un thread qui imprime "Running" chaque seconde.

En utilisant JProfiler, nous pouvons examiner l'état et l'impact de ce thread sur les ressources système.

## Exercice : Profilage d'une application

1. Configurez JProfiler pour une application Java de votre choix.
2. Effectuez un profilage de la mémoire et identifiez les classes utilisant le plus de mémoire.
3. Analysez les threads actifs et déterminez s'il existe des blocages.
4. Proposez des optimisations possibles basées sur vos observations.

Critères d'évaluation :

- Compréhension des profils de mémoire et de threads.
- Capacité à identifier et résoudre les problèmes de performance.
- Justification des optimisations suggérées.

## Synthèse

JProfiler est un outil essentiel pour le développement Java, fournissant des informations détaillées sur la performance et l'utilisation des ressources des applications.

Son utilisation compétente permet d'identifier les goulots d'étranglement et d'optimiser le code pour une meilleure performance et une plus grande efficacité. À l'avenir, vous pourriez explorer l'intégration continue de profilage pour une optimisation régulière des applications.

## Introduction

`jcmd` est un outil de diagnostic puissant inclus dans le JDK de Java.

Il permet d'interagir dynamiquement avec les processus Java pour effectuer diverses tâches de gestion, de diagnostic et de contrôle de performance.

Ce cours vous guidera à travers l'utilisation de `jcmd`, vous permettant d'améliorer l'efficacité de vos processus Java.

# Objectifs pédagogiques

- Explorer les fonctionnalités principales de `jcmd`.
- Utiliser `jcmd` pour diagnostiquer des problèmes dans des applications Java.
- Optimiser des applications Java en identifiant des goulets d'étranglement de performance.
- Apprendre à exécuter des commandes pour la collecte de données.

## Qu'est-ce que jcmand ?

`jcmd` est un utilitaire de ligne de commande utilisé pour communiquer avec une JVM active.

Contrairement à d'autres outils de diagnostic, `jcmd` offre une interface unifiée pour émettre des commandes de diagnostic parmi un large éventail.

## Les fonctionnalités de jcmand

`jcmd` fournit un accès à plusieurs types de commandes, telles que :

- L'affichage d'informations sur la configuration de la JVM.
- La collection de profils de pile (thread).
- L'exécution de garbage collection.
- La génération de snapshots de heap.

## Utilisation basique de jcmand

Pour utiliser `jcmd`, commencez par lister tous les processus Java actifs :

```
jcmd
```

Chaque processus a un ID qui peut être utilisé pour envoyer des commandes spécifiques :

```
jcmd <PID> help
```

Cette commande affiche toutes les commandes disponibles pour le processus spécifié.

## Exemple : Lister les threads

Pour lister les threads d'un processus spécifique, utilisez :

```
jcmd <PID> Thread.print
```

Cela vous montrera une liste des threads actifs dans le processus, avec des détails sur chaque pile.

# Exemple : Provoquer Garbage Collection

Voici comment vous pouvez demander à la JVM d'effectuer une collecte des ordures :

```
jcmd <PID> GC.run
```

Cela peut être utile pour libérer de la mémoire immédiatement dans un environnement de test ou de développement.

## Exercice 1 : Utilisation de base

1. Lister les processus Java actifs avec `jcmd`.
2. Choisir un processus (ID).
3. Utiliser `jcmd <PID> help` pour afficher les commandes disponibles.

Critères d'évaluation : Capacité à identifier les processus Java actifs et à explorer les commandes disponibles pour une instance spécifique.

## Exercice 2 : Diagnostic de Thread

1. Utilisez `jcmd` pour lister les threads d'une application Java en cours d'exécution.
2. Identifiez au moins un thread problématique en observant l'état et la pile d'appels.

Critères d'évaluation : Identifier les threads et interpréter les informations obtenues.

## Synthèse

`jcmd` est un outil polyvalent pour gérer et diagnostiquer des applications Java en temps réel.

En maîtrisant `jcmd`, les développeurs peuvent surveiller efficacement les performances, diagnostiquer les goulets d'étranglement et optimiser le comportement de la JVM.

Pour aller plus loin, explorez des cas d'utilisation avancés tels que l'analyse de heap dumps et la détection de deadlocks.

## Introduction

JConsole est un outil de surveillance Java qui permet de superviser et de gérer les performances des applications exécutées sur la machine virtuelle Java (JVM).

Il fournit une interface graphique pour observer divers paramètres comme l'utilisation de la mémoire, le chargement des classes, et l'activité des threads.

JConsole est principalement utilisé pour diagnostiquer les problèmes de performance et pour optimiser les applications Java en temps réel.

## Objectifs pédagogiques

- Comprendre le fonctionnement de JConsole et son utilisation.
- Apprendre à surveiller une application Java avec JConsole.
- Analyser les métriques fournies pour diagnostiquer des problèmes de performance.

- Optimiser l'application à partir des données observées.

## Fonctionnement de JConsole

JConsole est inclus dans le JDK et offre une interface graphique pour la surveillance des applications Java.

Il utilise le protocole JMX (Java Management Extensions) pour collecter des informations sur l'application.

- **Connexion à la JVM** : JConsole peut se connecter à n'importe quelle JVM en cours d'exécution locale ou distante via JMX.
- **Tableaux de bord** : Affiche des graphiques en temps réel pour la surveillance de l'utilisation CPU, de la mémoire, du chargement de classes et de l'activité des threads.
- **Outils de gestion** : Permet de gérer les threads et d'examiner les deadlocks potentiels.

## Surveillance avec JConsole

- **Utilisation mémoire** : Suivi des différents pools de mémoire, y compris l'empreinte du tas et de la pile.
- **Performances des threads** : Affichage de l'état des threads, du nombre de threads actifs et bloqués.
- **Chargement des classes** : Analyse du nombre de classes chargées et déchargées en temps réel.

## Exemples de surveillance

### Exemple 1 : Connexion à une JVM

**Pour se connecter à une JVM locale, démarrez JConsole et sélectionnez l'application cible dans la liste.**

### Exemple 2 : Monitoring de la mémoire

**Ouvrez l'onglet "Memory" pour observer l'historique de l'utilisation du tas et des collectes de déchets.**

### Exemple 3 : Analyse des threads

Dans l'onglet "Threads", vérifiez l'activité et identifiez les threads bloqués pour détecter les deadlocks potentiels.

## Exercices pratiques

### 1. Exercice 1 : Connexion et Surveillance

- Connectez-vous à une application Java locale avec JConsole.
- Identifiez les principales métriques de performance disponibles.

### 2. Exercice 2 : Analyse des performances

- Surveillez l'utilisation de la mémoire pour une application Java causant un niveau élevé de Garbage Collection.
- Proposez des améliorations.

### 3. Exercice 3 : Diagnostics avancés

- Identifiez un thread bloqué et proposez une solution pour résoudre un éventuel deadlock.

Critères d'évaluation :

- Capacité à se connecter correctement à une JVM.
- Aptitude à analyser et interpréter les données de performance.
- Propositions d'optimisation pertinentes.

## Synthèse et ouverture

JConsole est un outil puissant pour le diagnostic et l'optimisation des applications Java.

En surveillant activement les métriques de performance, vous pouvez identifier et résoudre efficacement les goulets d'étranglement.

**Une compréhension approfondie de ces mesures prépare le terrain pour l'utilisation d'outils plus avancés comme Java Flight Recorder pour une analyse plus détaillée des performances.**

## Java Flight Recorder

### Introduction

Java Flight Recorder (JFR) est un outil de profilage intégré dans la JVM qui permet de collecter des données de performance et d'exécution.

JFR offre des capacités avancées pour comprendre le comportement des applications Java en contexte de production avec une faible surcharge.

L'objectif est d'optimiser les performances en identifiant les goulets d'étranglement.

### Objectifs pédagogiques

- Comprendre le fonctionnement de Java Flight Recorder.
- Savoir configurer et démarrer un enregistrement JFR.
- Analyser les enregistrements JFR pour identifier les problèmes de performance.
- Pratiquer l'optimisation basée sur les données collectées.

### Fonctionnement et Architecture

Java Flight Recorder fonctionne en s'intégrant étroitement avec la JVM, étant capable de récupérer une quantité riche de données sur l'exécution et la performance de l'application.

Il utilise une approche d'enregistrement avec des événements qui sont collectés et peuvent être analysés ultérieurement.

- **Événements de la JVM** : Inclut le suivi des threads, l'optimisation du compilateur, les exceptions et l'utilisation de la mémoire.
- **Événements de l'application** : Permet de collecter des informations spécifiques à l'application, telles que le temps passé dans un bloc de code.
- **Surcharge minimale** : Conçu pour fonctionner en production avec un impact minimal sur la performance.

## Configuration et Usage de JFR

Pour utiliser JFR, vous devez généralement passer des options spéciales à la JVM lors du démarrage de votre application.

Cela inclut l'utilisation de différents paramètres pour ajuster les détails capturés par l'enregistrement.

- **Activation de JFR :** `java -XX:StartFlightRecording=name=Profiling,settings=profile,duration=60s -jar app.jar`
- **Paramètres d'enregistrement :** Peut être configuré pour obtenir des détails fins ou une vue d'ensemble selon le besoin.

Avec ces commandes, on peut contrôler la durée, la fréquence d'échantillonnage et bien d'autres aspects de l'enregistrement.

## Analyse des Enregistrements

Une fois le profil capturé, les fichiers JFR peuvent être chargés dans un outil d'analyse comme JMC (Java Mission Control) pour inspecter les données collectées.

Cela permet de visualiser les comportements de l'application et d'identifier les zones nécessitant une attention ou une optimisation.

- **Utilisation de JMC :** Chargez le fichier .jfr pour explorer les threads, la consommation CPU, et les délais de réponse.
- **Détection de goulets d'étranglement :** Repérer facilement où le temps est passé dans votre application et quelles fonctions consomment trop de temps.

## Exemples Pratiques

### 1. Activation Simple :

```
// Démarrage de JFR en ligne de commande  
java -XX:StartFlightRecording=duration=1m,filename=my_recording.jfr -jar MyApp.jar
```

Commence l'enregistrement pendant 1 minute et sauvegarde les données dans `my_recording.jfr`.

### 2. Exemple de Conférence :

Utilisez un profil pour réduire la consommation CPU pendant la collecte en modifiant les paramètres d'échantillonnage.

## Exercices Progressifs

1. **Basique :** Configurez une application simple pour utiliser JFR et capturez un profil de 1 minute.
2. **Analyse Avancée :** Sur un projet de taille moyenne, activez JFR, collectez des données et utilisez JMC pour identifier au moins un potentiel d'optimisation.
3. **Optimisation :** Après l'analyse, effectuez une modification suggérée par les données de JFR et mesurez les améliorations.

Critères de réussite : Identification correcte des goulets d'étranglement et mise en œuvre d'une optimisation mesurable.

## Synthèse

Java Flight Recorder est un outil puissant pour le profilage Java en production, offrant des insights précieux avec un minimum d'impact sur les performances.

L'analyse de ces données peut conduire à des optimisations significatives dans votre application.

Pour aller plus loin, explorez les customisations avancées pour des profils sur mesure et plongez plus profondément dans les analyses offertes par JMC.

# Introduction

Un "heap dump" est un instantané de la mémoire d'un programme Java, capturant l'état de tous les objets présents dans le tas à un moment donné.

Cette opération est essentielle pour analyser les fuites de mémoire, optimiser l'utilisation de la mémoire, et comprendre la structure des objets en cours d'exécution.

Elle est surtout utilisée dans les contextes de debugging et d'optimisation où la gestion de la mémoire est cruciale.

## Objectifs pédagogiques

- Comprendre ce qu'est un heap dump en Java.
- Identifier les outils pour générer et analyser un heap dump.
- Analyser un heap dump pour déceler les fuites de mémoire.
- Optimiser le code Java basé sur les analyses de heap dump.

## Contenu détaillé

Un heap dump est une technique qui permet de capturer une représentation complète de la mémoire d'une application Java à un instant T.

Il collecte toutes les instances d'objets du heap, leurs références et tailles.

## Utilisation des Heap Dumps

- **Détection de Fuites de Mémoire** : Identifier des objets orphelins qui ne sont plus nécessaires mais toujours référencés.
- **Optimisation de la Mémoire**: Vérifier l'efficacité de l'utilisation mémoire, améliorer les algorithmes ou structures de données.
- **Debugging**: Isoler et comprendre des anomalies de comportement en cours d'exécution.

## Outils et Techniques

Pour générer un heap dump en Java, plusieurs outils peuvent être utilisés :

- **jcmd** : Permet de créer un heap dump en exécutant `jcmd <PID> GC.heap_dump filename`.
- **jmap** : Utilisé pour générer un dump avec la commande `jmap -dump:format=b,file=<file> <PID>`.
- **VisualVM** : Offre une interface graphique pour générer et analyser les heap dumps.

Chaque outil présente des avantages selon le contexte d'exécution (environnement de production, développement, etc.).

## Exemples de code

### Exemple : Utilisation de jcmd

Pour capturer un heap dump avec jcmd, utilisez la ligne de commande suivante :

```
jcmd 12345 GC.heap_dump /path/to/heapdump.hprof
```

Ici, 12345 est le PID de l'application Java.

Le fichier .hprof peut ensuite être ouvert avec des outils comme VisualVM pour analyse.

## Exemple : VisualVM

1. **Installer VisualVM.**
2. **Lancer et connecter à la JVM** en cours d'exécution.
3. **Générer un heap dump** via l'interface graphique pour une analyse intuitive.

# Exercices et Travaux Pratiques

## Exercice 1

**Sujet :** Générer un heap dump d'une application Java en cours d'exécution.

**Instructions :**

- Identifiez un processus Java actif sur votre machine.
- Utilisez jmap pour créer un heap dump.
- Ouvrez le dump dans VisualVM et identifiez les plus gros consommateurs de mémoire.

**Critères d'évaluation :**

- Capacité à générer un heap dump avec succès.
- Analyse correcte de la mémoire pour des fuites potentielles.

## Exercice 2

**Sujet :** Analyse avancée de heap dump.

**Instructions :**

- Chargez un heap dump dans VisualVM.
- Identifiez tous les objets non collectés malgré une absence de référence.

**Critères d'évaluation :**

- Identification précise des objets inappropriés.
- Propositions d'optimisation pertinentes.

# Synthèse

Un heap dump est un outil puissant pour l'analyse mémoire en Java.

Il permet de déceler les fuites de mémoire et de comprendre le comportement de la mémoire d'une application.

Maîtriser les outils comme jcmd, jmap, et VisualVM est crucial pour optimiser l'usage mémoire. À l'avenir, nous explorerons comment intégrer ces pratiques dans un flux de travail DevOps pour une supervision continue.

# Thread dump

## Introduction

Un **thread dump** est un outil crucial pour analyser les performances et les problèmes de concurrence dans une application Java.

Il capture l'état actuel de tous les threads de la JVM, ce qui permet de diagnostiquer des blocages ou des performances dégradées.

Cette technique est particulièrement utile dans les scénarios de debugging et d'optimisation des applications concurrentes.

## Objectifs pédagogiques

- Comprendre le concept de thread dump.
- Identifier les situations justifiant un thread dump.
- Analyser un thread dump pour détecter des problèmes.
- Utiliser des outils pour générer et analyser un thread dump.

## Concept de Thread Dump

Un thread dump est une photographie instantanée des threads de la JVM, montrant leur état d'exécution.

Cela inclut les threads en cours, suspendus, ou bloqués.

Il fournit des informations essentielles telles que :

- Le nom des threads.
- L'état (RUNNABLE, BLOCKED, WAITING, etc.).
- La pile d'appel pour chaque thread.

Un thread dump est souvent utilisé pour diagnostiquer :

- Les blocages (deadlocks).
- L'utilisation anormale du CPU.
- Le thread starvation ou les lenteurs.

## Quand Utiliser un Thread Dump

On génère un thread dump dans plusieurs situations clés :

- Lorsque l'application ne répond pas.
- Pour analyser une utilisation excessive du CPU.
- Lors de soupçons de deadlock ou de contention excessive.
- Pour comprendre la répartition du travail entre différents threads.

## Générer un Thread Dump

Avec jcmd :

```
jcmd <pid> Thread.print
```

#### Avec jstack :

```
jstack <pid>
```

Ces commandes affichent l'état actuel de tous les threads d'une application Java en cours d'exécution. `pid` fait référence à l'identifiant du processus Java.

## Analyser un Thread Dump

Lors de l'analyse d'un thread dump :

1. Identifier les threads avec un long temps de blocage.
2. Surveiller les threads souvent en état BLOCKED ou WAITING.
3. Rechercher des cycles de dépendance entre threads (indique un deadlock possible).
4. Vérifier les threads consommatifs en cycles CPU (indiqués par RUNNABLE).

## Exemple 1 : Détection de Deadlock

Imaginons deux threads, A et B, essayant d'accéder à deux ressources dans un ordre inverse :

- **Thread A** tente de verrouiller la ressource 1 puis 2.
- **Thread B** fait l'inverse.

L'analyse du thread dump révélera une impasse, où chaque thread attend une ressource détenue par l'autre.

## Exemple 2 : Usage Intensif du CPU

Un thread RUNNABLE mais consomme beaucoup de CPU peut causer des problèmes de performance.

- **Thread Dump :** Observons le thread "Worker-Thread-1" :

```
"Worker-Thread-1" #17 prio=5 os_prio=0 tid=0x00007f6df0010800 nid=0x3abf runnable [0x00007f6df8d1a000]
  java.lang.Thread.State: RUNNABLE
    at java.util.HashMap.getNode(HashMap.java:580)
    at java.util.HashMap.get(HashMap.java:556)
```

Cela peut indiquer une boucle interne inefficace.

## Exercices Pratiques

### 1. Capture d'un Thread Dump

- Générez un thread dump de votre application Java en utilisant `jcmd`.

Notez les états des threads RUNNABLE et BLOCKED.

### 2. Analyse de Contention

- Sur un code Java multithreadé fourni, modifiez les sections de code pour éviter la contention de ressources détectée via un thread dump.

### 3. Deadlock Simulation

- Créez un simple programme multithreadé qui introduit un deadlock.

Capturez un thread dump et identifiez les threads et leurs dépendances.

#### 4. Optimisation de Performance

- Modifiez un programme Java dont le thread dump indique un usage élevé des ressources CPU, en ajustant la logique de boucles ou la gestion des ressources.

## Synthèse et Ouverture

Le thread dump est un composant indispensable du debugging et de l'optimisation d'applications Java concurrentes.

En identifiant et résolvant les problèmes de thread, on peut significativement améliorer les performances. À l'avenir, explorez comment les thread dumps s'intègrent avec d'autres outils de surveillance pour créer une image complète de la santé de votre application.

## Introduction

La détection de deadlocks est cruciale en programmation concurrente.

Un deadlock se produit lorsque deux ou plusieurs threads attendent indéfiniment des ressources verrouillées par chacun.

Comprendre comment identifier et résoudre un deadlock améliore la fiabilité et la performance des applications Java multithreadées.

## Objectifs pédagogiques

- Comprendre les causes courantes des deadlocks.
- Savoir utiliser des outils pour détecter les deadlocks.
- Acquérir des compétences pratiques pour éviter et résoudre les deadlocks.
- Appliquer des solutions de conception pour prévenir les deadlocks.

## Deadlocks en Java

Un deadlock se produit lorsqu'un thread attend indéfiniment une ressource verrouillée par un autre, qui attend à son tour une ressource verrouillée par le premier.

Les deadlocks peuvent survenir dans des systèmes multithreadés où des ressources limitées sont partagées entre plusieurs threads.

- **Causes** : ordonnancement non déterministe, ressources partagées, verrouillages circulaires.
- **Conséquence** : Les threads arrêtent de progresser, bloquant l'exécution du programme.

## Détection des deadlocks

- **Analyse de l'état des threads** : Utilisez des outils Java pour vérifier les états des threads en cours d'exécution.
- **Utilisation de thread dump** : Capture un snapshot des états des threads et des ressources pour détecter les cycles ou l'attente de ressources.

Les outils comme `jconsole`, `VisualVM` et `jcmand` peuvent être utilisés pour analyser et visualiser les deadlocks.

# Java Thread Dump

Un thread dump peut être généré en utilisant l'outil `jstack` qui fournit des informations sur chaque thread de la JVM.

Le dump identifie les threads en attente de ressources qui ne se libèrent pas, indiquant potentiellement la présence d'un deadlock.

Exécuter `jstack` :

```
jstack <process_id>
```

Cherchez les états comme `BLOCKED` et vérifiez les blocages circulaires.

## Exemple de Deadlock Java

Considérons le code suivant mettant en scène deux threads accédant à deux ressources.

```
class Resource {
    void metA(Resource b) {
        synchronized(this) {
            b.metB();
        }
    }

    synchronized void metB() {
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        final Resource res1 = new Resource();
        final Resource res2 = new Resource();

        Thread t1 = new Thread(() -> res1.metA(res2));
        Thread t2 = new Thread(() -> res2.metA(res1));

        t1.start();
        t2.start();
    }
}
```

Dans ce scénario, `metA` et `metB` provoquent un deadlock lorsque `t2` verrouille `res1` et attend `res2`, pendant que `t1` verrouille `res2` et attend `res1`.

## Exercices pratiques

### 1. Exercice de détection de deadlock :

- En utilisant le code précédemment présenté, introduisez un mécanisme pour détecter le deadlock avec `jstack`.

### 2. Exercice de résolution de deadlock :

- Modifiez l'ordre de verrouillage dans le code pour prévenir le deadlock. Évaluez votre solution avec des tests.

### Critères d'évaluation :

- Capacité à générer et analyser les threads dumps.
- Conception d'une solution sans deadlock.

# Synthèse

Comprendre les deadlocks et savoir les détecter est essentiel pour garantir la stabilité des applications Java.

En implémentant des solutions comme l'évitement de verrouillages circulaires et en utilisant des outils de surveillance, vous pouvez réduire la probabilité de deadlocks.

Continuons à explorer d'autres techniques avancées de debugging et d'optimisation pour améliorer les performances de nos systèmes.

## Introduction

L'exercice "Identifier et Corriger un Deadlock Multithreadé" vous plongera dans les complexités de la programmation concurrente en Java.

Un deadlock est une situation où deux threads ou plus se bloquent éternellement, chacun attendant que l'autre libère des ressources.

Comprendre et résoudre ces deadlocks est crucial pour garantir que les applications concurrentes fonctionnent correctement et efficacement.

## Objectifs pédagogiques

- Analyser du code multithreadé pour détecter des deadlocks
- Appliquer des techniques de debugging pour identifier les causes des deadlocks
- Implémenter des solutions pour prévenir et résoudre les deadlocks
- Évaluer le fonctionnement correct du code multithreadé après correction

## Comprendre les Deadlocks

Un deadlock survient lorsque deux threads ou plus tentent de verrouiller des ressources dans un ordre opposé, provoquant un blocage perpétuel.

En Java, cela implique généralement les objets `synchronized`.

Trouver et résoudre ces boucles d'attente est essentiel pour fiabiliser le code.

## Approches pour la Détection

1. **Thread Dump Analysis** : Utilisation d'outils comme `jconsole` ou `VisualVM` pour analyser les dumps de thread et identifier les deadlocks.
2. **Logs** : Ajouter des journaux pour suivre l'acquisition et la libération des verrous.
3. **Java Concurrency Utilities** : Utiliser des outils de diagnostic intégrés dans Java.

## Prévention et Résolution

- **Ordonnancement Consistant des Verrous** : Structurer le code pour acquérir les verrous dans un ordre cohérent.
- **Time-out sur les Verrous** : Utiliser `tryLock` avec un délai pour éviter l'attente infinie.
- **Thread Hierarchy** : Implémenter une hiérarchie stricte pour l'acquisition des ressources.

## Exemple

```
public class DeadlockExample {  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
  
    public void method1() {  
        synchronized (lock1) {  
            // Simulating work  
            Thread.sleep(50);  
            synchronized (lock2) {  
                // Critical section  
            }  
        }  
    }  
  
    public void method2() {  
        synchronized (lock2) {  
            // Simulating work  
            Thread.sleep(50);  
            synchronized (lock1) {  
                // Critical section  
            }  
        }  
    }  
}
```

Les méthodes ci-dessus démontrent un potentiel deadlock dû à l'ordre opposé des verrous.

## Solution Proposée

```
public class DeadlockResolved {  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
  
    public void safeMethod() {  
        synchronized (lock1) {  
            synchronized (lock2) {  
                // Critical section  
            }  
        }  
    }  
}
```

En modifiant l'ordre de verrouillage, ici consistant, nous évitons le deadlock.

## Exercices Pratiques

1. **Identifier le deadlock** : Prenez l'exemple initial et simulez-le avec plusieurs threads.

Utilisez jconsole pour identifier le deadlock.

2. **Résoudre le deadlock** : Modifiez le code pour corriger le deadlock, assurez-vous de tester l'absence de blocages avec des tests multithreadés.
3. **Refactoring** : Refactorez le code pour implémenter des ReentrantLock avec timeout.

## Synthèse

Les deadlocks constituent un défi courant dans la programmation concurrente, mais avec des techniques de diagnostic appropriées et des bonnes pratiques de code, ils peuvent être résolus efficacement.

Porter attention aux ordres de verrouillage et adopter des techniques de prévention comme l'utilisation de `tryLock` avec délais peut drastiquement réduire les risques de deadlocks dans vos applications Java.

## Ouverture

Explorez des outils avancés de debugging et de profilage, tels que Java Flight Recorder ou JProfiler, pour détecter des deadlocks et autres problèmes de performance.

Ces compétences étendront vos capacités à maintenir et optimiser des applications Java concurrentes à grande échelle.