

Java 21

Déclaration de variables

En Java, les variables doivent être déclarées avec un type spécifique.

Le typage est statique et impose de définir le type des données lors de la déclaration.

- Exemple de déclaration :

```
int age = 25;
```

- Types primitifs : int , double , char , boolean , etc.

Structures de contrôle

Les structures de contrôle permettent de gérer le flux de l'exécution du programme.

- **Conditions** : Utilisez if , else if , et else pour exécuter des blocs de code basés sur des conditions.

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

Boucles en Java

Les boucles sont essentielles pour exécuter du code plusieurs fois.

- **For** : Itération avec un compteur défini.

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

- **While** : Itération basée sur une condition vraie.

```
while (condition) {  
    // code à répéter  
}
```

Méthodes et fonctions

Les méthodes permettent de définir des blocs de code réutilisables.

- Définition d'une méthode :

```
public int add(int a, int b) {  
    return a + b;  
}
```

- Les méthodes doivent spécifier un type de retour ou `void` s'il n'y a pas de retour.

Présentation des chaînes

Les chaînes sont traitées sous forme d'objets de la classe `String`.

- Création :

```
String message = "Hello, World!";
```

- Concaténation :

```
String fullName = firstName + " " + lastName;
```

- Manipulation :

- `length()`
- `toUpperCase()`
- `substring()`

Opérateurs

Les opérateurs sont utilisés pour réaliser des calculs simples ou complexes.

- **Aritmétiques** : `+`, `-`, `*`, `/`, `%`
- **Relationnels** : `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logiques** : `&&`, `||`, `!`

Syntaxe des commentaires

Les commentaires aident à documenter le code.

- Commentaires sur une ligne : `// Ceci est un commentaire`
- Commentaires sur plusieurs lignes :

```
/*
 * Ceci est un commentaire
 * sur plusieurs lignes.
 */
```

Exercices de syntaxe

1. **Variables et types** : Déclarez trois variables de types différents et assignez-leur des valeurs.
2. **Conditions** : Écrivez une condition qui affiche "Bonjour" si une variable booléenne est vraie.
3. **Boucles** : Créez une boucle `for` pour imprimer les nombres de 1 à 10.

En vous familiarisant avec ces éléments fondamentaux, vous posez les bases solides pour progresser dans la programmation en Java.

Introduction aux Classes

Définition d'une Classe

Une classe en Java est un plan ou un modèle pour créer des objets.

Elle définit des propriétés (champs) et des comportements (méthodes) que les objets construits à partir d'elle possèdent.

Structure d'une Classe

Une classe est généralement composée de :

- Attributs : Variables membres de la classe qui contiennent des états de l'objet.

- Méthodes : Fonctions définies dans la classe pour déterminer le comportement de ses objets.
- Constructeur : Une méthode spéciale pour initialiser de nouveaux objets.

Exemple de Classe

Voici un exemple simple de classe en Java :

```
public class Chien {  
    // Attributs  
    String nom;  
    int age;  
  
    // Constructeur  
    public Chien(String nom, int age) {  
        this.nom = nom;  
        this.age = age;  
    }  
  
    // Méthode  
    void aboyer() {  
        System.out.println("Woof!");  
    }  
}
```

Cette classe "Chien" définit deux attributs, un constructeur, et une méthode.

Instanciation d'une Classe

Créer un objet en Java se fait par instanciation de la classe, à l'aide du mot-clé `new`, suivi d'un appel de constructeur :

```
Chien monChien = new Chien("Rex", 3);
```

Cela crée un nouvel objet de la classe `Chien` avec le nom "Rex" et l'âge de 3 ans.

Constructeurs

Un constructeur est utilisé pour initialiser un nouvel objet.

Il porte le même nom que la classe et n'a pas de type de retour.

- Il peut être surchargé pour accepter différents paramètres.

Exemple :

```
public Chien(String nom) {  
    this.nom = nom;  
    this.age = 0; // Définit un âge par défaut  
}
```

Accès aux Attributs et Méthodes

Les attributs et méthodes d'une classe sont typiquement accessibles via l'opérateur . sur l'objet.

Exemple :

```
monChien.aboyer(); // Appelle la méthode aboyer  
System.out.println(monChien.nom); // Accède à l'attribut nom
```

Modificateurs d'Accès

Les modificateurs d'accès déterminent la visibilité des classes, méthodes et attributs.

- public : Accès depuis n'importe quel autre code
- private : Accès seulement au sein de la classe

Exemple :

```
private int age; // Accessible uniquement au sein de la classe Chien
```

Avantages des Classes

Les classes permettent de :

- Modéliser des objets réels du monde.
- Réutiliser du code à travers l'héritage.
- Encapsuler les données pour limiter leur accès direct.

L'usage judicieux des classes simplifie le développement et l'entretien du code Java.

Principe DRY

Introduction au principe DRY

Le principe DRY (Don't Repeat Yourself) est une règle essentielle en programmation.

Elle vise à réduire la répétition de code, minimisant ainsi les erreurs et simplifiant la maintenance.

Avantages du DRY

- Réduction des erreurs grâce à une unique source de vérité.
- Code plus lisible et mieux organisé.
- Maintenance facilitée avec moins de duplication.

Mise en œuvre du DRY

Pour appliquer le DRY en Java, il est crucial d'identifier les répétitions dans votre code.

Utilisez des méthodes, classes, ou interfaces pour regrouper les comportements communs.

Exemple simple

Considérons deux méthodes répétitives.

Il est possible de les refactoriser en extrayant la logique commune dans une seule méthode, puis en l'appelant partout où c'est nécessaire.

Refactorisation

Initialement :

```
public void printWelcome() {  
    System.out.println("Welcome to Java!");  
}  
  
public void printGoodbye() {  
    System.out.println("Goodbye from Java!");  
}
```

Refactorisé :

```
public void printMessage(String message) {  
    System.out.println(message);  
}
```

Conclusion

L'application du principe DRY conduit à un code plus propre.

Il réduit non seulement la maintenance mais encourage aussi la réutilisation.

Un principe fondamental à adopter pour tout développeur.

Principe KISS

Définition du KISS

Le principe KISS signifie "Keep It Simple, Stupid".

Il met l'accent sur la simplicité dans la conception.

L'objectif est de rendre le code plus lisible et maintenable.

Pourquoi utiliser KISS ?

- **Lisibilité accrue** : Un code simple est plus facile à lire.
- **Maintenance facilitée** : Moins de complexité réduit les risques d'erreurs.
- **Évolutivité améliorée** : Un code simple est plus facile à adapter.

Application du KISS en Java

Choisissez des structures simples lors de l'écriture du code.
Évitez de surcharger les méthodes avec des responsabilités inutiles.
Divisez les tâches complexes en sous-tâches simples.

Exemple Java avec KISS

```
public class Calculator {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
}
```

À la place d'une seule méthode pour toutes les opérations, nous en avons une par opération.

Exercice : Refactorisation SOLID

But de l'Exercice

- Appliquer les principes SOLID pour améliorer le design de code.
- Comprendre comment SOLID garantit maintenabilité et flexibilité du code.

Classe Initiale

Imaginez une classe `OrderProcessor` qui gère la création, validation, et envoi de commandes.

Cette classe ne respecte pas les principes SOLID.

```
public class OrderProcessor {  
    public void createOrder() {  
        // logique pour créer une commande  
    }  
  
    public void validateOrder() {  
        // logique pour valider une commande  
    }  
  
    public void sendOrder() {  
        // logique pour envoyer une commande  
    }  
}
```

Tâche 1 : Principe de Responsabilité Unique (SRP)

- Divisez `OrderProcessor` en classes distinctes où chaque classe a une responsabilité unique.
- Assurez-vous qu'une classe ne réalise qu'une seule tâche.

Tâche 2 : Principe de Ouverture/Fermeture (OCP)

- Modifiez les classes pour qu'elles puissent être étendues sans modification.
- Concevez une méthode pour ajouter une nouvelle validation sans changer le code existant.

Exemple du Principe de Liskov Substitution (LSP)

- Assurez-vous que les sous-classes peuvent remplacer les instances de la classe parente sans affecter le comportement.

Tâche 3 : Principe d'Inversion des Dépendances (DIP)

- Refactorez le code pour que `OrderProcessor` dépende d'abstractions et non de classes concrètes.
- Utilisez des interfaces pour atteindre cet objectif.

Révisions Finales

- Vérifiez que chaque classe suit un ou plusieurs des principes SOLID.
- Testez les classes pour confirmer que la refactorisation n'introduit pas de bugs.

Objets

Qu'est-ce qu'un objet ?

- Un objet est une instance d'une classe.
- Il combine des données (attributs) et des comportements (méthodes).
- En Java, les objets permettent de modéliser des concepts réels dans le code.

Création d'un Objet

- Définissez d'abord une classe.
- Utilisez le mot-clé `new` pour instancier un objet.
- Exemple : `Personne p = new Personne();`

Attributs d'Objet

- Les attributs stockent les états d'un objet.
- Ils sont définis dans la classe.
- Exemple : `public String nom;` définit un attribut `nom`.

Méthodes d'Objet

- Les méthodes définissent le comportement d'un objet.
- Elles peuvent manipuler les attributs de l'objet.
- Exemple : `public void parler() { System.out.println("Bonjour !"); }`

Accès aux Membres

- Utilisez le point (.) pour accéder aux attributs et méthodes.
- Exemple : `p.parler();` appelle la méthode `parler` sur l'objet `p`.

Exercice Pratique

Créez une classe `Voiture` avec :

- Attributs `marque` et `vitesse`.
- Méthode `accelerer` qui augmente `vitesse`.
- Instanciez un objet et testez la méthode.

Interfaces

Qu'est-ce qu'une Interface?

Les interfaces définissent un contrat que les classes doivent respecter.

Elles sont complètement abstraites et contiennent principalement des méthodes sans implémentation.

Les classes qui implémentent une interface sont tenues de fournir des définitions concrètes pour toutes les méthodes de l'interface.

Déclaration d'une Interface

Pour déclarer une interface, utilisez le mot-clé `interface` suivi du nom de l'interface.

Voici un exemple simple d'une interface nommée `Volant` :

```
interface Volant {  
    void voler();  
    void atterrir();  
}
```

Implémentation d'Interfaces

Une classe implémente une interface en utilisant le mot-clé `implements`.

Elle doit fournir des implémentations pour toutes les méthodes de l'interface.

Exemples d'implémentations:

```
class Oiseau implements Volant {  
    public void voler() {  
        System.out.println("L'oiseau vole");  
    }  
  
    public void atterrir() {  
        System.out.println("L'oiseau atterrit");  
    }  
}
```

Interfaces et Héritage

Une interface peut hériter d'une ou plusieurs autres interfaces, étendant ainsi ses fonctionnalités.

Toute classe implémentant cette interface doit aussi implémenter les méthodes des interfaces parentes.

```
interface AvancéVolant extends Volant {  
    void planer();  
}
```

Utilisation des Interfaces Comme Types

Les interfaces peuvent être utilisées comme types de référence pour les objets.

Cela permet d'assurer une certaine flexibilité dans le code en autorisant des objets de diverses classes à être traités de manière uniforme.

```
Volant monVolant = new Oiseau();  
monVolant.voler();
```

Interfaces et Polymorphisme

Les interfaces sont au cœur du polymorphisme en Java.

Elles permettent de définir des méthodes polymorphiques, exécutant ainsi un comportement particulier pour chaque classe implémentant l'interface, tout en utilisant une référence d'interface.

Exercice Pratique

Créez une interface appelée `Conducteur` avec les méthodes `demarrer()` et `arreter()`.

Implémentez cette interface dans une classe `Auto`.

Assurez-vous que `Auto` fournit des définitions concrètes pour chaque méthode de l'interface.

Comprendre l'Héritage

Qu'est-ce que l'Héritage ?

- L'héritage permet à une classe de bénéficier des propriétés et méthodes d'une autre classe.
- On parle de classe parente (ou super classe) et de classe enfant (ou sous-classe).
- Favorise la réutilisation du code et l'organisation hiérarchique.

Syntaxe de l'Héritage

```
class SuperClasse {  
    void afficher() {  
        System.out.println("Bonjour de SuperClasse");  
    }  
}  
  
class SousClasse extends SuperClasse {  
    void afficher2() {  
        System.out.println("Bonjour de SousClasse");  
    }  
}
```

- Utilise `extends` pour hériter d'une classe.
- La sous-classe peut avoir ses propres méthodes et utiliser celles de la super classe.

Avantages de l'Héritage

- **Réutilisabilité** : Évite de réécrire du code pour des fonctionnalités similaires.
- **Extensibilité** : Facilite l'ajout de nouvelles fonctionnalités sans modifier le code existant.
- **Maintenabilité** : Simplifie les mises à jour en centrant le code commun.

Héritage et Constructeurs

- Une sous-classe n'hérite pas des constructeurs de sa super classe.
- Peut appeler le constructeur de la super classe via `super()`.

```
class SousClasse extends SuperClasse {  
    SousClasse() {  
        super();  
    }  
}
```

- `super()` doit être la première instruction dans le constructeur de la sous-classe.

Exercice Pratique

Objectif

Créer une hiérarchie de classes "Animal" et "Chien".

Consignes

- Créez une classe `Animal` avec une méthode `deplacement`.
- Créez une sous-classe `Chien` qui hérite de `Animal`.
- Ajoutez une méthode spécifique `aboyer` à `Chien`.
- Instanciez un `Chien` et appelez ses méthodes.

Polymorphisme

Définition du polymorphisme

Le polymorphisme en Java permet l'utilisation d'un objet de différentes manières.

L'objectif est de pouvoir traiter les objets de manière uniforme.

Cela est principalement réalisé grâce aux interfaces et l'héritage.

Utilité du polymorphisme

Le polymorphisme simplifie le code et renforce sa flexibilité.

En utilisant des méthodes génériques, vous pouvez réduire les duplications de code et faciliter la maintenance de votre application.

Exemple de polymorphisme

Imaginons une méthode qui accepte un paramètre de type `Animal` et appelle une méthode `makeSound()`.

Que ce soit un `Dog` ou un `Cat`, chacun aura sa propre implémentation de `makeSound()`.

Implémentation en Java

- Classes : `Animal`, `Dog`, `Cat`.
- Méthode : `makeSound()`.

```

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

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

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Meow");
    }
}

```

Appel polymorphique

Voici comment on créerait et utiliserait les objets de manière polymorphe :

```

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.makeSound();
        myCat.makeSound();
    }
}

```

Avantages

- **Flexibilité** : Ajoutez facilement de nouveaux `Animal` sans changer la logique existante.
- **Maintenance** : Diminuez les erreurs potentielles en réduisant le code dupliqué.

Polymorphisme dans les Interfaces

Les interfaces sont essentielles au polymorphisme.

Une interface définit un contrat que toutes les classes implémentantes doivent respecter, permettant un même traitement pour différents types d'objets.

Conclusion du polymorphisme

Le polymorphisme est un principe fondateur du paradigme orienté objet.

Il favorise un code modulaire et réutilisable, améliorant ainsi la qualité et la maintenabilité des projets logiciels.

Encapsulation

Définition

L'encapsulation est un principe fondamental de la programmation orientée objet.

Elle vise à restreindre l'accès direct aux données d'un objet et à utiliser des méthodes pour interagir avec les données.

Objectifs de l'encapsulation

- Protéger les données contre les modifications non contrôlées.
- Séparer l'interface publique d'une classe de son implémentation privée.
- Faciliter la maintenance et l'évolution du code.

Mise en œuvre en Java

En Java, l'encapsulation est souvent réalisée à l'aide de modificateurs d'accès, comme `private`, et de méthodes publiques appelées getters et setters.

Exemple de code

```
public class Person {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String newName) {  
        name = newName;  
    }  
}
```

Importance des getters et setters

- Ils permettent de contrôler l'accès et de valider les données avant modification.
- Ils offrent un point unique pour modifier l'accès aux données sans changer l'interface publique de la classe.

Avantages de l'encapsulation

- Améliore la sécurité des données.
- Diminue le couplage entre les composants.
- Augmente la flexibilité et la lisibilité du code.

Bonnes pratiques

- Toujours restreindre l'accès direct aux attributs : utilisez `private`.
- Fournir des getters et setters pour manipuler les données.
- Valider les changements de données dans les setters.

Exercice pratique

Créez une classe `BankAccount` avec un attribut `balance` privé.

Ajoutez des méthodes publiques pour déposer et retirer de l'argent, en s'assurant que le solde ne devienne jamais négatif.

Surcharge et Redéfinition

Surcharge : définition

La surcharge de méthodes permet plusieurs méthodes dans une classe ayant le même nom mais avec des paramètres différents.

- Différent nombre de paramètres.
- Différents types de paramètres.

Elle enrichit la flexibilité et la lisibilité du code.

Exemple de Surcharge

Examinez cet exemple de méthode `additionner` surchargée :

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

Ces deux méthodes traitent l'addition pour différents types numériques.

Redéfinition : définition

La redéfinition (ou surclassement) permet de remplacer la méthode d'une classe parente dans une classe enfant.

Utilisé pour fournir une implémentation spécifique pour un comportement déjà défini.

Exemple de Redéfinition

Considérez cet exemple de redéfinition avec une méthode `afficher` :

```
class Animale {  
    void afficher() {  
        System.out.println("Je suis un animal");  
    }  
}  
  
class Chien extends Animale {  
    @Override  
    void afficher() {  
        System.out.println("Je suis un chien");  
    }  
}
```

La méthode `afficher` est redéfinie dans la classe `Chien`.

Surchage vs Redéfinition

- **Surchage** : même nom de méthode, paramètres différents dans la même classe.
- **Redéfinition** : même signature de méthode, classes parent et enfant.

Ces concepts augmentent la flexibilité du design logiciel.

Portée des variables

Définition

La portée d'une variable en Java détermine le contexte dans lequel elle peut être utilisée.

Les variables peuvent être déclarées à différents niveaux de la structure du programme, influençant leur accessibilité et leur durée de vie.

Portée locale

Les variables locales sont déclarées à l'intérieur d'une méthode, d'un constructeur ou d'un bloc.

Elles ne sont accessibles qu'à cet endroit précis.

Leur durée de vie commence à la déclaration et se termine lorsque l'exécution quitte le bloc.

Exemple de portée locale

```
public void afficherNombre() {  
    int nombre = 10; // Variable locale  
    System.out.println(nombre);  
}
```

Dans cet exemple, `nombre` est une variable locale accessible uniquement dans la méthode `afficherNombre`.

Portée de classe

Les variables d'instance sont déclarées à l'intérieur de la classe et hors de ses méthodes.

Elles sont accessibles à toutes les méthodes de la classe.

Leur durée de vie correspond à celle de l'objet.

Exemple de portée de classe

```
public class Exemple {  
    private int nombre; // Variable d'instance  
  
    public void afficher() {  
        System.out.println(nombre);  
    }  
}
```

Dans cet exemple, `nombre` est une variable d'instance accessible dans toutes les méthodes de `Exemple`.

Portée statique

Les variables statiques sont déclarées avec le mot-clé `static`.

Elles sont partagées par toutes les instances de la classe et leur durée de vie commence au chargement de la classe.

Exemple de portée statique

```
public class Exemple {  
    public static int compteur = 0; // Variable statique  
  
    public static void incrementer() {  
        compteur++;  
    }  
}
```

Dans cet exemple, `compteur` est une variable statique, accessible sans créer une instance de `Exemple`.

En résumé

- **Locale** : à l'intérieur des méthodes.
- **Instance** : au niveau de la classe, hors des méthodes, propre à chaque objet.
- **Statique** : au niveau de la classe, partagé entre tous les objets.

Principes SOLID

Introduction à SOLID

Les principes SOLID sont un ensemble de cinq principes de conception destinés à rendre le code plus compréhensible, flexible, et maintenable.

Ils facilitent la gestion de la complexité logicielle.

Principe de Responsabilité Unique

Chaque classe doit avoir une seule responsabilité ou raison de changer.

Cela signifie que chaque classe doit gérer une tâche spécifique et bien définie dans le code.

Principe Ouvert/Fermé

Les logiciels doivent être ouverts à l'extension, mais fermés à la modification.

Cela implique la possibilité d'étendre les fonctionnalités d'une classe sans modifier son code source.

Principe de Substitution de Liskov

Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe mère sans altérer le fonctionnement du programme.

Autrement dit, les sous-type doivent être substituables à leur super-type.

Principe de Ségrégation des Interfaces

Une classe ne doit pas être forcée à implémenter des interfaces qu'elle n'utilisera pas.

Une interface doit être fine, c'est-à-dire spécifique à un ensemble réduit de fonctionnalités.

Principe d'Inversion des Dépendances

Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais tous deux doivent dépendre d'abstractions.

Cette approche permet de découpler les modules de code et facilite la maintenance.

Exemples et Application

Mettez en pratique les principes SOLID en refactorisant du code existant pour améliorer sa structure.

Par exemple, revoyez comment une classe peut être refactorisée pour isoler ses responsabilités ou pour respecter l'inversion des dépendances.

Syntaxe

La syntaxe de Java est la base de toute programmation dans ce langage.

Elle adhère à une structure stricte où chaque commande doit se terminer par un point-virgule.

Java est sensible à la casse, signifiant que `Variable` et `variable` sont distinctes.

Un programme se compose principalement de classes et de méthodes.

Classes

Une classe en Java est un plan pour créer des objets.

Elle définit les propriétés et les comportements que ses objets peuvent avoir.

Par exemple, une classe `Voiture` pourrait comprendre des propriétés comme `couleur` et `marque`, ainsi qu'un comportement comme `accélérer()`.

Objets

Les objets sont des instances de classes.

En déclarant `Voiture maVoiture = new Voiture();`, on crée un objet nommé `maVoiture` basé sur la classe `Voiture`.

Chaque objet a son propre ensemble de valeurs pour les propriétés définies dans sa classe.

Interfaces

Une interface en Java est un contrat que les classes peuvent signer.

Elle déclare des méthodes sans les implémenter, permettant à plusieurs classes d'implémenter la même interface avec un comportement spécifique.

Les interfaces facilitent le polymorphisme et la flexibilité du code.

Héritage

L'héritage permet la réutilisation de code en dérivant de nouvelles classes à partir de classes existantes.

Une classe enfant hérite des propriétés et méthodes de sa classe parent.

Par exemple, une classe `VoitureDeSport` pourrait hériter d'une classe `Voiture`, réutilisant ainsi son code et ses caractéristiques.

Polymorphisme

Le polymorphisme permet d'utiliser une classe enfant là où la classe parent est attendue.

Cela permet de créer du code plus générique et changeable.

Méthodes de la classe parent peuvent être redéfinies pour offrir un comportement spécifique dans la classe enfant, tout en utilisant les types de la classe parent.

Encapsulation

L'encapsulation protège les données d'une classe en restreignant l'accès direct à ses membres.

Le modèle commun consiste à déclarer les propriétés privées et fournir des méthodes publiques `get` et `set` pour les manipuler.

Cela garantit l'intégrité des données et réduit le couplage.

Surcharge et redéfinition

La surcharge signifie définir plusieurs méthodes avec le même nom mais des signatures différentes, permettant des comportements variés.

La redéfinition implique de fournir une nouvelle implémentation pour une méthode héritée, selon les besoins de la classe enfant.

Portée des variables

La portée détermine où une variable peut être accédée.

Les variables peuvent être locales, de classe (statique) ou d'instance.

Une variable locale est accessible seulement à l'intérieur de la méthode ou bloc où elle est déclarée.

Principes SOLID

SOLID est un ensemble de principes de conception logicielle pour améliorer la maintenabilité et l'évolutivité.

Il inclut la Responsabilité unique, la Ouverture/Fermeture, la Substitution de Liskov, la Ségrégation des interfaces, et l'Inversion de dépendance.

Ces principes encouragent un code plus modulaire et adaptable.

Principe DRY

DRY signifie "Don't Repeat Yourself" et encourage la réduction de la duplication dans le code.

En évitant les répétitions, on minimise les erreurs et simplifie la maintenance du code.

Cela conduit typiquement à l'extraction de codes redondants en méthodes ou classes réutilisables.

Principe KISS

KISS signifie "Keep It Simple, Stupid".

Ce principe met l'accent sur la simplicité et la clarté.

Des solutions simples sont souvent plus faciles à comprendre, à tester, et à maintenir.

Cela prévient la complexité excessive dans les projets de développement.

Exercice : Refactorisation SOLID

Refactorez la classe suivante en appliquant les principes SOLID :

```
class ReportGenerator {  
    public void generatePDF() {  
        // Code de génération ici  
        // Conseil : Appliquez le principe de responsabilité unique.  
    }  
    public void generateExcel() {  
        // Code de génération ici  
    }  
}
```

Identifiez les aspects non conformes et proposez des modifications pour respecter SOLID.

Introduction à List

Qu'est-ce qu'une List ?

Une `List` en Java est une interface qui représente une collection ordonnée d'éléments.

Elle permet de stocker des doublons et maintient l'ordre d'insertion.

C'est une des interfaces fondamentales de la bibliothèque des collections.

Principales implémentations

- **ArrayList** : Une liste basée sur un tableau dynamique, offrant une performance rapide en accès indexé.
- **LinkedList** : Basée sur une structure de données chaînée, elle est plus adaptée pour les opérations de manipulation du début ou de fin de liste.

Utilisation courante

```
List<String> myList = new ArrayList<>();  
myList.add("Apple");  
myList.add("Banana");  
System.out.println(myList.get(0)); // Affiche "Apple"
```

Caractéristiques des Lists

- Les éléments peuvent être accédés par leur position.
- Les Lists peuvent contenir des `null`.
- Elles offrent des méthodes de manipulation puissantes comme `add()`, `remove()`, et `get()`.

Manipulation d'une List

Pour ajouter ou supprimer des éléments à des positions spécifiques, utilisez `add(index, element)` et `remove(index)`.

Cela permet une manipulation fine de la liste, indispensable dans certaines applications.

Exemple pratique

```
List<Integer> numbers = new LinkedList<>();  
numbers.add(1);  
numbers.add(2);  
numbers.add(3);  
numbers.remove(1); // Supprime l'élément à l'index 1  
// Liste finale : [1, 3]
```

Bonnes pratiques

- **Choisir la bonne implémentation** : Utilisez `ArrayList` pour un accès rapide et `LinkedList` pour ajouter/supprimer fréquemment des éléments.
- **Utilisation de génériques** : Assurez-vous d'utiliser des génériques pour éviter les erreurs de type, rendant le code plus robuste.

Set

Introduction à Set

- **Definition :** L'interface `Set` est une collection qui ne permet pas de doublons.

Elle est utilisée pour modéliser des ensembles mathématiques.

- **Caractéristiques :** Les éléments ne sont pas ordonnés et chaque élément est unique.

Implémentations de Set

- **HashSet :** Basée sur une table de hachage.

Offre de bonnes performances $O(1)$ pour les opérations de base.

- **TreeSet :** Basée sur une structure d'arbre.

Maintient les éléments triés et offre des performances $O(\log n)$.

- **LinkedHashSet :** Conserve l'ordre d'insertion tout en offrant des performances proches de `HashSet` .

Opérations de base

- **Ajout d'un élément :** `boolean add(E e)` ajoute l'élément si non présent.
- **Suppression d'un élément :** `boolean remove(Object o)` retire l'élément spécifié.
- **Vérification de présence :** `boolean contains(Object o)` vérifie si un élément est présent.

Exemple d'utilisation

```
Set<String> fruits = new HashSet<>();
fruits.add("Pomme");
fruits.add("Banane");
fruits.add("Pomme"); // Pas ajouté, déjà présent
System.out.println(fruits); // Affiche [Pomme, Banane]
```

Comparaison avec List

- **Doublons** : Set n'en permet pas, List oui.
- **Ordonnancement** : List maintient l'ordre d'insertion, Set dépend de l'implémentation utilisée (non ordonné, trié ou ordre d'insertion).

Utilisation pratique

- **Modélisation d'ensembles** : Utile quand l'unicité des éléments est requise.
- **Filtrage des doublons** : Convertir une List en Set pour éliminer les doublons.
- **Vérification rapide** : Grâce à l'absence de doublons, les opérations de vérification sont optimisées.

Gestion d'exceptions

Introduction aux exceptions

Les exceptions en Java sont des événements anormaux qui interrompent le flux normal d'exécution d'un programme.

L'objectif de la gestion d'exceptions est de traiter ces erreurs pour éviter que le programme ne plante.

Cela garantit la robustesse et la fiabilité de votre application.

Types d'exceptions

Java définit trois types d'exceptions :

- **Checked Exceptions** : Doivent être déclarées ou traitées lors de la compilation (ex. IOException).
- **Unchecked Exceptions** : Sont généralement des erreurs de programmation (ex. NullPointerException).
- **Errors** : Indiquent des problèmes graves (ex. OutOfMemoryError).

Utilisation de try-catch

Le bloc `try-catch` est utilisé pour gérer les exceptions.

Le code susceptible de générer une exception est placé dans le bloc `try`, et le bloc `catch` capture et gère l'exception.

```
try {  
    int result = 10 / 0; // Provoque une ArithmeticException  
} catch (ArithmetricException e) {  
    System.out.println("Erreur : division par zéro");  
}
```

Structure try-catch-finally

Le bloc `finally` s'exécute après les blocs `try` et `catch`, qu'une exception ait été levée ou non.

Il est généralement utilisé pour libérer des ressources.

```
try {  
    // Code potentiellement problématique  
} catch (ExceptionType e) {  
    // Gestion de l'exception  
} finally {  
    // Nettoyage des ressources  
}
```

Lancer des exceptions

L'instruction `throw` permet de lancer manuellement une exception.

Cela est utile lorsque vous souhaitez vérifier des conditions spécifiques et signaler une erreur particulière.

```
if (age < 0) {  
    throw new IllegalArgumentException("L'âge ne peut pas être négatif");  
}
```

Exercice pratique

Créez une méthode qui prend un tableau d'entiers et divise chaque élément par un diviseur.

Gérez les exceptions telles que `ArithmeticException` pour la division par zéro et `NullPointerException` pour un tableau nul.

Utilisez des blocs `try-catch` pour assurer une exécution sans erreur.

try-with-resources

Concept de try-with-resources

Introduit dans Java 7, le try-with-resources simplifie la gestion des ressources.

Il garantit la fermeture automatique des ressources, comme les flux de fichiers, ce qui évite les fuites de ressources.

Syntaxe

La syntaxe du try-with-resources utilise des parenthèses pour déclarer et initialiser une ou plusieurs ressources.

Ces ressources doivent implémenter l'interface `AutoCloseable`.

Syntaxe Exemple

```
try (BufferedReader reader = new BufferedReader(new FileReader("file.txt"))) {  
    // Code pour lire le fichier  
} catch (IOException e) {  
    // Gestion des exceptions  
}
```

Fonctionnement

- Les ressources sont déclarées dans les parenthèses du bloc `try`.

- Chaque ressource est fermée automatiquement à la fin du bloc `try`, même en cas d'exception.

Avantages

- Réduit le code boilerplate.
- Diminue le risque de fuites de ressources.
- Facilite la gestion des exceptions en simplifiant le code.

Exercice

Créez un programme qui lit le contenu d'un fichier texte et l'affiche à l'écran.

Utilisez `try-with-resources` pour gérer le flux de lecture.

Exercice : lecture et écriture de fichiers

Objectif

- Lire et écrire des fichiers en Java
- Intégrer une gestion d'erreurs efficace

Lecture d'un fichier

Utiliser `Files.readAllLines()` pour lire un fichier ligne par ligne.

```
Path path = Paths.get("fichier.txt");
try {
    List<String> lignes = Files.readAllLines(path);
    lignes.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Écriture dans un fichier

Écrire du texte dans un fichier en utilisant `Files.write()`.

```
Path path = Paths.get("sortie.txt");
List<String> lignes = Arrays.asList("Ligne 1", "Ligne 2");
try {
    Files.write(path, lignes, StandardOpenOption.CREATE);
} catch (IOException e) {
    e.printStackTrace();
}
```

Gestion d'erreurs

Intégrer `try-catch` pour capturer les exceptions `IOException`.

- Exception courante lors de l'accès à un fichier.
- Toujours vérifier l'existence du fichier et les permissions.

try-with-resources

Utiliser `try-with-resources` pour assurer la fermeture automatique des ressources.

```
try (BufferedReader reader = Files.newBufferedReader(Paths.get("fichier.txt"))) {
    String ligne;
    while ((ligne = reader.readLine()) != null) {
        System.out.println(ligne);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Exercice pratique

1. Créez un fichier texte nommé `notes.txt`.
2. Écrivez-y votre nom et une ligne d'introduction.
3. Lisez le fichier pour afficher son contenu dans la console.
4. Gérez correctement les exceptions potentielles.

Introduction aux Maps

Les maps sont des collections d'associations entre une clé et une valeur.

Chaque clé dans une map est unique et chaque clé est associée à une seule valeur.

Principales Implémentations

- **HashMap** : Utilise une table de hachage.

Rapide pour les opérations de recherche.

- **TreeMap** : Triée selon l'ordre naturel des clés ou un comparateur personnalisé.
- **LinkedHashMap** : Maintient l'ordre d'insertion.

Utilisation d'une HashMap

Initialisation d'une `HashMap` et ajout d'entrées :

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

Accéder et Modifier

Pour accéder à une valeur via une clé ou modifier une entrée :

```
int age = ageMap.get("Alice"); // Accéder  
ageMap.put("Alice", 31); // Modifier
```

Suppression d'Entrées

Suppression d'une entrée dans une map :

```
ageMap.remove("Bob");
```

Vérifier l'Existence

Vérifiez la présence d'une clé ou valeur :

```
boolean exists = ageMap.containsKey("Alice");
```

Parcourir une Map

Parcourir une Map pour afficher toutes les paires clé-valeur :

```
for (Map.Entry<String, Integer> entry : ageMap.entrySet()) {  
    System.out.println(entry.getKey() + ": " + entry.getValue());  
}
```

Différences entre Map et Collection

Les maps ne sont pas des collections au sens conventionnel.

Elles structurent leurs éléments en paires et non en éléments individuels.

Queue

Définition de la Queue

Une Queue est une structure de données qui fonctionne selon le principe FIFO (First In, First Out).

C'est-à-dire que les éléments sont ajoutés à une extrémité et retirés de l'autre.

Cela permet de gérer un flux d'éléments de manière ordonnée.

Utilisation de l'interface Queue

L'interface Queue en Java est utilisée pour représenter les collections qui vont suivre les règles FIFO.

Elle fait partie du package `java.util`, ce qui facilite son intégration dans les applications Java.

Méthodes clés de Queue

Les principales méthodes de l'interface Queue incluent :

- `add()` : ajoute un élément à la queue.
- `remove()` : retire l'élément en tête de la queue.
- `peek()` : renvoie, mais ne retire pas, l'élément en tête.

Exemples d'implémentations

Les classes qui implémentent l'interface Queue incluent `LinkedList`, `PriorityQueue`, et `ArrayDeque`.

Chacune a ses propres avantages, comme le temps de traitement ou l'ordre des éléments.

Exemple pratique : LinkedList comme Queue

`LinkedList` peut être utilisée comme une Queue :

```
Queue<String> queue = new LinkedList<>();
queue.add("Element1");
queue.add("Element2");
System.out.println(queue.remove()); // Affiche "Element1"
```

Elle permet d'insérer et de supprimer des éléments efficacement.

Application : File d'attente

Les Queues sont souvent utilisées pour modéliser des files d'attente, telles que celles que l'on peut rencontrer dans un système de traitement de requêtes ou une file d'attente de messages.

Récapitulatif

La Queue est essentielle pour modéliser des scénarios où l'ordre d'entrée doit être préservé.

En exploitant l'interface Queue et ses implémentations, les développeurs peuvent gérer efficacement les collections d'éléments selon les besoins de leur application.

Stack

Qu'est-ce qu'une Stack ?

Une stack en Java est une structure de données qui suit le principe LIFO (Last In, First Out).

Cela signifie que l'élément ajouté en dernier sera le premier à être retiré.

Les stacks sont utiles lorsque l'ordre des opérations doit être inversé, comme dans l'évaluation d'expressions arithmétiques.

Utilisation de la Classe Stack

Java propose une classe `Stack` qui fait partie du package `java.util`.

Elle hérite directement de la classe `Vector` et offre des méthodes pour manipuler des stacks.

Voici une déclaration et instantiation basique d'une stack :

```
Stack<Integer> stack = new Stack<>();
```

Méthodes de la Classe Stack

La classe `Stack` fournit plusieurs méthodes utiles :

- `push()` : Ajoute un élément au sommet de la stack.
- `pop()` : Retire et retourne l'élément au sommet de la stack.
- `peek()` : Retourne l'élément en haut de la stack sans le retirer.
- `isEmpty()` : Vérifie si la stack est vide.

Exemple d'Utilisation

Examinons un exemple pratique d'utilisation des méthodes stack :

```
Stack<String> stack = new Stack<>();
stack.push("Java");
stack.push("Python");
System.out.println(stack.pop()); // Résultat : Python
System.out.println(stack.peek()); // Résultat : Java
System.out.println(stack.isEmpty()); // Résultat : false
```

Cet exemple démontre comment empiler et dépiler des éléments.

Cas d'Utilisation

Les stacks sont souvent utilisées pour :

- Undo functionalities dans les éditeurs de texte.
- Évaluation d'expressions mathématiques.
- Gestion d'appels récursifs.

Dans ces scénarios, la capacité à accéder rapidement à l'élément le plus récent est cruciale via la stack.

Avantages et Limitations

Avantages :

- Simple à utiliser pour le stockage temporaire de données.
- Fonctionnalité intégrée dans Java.

Limitations :

- Inefficace pour les recherches élémentaires.
- Consomme de la mémoire linéairement avec le nombre d'éléments.

Toujours évaluer si une stack est le meilleur choix en fonction des besoins de votre application.

API I/O (Input/Output)

Introduction à l'API I/O

L'API I/O en Java est essentielle pour la gestion des flux d'entrée et de sortie.

Elle permet de lire et écrire des données à partir de fichiers, de réseaux, etc.

Comprendre cette API est crucial pour manipuler les fichiers et les entrées utilisateur.

Structure de l'API I/O

L'API I/O se divise en deux catégories principales : les Streams et les Readers/Writers.

Les Streams s'occupent des données binaires, tandis que les Readers/Writers traitent les données textuelles.

Cette distinction permet un traitement adapté selon le type de données.

Utilisation des Streams

Les Streams gèrent les I/O binaires.

Les classes principales sont InputStream pour la lecture et OutputStream pour l'écriture.

Elles sont abstraites et ont plusieurs implémentations comme FileInputStream et FileOutputStream.

Exemple d'utilisation de Streams

Pour lire un fichier binaire, utilisez FileInputStream :

```
FileInputStream fis = new FileInputStream("example.bin");
int data = fis.read();
while (data != -1) {
    // Traitement des données
    data = fis.read();
}
fis.close();
```

Utilisation des Readers/Writers

Les Readers et Writers gèrent les I/O de caractères.

Les classes clé sont FileReader pour la lecture et FileWriter pour l'écriture.

Elles facilitent la manipulation de fichiers textuels.

Exemple d'utilisation de Readers

Pour lire un fichier texte, utilisez FileReader :

```
FileReader fr = new FileReader("example.txt");
int data = fr.read();
while (data != -1) {
    // Traitement des caractères
    data = fr.read();
}
fr.close();
```

Buffers pour optimisation

BufferInputStream et BufferOutputStream, ainsi que leurs équivalents pour character, optimisent les opérations I/O en réduisant le nombre d'accès disque en stockant temporairement les données en mémoire.

Exemple d'utilisation de Buffers

Pour une lecture plus performante, utilisez BufferedReader :

```
BufferedReader br = new BufferedReader(new FileReader("example.txt"));
String line = br.readLine();
while (line != null) {
    // Traitement de la ligne
    line = br.readLine();
}
br.close();
```

Écrire un fichier avec PrintWriter

PrintWriter facilite l'écriture de contenu formaté dans un fichier.

Il offre des méthodes pour écrire des primitives et des objets sous forme textuelle, ce qui est idéal pour la génération de rapports ou de logs.

Exemple PrintWriter

Pour écrire un texte dans un fichier :

```
PrintWriter pw = new PrintWriter(new FileWriter("output.txt"));
pw.println("Bonjour tout le monde!");
pw.close();
```

Gestion des exceptions

L'API I/O implique souvent des exceptions, notamment IOExceptions.

Utilisez try-catch-finally pour assurer une bonne gestion des erreurs et la fermeture des ressources.

try-with-resources

La déclaration try-with-resources garantit la fermeture automatique des ressources, simplifiant le code et réduisant les erreurs logiques lors du traitement des fichiers.

```
try (FileReader fr = new FileReader("example.txt")) {
    int data = fr.read();
    // Traitement...
} catch (IOException e) {
    // Gestion des erreurs
}
```

Conclusion API I/O

La maîtrise de l'API I/O est fondamentale pour toute application nécessitant la manipulation de données.

Des classes spécifiques adaptées facilitent la lecture et l'écriture de fichiers, gérant à la fois le texte et les données binaires.

Introduction à NIO

New I/O (NIO) est une collection d'APIs pour le traitement des entrées/sorties en Java.

Introduites dans Java 1.4, elles proposent des fonctionnalités avancées, notamment la manipulation des canaux et des buffers, pour une gestion performante des données.

Avantages de NIO

NIO offre plusieurs avantages :

- **Non-bloquant** : Gère plusieurs canaux de données simultanément.
- **Buffers** : Permettent une manipulation directe et plus efficace des data.
- **Sélecteurs** : Optimisent les opérations I/O en ne traitant que les canaux prêts.

Les Buffers

Les Buffers en NIO sont des conteneurs pour les data.

Ils encapsulent la gestion des data brutes, ce qui permet d'améliorer la performance des applications.

- **Capacité** : Quantité maximale qu'un buffer peut contenir.
- **Limite** : Restriction actuelle sur la lecture/écriture des data.
- **Position** : Indique où la prochaine donnée sera lue/écrite.

Utilisation des Canaux

Contrairement aux streams, les canaux dans NIO peuvent être lus et écrits.

Ils fournissent une manière plus flexible de travailler avec les data.

```
FileInputStream stream = new FileInputStream("data.txt");
FileChannel channel = stream.getChannel();
```

Les canaux permettent de lier les fichiers, réseaux et autres data sources à des buffers.

API NIO.2

Introduit avec Java 7, NIO.2 améliore considérablement l'API NIO, avec un focus particulier sur l'accès facilité aux systèmes de fichiers.

- **Classes Path et Files** : Simplifient les manipulations de fichiers et répertoires.
- **Interfaces WatchService** : Permettent de surveiller les modifications sur un répertoire.

La Classe Path

Path est une interface robuste permettant la manipulation des chemins de fichiers.

```
Path path = Paths.get("documents/report.txt");
```

Elle fournit des méthodes pour naviguer, analyser et manipuler les fichiers de manière intuitive.

Exemple avec NIO.2

Créer un fichier avec NIO.2 est simple.

Voici une courte démonstration :

```
Path path = Paths.get("example.txt");
try {
    Files.createFile(path);
    System.out.println("Fichier créé : " + path.toString());
} catch (IOException e) {
    e.printStackTrace();
}
```

Cet exemple montre comment créer un fichier en toute simplicité avec NIO.2.

Surveillance de répertoires

Avec NIO.2, il est possible de suivre en temps réel les changements dans un répertoire.

```
WatchService watchService = FileSystems.getDefault().newWatchService();
Path path = Paths.get("mon/dossier");
path.register(watchService, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
```

Grâce à `WatchService`, les applications peuvent réagir instantanément aux changements de fichiers.

NIO vs NIO.2

Alors que NIO offre des avantages en termes de performance, NIO.2 améliore la facilité d'utilisation et les fonctionnalités pour les opérations sur les fichiers.

- **NIO** : Plus bas niveau, orienté canaux et buffers.
- **NIO.2** : Inclut un meilleur support pour la gestion des fichiers et surveillance.

Exercice pratique

Créez un programme Java qui surveille un répertoire spécifique pour détecter la création de nouveaux fichiers et les liste dans la console.

Utilisez le `WatchService` de NIO.2 pour cela.

Définition

La sérialisation d'objets en Java consiste à convertir un objet en un flux de bytes.

Cela permet de sauvegarder l'état de l'objet dans un fichier ou de le transmettre par réseau.

C'est une fonctionnalité essentielle pour la persistance des données.

Interface Serializable

Pour sérialiser un objet, celui-ci doit implémenter l'interface `Serializable`.

Cette interface est un marqueur pour indiquer que l'objet peut être transformé en un flux de bytes.

Il n'a pas de méthodes spécifiques à implémenter.

Exemple pratique

Prenons un exemple simple d'une classe `Person` qui sera sérialisée.

Pour que cette classe soit sérialisable, elle doit implémenter `Serializable`.

```
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private int age;

    // Constructors, getters, setters
}
```

Processus de sérialisation

Pour sérialiser un objet, utilisez `ObjectOutputStream` associé à `FileOutputStream`.

Cela permet de sauvegarder l'objet dans un fichier.

```
Person person = new Person("Alice", 30);
try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("person.dat"))) {
    out.writeObject(person);
}
```

Cela écrit l'objet `person` dans un fichier nommé `person.dat`.

Idées clés à retenir

- La sérialisation simplifie la sauvegarde d'objets sur disque ou leur envoi à travers un réseau.
- Tous les objets ne sont pas sérialisables par défaut.

Implémentez `Serializable` pour rendre un objet sérialisable.

- Assurez-vous que toutes les propriétés d'un objet sérialisable sont elles-mêmes sérialisables, ou les marquer `transient` pour les ignorer.

Permissions et sécurité

La sérialisation peut poser des problèmes de sécurité, notamment lors de la désérialisation.

Vérifiez toujours l'origine des données avant de les désérialiser pour éviter l'exécution de code malveillant.

Désérialisation d'objets

Qu'est-ce que la désérialisation ?

La désérialisation est le processus inverse de la sérialisation.

Elle permet de reconstruire un objet Java à partir d'un flux de données.

Cela est essentiel pour lire des objets à partir de fichiers ou de réseaux.

Utilité de la désérialisation

La désérialisation est utilisée dans :

- La communication entre applications via des réseaux.
- La lecture de données d'un fichier où l'état des objets est conservé.
- Le passage d'objets entre différentes parties d'une application.

Exemple simple en Java

Pour désérialiser un objet en Java, suivez ces étapes :

1. Utilisez un `FileInputStream` pour lire le fichier contenant l'objet sérialisé.
2. Enveloppez-le dans un `ObjectInputStream`.
3. Appelez la méthode `readObject()` pour retrouver l'objet et casté au bon type.

Code d'exemple

Voici un exemple de désérialisation :

```
FileInputStream fileIn = new FileInputStream("person.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
Person person = (Person) in.readObject();
in.close();
fileIn.close();
```

Cela charge un objet `Person` précédemment sauvegardé dans le fichier "person.ser".

Exceptions à gérer

La désérialisation peut générer plusieurs exceptions :

- `IOException` : Problèmes d'entrée/sortie.
- `ClassNotFoundException` : La classe de l'objet déserialisé n'est pas trouvée.

Il est important de les gérer, souvent en les encapsulant dans un `try-catch`.

Bonnes pratiques

- Assurez-vous que le `serialVersionUID` est correctement utilisé.

Cela aide à maintenir la compatibilité entre versions de classes.

- Validez les données désérialisées avant de les utiliser, pour des raisons de sécurité.

Exercice pratique

1. Sérialisez un objet Java de votre choix dans un fichier.
2. Créez un programme qui désérialise cet objet.
3. Imprimez les propriétés de l'objet pour vérifier sa restauration correcte.

Essayez de manipuler les exceptions et assurez-vous de tester avec différentes classes d'objets.

Introduction à Optional

Optional en Java est une classe introduite dans Java 8. Elle est utilisée pour représenter un conteneur pouvant contenir une valeur unique ou être vide.

Son principal objectif est de réduire les erreurs dues à l'utilisation de valeurs null.

Création d'un Optional

Il existe plusieurs méthodes pour créer un objet Optional.

Vous pouvez utiliser `Optional.of(value)` pour un objet sécurisé avec une valeur non nulle, ou `Optional.empty()` pour un objet vide.

Utilisation d'Optional

Optional fournit des méthodes pour manipuler la présence ou l'absence de valeur. `isPresent()` vérifie si une valeur est présente, tandis que `ifPresent()` exécute une action si la valeur est non nulle.

Eviter NullPointerExceptions

L'un des objectifs principaux d'Optional est d'éviter les NullPointerExceptions.

Plutôt que de retourner `null`, une méthode peut retourner un Optional, signalant ainsi l'absence potentielle d'une valeur de manière sécurisée.

Exemple Pratique

Un exemple d'utilisation d'Optional est la méthode `String findNameById(String id)`.

**En utilisant Optional, la signature devient
Optional<String> findNameById(String id) , indiquant
clairement que le retour peut être vide.**

APIs standards

List

La classe `List` est une collection ordonnée permettant l'accès à ses éléments par leur indice.

Elle accepte les valeurs dupliquées et préserve l'ordre d'insertion.

Le type `ArrayList` est l'implémentation la plus utilisée.

Par exemple :

```
List<String> list = new ArrayList<>();
list.add("Java");
list.add("Python");
```

Set

Le `Set` est une collection qui n'accepte pas les valeurs dupliquées.

Les implémentations populaires incluent `HashSet` et `TreeSet`, qui n'ont pas d'ordre et un ordre trié respectivement.

Exemple d'utilisation :

```
Set<String> set = new HashSet<>();
set.add("Java");
set.add("Java"); // Ignoré car doublon
```

Map

Map est une collection de paires clé-valeur.

Chaque clé est unique, mais les valeurs peuvent se répéter. HashMap est souvent utilisé pour sa rapidité d'accès.

Exemple :

```
Map<String, Integer> map = new HashMap<>();
map.put("Java", 21);
```

Queue

La Queue est une collection ordonnée utilisée principalement pour les structures de données de type FIFO (First-In-First-Out). LinkedList et PriorityQueue sont des implémentations populaires.

Exemple :

```
Queue<String> queue = new LinkedList<>();
queue.add("First");
queue.add("Second");
```

Stack

Stack est une collection qui suit le principe LIFO (Last-In-First-Out).

Bien que Stack soit une classe, l'utilisation de Deque pour les nouvelles applications est recommandée.

Exemple avec Deque :

```
Deque<String> stack = new ArrayDeque<>();
stack.push("Bottom");
stack.push("Top");
```

API I/O (Input/Output)

L'API I/O standard permet de lire et écrire des données à partir de diverses sources comme fichiers ou réseaux.

Les classes de base incluent `InputStream` et `OutputStream` pour les opérations de byte streams.

Exemple basique de lecture de fichier :

```
try (InputStream in = new FileInputStream("example.txt")) {  
    int data = in.read();  
}
```

API NIO et NIO.2

NIO (New I/O) améliore l'I/O avec des buffers et des canaux, introduisant aussi NIO.2 pour une gestion plus simplifiée des fichiers et systèmes de fichiers.

Exemple d'utilisation avec `Paths` et `Files` :

```
Path path = Paths.get("example.txt");  
List<String> lines = Files.readAllLines(path);
```

Sérialisation d'objets

La sérialisation est le processus de conversion d'un objet en un format pouvant être stocké ou transmis.

Pour permettre à un objet d'être sérialisé, la classe doit implémenter `Serializable`.

Exemple de sérialisation :

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("object.dat"));  
out.writeObject(objectInstance);
```

Désérialisation d'objets

La désérialisation est l'opération inverse de la sérialisation, permettant de recréer des objets à partir de données sérialisées.

Exemple de désérialisation :

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("object.dat"));  
Object obj = in.readObject();
```

Classe Optional

`Optional` est un conteneur pour encapsuler des valeurs qui peuvent être nulles, évitant ainsi les exceptions `NullPointerException`.

Exemple :

```
Optional<String> optional = Optional.of("present");
optional.ifPresent(System.out::println);
```

Gestion d'exceptions

Java utilise des blocs `try`, `catch`, et `finally` pour gérer les exceptions.

Les exceptions peuvent être vérifiées (`Exception`) ou non-vérifiées (`RuntimeException`).

Exemple :

```
try {
    int divide = 10 / 0;
} catch (ArithmaticException e) {
    System.out.println("Cannot divide by zero");
}
```

try-with-resources

C'est un bloc `try` spécial qui assure la fermeture automatique des ressources, simplifiant le code et évitant les fuites de ressources.

Utilisé souvent avec I/O streams.

Exemple :

```
try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
    System.out.println(br.readLine());
}
```

Exercice : Lecture et écriture de fichiers

1. Créez une méthode qui lit le contenu d'un fichier texte et traite les erreurs de manière appropriée.
2. Implémentez la méthode pour écrire dans un fichier, en utilisant `try-with-resources`.
3. Assurez-vous de gérer les exceptions et de tester votre code avec différents scénarios de fichiers.

Introduction aux Lambda

Les expressions Lambda, introduites avec Java 8, permettent d'implémenter des interfaces fonctionnelles de manière concise.

Elles simplifient la syntaxe en remplaçant les classes anonymes.

Syntaxe des Lambdas

Une expression Lambda se compose de trois parties : des paramètres, une flèche (`->`) et un corps.

Le corps peut être une seule expression ou un bloc d'instructions.

Exemple :

```
(paramètres) -> expression  
(x, y) -> x + y
```

Utilisation des Lambdas

Les Lambdas sont souvent utilisées avec des interfaces fonctionnelles, telles que `Runnable` ou `Comparator`.

Elles améliorent la lisibilité et réduisent la verbosité.

Exemple :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
Collections.sort(names, (a, b) -> a.compareTo(b));
```

Avantages des Lambdas

- **Lisibilité** : Code plus concis et expressif.
- **Productivité** : Réduit le code boilerplate.
- **Flexibilité** : Facilite le passage des comportements en paramètre.

Interfaces fonctionnelles

Pour utiliser des Lambdas, une interface fonctionnelle est requise.

C'est une interface avec une unique méthode abstraite, souvent annotée avec `@FunctionalInterface`.

Exemple :

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}
```

Conclusion

Les expressions Lambda sont un élément clé de la programmation fonctionnelle en Java.

Elles offrent un moyen simple de passer des fonctions comme paramètre, améliorant la flexibilité et la lisibilité du code.

Références de méthode

Définition

Les références de méthode en Java 8 sont une fonctionnalité qui améliore la lisibilité du code en permettant de nommer directement une méthode existante.

Elles sont souvent utilisées avec des lambda expressions pour rendre le code plus concis et compréhensible.

Syntaxe

La syntaxe des références de méthode utilise le symbole `::` pour séparer le nom de la classe ou de l'instance du nom de la méthode.

Voici un exemple de la syntaxe générale :

- `ClassName::methodName` pour les méthodes statiques
- `instance::methodName` pour les méthodes d'instance

Types de références

Il existe quatre types de références de méthode en Java :

- Méthode statique : `ClassName::staticMethod`
- Méthode d'instance à partir d'un objet particulier : `instance::instanceMethod`
- Méthode d'instance d'un objet arbitraire d'un type particulier : `ClassName::instanceMethod`
- Constructeur : `ClassName::new`

Exemple pratique

Considérons un exemple simple avec les méthodes statiques et d'instance.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Utilisation de méthodes statiques
names.forEach(System.out::println);

// Utilisation de méthodes d'instance
names.stream().map(String::toUpperCase).forEach(System.out::println);
```

Dans cet exemple, `System.out::println` est une référence à une méthode statique et `String::toUpperCase` est une référence à une méthode d'instance.

Avantages

Les références de méthode rendent le code plus compact et lisible.

En remplaçant les expressions lambda redondantes, elles aident à réduire le risque d'erreur tout en améliorant la maintenance du code.

Limites

Lors de l'utilisation de références de méthode, assurez-vous que la signature de la méthode référencée correspond à celle de l'interface fonctionnelle utilisée.

Cela peut ne pas être possible dans certains cas de surcharge de méthode.

Introduction aux Interfaces Fonctionnelles

Les interfaces fonctionnelles sont une fondation clé en programmation fonctionnelle Java.

Elles permettent l'utilisation de lambda expressions pour simplifier et réduire la verbosité du code.

Function Interface

`Function` prend un argument et renvoie un résultat.

C'est idéal pour les transformations.

Exemple :

```
Function<Integer, String> intToString = x -> "Number: " + x;  
System.out.println(intToString.apply(5)); // Output: Number: 5
```

Predicate Interface

`Predicate` représente une fonction qui évalue une condition.

Il retourne un booléen.

Exemple :

```
Predicate<String> isEmpty = String::isEmpty;  
System.out.println(isEmpty.test("")); // Output: true
```

Supplier Interface

`Supplier` ne prend pas d'argument mais renvoie un résultat.

Utile pour les générateurs de valeurs.

Exemple :

```
Supplier<Double> randomValue = Math::random;  
System.out.println(randomValue.get()); // Output: Random number
```

Consumer Interface

`Consumer` accepte un argument mais ne renvoie pas de résultat.

Souvent utilisé pour appliquer des opérations.

Exemple :

```
Consumer<String> print = System.out::println;
print.accept("Hello, World!"); // Output: Hello, World!
```

Comparaison et Utilisation

- `Function` pour la transformation.
- `Predicate` pour les tests de condition.
- `Supplier` pour fourniture de données.
- `Consumer` pour exécution d'actions.

Ces interfaces facilitent le traitement des collections, le flux de données et la programmation asynchrone.

Exercice Pratique

Créez une petite application où vous utiliserez ces interfaces :

1. Utilisez `Function` pour convertir une liste d'entiers en leurs carrés.
2. Employez `Predicate` pour filtrer les nombres pairs.
3. Implémentez `Supplier` pour générer une liste de nombres aléatoires.
4. Appliquez `Consumer` pour afficher les nombres de la liste finale.

Tracez le flux de données dans chaque étape avec des commentaires afin de bien comprendre la logique.

Stream API

Introduction aux Streams

La Stream API, introduite dans Java 8, permet de traiter des collections de manière déclarative.

Cela simplifie l'application d'opérations comme le filtrage, le mapping, et la réduction sur des collections.

Création d'un Stream

Un Stream peut être créé à partir de collections comme des listes, des arrays, ou encore des générateurs.

Cela permet d'exécuter des opérations de manière fluide et versatile sur les données.

- Exemple : `List<String> maListe = Arrays.asList("a", "b", "c");`
- Utilisation : `Stream<String> monStream = maListe.stream();`

Opérations intermédiaires

Les Streams supportent des opérations intermédiaires qui transforment un Stream en un autre Stream.

Les opérations principales incluent `map`, `filter`, et `distinct`.

- `filter` : Sélectionne les éléments qui remplissent une condition.
- `map` : Transforme chaque élément en appliquant une fonction.

Opérations terminales

Une opération terminale produit un résultat ou un effet d'une chaîne de Streams.

Une fois qu'une opération terminale est appliquée, le Stream est consommé.

- Exemple : `forEach`, `collect`, `reduce`.
- `forEach` : Applique une action à chaque élément.
- `collect` : Convertit le Stream en une autre forme.

map, filter et reduce

Les opérateurs `map`, `filter` et `reduce` sont fondamentaux:

- `map` : Convertit les éléments d'un Stream.
- `filter` : Sélectionne les éléments selon un test.
- `reduce` : Combine les éléments pour produire un résultat unique.

Collectors

Les Collectors sont utilisés pour regrouper les éléments d'un Stream.

Ils permettent de convertir un Stream en une collection spécifique ou de regrouper les données.

- `Collectors.toList()` : Exemple d'un collecteur qui transforme un Stream en liste.

Exercice pratique

Créez un `Stream` à partir d'une liste de chaînes.

Appliquez `filter` pour garder les chaînes dont la longueur est supérieure à 3. Utilisez `map` pour convertir chaque chaîne en majuscule.

Collectez le tout dans une liste.

1. Créez une liste de chaînes.
2. Transformez-la en Stream.
3. Filtrez et mappez.
4. Collectez le résultat.

Opérateurs `map`, `filter`, `reduce`

Stream API

La Stream API en Java permet de traiter des collections de manière fonctionnelle.

Avec les opérateurs `map` , `filter` , et `reduce` , vous pouvez transformer, filtrer, et réduire des données efficacement.

Opérateur `map`

L'opérateur `map` transforme chaque élément d'un stream en appliquant une fonction.

- **Exemple :** Convertir une liste de noms en majuscules.

```
List<String> names = List.of("Alice", "Bob", "Charlie");
List<String> upperNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

Opérateur filter

L'opérateur `filter` sélectionne les éléments d'un stream en fonction d'un prédicat.

- **Exemple :** Filtrer les numéros pairs d'une liste.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());
```

Opérateur reduce

L'opérateur `reduce` agrège les éléments d'un stream en un seul résultat.

- **Exemple :** Calculer la somme d'une liste d'entiers.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
int sum = numbers.stream()
    .reduce(0, Integer::sum);
```

Exemple détaillé

En combinant ces opérateurs, vous pouvez créer des pipelines de traitement de données puissants.

- **Exemple :** Multiplier les nombres pairs par 2 et calculer la somme.

```
int result = numbers.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * 2)
    .reduce(0, Integer::sum);
```

Pratique avec map, filter, reduce

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

Utilisez les opérateurs pour :

1. Convertir les mots en majuscules.
2. Filtrer les mots qui commencent par une voyelle.
3. Compter le nombre de mots restants.

- **Code d'amorçage :**

```
List<String> words = List.of("apple", "banana", "avocado", "cherry", "amos", "echo");
```

Essayez de combiner map , filter , et reduce pour obtenir le résultat souhaité.

Collectors

Introduction aux Collectors

Les `Collectors` sont utilisés dans la Stream API de Java 8 pour regrouper, résumer et collecter des résultats sous une forme mutable comme une liste, un ensemble ou une carte.

Ils facilitent la manipulation et l'agrégation des données.

Comprendre leur rôle est essentiel pour tirer profit des Streams efficacement.

Utilisation de `Collectors.toList()`

`Collectors.toList()` permet de collecter les éléments d'un Stream dans une `List`.

C'est souvent l'option par défaut lorsque vous voulez simplement convertir un Stream en une collection de type Liste.

- Exemple :

```
List<Integer> numbers = Stream.of(1, 2, 3, 4).collect(Collectors.toList());
```

Utilisation de `Collectors.toSet()`

Avec `Collectors.toSet()` , vous collectez les éléments d'un Stream dans un `Set` , éliminant ainsi les doublons.

Utile lorsque l'ordre n'est pas important et que l'unicité des éléments est souhaitée.

- Exemple :

```
Set<Integer> uniqueNumbers = Stream.of(1, 2, 2, 3).collect(Collectors.toSet());
```

Groupement avec `Collectors.groupingBy()`

`Collectors.groupingBy()` classe les éléments d'un Stream basé sur une fonction de classification.

Il retourne une `Map` où la clé est le résultat de la fonction, et la valeur est une liste d'éléments.

- Exemple :

```
Map<Integer, List<String>> groupByLength =  
    Stream.of("cat", "dog", "elephant")  
        .collect(Collectors.groupingBy(String::length));
```

Somme avec `Collectors.summingInt()`

`Collectors.summingInt()` permet de calculer la somme des éléments d'un Stream en utilisant une méthode de conversion d'objet en entier.

- Exemple :

```
int totalLength = Stream.of("cat", "dog", "elephant")  
    .collect(Collectors.summingInt(String::length));
```

Concaténer des chaînes avec `Collectors.joining()`

`Collectors.joining()` fusionne les éléments d'un Stream en une seule chaîne.

Il peut prendre des délimiteurs pour personnaliser la sortie.

- Exemple :

```
String result = Stream.of("Java", "Python", "C++")  
    .collect(Collectors.joining(", "));
```

Exercice : Utilisation de Collectors

1. Récupérez une liste de noms.
2. Classez-les avec `Collectors.groupingBy()` selon la longueur des noms.
3. Obtenez la somme des caractères avec `Collectors.summingInt()`.

4. Concaténez-les en une chaîne avec `Collectors.joining(", ")`.

Essayez de combiner ces opérations pour manipuler et collecter les données efficacement en utilisant les `Collectors`.

API Date & Time (Introduction)

Nouveautés de Java 8

Java 8 a introduit une nouvelle API Date & Time pour corriger les limitations de l'ancienne API Date.

Elle offre des classes immuables et plus intuitives pour manipuler les dates et heures.

LocalDate

`LocalDate` représente une date sans heure, par exemple, "2023-10-25".

Elle est souvent utilisée pour des dates d'anniversaire, des événements futurs, etc.

Exemple de création :

```
LocalDate today = LocalDate.now();
LocalDate specificDate = LocalDate.of(2023, 10, 25);
```

Opérations avec LocalDate

Avec `LocalDate`, on peut ajouter ou soustraire des jours, mois, ou années :

```
LocalDate nextWeek = today.plusWeeks(1);
LocalDate lastMonth = today.minusMonths(1);
```

Ces opérations renvoient de nouveaux objets `LocalDate`, préservant ainsi l'immuabilité.

LocalTime

`LocalTime` est utilisé pour représenter une heure précise dans la journée, sans date.

Exemple : "15:30".

Initialisation :

```
LocalTime now = LocalTime.now();
LocalTime bedtime = LocalTime.of(22, 30);
```

Opérations avec LocalTime

LocalTime permet des manipulations d'heures et de minutes :

```
LocalTime inTwoHours = now.plusHours(2);
LocalTime thirtyMinutesAgo = now.minusMinutes(30);
```

Comme avec LocalDate, toutes les instances sont immutables.

ZonedDateTime

ZonedDateTime intègre une date, une heure, et une zone temporelle.

Il est utile pour les applications internationales :

```
ZonedDateTime meeting = ZonedDateTime.now();
```

Cela inclut des informations comme "2023-10-25T15:30+01:00[Europe/Paris]".

Avantages de l'API Date & Time

- **Immuabilité** : Toutes les classes sont immuables, ce qui renforce la sécurité des applications.
- **Lisibilité** : Syntaxe claire et simplifiée pour la création et manipulation des dates et heures.
- **Fusées horaires** : Gestion intégrée des zones temporelles avec ZonedDateTime .

Exercice pratique

Objectif

Créez un programme Java qui :

1. Affiche la date et l'heure actuelles.
2. Calcul la date 100 jours après la date courante.
3. Détermine l'heure exacte à San Francisco au moment actuel.

Indications

- Utilisez `LocalDate` pour obtenir et manipuler des dates.
- Travaillez avec `ZonedDateTime` pour intégrer et manipuler plusieurs zones horaires.

Pour San Francisco, utilisez la zone `ZoneId.of("America/Los_Angeles")`.

Terminez l'exercice en affichant les résultats en console.

Exercice : pipeline de transformation de données avec Streams

Introduction aux Streams

Java 8 a introduit les Streams, une API pour traiter les collections de manière fonctionnelle et déclarative.

Les Streams permettent d'effectuer des transformations et des agrégations sans écrire de code impératif.

Objectif de l'exercice

Utiliser les Streams pour créer un pipeline de transformation de données.

L'objectif est de filtrer, transformer et collecter les informations d'une liste donnée, en explorant les différentes méthodes fournies par l'API Stream.

Énoncé de l'exercice

1. **Créer une liste** de chaînes contenant des noms aléatoires.
2. **Filtrer la liste** pour ne garder que les noms commençant par la lettre 'A'.
3. **Transformer les noms filtrés** pour qu'ils soient tous en majuscules.

4. **Collecter les résultats** dans une nouvelle liste.

Étapes à suivre

- **Étape 1 : Initialiser la liste**
 - Créez une `List<String>` avec au moins 5 noms.
- **Étape 2 : Filtrage**
 - Utilisez la méthode `filter` pour sélectionner les noms commençant par 'A'.
- **Étape 3 : Transformation**
 - Appliquez la méthode `map` pour transformer chaque nom en majuscules.
- **Étape 4 : Collecte**
 - Utilisez la méthode `collect` pour rassembler le résultat dans une nouvelle liste.

Exemple de solution partielle

1. Initialiser une liste avec des noms :

```
List<String> names = Arrays.asList("Alice", "Bob", "Angela", "Steve", "Alex");
```

2. Filtrer les noms :

```
List<String> filteredNames = names.stream()  
    .filter(name -> name.startsWith("A"))  
    .collect(Collectors.toList());
```

3. Transformer en majuscules :

```
List<String> upperCaseNames = filteredNames.stream()  
    .map(String::toUpperCase)  
    .collect(Collectors.toList());
```

Conseils pour l'exercice

- Pensez à la lisibilité : chaque étape doit être claire et distincte dans votre code.
- Utilisez les méthodes `stream()`, `filter()`, `map()`, et `collect()` pour maintenir un pipeline propre et efficace.
- Assurez-vous de bien comprendre chaque méthode employée et son rôle dans le pipeline.

Extension de l'exercice

Essayez de modifier le pipeline pour inclure :

- Un tri des noms avant de les collecter.
- Un comptage des noms transformés.

- L'utilisation d'autres méthodes de la classe `Collectors` .

Cela vous permettra de renforcer votre compréhension et d'explorer d'autres fonctionnalités avancées des Streams.

Lambda expressions

Les expressions lambda, introduites dans Java 8, offrent une syntaxe concise pour implémenter des interfaces fonctionnelles.

Elles permettent de créer des fonctions anonymes avec une structure `(paramètres) -> {corps}` .

Par exemple, une expression lambda qui ajoute deux entiers pourrait s'écrire `(a, b) -> a + b` .

Références de méthode

Les références de méthode simplifient l'utilisation des lambdas lorsqu'elles font appel à des méthodes existantes.

Elles utilisent le symbole `::` .

Par exemple, `String::toUpperCase` est une référence à la méthode `toUpperCase` des objets de type `String` .

Interfaces fonctionnelles

Les interfaces fonctionnelles sont des interfaces avec un seul méthode abstraite.

Java 8 fournit de nombreuses interfaces fonctionnelles comme `Function` , `Predicate` , `Supplier` , et `Consumer` .

Elles sont essentielles dans l'utilisation des lambda expressions et des Streams.

Stream API

La Stream API permet des opérations fonctionnelles sur les collections en Java.

Elle facilite le traitement en parallèle et en chaîne de données telles que la transformation et l'agrégation, tout en conservant un style de programmation clair et déclaratif.

Opérateurs map, filter, reduce

- `map` assure la transformation des éléments : chaque élément d'un Stream est transformé selon une fonction.
- `filter` sélectionne les éléments selon un prédictat : seuls les éléments qui passent le test sont conservés.
- `reduce` agrège les éléments en un seul résultat, par exemple pour calculer une somme ou produit.

Collectors

Les `Collectors` sont des outils puissants pour accumuler les éléments d'un Stream dans diverses structures de données.

Par exemple, `Collectors.toList()` peut convertir un Stream en une `List`, tandis que `Collectors.groupingBy()` trie les éléments en groupes.

API Date & Time

Java 8 introduit l'API `Date & Time`, fournissant des classes comme `LocalDate`, `LocalTime`, et `ZonedDateTime`.

Elles améliorent la gestion des dates et heures, offrant immutabilité et une syntaxe plus intuitive pour la manipulation et l'affichage des données temporelles.

Exercice : pipeline de données

Créez un pipeline de transformation de données avec les Streams Java :

1. Commencez par une `List<Integer>` de valeurs aléatoires.
2. Filtrez les nombres pairs.
3. Multipliez chaque élément restant par 2.
4. Collectez et affichez les résultats dans une nouvelle liste.

Fondations et rappels

Syntaxe

La syntaxe en Java est stricte et suit une structure particulière.

Chaque programme Java commence par une classe qui doit avoir le même nom que le fichier.

Les méthodes, comme `main`, définissent le comportement.

Les accolades `{}` délimitent les blocs de code, et chaque instruction se termine par un point-virgule `;`.

Classes et Objets

Les classes sont les modèles de conception en Java.

Une classe définit les attributs et méthodes partagées par tous les objets créés à partir d'elle.

Les objets sont des instances de classes qui interagissent avec les autres en invoquant des méthodes.

Interfaces

Les interfaces en Java définissent un contrat que les classes implémentant doivent respecter.

Elles ne contiennent pas de code mais uniquement des déclarations de méthodes.

Utilisées souvent pour permettre le polymorphisme.

Héritage

L'héritage permet à une classe de dériver les caractéristiques d'une autre classe.

La classe dérivée ou "fille" hérite des attributs et méthodes de la classe "mère", favorisant la réutilisabilité du code.

Polymorphisme

Le polymorphisme permet à une même méthode d'avoir différentes implémentations.

En Java, cela se manifeste souvent à travers la surcharge (même méthode, paramètres différents) et la redéfinition (méthode redéfinie dans une sous-classe).

Encapsulation

L'encapsulation consiste à restreindre l'accès à certaines données en rendant les variables d'une classe privées.

Les autres classes interagissent avec celles-ci via des méthodes publiques dites "getter" et "setter".

Surcharge et Redéfinition

La surcharge permet d'avoir plusieurs méthodes du même nom dans une classe, différenciées par leurs paramètres.

La redéfinition, quant à elle, implique de redéfinir une méthode héritée pour qu'elle ait un comportement différent dans une sous-classe.

Portée des Variables

La portée détermine où une variable peut être utilisée.

En Java, il y a trois types de portée: locale (dans un bloc de code), de classe ou instance (dans toute la classe grâce à des variables de classe).

Principes SOLID

Les principes SOLID visent à guider le design des logiciels pour les rendre plus compréhensibles et extensibles :

- **S**ingle Responsibility: Une classe devrait avoir une seule responsabilité.
- **O**pen/Closed: Une classe devrait être ouverte à l'extension mais fermée à la modification.
- **L**iskov Substitution: Les objets peuvent être remplacés par les instances de leurs sous-classes.
- **I**nterface Segregation: Préférer de nombreuses interfaces spécifiques à une interface générale.
- **D**ependency Inversion: Dépendre d'abstractions, pas de concrétions.

Principe DRY

DRY – "Don't Repeat Yourself" – vise à réduire la duplication de code en s'assurant qu'une seule place dans le code contient une information.

Cela augmente la maintenabilité du code.

Principe KISS

"KISS" signifie "Keep It Simple, Stupid".

Il met l'accent sur la simplicité dans le design et la construction des systèmes, évitant la complexité accrue qui peut rendre l'entretien difficile et l'utilisation confuse.

Exercice : SOLID

1. Prenez une classe Java existante qui viole un des principes SOLID.
2. Identifiez les points de rupture en matière de design.
3. Refactorisez la classe en respectant les principes SOLID.
4. Documentez les changements apportés et les principes appliqués.

List

La collection `List` en Java est une interface qui représente une séquence ordonnée d'éléments.

Les implémentations courantes incluent `ArrayList` et `LinkedList`.

Elle permet des opérations telles que l'accès indexé et le parcours d'éléments.

Set

Un `Set` est une collection qui ne contient pas de doublons.

Le `HashSet` est l'une des implémentations les plus courantes.

Il offre des méthodes pratiques pour l'ajout, la suppression et la vérification de la présence d'un élément.

Map

Une `Map` associe des clés à des valeurs. `HashMap` est une implémentation courante.

Elle permet de récupérer, insérer ou supprimer des paires clé-valeur efficacement.

Les clés sont uniques, tandis que les valeurs peuvent être dupliquées.

Queue

`Queue` représente un ordre FIFO (First In, First Out).

Les implémentations populaires incluent `LinkedList` et `PriorityQueue`.

Utile pour les scénarios de traitement séquentiel.

Stack

Bien que depuis Java 1.2, `Vector` et `Stack` soient remplacés par le `Deque`, `Stack` reste une structure de données qui suit l'ordre LIFO (Last In, First Out). `push` et `pop` sont les opérations principales.

API I/O

L'API I/O en Java fournit les classes nécessaires pour lire et écrire des données. `FileReader`, `FileWriter`, `BufferedReader`, et `BufferedWriter` en sont des exemples.

Elle gère les flux d'entrée et de sortie pour divers formats.

API NIO et NIO.2

NIO/NIO.2 introduit des concepts avancés pour la gestion efficace des entrées/sorties, comme les buffers, channels, et l'accès asynchrone.

NIO.2 étend avec le système de fichiers et opérations atomiques.

Sérialisation

La sérialisation convertit un objet en un flux d'octets pour le stockage ou la transmission.

Java le fait via l'interface `Serializable`.

Ceci est utile pour sauvegarder l'état d'un objet.

Désérialisation

La désérialisation est l'inverse de la sérialisation.

Elle recrée un objet Java à partir d'un flux d'octets avec `ObjectInputStream`, permettant de restaurer les états d'objets.

Classe Optional

`Optional` évite les `NullPointerExceptions` en encapsulant potentiellement null.

Il offre une manière claire et fluide d'exprimer l'absence de valeur et d'effectuer des opérations conditionnelles.

Gestion d'exceptions

Le traitement des exceptions en Java utilise `try`, `catch`, `finally`, et `throw`.

Cela évite les plantages en gérant soigneusement les erreurs susceptibles de survenir à l'exécution.

Try-with-resources

Ce bloc assure la fermeture automatique des ressources.

Idéal pour le travail avec des classes qui implémentent `AutoCloseable`.

Il simplifie la gestion des ressources et améliore la lisibilité.

Exercice : I/O et erreurs

1. Créez un programme qui lit un fichier texte ligne par ligne.
2. Écrivez chaque ligne lue dans un autre fichier.
3. Utilisez `try-with-resources` pour gérer les exceptions potentielles et garantir la fermeture des ressources.

Lambda Expressions

Les lambda expressions en Java introduites avec Java 8 permettent de coder des instances de classes anonymes fonctionnelles sous forme de fonctions, simplifiant le code et facilitant la programmation fonctionnelle.

Références de méthode

Les références de méthode permettent d'utiliser des méthodes ou des constructeurs par leur nom.

Elles simplifient le lambda lorsqu'une méthode existante est déjà adaptée pour la tâche.

Interfaces fonctionnelles

Les interfaces fonctionnelles permettent d'utiliser des lambdas.

Parmi celles-ci :

- `Function (T -> R)`
- `Predicate (T -> boolean)`
- `Supplier () -> T`

- `Consumer` (`T -> void`)

Elles définissent exactement une méthode abstraite.

Stream API

La Stream API traite des séquences de données d'une manière déclarative.

Elle permet le mapping, le filtrage, la réduction, et d'autres opérations fonctionnelles sur des collections.

Map, Filter et Reduce

Les opérations de Stream :

- `map` : Transformation des éléments.
- `filter` : Sélectionner des éléments qui répondent à un critère.
- `reduce` : Combinaison des éléments pour produire un seul résultat.

Collectors

Les `Collectors` sont utilisés pour accumuler les résultats de Stream, comme le regroupement ou l'agrégation de données.

C'est un ensemble d'implémentations prédéfinies de l'interface `Collector`.

API Date & Time

Les API Date & Time avec `LocalDate`, `LocalTime`, et `ZonedDateTime` fournissent des moyens pratiques pour manipuler les dates, les heures, et les fuseaux horaires de manière fluide et immuable.

Exercice : Pipeline Stream

1. Chargez une collection de données brutes.
2. Appliquez `filter`, `map` et une opération `reduce` sur la collection.
3. Utilisez `Collectors` pour regrouper les données résultantes selon un critère spécifique.

Modularité avec Project Jigsaw

Introduction à Jigsaw

Project Jigsaw introduit la modularité à Java.

Cela permet de diviser les applications en modules distincts.

Chaque module a des dépendances explicites, ce qui améliore la maintenance et le réutilisabilité du code.

Structure d'un Module

Un module Java inclut un fichier `module-info.java` à sa racine.

Ce fichier définit le nom du module et les dépendances envers d'autres modules.

Cela clarifie ce qu'un module expose ou utilise des autres modules.

Exemple : Fichier `module-info.java`

```
module com.example.moduleA {  
    requires java.base;  
    exports com.example.publicapi;  
}
```

Dans cet exemple, le module `com.example.moduleA` utilise `java.base` et expose le package `com.example.publicapi`.

Avantages de la Modularité

- **Encapsulation Améliorée** : Les modules permettent de cacher les détails d'implémentation.
- **Maintenance Facilitée** : Modification possible des implémentations sans impacter les clients.
- **Démarrage Rapide** : Meilleur temps de démarrage dû à la suppression de code inutilisé.

Compilation et Exécution

Pour compiler un module, utilisez `javac --module-source-path` .

Exécutez-le avec `java --module` .

Ces commandes assurent une reconnaissance appropriée des modules et de leurs dépendances.

Exercice Pratique

1. Créez un module Java simple.
2. Ajoutez un fichier `module-info.java` .
3. Définissez des dépendances et exports.
4. Compilez et exécutez le module pour vérifier la configuration.

Cela renforce la compréhension des concepts modulaires introduits par Jigsaw.

Inférence de type local

Introduction à `var`

Java 10 introduit `var` pour simplifier la déclaration des variables locales.

Avec `var` , le compilateur infère automatiquement le type à partir de l'expression d'initialisation.

Exemple de `var`

```
var message = "Salut, Java 10!";
System.out.println(message);
```

Ici, `message` est automatiquement de type `String` .

Le mot-clé `var` réduit la verbosité sans perdre en lisibilité.

Limitations de var

`var` ne peut pas être utilisé pour les variables d'instance, les paramètres de méthode, ou les variables non initialisées.

Il est strictement pour les variables locales avec initialisation explicite.

```
// Cela NE fonctionne PAS
var count;
count = 10;
```

Avantages de var

- Réduit la verbosité: `var` peut simplifier votre code, en évitant de répéter les types.
- Maintient la sécurité de type: Le langage Java reste typé statiquement même avec `var`.

Bonnes pratiques

- Utilisez `var` lorsque le type réel est évident dans le contexte.
- Évitez `var` si cela dégrade la lisibilité ou la compréhension du code.

Un bon usage contribue à un code compact et maintenable.

Exercice pratique

Écrivez une méthode qui utilise `var` pour lire des entrées de l'utilisateur et les afficher.

Assurez-vous que le type des variables est évident pour garantir la clarté.

Switch expressions (Java 12)

Introduction aux switch expressions

Java 12 introduit les switch expressions, améliorant la traditionnelle structure switch.

Ces expressions rendent le code plus concis et moins sujet aux erreurs.

Elles permettent d'utiliser switch comme une expression qui retourne une valeur.

Syntaxe des switch expressions

La syntaxe des switch expressions utilise `->` pour associer les cas à leurs résultats.

Il n'est pas nécessaire d'avoir des `break` car chaque branche retourne automatiquement.

Les switch expressions peuvent être utilisées pour initialiser des variables.

```
int result = switch (day) {  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    default -> throw new IllegalArgumentException("Invalid day: " + day);  
};
```

Avantages des switch expressions

- **Concision** : Moins de code par rapport aux switches classiques.
- **Moins d'erreurs** : Pas de soucis de "fall-through" grâce à l'absence de `break`.
- **Readability** : Plus clair, les multiples valeurs peuvent être traitées ensemble.

Les switch expressions rendent le code plus maintenable et facile à comprendre.

Switch expressions vs switch statements

Les switch expressions retournent une valeur, contrairement aux statements classiques.

Chaque cas dans une switch expression est indépendant, éliminant les erreurs de `break` manquants.

Elles permettent de faire plus avec moins de code, sans changer l'état des objets.

Exemple pratique

Imaginons un programme qui retourne le nombre de jours restants avant le week-end.

```
int daysUntilWeekend = switch (day) {  
    case MONDAY -> 5;  
    case TUESDAY -> 4;  
    case WEDNESDAY -> 3;  
    case THURSDAY -> 2;  
    case FRIDAY -> 1;  
    case SATURDAY, SUNDAY -> 0;  
    default -> throw new IllegalArgumentException("Invalid day: " + day);  
};
```

Cet exemple montre l'utilisation de cases multiples et l'initialisation d'une variable avec le résultat.

Exercice : implémenter switch expressions

1. Créez une méthode qui prend en entrée un mois (en String).
2. Utilisez une switch expression pour renvoyer le nombre de jours dans ce mois.
3. Gérez les années bissextiles pour le mois de février.

Cet exercice aide à mettre en pratique l'utilisation et les avantages des switch expressions en Java 12.

Garbage Collector G1

G1 GC : Introduction

Le Garbage Collector (GC) G1 a été introduit dans Java 7 et est devenu le collecteur par défaut en Java 9.

Il est conçu pour optimiser les performances des applications nécessitant de gérer de grands volumes de données en RAM.

Contrairement aux autres GC, G1 segmente le tas en plusieurs régions de mémoire de taille égale.

Fonctionnement du G1 GC

- G1 procède à une collecte parallèle pour réduire les interruptions.
- Divise le tas en petites régions, facilitant une gestion plus souple de la mémoire.
- Utilise un modèle de collecte basé sur les "pauses", où l'objectif est de respecter une durée de pause prédéfinie.

Avantages du G1 GC

- Optimisation des performances avec des objectifs de pauses prédéfinies.
- Adaptabilité à la quantité de mémoire disponible grâce à sa segmentation en régions.
- Approprié pour les applications à grande échelle fonctionnant sur des serveurs qui nécessitent une gestion efficace de la mémoire.

Limitations du G1 GC

- Peut être plus compliqué à configurer correctement selon les besoins spécifiques de l'application.
- Dans certains cas, ne peut pas surpasser les performances des GC spécialisés si les pauses ne sont pas limitées.

Configurer G1 GC

Pour activer le G1 GC, utilisez les options suivantes lors de l'exécution de votre application Java :

- `-XX:+UseG1GC` : Pour activer le Garbage Collector G1.
- Ajustez les durées de pauses selon vos besoins avec `-XX:MaxGCPauseMillis=<ms>` . Cela permet de personnaliser la durée maximale de pause afin qu'elle s'aligne sur les exigences de performance de l'application.

Exemple de configuration

Considérons une application nécessitant une gestion fine des pauses :

`java -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -jar MonAppli.jar`
Ici, G1 GC est activé avec une pause maximum de 200 millisecondes, optimisant ainsi l'expérience utilisateur en réduisant les temps d'attente dus à la collecte.

Garbage Collector ZGC

Introduction au ZGC

Le Z Garbage Collector, ou ZGC, est un collecteur de déchets conçu pour minimiser les pauses lors de la collecte.

Introduit avec Java 11, il cible les applications nécessitant une gestion de mémoire à faible latence.

Caractéristiques du ZGC

- **Faible latence** : ZGC vise à maintenir des pauses de l'ordre de quelques millisecondes.
- **Scalabilité** : Il gère efficacement plusieurs terabytes de mémoire.
- **Concurrent** : La majorité du travail de mémoire est réalisée en parallèle de l'application.

Activer le ZGC

Pour utiliser ZGC, vous devez utiliser l'option de ligne de commande suivante lors de l'exécution de votre programme Java :

```
java -XX:+UseZGC -jar mon-application.jar
```

Assurez-vous de tester votre application sous diverses charges avec ZGC pour observer ses effets.

Avantages et Limitations

- **Avantages :**

- Minimise la latence, idéal pour les applications en temps réel.
- Gestion efficace de gros tas de mémoire.

- **Limitations :**

- Initialement compatible uniquement avec Linux/x64. Vérifiez les mises à jour pour d'autres plateformes.
- Peut nécessiter des ajustements spécifiques selon la charge de l'application.

Comparaison avec d'autres Collecteurs

ZGC se distingue par sa capacité à maintenir des pauses courtes par rapport aux autres collecteurs comme G1 ou CMS.

Ceci le rend particulièrement adapté aux systèmes exigeant une réactivité constante.

Exercice Pratique

- **Objectif :** Configurer et tester une application Java avec le ZGC.

- **Étapes :**

- Adaptez une application Java existante pour utiliser ZGC.
- Observez et documentez les performances sous différentes charges.
- Comparez les résultats avec d'autres collecteurs (ex: G1).

Disposez d'une application Java simple pour commencer et accroître sa complexité progressivement pour tester ZGC efficacement.

Exercice : Créer un mini-module Java

Objectif de l'exercice

L'objectif est de pratiquer la création d'un module Java avec des dépendances internes.

Vous allez explorer les concepts de modularité introduits avec Java 9.

Configuration du Projet

1. Créez un répertoire pour votre projet Java.
2. À l'intérieur du répertoire, créez un sous-dossier nommé `src`.
3. Dans `src`, créez un dossier pour notre module, nommez-le `com.example`.

Fichiers de Module

Dans `src/com.example`, créez un fichier `module-info.java` :

```
module com.example {  
    exports com.example.package1;  
    exports com.example.package2;  
}
```

Cela définit un module nommé `com.example` qui expose deux packages.

Création de Packages

Package 1

1. Créez un dossier `package1` dans `src/com.example`.
2. Dans `package1`, ajoutez une classe Java simple qui expose une méthode publique.

Exemple :

```
package com.example.package1;

public class HelloWorld {
    public void greet() {
        System.out.println("Hello, Module!");
    }
}
```

Dépendances Internes

Package 2

1. Créez un dossier package2 dans src/com.example .
2. Dans package2 , ajoutez une classe qui utilise HelloWorld .

Exemple :

```
package com.example.package2;

import com.example.package1.HelloWorld;

public class UseHelloWorld {
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld();
        hello.greet();
    }
}
```

Compilez votre projet en utilisant javac avec les options de module pour observer la structure modulaire.

Compilation et Exécution

Pour compiler votre projet, utilisez ces commandes à partir du répertoire src :

```
javac -d out --module-source-path . $(find . -name "*.java")
```

Pour exécuter votre programme :

```
java --module-path out -m com.example/com.example.package2.UseHelloWorld
```

Cela démontre comment les modules et leurs dépendances fonctionnent ensemble.

De Java 8 à Java 12

Modularité avec Project Jigsaw

Introduit avec Java 9, le Project Jigsaw permet de créer des modules Java.

Un module encapsule un ensemble de packages et établit des dépendances claires avec d'autres modules.

Cela améliore la gestion des dépendances et l'organisation du code.

Inférence de type local

Depuis Java 10, l'inférence de type local est possible grâce au mot-clé `var`.

Il permet de déclarer des variables en laissant le compilateur déterminer leur type.

Cela simplifie le code tout en maintenant la sécurité du typage statique de Java.

Switch expressions

Java 12 introduit les switch expressions, qui simplifient la syntaxe des instructions switch traditionnelles.

Désormais, vous pouvez utiliser `switch` comme une expression retournant une valeur.

Cela rend le code plus concis et moins sujet aux erreurs.

Garbage Collector G1

Le Garbage Collector G1 est devenu le ramasse-miettes par défaut à partir de Java 9. Il est conçu pour être plus performant sur les applications avec de grandes quantités de mémoire et promet des pauses GC plus courtes.

Garbage Collector ZGC

Introduit en Java 11, ZGC est un ramasse-miettes conçu pour gérer de très gros tas de mémoire avec des pauses extrêmement courtes.

Il est idéal pour des applications nécessitant une réponse rapide et un fonctionnement fluide.

Exercice

Créez un mini-module Java, nommé `com.example.math`, incluant au moins un package.

Définissez au moins une méthode publique et démontrez une dépendance interne en important un package.

Text Blocks ("""")

Introduction aux Text Blocks

Les Text Blocks sont introduits à partir de Java 13. Ils permettent de gérer des chaînes de caractères sur plusieurs lignes, simplifiant ainsi la manipulation de textes longs tels que le XML, JSON ou du code HTML.

Ils réduisent également le besoin d'échappement des caractères spéciaux.

Syntaxe des Text Blocks

Un Text Block commence et finit par trois guillemets ("""").

Il ne nécessite pas de concaténation explicite pour les chaînes multi-lignes.

Par exemple :

```
String json = """"
{
    "nom": "Marie",
    "âge": 30
}
""";
```

Avantages des Text Blocks

- **Lisibilité** : Facilite l'écriture et la lecture des longues chaînes de caractères.
- **Moins de concaténation** : Élimine le besoin de mettre des + à chaque fin de ligne.
- **Moins d'échappement** : Les caractères spéciaux n'ont pas besoin d'être précédés par un rétro-slash (\).

Usages Pratiques

Les Text Blocks sont idéaux pour écrire du SQL, du JSON, ou des documents HTML de manière plus propre et lisible :

```
String html = """  
<html>  
    <body>  
        <p>Bonjour tout le monde !</p>  
    </body>  
</html>  
""";
```

Conseils pour Text Blocks

- **Indentation** : Faites attention à l'indentation pour que le contenu soit correctement formaté.
- **Mise en forme** : Gardez un alignement cohérent pour améliorer la lisibilité du bloc texte.

Exercices Text Blocks

1. Créez un Text Block avec une structure JSON listant vos films préférés.
2. Écrivez un document HTML en utilisant un Text Block, comprenant un titre et deux paragraphes.

Ces exercices vous aideront à mieux comprendre l'implémentation et l'utilisation des Text Blocks.

Introduction aux Records

Les records ont été introduits dans Java 14 comme fonctionnalité en avant-première, puis stabilisés en Java 16. Ils simplifient la création de classes qui sont principalement des porteuses de données.

But des Records

Les records réduisent le code boilerplate pour les classes de données en générant automatiquement les méthodes `equals()`, `hashCode()`, et `toString()`.

Syntaxe de Base

La définition d'un record est simple.

Utilisez le mot-clé `record` suivi du nom de la classe et des composants de données :

```
public record Person(String name, int age) {}
```

Avantages des Records

- **Moins de code rédactionnel** : Plus besoin d'écrire manuellement `equals()`, `hashCode()`, et `toString()`.
- **Immutabilité** : Les records sont immuables par défaut, ce qui signifie que leurs champs ne peuvent pas être modifiés après l'initialisation.

Exemple avec Record

```
public record Car(String model, double price) {}

Car car = new Car("Tesla", 69999.99);
System.out.println(car.model()); // Tesla
System.out.println(car.price()); // 69999.99
```

Cas d'Utilisation

Les records sont idéaux pour les objets immuables qui représentent des données simples, comme les coordonnées géographiques, les entrées de liste, ou des réponses d'API.

Pros et Cons

- **Pro** : Syntaxe épurée et lisibilité améliorée.
- **Con** : Pas de prise en charge des changements d'état; pas destiné aux objets complexe avec une logique métier interne.

Comparaison avec les Classes

Contrairement aux classes traditionnelles, les records génèrent automatiquement un constructeur compact, des accesseurs pour chaque champ et des méthodes `equals()`, `hashCode()`, et `toString()`.

Pattern Matching pour `instanceof`

Introduction au Pattern Matching

Le pattern matching simplifie l'utilisation de `instanceof` en Java.

Il permet de réduire les lignes de code et améliore la lisibilité.

Introduit dans Java 16, cette fonctionnalité est utile pour travailler avec différents types d'objets.

Avant le Pattern Matching

Avant, vérifier un type et le caster nécessitait plusieurs lignes.

Exemple :

```
if (obj instanceof String) {  
    String s = (String) obj;  
    // Utilisation de s  
}
```

Cette approche était répétitive et sujet aux erreurs.

Utilisation avec Pattern Matching

Avec le pattern matching, le code devient plus concis.

Exemple :

```
if (obj instanceof String s) {  
    // Utilisation directe de s  
}
```

Le cast se fait automatiquement, réduisant le risque d'erreur.

Avantages et Simplicité

- Réduction du code standard.
- Amélioration de la lisibilité.
- Réduction des risques de cast incorrect.

Cette approche permet de se concentrer sur la logique métier.

Exemple Complet

Supposons que `obj` soit une instance de `Object` pouvant être une `String` :

```
void handle(Object obj) {  
    if (obj instanceof String s) {  
        System.out.println("Chaine de caractères : " + s);  
    } else {  
        System.out.println("Autre type");  
    }  
}
```

Ce code gère les chaînes différemment et sans cast explicite.

Exercice Pratique

Écrivez une méthode qui utilise le pattern matching pour traiter `Number`.

Dans cette méthode, vérifiez si un objet est un `Double` et ajoutez-le à une somme globale.

Ensuite, imprimez le résultat.

Pattern Matching pour switch

Introduction

Le pattern matching pour `switch` est une fonctionnalité introduite pour enrichir le `switch` traditionnel.

Il permet de combiner la puissance des motifs (patterns) avec la structure de contrôle `switch`, rendant le code plus expressif et moins verbeux.

Avantages

- **Lisibilité** : Simplifie la lecture des `switch` complexes en supprimant les conversions de type manuelles.
- **Sécurité** : Diminue le risque d'erreurs en vérifiant systématiquement les types.
- **Expressivité** : Permet d'utiliser des patterns plus variés, comme les conditions de type.

Syntaxe de base

L'approche traditionnelle du `switch` repose sur les valeurs discrètes.

Avec le pattern matching, nous pouvons utiliser des expressions conditionnelles et des types.

```
switch (obj) {  
    case Integer i -> System.out.println("Integer: " + i);  
    case String s -> System.out.println("String: " + s);  
    default -> System.out.println("Unknown type");  
}
```

Ce code vérifie le type de `obj` et exécute le bloc correct.

Exemple pratique

Prenons un exemple où l'on souhaite traiter différents types d'objets :

```
Object obj = ...; // un objet pouvant être de tout type

switch (obj) {
    case Integer i -> System.out.println("Nombre entier : " + i);
    case String s -> System.out.println("Chaîne : " + s);
    case null -> System.out.println("Null");
    default -> System.out.println("Type inconnu");
}
```

Le `switch` identifie le type de `obj` et imprime le message approprié.

Cas avec condition

Vous pouvez ajouter des conditions aux cas pour encore plus de précision :

```
switch (obj) {
    case Integer i when i > 10 -> System.out.println("Entier supérieur à 10 : " + i);
    case Integer i -> System.out.println("Entier : " + i);
    default -> System.out.println("Autre");
}
```

L'utilisation de `when` permet d'affiner la logique d'évaluation au sein d'un même type.

Conclusion

Le pattern matching pour `switch` enrichit Java en améliorant la façon dont les décisions conditionnelles sont prises.

Sa flexibilité et sa clarté font de lui un outil puissant dans le développement moderne avec Java 17.

Classes scellées

Les classes scellées permettent de restreindre l'héritage en Java.

Introduites entre Java 13 et Java 17, elles offrent un contrôle sur les classes dérivées autorisées, ce qui améliore la sécurité et la maintenance du code.

Déclaration : sealed

Une classe scellée est déclarée avec le mot clé `sealed`.

Celle-ci doit spécifier quelles classes sont autorisées à hériter d'elle, via la clause `permits`.

Exemple :

```
public sealed class Shape permits Circle, Square {}
```

Classes dérivées

Les classes dérivées d'une classe `sealed` doivent spécifier comment elles poursuivent l'héritage : avec `sealed`, `non-sealed` ou `final`.

- `sealed` : Continue l'héritage restreint.
- `final` : Empêche toute sous-classe dérivée.
- `non-sealed` : Ouvre l'héritage sans restriction.

Exemple : héritage scellé

Prenons l'exemple suivant pour illustrer l'utilisation des classes scellées :

```
public sealed class Shape permits Circle, Square {}

public final class Circle extends Shape {}

public non-sealed class Square extends Shape {}
```

- `Circle` est final, donc pas de sous-classe possible.
- `Square` est non-sealed, ce qui autorise d'autres classes à l'étendre.

Utilité des classes scellées

Les classes scellées facilitent la maintenance du code en :

- Précisant explicitement les sous-classes autorisées.
- Réduisant les erreurs potentielles dues à des extensions inattendues.

- Simplifiant la compréhension des hiérarchies complexes.

Bonnes pratiques

- Utiliser les classes scellées pour les hiérarchies où l'extension doit être strictement contrôlée.
- Combiner avec records et pattern matching pour un code plus expressif et sûr.
- Toujours mettre à jour la clause permits lors de modifications des sous-classes.

Exercice : refactoriser une hiérarchie

Objectif de l'exercice

- Refactoriser une hiérarchie de classes Java existantes pour utiliser les records et les sealed classes .
- Simplifier la gestion des données immuables.
- Appliquer des contrôles d'héritage plus stricts dans la hiérarchie de classes.

Présentation du contexte

Imaginons que vous avez une hiérarchie de classes représentant des formes géométriques :

- Forme est la classe de base.
- Cercle , Rectangle , et Triangle en sont des sous-classes.

Votre objectif est de moderniser cette hiérarchie.

Étape 1 : Utiliser les records

- Convertir les classes Cercle , Rectangle , et Triangle en records pour bénéficier de l'immuabilité.
- Exemple pour Cercle :

```
public record Cercle(double rayon) { }
```

Les records réduisent le code boilerplate, fournissent automatiquement des méthodes comme equals() et hashCode() .

Étape 2 : Introduction des sealed classes

- Déclarer la classe Forme comme une classe scellée (sealed) pour contrôler l'héritage.
- Exemple :

```
public sealed class Forme
    permits Cercle, Rectangle, Triangle {
}
```

Cette déclaration sealed garantit que seules les classes explicitement permises peuvent hériter de Forme .

Étape 3 : Implémentation complète

- Transformez Rectangle et Triangle en records .
- Assurez-vous que chaque record est référencé dans permits dans Forme .

Prenons l'exemple de Rectangle :

```
public record Rectangle(double largeur, double hauteur) { }
```

Vérifiez que votre code compile et teste qu'il fonctionne comme prévu.

Exercice final

- Transformez toute hiérarchie de classes géométriques en utilisant records et sealed classes .
- Testez en créant des instances de chaque forme et vérifiez leur comportement immuable et scellé.

Relisez votre code pour vous assurer qu'il respecte les principes modernes de Java 17.

De Java 13 à Java 17

Text Blocks (""")

Les Text Blocks, introduits avec Java 13, facilitent la manipulation des chaînes de caractères multi-lignes.

Ils permettent d'éviter une syntaxe verbeuse en supprimant la nécessité d'échapper des caractères.

Exemple de Text Blocks

Voici un exemple de Text Block :

```
String text = """
Ceci est un exemple
de Text Block
en Java.
""",
```

Cela réduit les erreurs et améliore la lisibilité.

Records

Introduits dans Java 14, les Records simplifient la création de classes qui sont principalement des porteurs de données.

Ils réduisent la quantité de code nécessaire en générant automatiquement les getters, `equals()`, `hashCode()` et `toString()`.

Exemple de Record

Définir un Record pour un `Point` :

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

Les Records éliminent le besoin d'écrire des constructeurs et des méthodes courantes.

Pattern Matching pour `instanceof`

Depuis Java 16, le Pattern Matching pour `instanceof` simplifie et sécurise la vérification de type en évitant le casting manuel d'objets.

Exemple de Pattern Matching

Avec Java 16 :

```
if (obj instanceof String s) {  
    int length = s.length();  
}
```

Cela rend le code plus propre et réduit les erreurs potentielles de `ClassCastException`.

Pattern Matching pour switch

Introduit en Java 17, le Pattern Matching pour `switch` améliore la puissance de l'instruction `switch` en permettant de l'utiliser avec des motifs, prenant en charge la déstructuration d'objets.

Exemple de Pattern Matching switch

Pattern Matching `switch` :

```
switch (shape) {  
    case Circle c -> handleCircle(c);  
    case Rectangle r -> handleRectangle(r);  
    default -> handleUnknown();  
}
```

Cela introduit une logique de décision plus flexible.

Classes scellées

Les classes scellées, introduites en Java 17, permettent de restreindre quelles classes peuvent hériter d'une classe.

Elles offrent un meilleur contrôle du modèle d'héritage et peuvent améliorer la sécurité du code.

Exemple de Classes Scellées

```
sealed abstract class Shape permits Circle, Square {}
```

Cela garantit que seules `Circle` et `Square` peuvent hériter de `Shape`.

Exercice : refactorisation avec Records et Sealed Classes

Refactorisez la hiérarchie suivante :

1. Convertissez la classe `Person` en un Record.
2. Transformez la classe `Vehicle` en classe scellée, ne permettant que les sous-classes `Car` et `Bike`.

Cet exercice aide à appliquer l'évolution de Java 13 à Java 17 clef dans la conception de systèmes Java modernes.

Virtual Threads (Project Loom)

Introduction aux Virtual Threads

Les virtual threads sont une nouvelle fonctionnalité introduite dans Project Loom.

Ils permettent de créer des threads allégés, optimisant la gestion des ressources.

Limites des Threads Classiques

Les threads classiques sont coûteux en mémoire et en gestion.

Ils sont souvent limités par le système d'exploitation, entravant leur scalabilité.

Avantages des Virtual Threads

- **Légers** : Consomment moins de mémoire.
- **Scalables** : Peuvent être créés en masse par rapport aux threads classiques.
- **Efficientes** : Facilite la création de milliers de threads simultanés.

Utilisation de Virtual Threads

Les virtual threads sont intégrés dans l'API standard de Java.

Ils s'utilisent presque comme des threads classiques :

```
Thread.ofVirtual().start(() -> System.out.println("Hello from a virtual thread!"));
```

Use Case : Serveurs Haute Concurrence

Les virtual threads sont idéaux pour des serveurs nécessitant des milliers de connexions simultanées, améliorant la réactivité et la performance.

Performances et Virtual Threads

En répartissant la charge, les virtual threads réduisent les temps de latence, car ils minimisent les commutations de contexte et optimisent l'utilisation CPU.

Intégration dans des Applications Existantes

Les virtual threads peuvent être introduits progressivement, sans réécriture massive, permettant une migration en douceur depuis les threads traditionnels.

Démarrer avec Virtual Threads

Pour débuter, installez une version Java supportant Project Loom et commencez par transformer quelques threads critiques en virtual threads pour observer les gains.

Exercice Pratique

Implémentez un service simple simulant 10 000 clients simultanés, chacun exécuté dans un virtual thread, pour observer les améliorations en termes de ressources et de performance.

Structured Concurrency

Concept de Concurrence Structurée

La concurrence structurée vise à simplifier la gestion des tâches concurrentes.

Elle organise les tâches en blocs logiques pour une gestion épurée.

Avantages de la Concurrence Structurée

- **Lisibilité améliorée :** Facilite la lecture et la compréhension du code.
- **Gestion simplifiée :** Les tâches sont organisées et contrôlées ensemble.
- **Débogage efficace :** Réduit la complexité du suivi des erreurs.

Implémentation de la Concurrence Structurée

Dans Java 21, la concurrence structurée est intégrée pour organiser les threads de manière hiérarchique, limitant les effets secondaires et les fuites de ressources.

Exemple de Code en Java 21

La structure d'un bloc de concurrence pourrait se faire comme suit :

1. Démarrage de tâches.
2. Attente de leur achèvement.
3. Gestion des exceptions de manière unifiée.

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {  
    Future<String> task1 = scope.fork(() -> taskMethod1());  
    Future<String> task2 = scope.fork(() -> taskMethod2());  
  
    scope.join();  
    scope.throwIfFailed();  
  
    String result1 = task1.resultNow();  
    String result2 = task2.resultNow();  
}
```

Points Clés

- **Structure et contrôle :** Établit des limites claires entre tâches parentes et enfant.
- **Simplification du code :** Réduit la complexité lors du lancement et de l'arrêt des tâches.

Pratiques Recommandées

- Utilisez la concurrence structurée pour les opérations nécessitant la gestion de plusieurs sous-tâches.
- Assurez-vous que toutes les tâches sont correctement encadrées pour éviter les fuites de thread.

Conclusion

La concurrence structurée introduite dans Java 21 facilite la gestion des tâches parallèles, rendant le code plus simple et robuste.

Scoped Values

Introduction aux Scoped Values

Scoped Values introduit dans Java 21 permet de créer des valeurs spécifiques accessibles uniquement dans un certain contexte d'exécution.

Cela facilite le passage de données de configuration ou de contexte dans des environnements fortement concurrents sans dépendre de variables thread locales, limitant ainsi les effets de bord.

Contexte d'Utilisation

Les Scoped Values sont utiles dans les situations où vous devez isoler des données pour chaque requête utilisateur ou transaction, notamment lorsqu'elles exécutent de multiples tâches concurrentes.

Cela permet une gestion plus propre et sécurisée des données sensibles et contextuelles.

Comparaison avec les ThreadLocals

Contrairement aux `ThreadLocals`, qui stockent les données à l'échelle d'un thread, les Scoped Values associent les données à un contexte de portée défini.

Cela minimise les risques liés au partage de données entre différents flux d'exécution, rendant le code plus robuste.

Exemple de Scoped Values

Imaginons un serveur de traitement de requêtes où chaque requête est isolée avec ses propres données de contexte :

- Créez un `ScopedValue` pour stocker le contexte d'une requête.
- Passez ce contexte à chaque étape du traitement, garantissant que chaque tâche utilise la bonne configuration sans interférence.

```
ScopedValue<String> userContext = ScopedValue.newInstance();
```

Avantages des Scoped Values

- Permet un passage de données sécurisé et localisé dans des architectures multi-threading.
- Réduit la complexité et les erreurs potentielles dues au partage de données globales.
- Facilite le nettoyage automatique, évitant les fuites comme avec `ThreadLocal`.

Pratiques Recommandées

Pour utiliser efficacement les Scoped Values :

- Toujours définir le contexte dès l'entrée dans une nouvelle portée pour garantir l'intégrité des données.
- Limiter la portée autant que possible pour éviter la pollution de l'état.
- Utiliser en combinaison avec les virtual threads pour une gestion fine du déroulement concurrent.

Exercice Pratique

1. Implémentez un programme Java qui utilise `ScopedValue` pour gérer des préférences utilisateur lors de traitement de requêtes.
2. Simulez le traitement de plusieurs requêtes simultanées et assurez-vous que chaque requête accède à ses propres préférences de manière isolée.

Record Patterns

Introduction aux Record Patterns

Les record patterns, introduits dans Java 21, permettent d'extraire des données d'objets de type record de manière concise.

Ils simplifient l'écriture de code en rendant la décomposition de données plus lisible.

Concept de Record

Un record en Java est une classe transparente dont le but est principalement de transporter des données immuables.

Il est défini d'une manière concise et possède des fonctionnalités intégrées comme les getters et la méthode `toString()`.

Syntaxe des Record Patterns

Les record patterns utilisent une syntaxe permettant d'intégrer la décomposition directement dans le code.

Voici un exemple basique :

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

void printPoint(Point point) {
    if (point instanceof Point(int x, int y)) {
        System.out.printf("x: %d, y: %d%n", x, y);
    }
}
```

Dans cet exemple, les valeurs `x` et `y` sont extraites directement du record `Point`.

Utilisation des Record Patterns

En liant des variables à des éléments dans un record, les record patterns offrent une manière plus naturelle et intuitive de travailler avec ces données.

Ils simplifient les contrôles de flux et les expressions conditionnelles.

Exemple Pratique

Illustrons l'utilité des record patterns avec un exemple :

```

record Rectangle(int width, int height) {}

void processRectangle(Rectangle rect) {
    switch (rect) {
        case Rectangle(int w, int h) when (w > 0 && h > 0) ->
            System.out.printf("Valid rectangle: %d x %d%n", w, h);
        case Rectangle(int w, int h) ->
            System.out.println("Invalid dimensions");
        default ->
            System.out.println("Unknown shape");
    }
}

```

Cet exemple utilise les record patterns dans une expression `switch` pour gérer les différentes dimensions du rectangle.

Avantages des Record Patterns

- **Clarté** : Améliorent la lisibilité du code en réduisant le besoin d'extraction manuelle des données.
- **Concision** : Réduisent la verbosité, ce qui diminue les risques d'erreurs.

Grâce à ces fonctionnalités, les record patterns représentent un ajout significatif à la manipulation des données en Java.

Guarded Patterns

Introduction aux Guarded Patterns

Les "Guarded Patterns" sont introduits pour enrichir la correspondance de motifs en Java.

Ils permettent d'ajouter des conditions supplémentaires dans `switch` et `instanceof`.

Cela accroît la flexibilité des structures conditionnelles.

Utilisation dans Switch

Avec les "Guarded Patterns", vous pouvez ajouter des gardes dans les expressions `switch`.

Cela permet d'exécuter un cas spécifique uniquement si une condition est remplie en plus du type vérifié.

Exemple :

```
switch (object) {  
    case Integer i && i > 10 -> System.out.println("Integer greater than 10");  
    default -> System.out.println("Other");  
}
```

Utilisation dans instanceof

Les "Guarded Patterns" permettent également d'utiliser des conditions dans le mot-clé `instanceof`.

Cela ajoute des filtres lors de l'évaluation des instances de classes.

Exemple :

```
if (obj instanceof String s && s.length() > 5) {  
    System.out.println("String longer than 5 characters");  
}
```

Avantages des Guarded Patterns

- **Clarté** : Facilite la lecture et l'écriture de conditions complexes.
- **Précision** : Réduit le besoin de blocs `if` imbriqués pour vérifier des conditions supplémentaires.
- **Flexibilité** : Permet de coder des structures conditionnelles plus robustes et modulaires.

Exercice Pratique

1. Créez une méthode qui utilise un `switch` avec des "Guarded Patterns" pour distinguer entre trois types d'objets différents avec des conditions spécifiques.
2. Testez cette méthode avec des exemples variés pour vous familiariser avec le fonctionnement des "Guarded Patterns".

Foreign Function & Memory API

Introduction

La Foreign Function & Memory API permet une interaction directe avec des bibliothèques natives.

Elle améliore la sécurité et la performance par rapport aux anciennes méthodes comme Java Native Interface (JNI).

Fonctions Natives

Les fonctions natives sont des fonctions écrites dans un langage autre que Java, comme le C.

L'API facilite l'appel de ces fonctions directement depuis Java sans avoir besoin de JNI.

Gestion de la Mémoire

Avec l'API, il est possible de gérer la mémoire native de manière sécurisée.

Elle offre des abstractions pour allouer, lire et écrire de la mémoire hors du tas Java.

Sécurité et Performance

L'API assure une meilleure sécurité en minimisant les erreurs communes du JNI.

Elle est optimisée pour la performance, ce qui améliore l'efficacité des appels et de la gestion mémoire.

Exemple Pratique

Utilisation de l'API pour appeler une fonction C :

1. Déclarer une interface Java représentant la fonction native.
2. Utiliser l'API pour mapper et appeler cette fonction.

Exercice

Implémentez un programme Java qui utilise une fonction native pour ajouter deux nombres.

- Écrivez la fonction en C.
- Utilisez la Foreign Function & Memory API pour l'appeler en Java.
- Assurez un nettoyage correct de la mémoire native après l'exécution.

Exercice : implémenter un service utilisant des virtual threads

Objectif de l'exercice

- Apprendre à utiliser les virtual threads introduits par le Project Loom.
- Implémenter un service Java capable de gérer des tâches concurrentes efficacement.

Introduction aux virtual threads

Les virtual threads permettent de gérer des milliers de threads légers facilement.

Contrairement aux threads classiques, ils sont non-bloquants et gèrent mieux la mémoire.

Définir le service

1. Créez une classe Java nommée `VirtualThreadService`.
2. Implémentez une méthode `runTask` qui exécute des tâches en virtual threads.

Création de virtual threads

- Utilisez `Thread.startVirtualThread` pour créer des threads.
- Exemple de code à implémenter :

```
Thread.startVirtualThread(() -> {
    // Code de la tâche
    System.out.println("Tâche exécutée");
});
```

Implémenter runTask

1. La méthode `runTask` prend une tâche sous forme de `Runnable` .
2. Utilisez des virtual threads pour exécuter cette tâche.

Exercice pratique

1. Ajoutez une méthode `public void executeBatch(List<Runnable> tasks)` .
2. Cette méthode doit exécuter chaque tâche de la liste dans un virtual thread.

Tester le service

- Créez un `main` pour instancier `VirtualThreadService` .
- Ajoutez plusieurs tâches dans une liste et utilisez `executeBatch` pour les exécuter.

Critères de réussite

- Les tâches s'exécutent simultanément et indépendamment.
- Aucune tâche ne bloque une autre, démontrant la puissance des virtual threads.

Points de réflexion

- En quoi les virtual threads améliorent la gestion de la concurrence ?
- Comparez leur performance avec les threads classiques dans de petites applications de test.

De Java 18 à Java 21

Virtual Threads (Project Loom)

Java 21 introduit les virtual threads, permettant de créer un grand nombre de threads légers.

Ils facilitent la gestion de la concurrence et augmentent les performances des applications basées sur le multithreading.

Structured Concurrency

Structured Concurrency améliore la lisibilité et la maintenance du code concurrent.

Elle encadre les opérations asynchrones de manière plus sécurisée et robuste.

Scoped Values

Les Scoped Values remplacent les ThreadLocals pour une gestion des valeurs contextuelles plus sûre.

Elles limitent les fuites de mémoire et améliorent la modularité du code.

Record Patterns

Les Record Patterns simplifient l'extraction et le traitement des données stockées dans les records.

Ils offrent une syntaxe plus concise pour travailler avec les objets immuables.

Guarded Patterns

Guarded Patterns ajoutent des conditions logiques aux modèles de correspondance.

Ils améliorent la précision et flexibilité lors de la composition de conditions dans le code.

Foreign Function & Memory API

Cette API facilite l'appel de fonctions externes et la manipulation de mémoire off-heap.

Elle remplace les interfaces JNI, rendant l'interopérabilité avec des bibliothèques natives plus simple et performante.

Exercice : Implémenter un Service

Objectif

Créer un service simple utilisant des virtual threads.

Enoncé

1. Créez un projet Java 21.
2. Implémentez un service qui utilise des virtual threads pour traiter plusieurs tâches simultanément, comme le simple calcul de sommes de tableaux d'entiers.
3. Assurez-vous que le service fonctionne efficacement avec un grand nombre de tâches.

Exemple

```
public class VirtualThreadService {  
    public static void main(String[] args) {  
        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
            List<Future<Integer>> results = IntStream.range(0, 1000)  
                .mapToObj(i -> executor.submit(() -> sumArray(new int[]{1, 2, 3, 4, 5})))  
                .collect(Collectors.toList());  
  
            results.forEach(future -> {  
                try {  
                    System.out.println(future.get());  
                } catch (InterruptedException | ExecutionException e) {  
                    e.printStackTrace();  
                }  
            });  
        }  
  
        static int sumArray(int[] array) {  
            return Arrays.stream(array).sum();  
        }  
    }  
}
```

Suivez ces étapes pour comprendre et constater l'efficacité des virtual threads sur le traitement concurrentiel.

Évolution du langage (Java 8 → 21)

De Java 8 à Java 12

Modularité avec Project Jigsaw

Java 9 a introduit Project Jigsaw, apportant la modularité au langage.

Les modules permettent de mieux organiser le code, réduire la taille des applications, et gérer les dépendances plus efficacement.

Un module est déclaré dans un fichier `module-info.java` spécifiant ses dépendances et les parties exportées.

Exemple de module :

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

Inférence de type local

Introduite avec Java 10, l'inférence de type local permet d'utiliser le mot-clé `var` pour déclarer des variables.

Java déduit automatiquement le type à partir de l'expression à droite de l'affectation, simplifiant le code tout en conservant la vérification de type.

Exemple :

```
var list = new ArrayList<String>();
```

Switch expressions

Java 12 a introduit des améliorations pour `switch`, permettant de l'utiliser comme une expression.

Cette fonctionnalité simplifie le code en évitant les déclarations de casse en cascade et les oubli de `break`.

Exemple :

```
int result = switch (day) {  
    case MONDAY, FRIDAY, SUNDAY -> 6;  
    case TUESDAY -> 7;  
    default -> 0;  
};
```

Garbage Collector G1

Le Garbage Collector G1, introduit pour optimiser les pauses et les performances des applications, est maintenant le GC par défaut depuis Java 9. Il réduit les temps de pause en organisant le tas en régions puis collectant les régions avec le plus de garbage.

Garbage Collector ZGC

ZGC, introduit en tant que GC expérimental dans Java 11, vise à gérer de très gros tas avec des pauses quasi constantes et très courtes.

Idéal pour les applications nécessitant une faible latence quel que soit le tas.

Exercice : mini-module Java

Créez un mini-module Java.

Incluez une dépendance interne et exportez une fonction simple en utilisant `module-info.java`.

Assurez-vous que votre module suit les principes de modularité.

De Java 13 à Java 17

Text Blocks

Les Text Blocks, introduits en Java 13, facilitent le travail avec des chaînes multiligne.

En utilisant une syntaxe simple (`"""`), vous réduisez la nécessité d'utiliser des caractères d'échappement et améliorez la lisibilité.

Exemple :

```
String html = """""  
    <html>  
        <body>  
            <p>Hello, world</p>  
        </body>  
    </html>  
""";
```

Records

Java 16 a intégré les Records, simplifiant la création de classes uniquement porteuses de données.

Les Records génèrent automatiquement les accesseurs et constructeurs, réduisant le besoin de code boilerplate.

Exemple :

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

Pattern Matching instanceof

Avec Java 16, le pattern matching pour `instanceof` fournit une manière plus expressive d'effectuer des vérifications de type.

Il simplifie le code en combinant la vérification de type et la conversion en une seule opération.

Exemple :

```
if (obj instanceof String s) {  
    System.out.println(s.toLowerCase());  
}
```

Pattern Matching switch

Cette extension du switch, introduite avec Java 17, permet d'utiliser des motifs pour créer des branches basées sur le type et le contenu des objets.

Classes scellées

Les classes scellées (Java 17) permettent de contrôler quels autres classes ou interfaces sont autorisées à les étendre ou les implémenter.

Cela favorise un design plus fin et des règles d'héritage contrôlées.

Exemple :

```
public sealed class Shape permits Circle, Square {}  
  
final class Circle extends Shape {}  
final class Square extends Shape {}
```

Exercice : refactoriser hiérarchie

Refactorisez une hiérarchie de classes existantes pour utiliser des Records et des sealed classes.

Identifiez où sceller le design et comment tirer parti des nouveaux concepts pour une gestion plus performante du modèle.

De Java 18 à Java 21

Virtual Threads

Introduits via Project Loom, les Virtual Threads révolutionnent la programmation concurrente.

Ces threads légers permettent de gérer des milliers de threads avec des ressources minimales, optimisant la scalabilité des applications.

Structured Concurrency

La concurrence structurée offre un modèle pour coordonner la gestion des tâches asynchrones, permettant une gestion plus prédictive des erreurs et de la performance.

Scoped Values

Les valeurs à portée offrent une manière d'acheminer les informations de manière sécurisée et isolée dans une hiérarchie d'exécution concurrente.

Record Patterns

Améliore la déstructuration d'objets complexes, permettant un accès plus direct et structuré aux valeurs internes des records grâce aux pattern matching avancés.

Guarded Patterns

Offrent un moyen d'ajouter des conditions dans les déclarations de motifs, renforçant encore l'expressivité et la clarté des codes.

Foreign Function & Memory API

Cette API facilite les interactions avec le code et la mémoire non Java, remplaçant la complexité du JNI avec une interface plus simple et plus sûre pour les manipulations externes.

Exercice : service avec virtual threads

Implémentez un service de traitement de requêtes utilisant des virtual threads pour optimiser la gestion de la concurrence.

Identifiez des scénarios pratiques où l'efficacité des virtual threads améliore le traitement des tâches simultanées.

Types génériques bornés

Introduction aux bornes

Les types génériques bornés permettent de restreindre les types que les paramètres génériques peuvent accepter.

Cela aide à créer des méthodes plus flexibles et sûres.

Les bornes sont spécifiées avec les mots-clés `extends` et `super`.

Mot-clé `extends`

Lorsqu'un paramètre générique utilise `extends`, il indique une borne supérieure.

C'est-à-dire que le type doit être une sous-classe ou implémenter une interface précise.

Exemple d'utilisation :

```
public <T extends Number> void add(T a, T b) {  
    System.out.println(a.doubleValue() + b.doubleValue());  
}
```

Dans cet exemple, `T` peut être n'importe quelle sous-classe de `Number`.

Mot-clé `super`

Le mot-clé `super` désigne une borne inférieure.

Il permet de spécifier que le type doit être une superclasse d'un certain type.

Cela est utile pour consommer un type.

Exemple d'utilisation :

```
public void processList(List<? super Integer> list) {  
    list.add(new Integer(10));  
}
```

Ici, `list` peut être un `List` d'`Integer` ou de n'importe quelle superclasse de `Integer`.

Avantages des types bornés

- **Flexibilité accrue** : Permet de créer des méthodes qui fonctionnent avec une hiérarchie de classes.
- **Sécurité renforcée** : Limite le risque de recevoir des types inattendus.
- **Lisibilité du code** : Spécification claire des attentes de types.

Exercez-vous à identifier quand utiliser `extends` et `super` pour des cas pratiques dans votre code.

Introduction aux Wildcards

Qu'est-ce qu'un Wildcard?

Les wildcards en Java permettent de créer des types génériques flexibles.

Ils sont représentés par le point d'interrogation `?`.

Les wildcards facilitent le traitement de divers types sans spécifier précisément le type générique.

Utilisation de Wildcards

Les wildcards sont souvent utilisés dans les méthodes et les structures de données.

Ils permettent une représentation plus générale qui accepte divers types de données.

Cela est particulièrement utile lors de la manipulation de collections contenant des types variés.

Wildcard avec `extends`

`<? extends Type>` permet de lire des objets d'un type descendant de `Type` mais pas de les modifier.

Idéal pour les méthodes qui doivent être compatibles avec tous les types dérivés de `Type`.

Exemple : `List<? extends Number>` inclut `Integer`, `Double`, etc.

Wildcard avec `super`

`<? super Type>` est utilisé pour ajouter des objets dans une collection.

Ce type de wildcard accepte `Type` et tous ses super-classes.

Exemple : `List<? super Integer>` permet d'ajouter des `Integer`, mais peut contenir des `Object`.

Exemple pratique

```
public static void processNumbers(List<? extends Number> list) {  
    for (Number n : list) {  
        System.out.println(n);  
    }  
}
```

La méthode `processNumbers` affiche tous les éléments d'une liste de `Number` ou ses sous-classes.

Cas d'utilisation des Wildcards

Utiliser `extends` pour lire, `super` pour écrire.

Choisir le wildcard approprié conduit à des API plus robustes et flexibles.

Assure une meilleure maintenabilité du code Java générique.

Interface Comparable

L'interface Comparable

L'interface `Comparable<T>` permet de définir un ordre naturel pour les objets d'une classe.

En implémentant la méthode `compareTo()`, les objets peuvent être triés de manière cohérente.

Utilisation de `compareTo()`

La méthode `compareTo(T o)` retourne un entier :

- Négatif si l'objet courant est "inférieur" à `o`.
- Zéro si égal à `o`.
- Positif si "supérieur" à `o`.

Exemple : Tri de chaînes

```
class Produit implements Comparable<Produit> {
    String nom;

    @Override
    public int compareTo(Produit autre) {
        return this.nom.compareTo(autre.nom);
    }
}
```

Ici, les produits seront triés par ordre alphabétique de leur nom.

Bonne pratique

Implémenter Comparable permet de s'assurer que les objets sont triés de façon naturelle.

Cela facilite l'utilisation de structures de données comme les TreeSet ou pour le tri d'éléments dans une List avec Collections.sort().

Exercice pratique

Créez une classe Personne avec un attribut age .

Implementez Comparable pour trier les objets Personne par ordre croissant d'âge.

Testez votre implémentation en ajoutant plusieurs objets Personne dans une liste et en les triant.

Comparator

Un Comparator en Java est une interface fonctionnelle utilisée pour définir un ordre sur un ensemble d'objets.

- Permet de créer des critères de tri personnalisés.
- Rend possible le tri d'objets même s'ils ne sont pas comparables intrinsèquement.

Utilisation du Comparator

Pour utiliser un Comparator , vous devez implémenter sa méthode abstraite compare .

- La méthode compare prend deux arguments et retourne un entier.
 - Retourne un nombre négatif si le premier argument est plus petit.

- Retourne zéro si les arguments sont égaux.
- Retourne un nombre positif si le premier argument est plus grand.

Exemple de Comparator

Un exemple simple peut être de comparer des chaînes par leur longueur :

```
Comparator<String> comparator =
    (str1, str2) -> Integer.compare(str1.length(), str2.length());
```

- Ce code crée un `Comparator` qui compare deux chaînes en fonction de leur longueur.

Avantages de Comparator

- **Flexibilité** : Permet de définir plusieurs critères de tri pour le même objet.
- **Simplicité** : Moins intrusive que `Comparable`, car il ne modifie pas la classe de l'objet.

Exemple de Comparaison Multiple

En combinant plusieurs `Comparator`, vous pouvez obtenir un tri complexe :

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

- Ce code trie d'abord par nom, puis par âge en cas d'égalité des noms.

Interface fonctionnelle

Étant une interface fonctionnelle, elle peut être utilisée avec les expressions lambda, simplifiant le code.

- Encourage un style de programmation fonctionnel, améliorant la lisibilité et la maintenabilité du code.

Exercices Pratiques

1. Créez un `Comparator` pour trier des `Person` par `nom` et ensuite par `age`.
2. Modifiez l'exemple précédent pour ignorer la casse lors de la comparaison des noms.

Appliquez les solutions pour transformer votre compréhension théorique en compétences pratiques!

Tri personnalisé

Introduction au tri personnalisé

Le tri personnalisé en Java permet de définir la manière dont les objets d'une collection doivent être organisés selon des critères spécifiques.

Avec les interfaces `Comparable` et `Comparator`, Java offre des moyens flexibles pour personnaliser ce processus.

Utilisation de Comparator

`Comparator` est une interface fonctionnelle.

En utilisant cette interface, vous pouvez définir des critères de tri sans modifier les objets eux-mêmes.

Cela se fait en implémentant la méthode `compare`.

Exemple d'utilisation

Pour trier une liste par un attribut particulier d'un objet, vous pouvez créer une classe qui implémente `Comparator`.

Exemple :

```
public class PersonNameComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

Tri avec des lambdas

Depuis Java 8, les expressions lambda simplifient la création de comparateurs.

Elles permettent d'écrire du code plus concis :

```
list.sort((p1, p2) -> p1.getName().compareTo(p2.getName()));
```

Cela supprime le besoin de classes supplémentaires dédiées aux comparateurs.

Combinaison de plusieurs critères

Avec la méthode `thenComparing`, vous pouvez combiner plusieurs critères de tri.

Cela permet de définir un ordre secondaire lorsque le critère principal résulte en une égalité.

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

Exemple : Tri d'une collection

Considérez une collection de personnes que vous souhaitez trier par nom, puis par âge si les noms sont identiques.

Vous pouvez utiliser une combinaison de comparateurs pour y parvenir :

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

Exercice : Implémenter un comparateur

Énoncé

Créez une classe `PersonSalaryComparator` qui implémente `Comparator<Person>` pour comparer des objets `Person` selon leur salaire.

1. Implémentez la méthode `compare`.
2. Testez votre comparateur sur une liste de personnes pour les trier par salaire croissant.

Objectif

Comprendre l'implémentation pratique des comparateurs pour des critères de tri personnalisés.

Stream parallèle

Introduction aux Streams parallèles

Les Streams parallèles en Java permettent de diviser rapidement les opérations sur les collections en tâches simultanées, exploitant le multicœur des processeurs.

Cela résulte en des temps d'exécution généralement plus courts pour certaines opérations intensives.

- Favorisent l'efficacité.
- Exploitent le parallélisme matériel.
- Intégrés dans l'API `java.util.stream`.

Utilisation de Streams parallèles

Pour convertir un Stream en un Stream parallèle, on utilise la méthode `parallel()`, ou directement `parallelStream()` sur des collections.

Cette simple modification indique à Java d'exécuter le traitement en parallèle, sans changer la logique de traitement.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.parallelStream().forEach(System.out::println);
```

Avantages des Streams parallèles

Les Streams parallèles peuvent réduire considérablement le temps de traitement lors de:

- Calculs intensifs.
- Traitements de grandes quantités de données.
- Fonctionnement en environnement multicœur.

Attention, tous les traitements ne bénéficient pas du parallélisme, surtout si l'opération est légère.

Limitations des Streams parallèles

- **Overhead:** Le coût du passage au parallélisme peut occulter les bénéfices pour les petites tâches.
- **Sécurité des threads:** Assurez-vous que les opérations sont thread-safe.
- **Ordre:** Les streams parallèles peuvent modifier l'ordre des résultats.

Bonnes pratiques

Il est crucial d'évaluer le rapport coût/bénéfice du parallélisme et de s'assurer que le traitement demeure cohérent et sans effets secondaires indésirables.

Testez les performances pour valider les gains attendus.

Exercice pratique

Convertissez un `Stream` classique en un `Stream` parallèle pour traiter une liste de nombres.

Imprimez les résultats uniquement s'ils sont pairs, puis mesurez le temps d'exécution.

1. Utiliser `forEach` pour traiter chaque élément.
2. Mesurer le temps avant et après l'opération.
3. Comparer avec un Stream séquentiel.

Collectors.groupingBy

Définition

`Collectors.groupingBy` est un outil puissant de l'API Stream de Java, introduit avec Java 8. Il permet de regrouper des éléments d'un Stream en fonction d'une classification.

Fonctionnement

La méthode `groupingBy` divise les éléments d'un Stream en groupes en fonction d'une fonction de classification.

Chaque groupe est ensuite collecté dans une `Map`, où la clé est la classification et la valeur est une liste des éléments correspondants.

Exemples Usuels

Un usage courant de `groupingBy` est de regrouper des objets par une de leurs propriétés.

Par exemple, regrouper une liste de personnes par leur ville.

- **Classement par ville :** Chaque clé est une ville et chaque valeur est une liste de personnes.

```
Map<String, List<Person>> peopleByCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity));
```

Avantages

- Simplifie le code pour le regroupement de données.
- Rend le code plus lisible et expressif.
- Réduit la nécessité de boucles imbriquées.

Cas d'Utilisation

Regrouper des transactions bancaires par type pour analyser les dépenses, segmenter des produits par catégorie dans une boutique en ligne, etc.

Ces exemples montrent comment `groupingBy` facilite le traitement de datasets structurés.

Limites

Bien que puissant, `Collectors.groupingBy` peut être inefficace avec de très grands datasets en mémoire limitée.

Il est aussi important de choisir des clés de classification qui garantissent des groupes gérables en taille.

Exercice Pratique

Considérant une liste d'employés, écrivez un code pour les regrouper par département et imprimez chaque département avec ses employés.

- **Étape 1 :** Créer une liste d'employés avec leurs départements.
- **Étape 2 :** Utiliser `Collectors.groupingBy` pour les regrouper.
- **Étape 3 :** Afficher le département suivi de la liste d'employés.

Collectors.mapping

Introduction aux Collectors

Les collecteurs en Java sont utilisés pour regrouper et transformer des données d'un flux (stream).

Ils fournissent des moyens puissants pour traiter les données selon différentes stratégies.

L'un des collecteurs les plus flexibles est `Collectors.mapping`.

Collectors.mapping

`Collectors.mapping` permet de transformer les éléments d'un `Stream` avant de les collecter.

Il fonctionne souvent en combinaison avec d'autres collecteurs, comme `Collectors.groupingBy`.

Cette méthode est idéale lorsque vous souhaitez appliquer une transformation sur les éléments d'un flux tout en regroupant les résultats.

Utilisation de `Collectors.mapping`

- `Collectors.mapping()` prend deux arguments :
 - Une fonction de transformation.
 - Un collecteur qui spécifie comment traiter les éléments transformés.
- Exemple d'utilisation : Lorsqu'on souhaite transformer les éléments avant de les collecter sous une forme particulière.

Exemple de Transformation

Supposons que nous souhaitons collecter les noms en majuscules d'une liste d'objets `Person`.

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

Ici, les noms sont extraits et transformés avant d'être collectés en listes, regroupées par ville.

Combinaison de Collecteurs

`Collectors.mapping` est souvent utilisé avec `Collectors.groupingBy`.

Cette combinaison permet de regrouper les éléments selon une clé, puis d'appliquer une transformation spécifique.

Cela offre une grande flexibilité pour manipuler des flux de données complexes.

Conclusion

`Collectors.mapping` enrichit la manière d'utiliser les collecteurs en appliquant des transformations.

Son association avec d'autres collecteurs permet une manipulation plus fine et puissante des données.

Expérimitez pour découvrir la pluralité des solutions qu'il offre pour traiter les données en Java.

Collectors.teeing

Introduction à Collectors.teeing

`Collectors.teeing` est une fonctionnalité introduite avec Java 12. Il permet de combiner deux collecteurs dans un même processus de réduction.

- Il est utile pour effectuer deux opérations de collecte indépendamment et ensuite combiner leurs résultats.

Syntaxe de Collectors.teeing

La méthode `Collectors.teeing` utilise trois arguments :

- Deux collecteurs pour effectuer les opérations de collecte.
- Une fonction de fusion pour combiner les résultats des deux collecteurs.

Exemple d'utilisation :

```
Collectors.teeing(collector1, collector2, (result1, result2) -> combine)
```

Exemple Pratique

Considérons un scénario où nous voulons calculer à la fois la somme et le nombre d'éléments d'une liste de nombres :

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

Map<String, Integer> result = numbers.stream()
    .collect(Collectors.teeing(
        Collectors.summingInt(n -> n),
        Collectors.counting(),
        (sum, count) -> Map.of("sum", sum, "count", count)
    ));
```

Dans cet exemple, `sum` et `count` sont calculés indépendamment, puis combinés dans une `Map`.

Avantages de Collectors.teeing

- **Simplicité** : Évite des déroulements complexes de flux pour combiner les résultats.
- **Clarté** : Les opérations restent propres et lisibles.
- **Efficacité** : Réduit le nombre de parcours en utilisant un simple flux.

Conclusion

`Collectors.teeing` est un outil très pratique pour les traitements parallèles qui nécessitent la combinaison de résultats.

Il renforce la puissance des API Stream et simplifie le code en évitant des étapes manuelles de fusion des résultats.

Utilisez cette méthode pour optimiser et clarifier votre logique de collecte.

Exercice : Trier et regrouper un dataset complexe

Contexte de l'exercice

Dans cet exercice, vous allez manipuler un dataset complexe tirant parti des génériques et des collections en Java.

- Vous utiliserez des `List`, `Map` et des classes utilitaires pour trier et regrouper des données efficacement.
- Vous allez exploiter les `Comparator` et les fonctionnalités des interfaces de collections modernes.

Description du dataset

Imaginez un dataset constitué d'une liste de personnes, où chaque personne est caractérisée par :

- Un nom (`String`)
- Un âge (`int`)
- Une ville de résidence (`String`)

Votre objectif est de créer un programme Java qui trie cette liste de personnes par âge et, simultanément, regroupe les personnes par leur ville de résidence.

Objectifs de l'exercice

1. **Trier par âge** : Utilisez un `Comparator` pour trier la liste de personnes par âge croissant.
2. **Regrouper par ville** : Utilisez les fonctionnalités de l'API `Collectors` pour regrouper les personnes par ville dans une `Map`.

Avant de commencer, assurez-vous que vous comprenez comment utiliser les `Collectors.groupingBy` et `Comparator.comparing` pour réaliser ces objectifs.

Énoncé pratique

1. Créez une classe `Person` avec les propriétés `name`, `age` et `city`.
2. Instanciez une liste de personnes ayant des âges et des villes distincts.
3. Utilisez un `Comparator` pour trier cette liste de manière croissante par rapport à l'âge des personnes.

4. Employez `Collectors.groupingBy` pour regrouper les résultats par ville, renvoyant une `Map<String, List<Person>>`.

Points à considérer

- **Immutabilité** : Dans quelle mesure pouvez-vous rendre vos objets immuables ?
- **Stream API** : Comment les flux peuvent-ils simplifier votre code ?
- **Bonne pratique** : Pensez à utiliser des méthodes de référence pour améliorer la lisibilité.

Ces questions peuvent vous guider pour réfléchir aux solutions les plus optimales.

Validation de votre code

Une fois que vous avez implémenté votre solution, vérifiez :

- Que la liste est correctement triée par âge.
- Que le regroupement par ville est exact et complet.

Comparez votre approche avec d'autres pour identifier d'éventuelles améliorations ou alternatives.

Types génériques bornés

Les types génériques bornés permettent de restreindre les types pouvant être utilisés dans une déclaration générique.

Cela se fait avec `extends` et `super`.

- `extends` : Utilisé pour établir une borne supérieure.

Par exemple, `<? extends Number>` signifie que le type doit être une sous-classe de `Number`.

- `super` : Utilisé pour établir une borne inférieure.

Par exemple, `<? super Integer>` autorise un type `Integer` ou un de ses super-types.

Wildcards

Les wildcards sont utilisées avec les génériques pour introduire une flexibilité dans la liaison de types.

- `<?>` représente un type inconnu.
- Aide à écrire des méthodes qui peuvent fonctionner avec n'importe quel type, tout en maintenant la sécurité de type.

Comparable

Comparable est une interface utilisée pour définir l'ordre naturel des objets.

- Implémentez `compareTo(T o)` pour comparer "this" à l'objet "o".
- Retourne un entier négatif, zéro, ou positif selon les cas.

Exemple :

```
public class MyClass implements Comparable<MyClass> {  
    @Override  
    public int compareTo(MyClass o) {  
        return this.value - o.value;  
    }  
}
```

Comparator

Comparator est une interface pour définir un ordre de tri personnalisé.

- Implémentez `compare(T o1, T o2)` pour définir l'ordre entre `o1` et `o2`.
- Utile pour des tris différents sans modifier les objets eux-mêmes.

Exemple :

```
Comparator<MyClass> comparator =  
    (o1, o2) -> Integer.compare(o1.getValue(), o2.getValue());
```

Tri personnalisé

Le tri personnalisé permet d'organiser une collection en fonction de critères définis par le programmeur.

- Utilise `Collections.sort(list, comparator)` en Java.
- Essentiel pour structurer des données selon divers attributs.

Exemple :

```
Collections.sort(myList, (a, b) -> a.getName().compareTo(b.getName()));
```

Stream parallèle

Les Streams parallèles permettent le traitement en parallèle des collections.

- Utilisent `parallelStream()` sur une collection pour traiter en parallèle.
- Utiles pour améliorer la performance avec de grands datasets.

Attention : Veillez à ce que les opérations soient thread-safe.

Collectors.groupingBy

`Collectors.groupingBy` est utilisé pour regrouper les éléments d'un Stream par clé.

- Crée une Map où les clés sont produites par une fonction de classification.
- Permet de regrouper par comportements communs ou propriétés.

Exemple :

```
Map<Category, List<Item>> itemsGrouped =
    itemList.stream().collect(Collectors.groupingBy(Item::getCategory));
```

Collectors.mapping

`Collectors.mapping` permet d'appliquer une transformation sur les éléments avant de les collecter.

- Utilisable en combinant d'autres collectors pour des transformations complexes.

Exemple :

```
Map<String, List<String>> nameByCategory =
    items.stream().collect(Collectors.groupingBy(
        Item::getCategory,
        Collectors.mapping(Item::getName, Collectors.toList())))
    );
```

Collectors.teeing

`Collectors.teeing` combine deux collecteurs pour produire un résultat unique.

- Exemple de combinaison de somme et moyenne simultanément.
- Nécessite Java 12 ou plus.

Exemple :

```
DoubleSummaryStatistics stats =  
    items.stream().collect(  
        Collectors.teeing(  
            Collectors.summingDouble(Item::getValue),  
            Collectors.counting(),  
            (sum, count) -> new DoubleSummaryStatistics(count, sum/count)  
        )  
    );
```

Exercice : Trier et regrouper

Votre tâche :

1. Créez une classe `Product` avec des attributs `String name`, `double price`.
2. Créez une liste de `Product` et triez-la par prix ascendant.
3. Utilisez `Collectors.groupingBy` pour regrouper les produits par une certaine catégorie (par exemple, par première lettre du nom).
4. Affichez la liste triée et les groupes obtenus.

Exemple de données :

```
List<Product> products = List.of(  
    new Product("Apple", 1.2),  
    new Product("Banana", 0.5),  
    new Product("Peach", 1.5)  
>;
```

Record avancé

Introduction aux Records

Les records sont une fonctionnalité introduite pour simplifier la création de classes immuables.

- Ils réduisent le code boilerplate.

- Idéal pour stocker des données immuables.

Déclaration d'un Record

Un record est une classe spéciale avec des caractéristiques prédéfinies.

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

- La déclaration `Point` crée automatiquement un constructeur, des accesseurs et des méthodes utilitaires comme `equals()`, `hashCode()`, et `toString()`.

Caractéristiques des Records

Les records simplifient la gestion des données en Java.

- Immutabilité garantie par défaut.
- Égalité basée sur les valeurs, non sur la référence d'objet.
- Utilisation automatique de `hashCode()` et `equals()`.

Cas d'Utilisation des Records

Les records sont utiles pour :

- Représenter des données transférées entre services.
- Structurer des réponses d'API sans nécessiter de logique métier complexe.
- Remplacer efficacement des DTOs standard.

Ajouter des Comportements aux Records

Les records peuvent contenir des méthodes supplémentaires.

```
public record Circle(double radius) {  
    public double area() {  
        return Math.PI * radius * radius;  
    }  
}
```

- Les records peuvent inclure des validations dans un constructeur compact.

Limites des Records

Bien que puissants, les records ne sont pas toujours le bon choix.

- Ne pas utiliser pour des objets avec état mutable.
- Non compatibles avec l'héritage classique (les records sont implicitement `final`).

Exercice sur les Records

Créer un record `Rectangle` avec deux champs `width` et `height`.

- Implémente une méthode pour calculer l'aire.
- Testez la création et l'égalité de deux `Rectangle`.

Introduction

Que sont les Sealed interfaces?

Les **sealed interfaces** en Java permettent de restreindre les interfaces qui peuvent les implémenter.

Introduites pour offrir plus de contrôle sur les relations de hiérarchie, elles garantissent une organisation plus prédictible et sécurisée du code.

Avantages des Sealed interfaces

- **Contrôle de la hiérarchie** : Empêche l'implémentation non souhaitée d'une interface.
- **Lisibilité** : Facilite la compréhension de la hiérarchie de types autorisés.
- **Maintenance** : Réduit les erreurs lors des modifications de code.

Déclaration d'une Sealed interface

Pour déclarer une sealed interface, utilisez le mot-clé `sealed` et spécifiez les interfaces ou classes permises avec `permits`.

```
public sealed interface Animal permits Dog, Cat {}
```

Classes permises

Les classes implémentant une sealed interface doivent être déclarées comme `final`, `sealed`, ou `non-sealed` pour indiquer leur statut dans la hiérarchie.

Exemples d'utilisations

Imaginons une hiérarchie d'animaux :

```
sealed interface Animal permits Dog, Cat {}

final class Dog implements Animal {}

final class Cat implements Animal {}
```

Ce code indique que seuls les types `Dog` et `Cat` peuvent être des `Animal`.

Exercice pratique

1. Déclarez une sealed interface `Vehicle` qui peut être implémentée seulement par `Car` et `Bike`.
2. Implémentez `Car` et `Bike` en tant que classes finales.

Résumé et bonnes pratiques

- Utilisez des sealed interfaces pour clarifier et restreindre les hiérarchies de type.
- Spécifiez explicitement quelles classes ou interfaces peuvent s'implémenter.
- Favorisez l'utilisation de sealed interfaces pour des architectures de code robustes et maintenables.

Immutabilité

Qu'est-ce que l'immuabilité?

L'immuabilité désigne la propriété d'un objet dont l'état ne peut pas être modifié après sa création.

En Java, cela signifie que tous les champs d'un objet immuable sont final et initiaux uniquement via le constructeur.

Ce concept est crucial pour éviter les effets de bord imprévus dans le code.

Avantages de l'immuabilité

- **Simplicité** : Code plus facile à comprendre et à maintenir.
- **Sûreté** : Réduit les erreurs liées aux modifications indésirables.
- **Thread-safety** : Les objets immuables sont intrinsèquement sûrs vis-à-vis de la concurrence.
- **Caching** : Facilite l'implémentation de caches efficaces puisque l'état reste constant.

Créer un objet immuable

Pour rendre une classe Java immuable, suivez ces règles :

- Déclarez la classe comme final.
- Rendez tous les champs private et final.
- Fournissez des constructeurs pour initialiser les champs.
- N'offrez aucun setter.
- Ne retournez pas de références aux champs mutables.

Exemple d'une classe immuable

```
public final class Person {  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
    public int getAge() { return age; }  
}
```

Ce code montre une classe `Person` immuable avec deux propriétés initialisées via le constructeur.

Limites de l'immuabilité

L'immuabilité peut conduire à des augmentations de mémoire si des copies d'objets sont fréquemment nécessaires.

Par ailleurs, elle peut rendre les modifications d'objets plus coûteuses.

Cependant, ces inconvénients sont souvent compensés par les nombreux avantages en termes de sécurité et de maintenance.

Conclusion sur l'immuabilité

L'immuabilité est une pratique recommandée en Java, surtout dans des contextes où la sûreté, la simplicité et la maintenance sont primordiales.

Adapter son code pour utiliser des objets immuables dès que possible peut fournir une base saine et robuste, propice à des applications stables et performantes.

Exercice : Pratiquer l'immutabilité

- Créez une classe immuable `Book` avec les propriétés suivantes : `title` (`String`), `author` (`String`), et `year` (`int`).
- Assurez-vous que toutes les règles de l'immuabilité sont respectées.
- Testez votre classe en instanciant des objets `Book` et en accédant à leurs propriétés.

Thread-safety

Qu'est-ce que la Thread-safety ?

La thread-safety signifie qu'un programme fonctionne correctement lorsqu'il est exécuté par plusieurs threads en parallèle.

Elle est cruciale en programmation concurrente pour éviter des comportements indéterminés ou des corruptions de données.

Les développeurs doivent veiller à protéger les ressources partagées pour garantir cette sécurité.

Ressources partagées

Les ressources partagées sont des variables ou objets accessibles par plusieurs threads.

Protéger ces ressources empêche les accès concurrents non sécurisés.

Java fournit des outils comme les verrous pour gérer cet accès concurrent.

Mécanismes de Synchronisation

Java offre des mots-clés comme `synchronized` pour la synchronisation.

Cela permet de contrôler l'accès à une section de code critique.

Seul un thread est autorisé à accéder à un bloc synchronisé à la fois.

Exemple : Bloc Synchronized

```
public synchronized void increment() {  
    count++;  
}
```

Ce code protège une variable `count` contre les accès concurrents.

Un seul thread peut exécuter `increment()` en même temps.

Cela assure l'intégrité des opérations sur `count`.

Utilisation de Locks

Les ReentrantLock fournissent une alternative plus flexible aux blocs synchronized .

Ils permettent de mieux gérer les scénarios complexes de thread-safety.

Un Lock offre des méthodes comme lock() et unlock() .

Exemple : ReentrantLock

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

Le ReentrantLock protège la section critique.

Il garantit que même en cas d'exception, le verrou est libéré.

Il est utile pour créer des implémentations thread-safe.

Volatile Keyword

Le mot-clé volatile assure que les modifications à une variable sont visibles à tous les threads.

Il garantit la cohérence des lectures et écritures sans blocage de threads.

Il ne protège pas contre les opérations atomiques multi-étapes.

Exemple : Volatile

```
private volatile boolean flag = true;
```

Ce mot-clé informe le JVM que flag peut être modifié.

Tous les threads accèderont à sa valeur mise à jour immédiatement.

Il convient aux états utilisés dans des conditions d'arrêt de boucle.

Bonnes Pratiques

- Toujours utiliser des mécaniques de synchronisation sur les ressources partagées.
- Éviter de concevoir des sections critiques trop larges pour réduire le temps de verrouillage.
- Tester régulièrement la thread-safety du code à l'aide des outils de test adaptés.

Introduction au Pattern Builder

Définition du Pattern Builder

Le Pattern Builder est un patron de conception qui facilite la création d'objets complexes en les construisant pas à pas.

Il est particulièrement utile pour créer des objets immuables, où chaque modification produit une nouvelle instance.

Avantages du Pattern Builder

- Simplifie la création d'objets complexes.
- Améliore la lisibilité et la compréhension du code.
- Facilite la gestion de l'immuabilité et de la thread-safety.

Immutabilité et Pattern Builder

L'immuabilité garantit qu'un objet ne peut être modifié après sa création.

Le Pattern Builder permet de construire ces objets en fournissant un moyen flexible et sûr.

Implémentation de base

Exemple d'utilisation du Pattern Builder pour un objet `Person`.

```

public 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 setName(String name) {
            this.name = name;
            return this;
        }

        public Builder setAge(int age) {
            this.age = age;
            return this;
        }

        public Person build() {
            return new Person(this);
        }
    }
}

```

Utilisation du Builder

Pour utiliser le Builder :

```

Person person = new Person.Builder()
    .setName("Alice")
    .setAge(30)
    .build();

```

Chaque appel à un setter du Builder retourne l'instance du Builder, permettant de chaîner les appels.

Bonne pratique : Immuabilité

En rendant les objets immuables, on évite les effets de bord, améliorant ainsi la fiabilité et la testabilité du code.

Le Pattern Builder aide à encapsuler toutes les modifications dans la phase de construction.

Exercice pratique

Créez un objet `Car` utilisant le Pattern Builder.

Incluez des propriétés comme `model`, `engineType` et `year`.

Assurez-vous que votre `Car` est immuable.

Garbage Collection tuning

Qu'est-ce que le GC tuning ?

Le Garbage Collection (GC) tuning consiste à ajuster les paramètres du GC dans la JVM pour optimiser la gestion de la mémoire.

Cela permet d'améliorer la performance des applications Java en réduisant les pauses et en augmentant le débit.

Importance du GC tuning

Le tuning du GC est crucial pour les applications à grande échelle où la gestion inefficace de la mémoire peut entraîner des ralentissements.

Une gestion optimisée améliore la réactivité et la capacité du système à gérer des charges importantes.

Types de collecteurs Java

- **Serial GC** : Adapté pour les applications à thread unique.
- **Parallel GC** : Conçu pour les systèmes multicœurs.
- **G1 GC** : Optimisé pour les pauses fréquentes mais courtes.
- **ZGC et Shenandoah** : Ciblent de faibles latences.

Paramètres du GC

Ajuster les options de la JVM comme `-Xms`, `-Xmx` (définir la taille minimale et maximale du tas) et choisir le bon collecteur avec `-XX:+UseG1GC` ou `-XX:+UseZGC` peut considérablement affecter les performances.

Outils de profiling

Utilisez des outils comme JVisualVM et JProfiler pour surveiller le comportement de la mémoire en temps réel.

Ces outils aident à identifier les fuites de mémoire et à ajuster le tuning du GC.

Stratégies de tuning

- **Augmenter la taille de l'ancien espace** : Peut réduire la fréquence des collections majeures.
- **Monitorer les pauses** : Identifiez les pauses longues et ajustez les paramètres du GC pour les réduire.
- **Profiling régulier** : Exécutez régulièrement des profils pour adapter les paramètres selon les évolutions de l'application.

Exercice pratique

1. Choisissez une application Java lente.
2. Utilisez JVisualVM pour surveiller la mémoire.
3. Identifiez les points de blocage.
4. Ajustez les paramètres du GC en conséquence.
5. Testez les améliorations de performance.

Chaque étape doit être documentée pour évaluer les changements de performance apportés.

JVM tuning

Introduction à JVM tuning

La JVM (Java Virtual Machine) est responsable de l'exécution des applications Java.

Pour optimiser ses performances, le tuning de la JVM est crucial.

Cela implique de régler divers paramètres qui influencent l'allocation de mémoire, la gestion des threads et le Garbage Collector.

Pourquoi tuner la JVM ?

Tuner la JVM peut améliorer la performance d'une application en :

- Réduisant la latence
- Augmentant le débit
- DéTECTANT et résolvant les points de contention

Ce processus est essentiel pour maximiser l'efficacité sur différents environnements matériels.

Paramètres de mémoire

Les paramètres de mémoire sont utilisés pour ajuster l'allocation des ressources :

- `-Xms` : Définit la taille initiale du tas.
- `-Xmx` : Définit la taille maximale du tas.
- Ajuster ces paramètres influence la vitesse de démarrage et les performances à long terme.

Gestion de la mémoire

La gestion de la mémoire inclut l'ajustement du tas et la configuration de la région de la jeune génération :

- Un tas bien ajusté prévient les OutOfMemoryErrors.
- La jeune génération doit être optimisée pour améliorer le Garbage Collection.

Garbage Collection tuning

La sélection et la configuration du collecteur de déchets influencent la performance :

- Le GC par défaut est adapté à la plupart des situations.
- Modifier les options comme `-XX:+UseG1GC` peut améliorer les performances selon les besoins.

Conclusion et bonnes pratiques

Pour une optimisation optimale des performances :

- Profiler régulièrement l'application
- Adapter les paramètres aux besoins spécifiques de l'application
- Surveiller en continu après chaque modification pour garantir l'efficacité du changement

Exercice pratique

1. Mesurez les performances actuelles de votre application avec JVisualVM.
2. Modifiez les paramètres de la JVM, comme `-Xms` et `-Xmx`.
3. Comparez les performances avant et après l'optimisation pour valider vos ajustements.

Exercice : Profiler un Code Lent

Comprendre le Profilage

Le profiling est une technique pour analyser le comportement d'une application.

Il identifie les parties du code consommant le plus de ressources, généralement en termes de temps de calcul. Utiliser un profiler comme VisualVM peut aider à trouver des goulots d'étranglement.

Utiliser VisualVM

Téléchargez et installez VisualVM pour analyser l'application.

Connectez VisualVM à votre application Java en cours d'exécution.

Surveillez les performances en temps réel, concentrez-vous sur la CPU et la mémoire.

Identifier les Goulots d'Étranglement

Passez en revue les résultats du profiler pour identifier les méthodes les plus gourmandes.
Cherchez les méthodes qui prennent le plus de temps d'exécution.
Notez celles-là pour une optimisation potentielle.

Améliorer les Performances

Optimisez le code en réduisant la complexité des algorithmes.
Utilisez des structures de données et collections appropriées.
Réduisez les appels inutiles à des méthodes coûteuses.

Réévaluer les Performances

Après optimisation, relancez le profilage pour vérifier les améliorations.
Comparez les nouveaux résultats avec les précédents.
Assurez-vous que les temps d'exécution ont significativement diminué.

Bonnes Pratiques

Écrivez du code clair et lisible, facilitant l'optimisation future.
Réalisez un profilage régulièrement, particulièrement lors des mises à jour de code.
Utilisez des outils de monitoring en production pour éviter les surprises.

Exercice

1. Exécutez un programme Java simple, mais inefficace.
2. Profiler ce programme avec VisualVM pour identifier les méthodes lentes.
3. Réécrivez ces méthodes pour améliorer les performances.
4. Comparez les temps d'exécution avant et après.

Record avancé

Les records en Java 21 permettent de définir facilement des classes immuables.

Contrairement aux classes traditionnelles, les records synthétisent automatiquement des méthodes comme `equals`, `hashCode`, et `toString`.

Cela rend le code plus lisible et réduit les erreurs.

Exemple de record avancé

```
public record Point(int x, int y) {
    public Point {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException("Les coordonnées doivent être positives.");
        }
    }
}
```

Le constructeur compact dans cet exemple vérifie que les coordonnées sont positives, montrant comment ajouter une logique métier dans un record.

Sealed interface

Les `sealed interfaces` restreignent les classes pouvant implémenter une interface.

Cela améliore le contrôle sur la hiérarchie de classes et assure l'intégrité du modèle de votre application.

Exemple de sealed interface

```
public sealed interface Shape permits Circle, Square {
}

public final class Circle implements Shape {
    // Implémentation du cercle
}

public final class Square implements Shape {
    // Implémentation du carré
}
```

Ici, seules les classes `Circle` et `Square` peuvent implémenter l'interface `Shape`, assurant un design précis et contrôlé.

Immutabilité

L'immutabilité signifie qu'une fois un objet créé, son état ne peut plus changer.

Cela simplifie le raisonnement sur le code et évite les effets secondaires indésirables.

Thread-safety

Un code thread-safe fonctionne correctement lorsqu'il est exécuté simultanément par plusieurs threads.

Les objets immuables aident souvent à garantir la thread-safety, car leur état ne peut pas être modifié.

Pattern builder immuable

Le pattern builder immuable est utilisé pour créer des objets complexes de manière sûre et flexible.

Chaque appel de la méthode du builder retourne une nouvelle instance jusqu'à atteindre le produit final.

Exemple de pattern builder immuable

```
public 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 setName(String name) {  
            this.name = name;  
            return this;  
        }  
  
        public Builder setAge(int age) {  
            this.age = age;  
            return this;  
        }  
  
        public Person build() {  
            return new Person(this);  
        }  
    }  
}
```

Ce pattern permet de créer un objet `Person` de manière fluide et immuable.

Garbage Collection tuning

Le tuning du Garbage Collection (GC) consiste à ajuster les paramètres de la JVM pour optimiser l'usage de la mémoire et minimiser les pauses de collecte de déchets.

Une bonne compréhension des types de GC et de leur fonctionnement est essentielle pour des applications performantes.

JVM tuning

Le tuning de la JVM améliore la performance globale de l'application en ajustant les paramètres comme la taille du tas, le GC et les options de compilation.

Cela nécessite de surveiller le comportement de l'application et d'ajuster en fonction des besoins spécifiques.

Exercice : Profilage et optimisation

1. Exécutez un programme Java intentionnellement lent.
2. Utilisez des outils de profilage pour identifier les goulots d'étranglement.
3. Mesurez les statistiques du Garbage Collection.
4. Ajustez les paramètres de la JVM et réalisez des optimisations au code, jusqu'à ce que les performances s'améliorent.

Cet exercice vous aidera à pratiquer l'analyse et l'optimisation des performances d'une application Java.

module-info.java

Introduction à module-info.java

Le fichier `module-info.java` est au cœur de l'architecture modulaire de Java.

Introduit avec Java 9, il définit les dépendances et services d'un module.

C'est un fichier obligatoire pour tout projet modulable.

Structure de module-info.java

Un fichier `module-info.java` commence par la déclaration du module avec le mot-clé `module` suivi du nom du module.

Il liste les modules requis et les packages exportés.

Exemple de module-info.java

Prenons un exemple basique :

```
module com.example.myapp {  
    requires java.sql;  
    exports com.example.myapp.util;  
}
```

- `requires` indique les modules dont le module a besoin.
- `exports` indique quels packages sont accessibles par d'autres modules.

Déclaration des nécessite

La clause `requires` permet de spécifier les dépendances.

Chaque module nécessaire doit être déclaré, activant ainsi un contrôle strict des dépendances.

Exportation de packages

Avec `exports`, vous pouvez exposer certains packages de votre module.

Seuls ces packages seront accessibles aux autres modules, assurant l'encapsulation.

Avantages de module-info.java

- Meilleure encapsulation : Seuls les packages explicitement exportés sont accessibles.
- Meilleure gestion des dépendances : Les dépendances sont déclarées et vérifiées à la compilation.

Écriture d'un module-info.java

Pour créer un fichier `module-info.java` :

1. Créez un fichier nommé `module-info.java` à la racine du dossier source.
2. Déclarez le module avec `module <nom_du_module> { } .`
3. Ajoutez `requires` et `exports` selon les besoins du projet.

Exercices Pratiques

1. Créez un projet Java avec un module `com.example.hello` .
2. Ajouter un fichier `module-info.java` qui :
 - Requiert le module `java.base` .
 - Exporte un package `com.example.hello` .

Vérifiez que votre fichier source s'exécute correctement en utilisant `javac --module-path` pour compiler.

Découpage en modules

Pourquoi découper en modules ?

Découper une application en modules indépendants :

- Améliore la maintenance du code.
- Facilite les tests et le déploiement.
- Encourage la réutilisabilité des composants.

Avantages des modules

Les modules permettent :

- Une gestion claire des dépendances.
- Une encapsulation renforcée.
- Une amélioration des performances grâce à un chargement optimisé.

Identifier les modules

Pour découper en modules :

- Analysez les fonctionnalités communes dans votre application.
- Regroupez les classes et interfaces en fonction de leur responsabilité.

Définir les interfaces

Dans un module :

- Les interfaces publiques définissent les points d'interaction.
- Les interfaces permettent de cacher les implémentations internes.

module-info.java

Chaque module possède un fichier `module-info.java` :

- Déclare les dépendances du module.
- Spécifie les packages exportés pour être utilisés par d'autres modules.

Exercice pratique

Créez une application Java modulaire :

1. Définissez au moins deux modules indépendants.
2. Chaque module doit avoir son `module-info.java`.
3. Implémentez l'interaction entre les modules via des interfaces publiques.

Définition du Module Core

Le module core est un composant central d'une application modulaire en Java.

Son rôle principal est de fournir les fonctionnalités clés nécessaires au fonctionnement de l'application.

Fonctionnalités du Module Core

- **Gestion des Connexions** : Il peut gérer la connexion aux bases de données ou aux services externes.
- **Validation des Données** : Vérifie l'intégrité et la validité des données traitées.
- **Logique Métier** : Intègre la logique principale qui drive l'application.

module-info.java

Le fichier `module-info.java` est crucial pour définir un module.

Dans le module core, ce fichier spécifie les dépendances, les services fournis et les entités exportées.

Exemple module-info.java

```
module com.example.core {  
    requires java.sql;  
    exports com.example.core.util;  
    provides com.example.core.service with com.example.core.service.impl.ServiceImpl;  
}
```

Clés pour le Fichier module-info.java

- **requires** : Indique les modules nécessaires (ex: `java.sql`).
- **exports** : Définit les paquets accessibles par d'autres modules.
- **provides...with** : Spécifie l'implémentation d'un service.

Importance du Module Core

Le module core est essentiel car il centralise les composantes fondamentales de l'application.

Cela facilite la maintenance, la compréhension et l'évolution du projet.

Exercice Pratique

1. Créez un module core dans un projet Java 21.
2. Définissez les fonctionnalités de base dans `module-info.java`.
3. Implémentez une fonction simple comme "calculer la somme de deux nombres" pour une vue pratique.

Module service

Concept du Module Service

Les modules services permettent de définir des services réutilisables et interchangeables au sein de votre application Java.

Ils facilitent l'extension et la modification des fonctionnalités sans affecter le système global.

Un module service est souvent utilisé en lien avec des API et d'autres modules.

Interface de Service

Une interface de service définit les fonctionnalités qu'un module service doit implémenter.

Elle agit comme un contrat que les implémentations de service devront respecter.

Cette interface est souvent exposée par un module API pour être utilisée par d'autres parties du code.

Implémentation d'un Service

Une implémentation de service est une classe qui concrétise l'interface de service.

Elle fournit le comportement spécifique désiré pour le service décrit.

Une même interface peut avoir plusieurs implémentations, facilitant la modularité.

Déclaration de Fournisseur de Service

Au sein du fichier `module-info.java`, un module service précise quelle implémentation est utilisée.

La syntaxe est :

```
provides <interface> with <implémentation>;
```

Cela enregistre l'implémentation auprès du Java Module System.

Utilisation avec ServiceLoader

ServiceLoader est une classe Java qui facilite le chargement dynamique de services.
Elle permet de découvrir et utiliser différentes implémentations d'un service à l'exécution.
ServiceLoader explore les modules disponibles pour rechercher les fournisseurs de service.

Exemple Pratique

Supposons une interface de service `PaymentService` avec deux implémentations.
Implémentez les classes `CreditCardPayment` et `PaypalPayment`.
Ensuite, utilisez `ServiceLoader` pour choisir l'implémentation au moment de l'exécution.

Écriture du module-info.java

Dans `module-info.java`, vous devez indiquer quel service est fourni :

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

Cette configuration dit au système quel fournisseur de service est associé au module.

Importance pour les Applications

Les modules services ajoutent flexibilité et adaptabilité à un logiciel.
Ils permettent de changer facilement les fournisseurs sans modifier le code de consommation.
Cela est essentiel pour maintenir et évoluer les applications complexes dans le temps.

Module API

Introduction au Module API

Les modules API définissent les interfaces publiques d'un système modulaire.

Ils exposent les fonctionnalités destinées à être utilisées par d'autres modules ou applications externes.

Le module API joue un rôle central en définissant les contrats d'interaction.

Rôle des Modules API

- **Encapsulation :** Les API permettent de masquer l'implémentation interne tout en exposant des fonctionnalités essentielles.
- **Stabilité :** En fournissant des interfaces stables, elles facilitent la maintenance et l'évolution du système.
- **Réutilisation :** Les modules API favorisent la réutilisabilité des fonctionnalités à travers plusieurs projets.

Déclaration d'une API

Un module API est souvent défini par un fichier `module-info.java` qui expose des packages.

Exemple :

```
exports com.example.api;
```

Cette ligne rend le package `com.example.api` accessible aux modules qui le nécessitent.

Exemple de Module API

Imaginez une bibliothèque de calculs mathématiques :

- Les implémentations de calcul restent cachées.
- Le module API expose uniquement les méthodes nécessaires comme le calcul de la somme, produit, etc.

Utilisation d'un Module API

Pour utiliser un module API, un module consommateur doit déclarer une dépendance :

```
requires com.example.mathapi;
```

Avec cette déclaration, les fonctionnalités exposées par l'API deviennent accessibles.

Avantages des Modules API

- **Évolution du code :** Les API permettent de modifier l'implémentation sans affecter le code client.
- **Collaborativité :** L'équipe dédiée à l'API peut travailler indépendamment des implémentateurs.

Exercice Pratique

Créez un module `mathapi` avec des opérations de base (addition, multiplication).

Exposez ces opérations via des interfaces publiques.

Ensuite, créez un module `app` qui utilise `mathapi` pour réaliser des calculs.

ServiceLoader

Qu'est-ce que ServiceLoader ?

ServiceLoader est un mécanisme de découverte de services dans Java.

Il permet de charger des implémentations de services au moment de l'exécution sans avoir besoin de connaître la classe concrète à l'avance.

Fonctionnement

- Utilise un fichier de configuration pour associer une interface à ses implémentations.
- Simplifie l'extension et la modularité d'une application Java.
- Permet le chargement dynamique en fonction des services disponibles.

Avantages

- **Modularité** : Favorise le découplage en séparant l'interface des implémentations.
- **Extensibilité** : Permet d'ajouter de nouvelles implémentations sans modifier le code existant.
- **Maintenance** : Réduit les dépendances directes entre classes, facilitant l'évolution du code.

Exemples pratiques

Pour utiliser le ServiceLoader, créez un fichier dans `META-INF/services` avec le nom de l'interface et listez les implémentations possibles.

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

Pour la classe implémentée :

```
public class PaypalService implements PaymentService {  
    public void processPayment() {  
        System.out.println("Processing payment with PayPal.");  
    }  
}
```

Puis, dans `META-INF/services/com.example.PaymentService` :

```
com.example.PaypalService
```

Utilisation dans le code

Pour charger et utiliser les services, ServiceLoader s'utilise comme suit :

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

Cette boucle itère sur chaque implémentation de `PaymentService` trouvée et l'appelle.

Injection de dépendances

ServiceLoader est souvent utilisé pour l'injection de dépendances, car il permet de remplacer des implémentations sans changer le code client.

Cela est particulièrement utile pour les plugins ou les composants interchangeables.

Exercice pratique

Créer un projet modulaire où :

1. Définissez une interface de service.
2. Implémentez plusieurs versions de ce service.
3. Utilisez ServiceLoader pour charger et exécuter ces implémentations.

Assurez-vous que votre projet est bien structuré en modules et que chaque module est indépendant.

Injection de dépendances via ServiceLoader

Qu'est-ce que ServiceLoader ?

ServiceLoader est une fonctionnalité de Java qui permet de découvrir et de charger des implémentations de services définies dans des modules différents.

Il facilite l'injection de dépendances en permettant de découpler les interfaces des implémentations, assurant ainsi la modularité et l'extensibilité.

Fonctionnement de ServiceLoader

1. **Définir un service** : Une interface ou une classe abstraite.
2. **Créer des implémentations** : Classes concrètes implémentant le service.
3. **Utiliser ServiceLoader** : Chargement dynamique des implémentations à l'exécution.

Chaque implémentation doit être déclarée dans un fichier de configuration nommé comme l'interface du service dans le répertoire `META-INF/services`.

Exemple d'utilisation

Supposons une interface `PaymentService`.

Voici comment configurer ServiceLoader :

- Déclarez votre service dans le fichier `META-INF/services/com.example.PaymentService` avec le contenu suivant :

```
com.example.impl.PaypalService
```

- Utilisez ServiceLoader pour charger les services :

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

Avantages de ServiceLoader

- **Flexibilité** : Permet l'ajout de nouvelles implémentations sans modifier le code existant.
- **Modularité** : Favorise un design modulaire en regroupant les services et implémentations dans des modules distincts.
- **Extensibilité** : Nouvelle implémentation ajoutée en éditant simplement le fichier de configuration, pas de recompilation nécessaire.

Limites de ServiceLoader

- **Performance** : Peut être moins efficace si de nombreuses implémentations sont chargées ou testées.
- **Absence de contrôle de version** : Contrairement à certains frameworks de dépendance, ServiceLoader ne permet pas de spécifier la version des implémentations à charger.

Exercice Pratique

Créez un projet modulaire avec ServiceLoader :

1. **Définissez un module `payment.api`** contenant l'interface `PaymentService`.
2. **Implémentez deux modules** : `paypal` et `stripe` contenant chacun une implémentation différente de `PaymentService`.
3. **Utilisez ServiceLoader dans un module `payment.client`** pour charger dynamiquement et utiliser ces services.

Vérifiez que l'ajout de nouvelles implémentations dans un nouveau module est possible sans changer le code client.

Exercice : création d'un projet modulaire complet

Objectif de l'exercice

- Comprendre l'architecture modulaire en Java.
- Créer un projet en plusieurs modules.
- Utiliser `module-info.java` pour définir les dépendances.

Étape 1 : Initialiser le projet

1. Créez un nouveau projet Java.
2. Créez trois répertoires pour les modules : `core` , `service` , `api` .
3. Chaque répertoire contiendra son propre `module-info.java` .

Étape 2 : Créer le module core

1. Dans le répertoire `core` , créer les classes essentielles.
2. Exemple : une classe `CoreUtils` avec des méthodes utilitaires.
3. Définir un `module-info.java` pour exporter les fonctionnalités.

Exemple de module-info.java

Title: core/module-info.java

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

- Ce fichier indique que le module `core` expose `com.example.core` .

Étape 3 : Créer le module service

1. Module service devra utiliser core .
2. Créez des classes pour gérer la logique métier.

Exemple : classe ServiceManager utilisant CoreUtils .

Exemple de module-info.java

Title: service/module-info.java

```
module service {  
    requires core;  
    exports com.example.service;  
}
```

- Le module service dépend du module core .

Étape 4 : Créer le module API

1. Interfacez les fonctionnalités avec api .
2. Utilisez ServiceLoader pour la découverte dynamique.

Exemple de ServiceLoader

Title: Utilisation de ServiceLoader

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

- ServiceLoader aide à gérer l'injection de dépendances.

Étape 5 : Compiler et exécuter

1. Compilez les modules séparément.
2. Exécutez une application utilisant ces modules.

Validation des compétences

- Comprenez-vous comment découper un projet ?
- Savez-vous configurer `module-info.java` ?
- Pouvez-vous utiliser `ServiceLoader` efficacement ?

Réflexion post-exercice

- Quels bénéfices tirez-vous de la modularité ?
- Comment ceci impacte-t-il la maintenance de votre code ?

Ressources supplémentaires

- Documentation officielle de Java
- Tutoriels en ligne pour approfondir vos connaissances sur Jigsaw (le système de module Java).

Architecture modulaire

module-info.java

`module-info.java` est le fichier de configuration de module en Java.

Il définit un module, ses dépendances et les packages qu'il expose.

Exemple de déclaration :

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

Découpage en modules

Dans une architecture modulaire, le code est séparé en modules indépendants.

Chaque module a sa responsabilité distincte, ce qui facilite la maintenance et le développement.

- **Indépendance** : Les modules n'exposent que ce qui est nécessaire.
- **Réutilisation** : Les modules peuvent être facilement réutilisés ou remplacés.

Module core

Le module core contient les fonctionnalités de base de l'application.

Il sert de fondation pour les autres modules.

- Ne dépend d'aucun autre module.
- Expose des fonctionnalités que d'autres modules peuvent utiliser.

Module service

Le module service encapsule la logique métier spécifique de l'application.

Il utilise les services fournis par le module core.

- Peut dépendre du module core.
- Peut exposer ses propres services.

Module API

Le module API fournit les interfaces et types, permettant à d'autres modules d'interagir sans connaître les implémentations internes.

- Facilite la modularité et l'interopérabilité.
- Permet de changer les implémentations sans affecter d'autres modules.

ServiceLoader

Le ServiceLoader est une fonctionnalité de Java pour la gestion des pluggins et l'implémentation de services dynamiques.

- Permet de découvrir et utiliser des implémentations de service au moment de l'exécution.
- Améliore la flexibilité et l'extensibilité.

Injection de dépendances

Avec ServiceLoader, l'injection de dépendances se fait dynamiquement.

- Trouve et charge automatiquement les implémentations disponibles.
- Réduit le couplage et améliore l'extensibilité.

Exercice : Projet modulaire

1. Créez un projet Java avec au moins trois modules : core, service, et API.
2. Dans `module-info.java`, spécifiez les dépendances et exportez les packages nécessaires.
3. Utilisez ServiceLoader pour implémenter et charger dynamiquement un service du module service.

Tâchez de bien séparer les responsabilités et d'appliquer les règles d'une architecture modulaire propre.

Généricité et collections

Types génériques bornés

Les types génériques bornés utilisent `extends` et `super` pour restreindre les types utilisés. `extends` limite le type au sous-type ou implémenteur d'une classe/interface. `super` limite au supertype du type indiqué.

Ces bornes permettent une flexibilité accrue tout en garantissant la sécurité des types.

Wildcards

Les wildcards ? représentent un type inconnu dans les génériques.

Utilisées souvent avec `extends` ou `super`, elles permettent de gérer des collections d'objets de types variés tout en assurant la sécurité des types, ce qui simplifie la manipulation des collections.

Comparable

L'interface `Comparable<T>` est utilisée pour définir un ordre naturel pour les objets.

Une classe implémentant `Comparable` doit redéfinir la méthode `compareTo`, qui retourne un entier basé sur la relation entre l'objet appelant et l'objet comparé : négatif si inférieur, zéro si égal, et positif si supérieur.

Comparator

`Comparator<T>` est une interface fonctionnelle pour définir des ordres personnalisés.

Elle fournit des méthodes par défaut comme `thenComparing` pour chaîner les comparateurs.

Les instances de `Comparator` sont utilisées notamment pour trier des collections via des méthodes comme `Collections.sort`.

Tri personnalisé

Pour effectuer un tri personnalisé, implémentez un `Comparator` ou utilisez des lambdas.

Cela vous permet de trier des collections selon divers critères dynamiques.

Par exemple :

- Trier par propriété puis par une autre.
- Inverser l'ordre naturel.

Stream parallèle

Les Streams en Java facilitent le traitement de collections/de flux.

Les streams parallèles exploitent plusieurs threads pour accélérer le traitement des données.

Utilisez `parallelStream()` pour tirer parti des cœurs multiples d'un CPU.

Collectors.groupingBy

`Collectors.groupingBy` permet de regrouper les éléments d'une collection en fonction d'un critère de classement.

C'est un moyen puissant d'organiser les données en sous-collections, facilitant leur traitement ultérieur par critères spécifiques.

Collectors.mapping

`Collectors.mapping` est utilisé lors de regroupement pour appliquer une fonction de transformation à chaque élément avant de les regrouper.

Il simplifie la modification des éléments collectés pendant des opérations groupées complexes.

Collectors.teeing

Introduit dans Java 12, `Collectors.teeing` permet de combiner deux collecteurs en un seul.

Il est utilisé pour exécuter deux opérations distinctes sur le même ensemble de données et combiner les résultats de manière flexible.

Exercice : Trier et regrouper dataset

- Chargez un dataset Java contenant des objets avec divers attributs.
- Utilisez `Comparable` ou `Comparator` pour trier le dataset selon différents critères.
- Implémentez `Collectors.groupingBy` pour regrouper les données basé sur un attribut commun.
- Les solutions doivent illustrer les concepts de générique.

Fonctionnalités modernes

Record avancé

Les records, introduits en Java 16, simplifient la création de classes immuables.

Utilisez les records pour représenter des données tacitement et fournir une implémentation automatique d'égal, hashCode et toString, améliorant la concision du code.

Sealed interface

Les interfaces scellées restreignent quelles classes peuvent les implémenter.

En déclarant les sous-types autorisés dans la même unité de compilation, elles offrent un contrôle fin sur l'extension, facilitant la maintenance et l'évolution du code.

Immutabilité

L'immutabilité garantit que l'état d'un objet ne change pas après sa construction.

Les objets immuables simplifient le partage entre threads et améliorent la sécurité du code.

En Java, privilégiez les classes finales avec des champs finaux.

Thread-safety

La sécurité des threads assure qu'un programme fonctionne correctement en présence d'exécutions simultanées.

Java propose des outils comme `synchronized`, `volatile`, et les collections thread-safe pour gérer l'accès concurrent aux données partagées.

Pattern builder immuable

Le pattern Builder permet de construire des objets de manière flexible.

Un Builder immuable améliore la lisibilité et la sécurité, surtout dans les environnements concurrentiels.

Utilisez-le pour créer des objets complexes tout en garantissant leur immutabilité.

Garbage Collection tuning

Java propose divers Garbage Collectors pour gérer la mémoire.

Optimiser leur comportement via tuning améliore les performances applicatives, réduisant la latence et augmentant le débit en ajustant les paramètres de la JVM.

JVM tuning

Le tuning de la JVM consiste à ajuster les paramètres d'exécution pour améliorer les performances applicatives.

Optimisez la mémoire, la GC et les fils d'exécution pour tirer le meilleur parti de l'infrastructure sous-jacente.

Exercice : Performance profiler

- Analyser un code Java pour identifier les goulots d'étranglement.
- Appliquer des techniques de tuning JVM et garbage collection.
- Utiliser un profiler pour évaluer les améliorations de performance.
- Discuter des gains obtenus après amélioration.

Architecture modulaire

module-info.java

`module-info.java` définit un module Java et liste ses dépendances directes permettant une encapsulation forte et des systèmes évolutifs.

Ce fichier est essentiel pour obtenir les avantages de modularisation introduits par le projet Jigsaw.

Découpage en modules

La modularisation en Java 9 découpe un programme en modules fonctionnellement distincts favorisant la réutilisation et la maintenance.

Un module regroupe classes et interfaces autour d'une fonction ou un domaine.

Module core

Un module core contient la logique fondamentale réutilisable dans plusieurs parties d'un système.

Il est généralement indépendant et pourrait être utilisé par d'autres modules ou systèmes séparés.

Module service

Le module service encapsule la logique spécifique d'un service, permettant une séparation claire et une évolution indépendante de la fonctionnalité au sein d'un système modulaire.

Module API

Un module API définit l'ensemble des interfaces publiques accessibles aux autres modules.

Il encapsule la complexité interne et offre une surface de contact stable pour les interactions externes.

ServiceLoader

`ServiceLoader` est une mécanique pour découvrir et charger des implémentations de services (interfaces) de manière dynamique.

Il offre une facilité d'extension et de remplacement de composants à l'exécution.

Exercice : Projet modulaire

- Concevez un projet Java avec plusieurs modules : core, service, et API.
- Implémentez un système de découverte de services avec `ServiceLoader`.
- Démontrez la découpe et l'interdépendance des modules par un exemple fonctionnel.

CompletableFuture

Introduction

CompletableFuture est une classe de l'API de Java 8 qui simplifie l'écriture de code asynchrone.

Elle permet d'exécuter du code de manière indépendante et de récupérer le résultat une fois l'opération terminée.

Création d'un CompletableFuture

Pour créer une instance vide de CompletableFuture, vous pouvez utiliser la méthode statique `CompletableFuture.supplyAsync()`.

Cela démarrera l'exécution d'une tâche de manière asynchrone.

Exemple de création

Pour créer un CompletableFuture qui retourne un résultat :

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

Ce code exécute la tâche `() -> "Hello, World!"` dans un thread séparé.

Gestion des résultats

Une fois la tâche asynchrone terminée, vous pouvez récupérer le résultat à l'aide de la méthode `get()`.

Attention, cette méthode est bloquante, ce qui signifie qu'elle attendra que le résultat soit disponible.

Exemple de récupération

Pour obtenir le résultat :

```
String result = future.get();
System.out.println(result);
```

Ce code bloquera jusqu'à ce que le résultat "Hello, World!" soit disponible.

Combinaison de tâches

CompletableFuture permet également de combiner plusieurs tâches asynchrones à l'aide des méthodes `thenApply`, `thenAccept`, et `thenRun`.

Exemple de combinaison

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello")
    .thenApply(s -> s + ", World!")
    .thenApply(String::toUpperCase);
```

Ici, `thenApply` est utilisé pour chaîner des transformations de résultats de manière asynchrone.

Erreurs et exceptions

CompletableFuture gère les exceptions de manière fluide.

Utilisez `exceptionally` pour définir un traitement d'erreur lorsque celle-ci se produit.

Exemple de gestion d'erreur

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) {
        throw new RuntimeException("Something went wrong");
    }
    return "Success!";
}).exceptionally(ex -> "Error: " + ex.getMessage());
```

Cet exemple montre comment gérer une exception en retournant un message d'erreur spécifique.

Conclusion

CompletableFuture facilite l'écriture de code asynchrone et la gestion des opérations à long terme de manière non bloquante.

Son utilisation est essentielle pour tirer parti de la puissance des applications Java modernes.

Chaines d'exécution asynchrone

Introduction

Les chaînes d'exécution asynchrone permettent de gérer des tâches qui peuvent être exécutées de manière non bloquante.

Elles améliorent la performance des applications en utilisant au mieux les ressources disponibles.

Asynchrone vs Synchrone

- **Synchrone** : Les tâches attendent la fin de l'autre.
- **Asynchrone** : Les tâches s'exécutent indépendamment les unes des autres.
- L'asynchronisme utilise les ressources de façon plus efficiente en optimisant l'exécution parallèle.

Completer la Future

L'interface `CompletableFuture` en Java 21 permet de créer des chaînes asynchrones.

Elle offre une flexibilité pour combiner différentes opérations non bloquantes en séquence.

Chaining CompletableFutures

- Utiliser `thenApply()` , `thenAccept()` , et `thenRun()` pour enchaîner des tâches.
- Facilite la composition de plusieurs appels de méthode qui dépendent du même résultat.

Exemple :

```
CompletableFuture.supplyAsync(() -> fetchData())
    .thenApply(data -> process(data))
    .thenAccept(result -> display(result));
```

Gestion des Erreurs

`CompletableFuture` permet de gérer les exceptions en utilisant `exceptionally()` ou `handle()`.

Cela assure que les erreurs dans les tâches asynchrones peuvent être capturées et gérées sans bloquer le flux.

Avantages des Chaînes Asynchrones

- **Performance** : Permet l'exécution parallèle des tâches.
- **Non-bloquant** : Optimise l'accès aux ressources sans attendre.
- **Lisibilité** : Les méthodes de chaîne améliorent la compréhension du flux d'exécution.

Cas Pratiques

Mettre en place des chaînes d'exécution asynchrone peut optimiser les applications nécessitant des appels réseau ou E/S intensifs, où l'attente d'une réponse est coûteuse.

Exercice Pratique

Implémentez un système qui récupère des données d'une API, les traite, puis affiche les résultats en utilisant `CompletableFuture` pour orchestrer ces étapes de manière asynchrone.

ForkJoinPool

Introduction à ForkJoinPool

ForkJoinPool est une nouveauté pour gérer la concurrence en divisant récursivement les tâches jusqu'à atteindre un seuil.

Cela améliore l'efficacité du traitement parallèle en exploitant plusieurs cœurs.

Fonctionnement de ForkJoinPool

ForkJoinPool utilise un algorithme "work-stealing" pour équilibrer la charge, ce qui permet à un thread inactif de "voler" du travail à d'autres threads surchargés.

Cela maximise l'utilisation des ressources CPU.

Création de ForkJoinPool

Pour utiliser un ForkJoinPool, créez une instance via son constructeur ou en appelant ForkJoinPool.commonPool().

Le second option est pratique pour des tâches courantes sans besoin de personnalisation.

Exemple de ForkJoinTask

ForkJoinTask représente une tâche pouvant être divisée.

Utilisez RecursiveTask pour retourner un résultat ou RecursiveAction pour une tâche sans retour.

Exemple : calculer la somme d'un tableau.

Implémentation d'une RecursiveTask

```
public class SumTask extends RecursiveTask<Integer> {
    private final int[] array;
    private final int start, end;

    public SumTask(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= THRESHOLD) {
            int sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            int mid = (start + end) / 2;
            SumTask leftTask = new SumTask(array, start, mid);
            SumTask rightTask = new SumTask(array, mid, end);
            leftTask.fork();
            int rightResult = rightTask.compute();
            int leftResult = leftTask.join();
            return leftResult + rightResult;
        }
    }
}
```

Utilisation de ForkJoinPool

Pour exécuter une ForkJoinTask, soumettez-la au ForkJoinPool avec invoke() ou submit().

Exemple :

```
ForkJoinPool pool = ForkJoinPool.commonPool();
SumTask task = new SumTask(array, 0, array.length);
int result = pool.invoke(task);
```

Ce code soumet une tâche au pool commun pour obtenir le résultat.

Avantages de ForkJoinPool

ForkJoinPool permet une meilleure scalabilité en utilisant efficacement plusieurs coeurs sans surcharge de gestion manuelle des threads.

Il est idéal pour les tâches récursives importantes.

Utilisation optimale

Pour des performances optimales, ajustez le seuil de division de tâches selon les caractéristiques du matériel et la taille de la tâche.

Cela empêche une surcharge due à trop d'opérations parallèles.

ExecutorService

Introduction à ExecutorService

- **ExecutorService** est une interface pour gérer des tâches concurrentes.
- Facilite la gestion des threads en coordonnant leur exécution.
- Sert à soumettre des tâches sans se préoccuper de leur création ou gestion des threads.

Création d'un ExecutorService

Exemple d'initialisation d'un pool de threads fixe :

```
ExecutorService executor = Executors.newFixedThreadPool(4);
```

- Ici, 4 threads sont créés pour gérer les tâches en parallèle.
- Structuré pour améliorer les performances dans les opérations multitâches.

Soumission de tâches

Soumettre une tâche via **ExecutorService** :

```
executor.execute(() -> {
    System.out.println("Tâche exécutée.");
});
```

- Méthode `execute` utilise un **Runnable** pour exécuter des tâches.
- Approprié pour des opérations qui n'ont pas besoin de retour.

Résultats avec Callable

Utiliser **Callable** pour des tâches retournant un résultat :

```
Future<Integer> result = executor.submit(() -> {
    return 5 + 3;
});
```

- **Callable** fournit une méthode `call` qui retourne une valeur.
- Utilisation de `Future` pour obtenir le résultat du calcul.

Arrêt de l'ExecutorService

Désactiver après usage :

```
executor.shutdown();
```

- `shutdown()` arrête la prise de nouvelles tâches mais continue celles en cours.
- Utiliser `shutdownNow()` pour arrêter immédiatement tous les threads.

Exemple pratique

Créer un programme gérant plusieurs tâches simultanées :

- Initialiser un ExecutorService avec un thread pool.
- Soumettre plusieurs tâches imprimant des messages différents.
- Arrêter le service après exécution de toutes les tâches soumises.

synchronized

Définition de synchronized

Le mot clé `synchronized` en Java est utilisé pour contrôler l'accès aux méthodes ou blocs de code partagés.

Il garantit qu'un seul thread peut exécuter un bloc de code synchronisé à la fois, prévenant ainsi les accès concurrents problématiques.

Utilisation de synchronized

On peut utiliser `synchronized` pour synchroniser :

- **Méthodes** : en ajoutant `synchronized` dans la déclaration de méthode.
- **Blocs** : pour synchroniser une portion spécifique d'une méthode.

Exemple de Méthode synchronized

Voici comment utiliser `synchronized` dans une méthode :

```
public synchronized void increment() {  
    counter++;  
}
```

En synchronisant cette méthode, seul un thread à la fois peut modifier la valeur de `counter`.

Exemple de Bloc synchronized

L'utilisation dans un bloc `synchronized` peut ressembler à ceci :

```
public void increment() {  
    synchronized(this) {  
        counter++;  
    }  
}
```

Cela synchronise uniquement la section critique de la méthode, améliorant potentiellement les performances.

Avantages et Limites

- **Avantages** : Simplicité et sécurité des threads.

- **Limites** : Peut mener à des performances réduites et au blocage de threads.

Il est crucial d'utiliser `synchronized` judicieusement pour éviter les problèmes de performance tels que les goulots d'étranglement.

Introduction à ReentrantLock

ReentrantLock est une classe qui fournit des mécanismes de synchronisation avancés en Java, introduits pour résoudre les limitations de l'utilisation de `synchronized`.

Pourquoi ReentrantLock?

- **Contrôle Fin**: Offre un contrôle plus fin sur le verrouillage.
- **Verrouillages Explicites**: Permet d'obtenir et de libérer des verrouillages explicitement.
- **Fonctionnalités Avancées**: Prise en charge des verrous réentrants et possibilité de spécifier une équité.

Utilisation de ReentrantLock

La classe ReentrantLock peut remplacer `synchronized` pour offrir plus de flexibilité.

Exemple d'utilisation :

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

Avantages par rapport à synchronized

- **Équité** : ReentrantLock peut être établi avec une équité afin que les threads soient traités équitablement.
- **Essayer sans Bloquer** : Vous pouvez essayer d'accéder à un verrou sans forcer l'attente.
- **Interruptibilité** : Les threads peuvent être interrompus en attente d'un verrou.

Implémentation Pratique

Dans l'utilisation typique, on enveloppe les codes critiques avec lock() et unlock() pour garantir un accès exclusif, tout en gérant correctement les exceptions avec finally.

Cas d'Utilisation de ReentrantLock

ReentrantLock est idéal lorsque les besoins de synchronisation sont complexes ou lorsque des fonctionnalités avancées de verrouillage sont requises, comme le recours à des locks pour différents niveaux d'accès ou la possibilité de manipuler directement les états du verrou.

Limites de ReentrantLock

Bien qu'avancé, l'utilisation incorrecte de ReentrantLock peut conduire à des erreurs de synchronisation difficiles à détecter, telles que des impasses ou des comportements inattendus si les locks ne sont pas correctement libérés.

Atomic Variables

Définition

Les Atomic Variables en Java sont des variables qui supportent des opérations atomiques sans utiliser de mécanismes de verrouillage explicites.

Cela signifie qu'elles permettent une mise à jour sûre et cohérente des données dans des environnements multithreads.

Utilisation

Les atomic variables sont souvent utilisées pour éviter les problèmes de concurrence, comme les conditions de course.

Elles garantissent que seules les opérations atomiques comme l'incrémentation ou la comparaison et l'échange peuvent modifier la variable.

Exemples d'Atomic Variables

Java fournit plusieurs classes atomiques comme `AtomicInteger` , `AtomicLong` , et `AtomicReference` .

Ces classes offrent des méthodes atomiques prêtes à l'emploi telles que `incrementAndGet()` et `compareAndSet()` , qui facilitent la gestion des calculs concurrents sans verrouillage.

Exemple d'`AtomicInteger`

La classe `AtomicInteger` permet des opérations sur des entiers de manière atomique.

Exemple :

```
AtomicInteger atomicInt = new AtomicInteger(0);
int newValue = atomicInt.incrementAndGet(); // Augmente la valeur atomiquement
```

Exercice Pratique

Créez une classe Java utilisant `AtomicInteger` pour gérer un compteur partagé entre plusieurs threads.

Implémentez une méthode qui augmente atomiquement le compteur et vérifiez son fonctionnement dans un environnement multithread.

Avantages des Atomic Variables

- Pas besoin de synchronisation explicite.
- Meilleure performance dans certains cas que l'utilisation de `synchronized` .
- Simplicité et clarté du code en multitâche.

Exercice : implémenter un pipeline parallèle de calcul

Objectifs de l'exercice

- Comprendre les bases de la concurrence moderne en Java.
- Implémenter un pipeline de calcul utilisant les fonctionnalités asynchrones.
- Pratiquer l'utilisation de `CompletableFuture` pour gérer des tâches concurrentes.

Description de l'exercice

L'objectif est d'implémenter un système où plusieurs étapes de calcul s'exécutent parallèlement, chaque étape prenant le résultat de la précédente.

Le pipeline doit être non-bloquant et tirer parti de la concurrence moderne en Java 21.

Étape 1 : Initialisation

1. Créez un projet Java.
2. Ajoutez la classe `Main`.
3. Importez `java.util.concurrent.CompletableFuture`.

Étape 2 : Définir les étapes de calcul

- Définissez trois fonctions représentant chaque étape du calcul.
- Chaque fonction doit simuler une charge de travail (par exemple, un calcul mathématique simple) et retourner un `CompletableFuture`.

Exemple :

```
public CompletableFuture<Integer> step1(int input) {  
    return CompletableFuture.supplyAsync(() -> {  
        // Simuler un calcul  
        return input * 2;  
    });  
}
```

Étape 3 : Chainer les étapes

- Utilisez les méthodes `thenCompose` ou `thenApply` pour enchaîner les étapes de calcul.
- Commencez par récupérer l'entrée de la première étape.

Exemple :

```
CompletableFuture<Integer> result =  
    step1(5)  
    .thenCompose(this::step2)  
    .thenCompose(this::step3);
```

Étape 4 : Gestion des résultats

- Assurez-vous que le résultat final du calcul est correctement récupéré.
- Utilisez la méthode `join()` pour attendre la fin du calcul complet et afficher le résultat.

Exemple :

```
int finalResult = result.join();  
System.out.println("Final result: " + finalResult);
```

Étape 5 : Améliorations et Tests

- Testez votre pipeline avec différentes valeurs d'entrée pour s'assurer de son bon fonctionnement.
- Expérimitez avec des optimisations comme l'utilisation de `exceptionally` pour gérer les erreurs possibles durant le calcul.

Objectifs de Révision

- Revoir les concepts de bases des `CompletableFuture`.
- Analyser l'importance de l'asynchronisme dans le traitement parallèle.
- Améliorer le pipeline avec de meilleures pratiques pour le développement moderne.

Concurrence moderne

CompletableFuture

`CompletableFuture` permet de gérer des tâches asynchrones en Java.

Il offre la possibilité d'enchaîner des opérations de manière non bloquante.

Exemple simple :

```
CompletableFuture.supplyAsync(() -> "Hello")
    .thenApply(result -> result + " World")
    .thenAccept(System.out::println);
```

Ce code enchaîne des tâches sans bloquer le thread principal.

Chaînes d'exécution asynchrone

Les chaînes d'exécution asynchrone associent plusieurs futures pour réaliser des tâches distinctes.

La composition des futures réduit la complexité des callbacks.

Exemple :

```
CompletableFuture.supplyAsync(() -> fetchData())
    .thenApply(data -> processData(data))
    .thenAccept(result -> displayResult(result));
```

Chaque étape est exécutée sans bloquer le fil d'exécution principal.

ForkJoinPool

Le `ForkJoinPool` est conçu pour les calculs parallèles.

Il améliore l'efficacité en divisant les tâches récursivement en sous-tâches plus petites.

Les méthodes principales sont `fork()` pour découper la tâche et `join()` pour la recombiner.

ExecutorService

`ExecutorService` gère des ensembles de threads.

Il simplifie le démarrage et l'arrêt des threads.

Exemple d'utilisation :

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.submit(() -> System.out.println("Task 1"));
executor.shutdown();
```

L'arrêt d'un ExecutorService est essentiel pour libérer les ressources.

synchronized

Le mot-clé `synchronized` garantit qu'un bloc de code ou une méthode n'est exécuté que par un thread à la fois.

Exemple :

```
public synchronized void incrementCounter() {
    counter++;
}
```

Cela évite les problèmes de concurrences sur les variables partagées.

ReentrantLock

`ReentrantLock` est une alternative avancée à `synchronized`.

Pour l'utiliser :

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

Il offre plus de flexibilité, comme l'acquisition conditionnelle de verrou.

Atomic variables

Les `Atomic variables` permettent des opérations atomiques sans `synchronized`.

Exemple :

```
AtomicInteger atomicInteger = new AtomicInteger(0);
atomicInteger.incrementAndGet();
```

Elles assurent la sécurité des threads sur des opérations complexes.

Exercice : pipeline parallèle

Implémentez un pipeline de calcul parallèle en utilisant des concepts de concurrence moderne.

Objectifs :

- Charger des données asynchrones.
- Traiter les données en parallèle via `ForkJoinPool`.
- Utiliser `CompletableFuture` pour gérer les opérations asynchrones et combiner les résultats.

Solution attendue : Un programme Java qui démontre la filiation de ces tâches de manière efficace.

Introduction aux Reactive Streams

Les Reactive Streams fournissent une approche standardisée pour traiter les flux de données asynchrones.

Conçus pour manipuler avec efficacité des flux potentiellement infinis, ils offrent des solutions aux problèmes de concurrence en Java.

Avantages des Reactive Streams

- **Gestion de Flux Asynchrones** : Traite efficacement des données sans bloquer le système.
- **Backpressure** : Contrôle du flux de données pour éviter la surcharge.
- **Interopérabilité** : Norme compatible avec différents frameworks de programmation réactive.

Composants des Reactive Streams

Les Reactive Streams se composent de quatre interfaces principales :

- **Publisher** : Émet des éléments à un Subscriber.
- **Subscriber** : Consomme des éléments d'un Publisher.

- **Subscription** : Lien entre Publisher et Subscriber qui régule le flux avec un système de demandes.
- **Processor** : Composant intermédiaire qui agit à la fois comme Publisher et Subscriber.

Exemple : Publisher et Subscriber

Considérez un système simple où un Publisher envoie des données à un Subscriber.

- **Publisher**: Produit des données (ex : sensor de température).
- **Subscriber**: Utilise ces données (ex : afficheur de température).

Ce modèle assure une transmission fluide et contrôlée des données.

Backpressure

Le concept de backpressure dans Reactive Streams gère le débit entre Publisher et Subscriber.

- Le Subscriber peut demander un nombre spécifique d'éléments à traiter.
- Cela évite la surcharge s'il ne peut pas consommer les données à la même vitesse que le Publisher les émet.

Mise en œuvre avec Java

Reactive Streams nativement intégrés à certaines parties de l'API Java 21 simplifient leur usage :

```
Flow.Publisher<String> publisher = ....;
Flow.Subscriber<String> subscriber = ....;

publisher.subscribe(subscriber);
```

Cette implantation montre comment un Publisher et un Subscriber interagissent en Java.

Conclusion

Les Reactive Streams standardisent la gestion des flux asynchrones en permettant un contrôle précis du débit, assurant une gestion fluide et efficace des données.

Sa compatibilité avec diverses bibliothèques en fait un choix pragmatique pour la programmation réactive en Java.

Introduction à Backpressure

Définition de Backpressure

Backpressure désigne la gestion des flux de données lorsque le consommateur est plus lent que le producteur.

Il est essentiel dans un système réactif pour éviter les débordements de mémoire, en régulant la vitesse à laquelle les données sont émises.

Importance du Backpressure

Le Backpressure est crucial pour garantir la stabilité et la performance des applications, en prévenant les surcharges et en optimisant l'utilisation des ressources.

Sans cela, les systèmes peuvent devenir instables ou inaccessibles à cause d'une surcharge.

Implémentation du Backpressure

Dans des frameworks comme Reactive Streams, le Backpressure s'implémente par des mécanismes de demande (request-n) entre le consommateur et le producteur.

Le consommateur contrôle ainsi le volume de données reçues.

Exemples de Backpressure

- **Demandes limitées** : Le consommateur peut demander une quantité précise de données.
- **Pause du flux** : Permet de gérer le rythme des données selon la charge de traitement du consommateur.

Backpressure dans Project Reactor

Project Reactor offre des opérateurs spécifiquement conçus pour gérer le Backpressure, tels que `limitRate`, qui aide à réguler le flux de données entre le producteur et le consommateur.

Cas d'usage du Backpressure

Utiliser le Backpressure est conseillé lors de:

- Traitement de grandes quantités de données où le traitement est plus lent que la production.
- Services en temps réel nécessitant une régulation stricte du débit de données.

Exercice : Appliquer le Backpressure

Créez un système simple où un producteur envoie des journaux à un consommateur.

Implémentez le Backpressure pour s'assurer que le consommateur ne reçoit que ce qu'il peut traiter confortablement.

Introduction à Project Reactor

Project Reactor est une bibliothèque Java pour la programmation réactive.

Elle fournit des API robustes pour travailler avec des flux de données asynchrones et non bloquants.

Mono et Flux

- **Mono** : représente un flux réactif contenant 0 ou 1 élément.

Similaire à `Optional` mais asynchrone.

- **Flux** : gère plusieurs éléments comme une liste, mais en mode non bloquant et asynchrone.

Utilisation de Mono

Mono permet de gérer des résultats uniques éventuels de manière asynchrone.

Exemple :

```
Mono<String> mono = Mono.just("Hello, Reactor!");
mono.subscribe(System.out::println);
```

Création de Flux

Flux permet de gérer plusieurs éléments dans un flux réactif.

Exemple :

```
Flux<Integer> numbers = Flux.just(1, 2, 3, 4, 5);
numbers.subscribe(System.out::println);
```

Opérateurs de transformation

Les opérateurs comme `map` , `flatMap` et `filter` permettent de transformer les données dans un flux.

Exemple d'usage avec `map` :

```
Flux<Integer> squares = Flux.just(1, 2, 3).map(n -> n * n);
squares.subscribe(System.out::println);
```

Backpressure dans Flux

Backpressure gère le flot de données pour éviter de surcharger les consommateurs.

- Demand-Control permet de limiter la quantité d'éléments produits.
- Utilisation d'opérateurs comme `limitRate()` pour gérer le débit.

Error Handling

Project Reactor fournit des moyens pour gérer les erreurs de manière réactive avec des opérateurs comme `onErrorResume` ou `onErrorReturn`.

Exemple :

```
Flux<Integer> fluxWithError = Flux.just(1, 2, 0)
    .map(n -> 10 / n)
    .onErrorReturn(-1);
fluxWithError.subscribe(System.out::println);
```

Exercice : Système de Notifications

Créez un système réactif de notifications :

1. Utilisez `Flux` pour simuler des événements de notification.
2. Transformez les événements avec `map` pour personnaliser les messages.
3. Gérez les erreurs grâce à `onErrorReturn`.
4. Implementez le backpressure pour contrôler le flux de notifications.

Inspirez-vous des exemples précédents pour structurer votre code.

RxJava

Introduction à RxJava

RxJava est une bibliothèque Java pour la programmation réactive, permettant de composer des opérations asynchrones.

Elle est basée sur le pattern "Observable", qui permet de réagir à des flux de données en temps réel.

Concepts de base

- **Observable** : Émet des données sur lesquelles on peut réagir.
- **Observer** : Consomme les données émises par un Observable.
- **Subscription** : Lien entre un Observable et un Observer.

Créer un Observable

Pour créer un Observable, utilisez `Observable.create()` et définissez la logique d'émission des données que les Observers consommeront.

Exemple simple d'un Observable émettant trois entiers successivement :

```
Observable<Integer> observable = Observable.create(emitter -> {
    emitter.onNext(1);
    emitter.onNext(2);
    emitter.onNext(3);
    emitter.onComplete();
});
```

Observer et Subscription

Un Observer doit implémenter trois méthodes : `onNext` , `onError` , et `onComplete` .

Ces méthodes traitent respectivement les données émises, les erreurs et la fin de l'émission.

```
Observer<Integer> observer = new Observer<Integer>() {
    @Override
    public void onNext(Integer item) {
        System.out.println("Received: " + item);
    }

    @Override
    public void onError(Throwable error) {
        error.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done!");
    }
};

observable.subscribe(observer);
```

Opérateurs principaux : map

L'opérateur `map` transforme chaque élément émis en appliquant une fonction.

Il est souvent utilisé pour effectuer des calculs ou des transformations sur les données.

Exemple : Transformer une séquence d'entiers en leurs carrés.

```
observable.map(item -> item * item)
    .subscribe(System.out::println);
```

Opérateurs principaux : flatMap

flatMap transforme chaque élément émis en un Observable, et fusionne leurs émissions en un seul Observable.

Idéal pour gérer des opérations asynchrones dépendantes.

Exemple : Émettre plusieurs événements à partir d'un seul élément initial.

```
Observable.just(1, 2, 3)
    .flatMap(item -> Observable.just(item * 10, item * 100))
    .subscribe(System.out::println);
```

Filtrer des données avec filter

L'opérateur filter permet d'éliminer des éléments ne satisfaisant pas un critère spécifique.

Très utile pour affiner vos jeux de données.

Exemple : Filtrer les nombres pairs.

```
Observable.just(1, 2, 3, 4)
    .filter(item -> item % 2 == 0)
    .subscribe(System.out::println);
```

Fusionner des flux avec merge

L'opérateur merge combine plusieurs Observables en un seul, émettant des valeurs dès que l'une d'elles est disponible.

Exemple : Fusionner deux Observables d'entiers.

```
Observable<Integer> obs1 = Observable.just(1, 2);
Observable<Integer> obs2 = Observable.just(3, 4);
Observable.merge(obs1, obs2)
    .subscribe(System.out::println);
```

Exercice : Système de notifications réactif

Créez un système simple de notifications en utilisant RxJava.

L'objectif est de simuler l'émission de notifications via un Observable, et de les filtrer pour afficher uniquement celles qui sont urgentes.

Consignes :

1. Créez un Observable émettant des notifications avec différents niveaux de priorité.
2. Utilisez `filter` pour conserver uniquement celles marquées comme urgentes.
3. Affichez les notifications urgentes en utilisant un Observer.

Observable : Introduction

Les Observables sont un composant clé en programmation réactive.

Ils émettent une série d'événements ou de données auxquels les Observers peuvent réagir.

Cette interaction suit le modèle Observer, favorisant un flux de données asynchrone.

Fonctionnement d'un Observable

- **Émission** : Un Observable peut émettre une séquence de données.
- **Observation** : Les Observers s'abonnent pour réagir aux données émises.
- **Asynchronie** : Les opérations sont exécutées de manière asynchrone.

Création d'un Observable

Exemple simple pour créer un Observable en Java avec RxJava :

```
Observable<String> observable = Observable.just("Bonjour", "le", "monde");
```

Ce code crée un Observable qui émet les chaînes "Bonjour", "le", "monde".

Subscribing à un Observable

Après création, un `Observer` doit s'abonner pour traiter les données émises par un `Observable`.

```
observable.subscribe(item -> System.out.println(item));
```

Ici, chaque chaîne sera imprimée, illustrant la réaction aux données émises.

map

Introduction à map

La fonction `map` est essentielle en programmation réactive.

Elle transforme les éléments d'une collection ou d'un flux en appliquant une fonction spécifiée.

Fonctionnement de map

- **Transformation** : `map` applique une fonction à chaque élément émis par un Observable.
- **Renvoie** : Un nouvel Observable est renvoyé avec les éléments transformés.
- **Sans effet de bord** : N'affecte pas les éléments originaux.

Exemple simple avec map

Pour utiliser `map`, appliquez une fonction lambda sur chaque élément du flux.

```
Observable.just(1, 2, 3)
    .map(x -> x * 2)
    .subscribe(System.out::println);
```

Ce code double chaque nombre émis.

Utilisation de map avec chaînes

La fonction `map` peut manipuler des données de types variés, notamment des chaînes de caractères.

```
Observable.just("a", "b", "c")
    .map(String::toUpperCase)
    .subscribe(System.out::println);
```

Ici, chaque lettre est transformée en majuscule.

Cas d'usage de map

- **Conversion de données** : Transformer des données brutes en format utilisable.
- **Préparation de données** : Adapter les données avant un traitement plus complexe.
- **Interopérabilité** : Modifier les formats pour l'intégration avec d'autres systèmes.

map vs. autres opérations

La fonction `map` modifie les valeurs mais ne change pas leur nature, contrairement à `flatMap`, qui gère les transformations asynchrones et les flux secondaires.

Qu'est-ce que flatMap ?

Introduction à flatMap

FlatMap est une méthode utilisée en programmation réactive pour transformer un élément en plusieurs éléments sur un flux réactif.

Contrairement à `map`, qui transforme un élément en un autre, `flatMap` permet d'aplatir les flux de manière à gérer des séquences asynchrones qui produisent des éléments multiples.

Fonctionnement de flatMap

- **Transformation** : Convertir un élément en un flux réactif.
- **Aplatissement** : Combine les flux produits en un seul flux.
- **Asynchronisme** : Idéal pour gérer des appels à des services externes ou des opérations parallèles.

Exemple pratique flatMap

Imaginez que chaque utilisateur fasse plusieurs actions.

Avec `flatMap`, nous pouvons transformer une liste d'utilisateurs en actions corrélées.

```
Flux<User> users = getUsers();
users.flatMap(user -> getActionsForUser(user))
    .subscribe(action -> System.out.println(action));
```

Pourquoi utiliser flatMap ?

- **Gestion de l'asynchronisme** : Simplifie l'exécution d'opérations asynchrones.
- **Combinaison de flux** : Gère des séquences complexes et imbriquées.
- **Flexibilité** : Adapte facilement des flux dynamiques.

Exercice interactif : flatMap

`flatMap` est utile dans les applications réactives.

Essayez d'utiliser `flatMap` pour transformer une collection de cours en actions associées à chaque cours.

Voici comment commencer :

```
Flux<Course> courses = getCourses();
courses.flatMap(course -> getActionsForCourse(course));
// Implémentez la suite pour traiter chaque action.
```

Identifiez les actions possibles pour chaque cours et affichez-les.

Java 21 – Développement

Programmation concurrente et réactive

Programmation réactive

filter

Qu'est-ce que filter ?

La méthode `filter` est une opération de transformation en programmation réactive.

Elle permet de sélectionner les éléments d'un flux de données en fonction de certains critères.

Cela se fait généralement en appliquant une fonction prédicteur qui retourne `true` ou `false` pour chaque élément, ne conservant que ceux pour lesquels la fonction retourne `true`.

Utilisation de filter

En utilisant un flux réactif, `filter` prend en entrée un flux et applique un prédicat.

Par exemple, dans RxJava :

```
Observable<Integer> numbers = Observable.just(1, 2, 3, 4, 5);
numbers.filter(n -> n % 2 == 0)
    .subscribe(System.out::println);
```

Cet exemple affichera uniquement les nombres pairs : 2 et 4.

Exemple pratique : filtres

Imaginons un système de notifications.

On utilise `filter` pour recevoir uniquement les notifications importantes :

```
Observable<Notification> notifications = getNotifications();
notifications.filter(Notification::isImportant)
    .subscribe(System.out::println);
```

Ceci souscrit uniquement aux notifications marquées comme importantes.

Quand utiliser `filter` ?

`filter` est idéal lorsqu'il faut gérer un flux de données potentiellement vaste mais qu'on ne s'intéresse qu'à un sous-ensemble spécifique de ces données.

Cela permet de réduire la complexité du traitement en éliminant les éléments non pertinents dès le début.

merge

Qu'est-ce que `merge` ?

La fonction `merge` est utilisée en programmation réactive pour combiner plusieurs flux (ou streams) en un seul.

Elle permet de traiter les valeurs émises par ces flux de manière simultanée.

Usage de `merge`

- La méthode `merge` prend en entrée plusieurs flux.
- Elle émet les valeurs de chaque flux dès qu'elles sont disponibles, dans l'ordre de leur arrivée.
- Elle est utile pour regrouper les résultats de plusieurs tâches ou sources d'événements.

Exemple d'utilisation

Considérons deux flux de données A et B.

En utilisant `merge`, nous combinons ces flux pour écouter les émissions de données de l'un ou l'autre.

```
Flux<String> flux1 = Flux.just("Donnée 1", "Donnée 2");
Flux<String> flux2 = Flux.just("Donnée A", "Donnée B");
Flux<String> résultatMerge = Flux.merge(flux1, flux2);
```

Avantages de merge

- Permet une gestion simplifiée des flux multiples.
- Réduit le besoin d'écouter séparément chaque flux et de combiner manuellement les résultats.
- Efficace pour les systèmes réactifs nécessitant une réactivité rapide aux événements.

Précautions à prendre

- Attention à la consommation de mémoire : `merge` peut émettre des éléments rapidement, ce qui peut surcharger la mémoire si les consommateurs ne sont pas assez rapides.
- La gestion de l'ordre n'est pas garantie, donc `merge` convient mieux aux cas où l'ordre d'émission n'est pas critique.

Évaluer vos connaissances

Essayez de créer un flux réactif en combinant trois flux de chaînes différentes, puis affichez chaque émission.

Expérimitez pour comprendre comment `merge` fonctionne avec des flux rapides et lents.

Exercice : Créer un Système de Notifications Réactif

Introduction à la Programmation Réactive

La programmation réactive permet de concevoir des applications en se basant sur la propagation des changements.

Ce paradigme repose sur l'idée que les systèmes réactifs réagissent aux flux de données de manière asynchrone et non-bloquante.

Objectif de l'Exercice

L'objectif est de créer un système de notifications réactif qui réagit aux événements tels que l'ajout ou la suppression de notifications.

Ce système doit être capable de gérer un flux continu de notifications en utilisant RxJava.

Concepts à Utiliser

- **Observable** : Un flux de données émettant des événements.
- **map, flatMap** : Transformations des données.
- **filter** : Filtrer les événements.
- **merge** : Combiner plusieurs flux.
- **Backpressure** : Gestion de la pression exercée par un flux de données pouvant être plus rapide que le traitement.

Étapes de Réalisation

1. Créer un Observable :

- Générer un flux représentant les notifications reçues.

2. Filtrer les Notifications :

- Appliquer la fonction `filter` pour ne garder que les notifications importantes.

3. Transformation des Données :

- Utiliser `map` pour formater les notifications avant affichage.

4. Combiner plusieurs Flux :

- Utiliser `merge` pour combiner les notifications provenant de différentes sources.

5. Gérer la Backpressure :

- Implémenter une stratégie de gestion de la pression pour s'assurer que le système ne soit pas surchargé.

Énoncé de l'Exercice

1. Initialisation :

- Créez un Observable pour les notifications de type `String` .

2. Filtrage :

- Filtrez les notifications selon un critère de priorité, par exemple `propriété == "important"` .

3. Transformation :

- Transformez les notifications pour qu'elles incluent la date de réception.

4. Combinaison :

- Combinez ces notifications avec un autre flux de messages système important.

5. Backpressure :

- Assurez-vous que le traitement en temps réel supporte une surabondance de notifications par une stratégie de buffering ou d'échantillonnage.

Structure du Code

```
import io.reactivex.rxjava3.core.Observable;

public class NotificationSystem {
    public static void main(String[] args) {
        Observable<String> notifications = Observable.just("notification1", "notification2", "notification3");

        notifications
            .filter(notification -> notification.contains("important"))
            .map(notification -> notification + " received at " + LocalDateTime.now())
            .subscribe(System.out::println);
    }
}
```

Exécution et Vérification

- Testez votre système en observant la console pour vous assurer que seules les notifications importantes sont affichées avec la date de réception.
- Expérimitez avec différents événements pour voir comment le système les gère.

Programmation réactive

Reactive Streams

Reactive Streams est une initiative pour fournir un standard de traitement asynchrone avec un flux de données non bloquant.

Ce standard permet un flux de communication entre les composants avec gestion du contrôle de flux, appelé backpressure, pour éviter les surcharges de mémoire.

Backpressure

Le backpressure est une technique permettant de gérer le flux de données entre un producteur et un consommateur.

Lorsqu'un consommateur est submergé par les données, le backpressure ralentit le producteur.

Ceci assure une utilisation efficace des ressources et évite l'épuisement de la mémoire.

Project Reactor

Project Reactor est une bibliothèque Java pour la programmation réactive, basée sur le standard Reactive Streams.

Elle fournit des API puissantes comme Flux et Mono pour créer des applications asynchrones.

Découvrez comment il simplifie la gestion des flux de données grâce à une syntaxe fluide.

RxJava

RxJava est une implémentation de ReactiveX en Java, permettant une programmation asynchrone et basée sur des événements.

Avec des abstractions comme Observable et Flowable, RxJava facilite la manipulation des flux de données en supportant des opérations complexes de transformation.

Observable

Un Observable est un objet qui émet une séquence de valeurs ou d'événements.

Les abonnés s'y inscrivent pour recevoir les données émises.

Utilisez Observable pour modéliser des flux de données dans vos applications réactives.

Il permet la composition facile de flux.

map

L'opérateur `map` transforme chaque élément émis par un Observable à l'aide d'une fonction fournie.

Par exemple, utilisez `map` pour convertir un flux de températures de degrés Celsius en degrés Fahrenheit.

Cela assure la transformation des données dans le flux.

flatMap

`flatMap` permet de transformer un élément émis en un autre flux Observable, puis fusionne les résultats dans un seul flux.

C'est utile pour gérer des opérations asynchrones dépendantes.

Par exemple, utiliser `flatMap` pour récupérer et combiner les détails de l'utilisateur.

filter

L'opérateur `filter` laisse passer seulement les éléments qui satisfont une condition spécifiée.

Utilisez `filter` pour éliminer les données non désirées d'un flux.

Par exemple, filtrez les températures en dessous de 20°C pour ne traiter que celles qui dépassent ce seuil.

merge

L'opérateur `merge` combine plusieurs flux Observable en un seul.

Utilisez-le pour unifier les données provenant de différentes sources.

Par exemple, `merge` peut agréger les notifications provenant de diverses applications pour un traitement centralisé.

Exercice : Système de notifications

Créez un système de notifications réactif avec les outils suivants :

- Émettez des notifications en simulant différentes sources (emails, SMS, push).
- Utilisez `merge` pour unifier ces sources de notifications.
- Appliquez `filter` pour n'afficher que les notifications importantes.
- Transformez le message de notification avec `map` pour ajout de contexte.

Testez votre système pour vous assurer qu'il gère efficacement les notifications.

VisualVM

Introduction à VisualVM

VisualVM est un outil de profilage pour les applications Java.

Il permet de surveiller, dépanner et perfectionner les performances des applications Java en temps réel.

VisualVM est intégré au JDK et fournit une interface graphique pour analyser la consommation CPU, la mémoire et les threads.

Installation et démarrage

1. **Installation** : VisualVM est inclus avec le JDK, aucune installation supplémentaire n'est requise.
2. **Démarrage** : Exécutez `jvisualvm` à partir de la ligne de commande après avoir installé le JDK.
3. **Interface** : L'interface utilisateur de VisualVM montre les applications Java en cours d'exécution que vous pouvez sélectionner pour analyser.

Surveillance des performances

VisualVM offre un aperçu des performances des applications :

- **CPU Usage** : Identifie les méthodes consommatrices en ressources.
- **Heap Memory** : Analyse la consommation de mémoire et détecte les fuites potentielles.
- **Thread Activity** : Suit l'activité des threads et diagnostique les blocages.

Analyse de la mémoire

VisualVM permet d'analyser la mémoire de votre application :

- **Heap Dumps** : Prenez un instantané de la mémoire pour analyser les objets et leur allocation.
- **GC Activity** : Surveillez l'activité du ramasse-miettes pour optimiser la gestion de la mémoire.

Diagnostic avancé

Pour une analyse approfondie :

- **Profiling** : Collecte des données détaillées sur le CPU et la mémoire pour augmenter la performance.
- **Monitoring Threads** : Repérez les threads qui causent des goulets d'étranglement.
- **Deadlock Detection** : Identifie les situations où plusieurs threads se bloquent mutuellement.

Exercices pratiques

Exercice : Suivez ces étapes :

1. Lancez une application Java.
2. Utilisez VisualVM pour surveiller ses performances.
3. Identifiez un pic de consommation CPU.
4. Observez les objets mémoire avec Heap Dump.
5. Trouvez et résolvez un blocage de thread, si présent.

Cet exercice vous permettra de mettre en pratique la surveillance, le profilage et le dépannage des problèmes de performance avec VisualVM.

JProfiler

Introduction à JProfiler

JProfiler est un outil puissant pour profiler des applications Java.

Il aide à analyser les performances et à détecter les goulets d'étranglement.

Avec JProfiler, on peut obtenir des informations détaillées sur la mémoire, l'utilisation du CPU, et le comportement des threads.

Installation de JProfiler

Pour installer JProfiler, téléchargez-le depuis le site officiel.

L'installation est simple; l'assistant de configuration vous guidera à travers le processus.

Une fois installé, JProfiler peut être intégré à votre environnement IDE préféré pour une utilisation simplifiée.

Utilisation de JProfiler

Pour commencer à utiliser JProfiler, lancez votre application Java via l'interface de JProfiler.

Sélectionnez les métriques que vous souhaitez profiler, telles que l'utilisation du CPU, la mémoire, ou les threads.

JProfiler offre une vue graphique pour une analyse facile.

Analyse des performances

JProfiler vous permet de visualiser l'utilisation des ressources de votre application.

Par exemple, vous pouvez identifier les méthodes consommant le plus de CPU ou de mémoire.

Utilisez cette information pour optimiser le code et améliorer l'efficacité de l'application.

Analyse des threads

Une des fonctionnalités clés de JProfiler est l'analyse des threads actifs.

Cela inclut la visualisation des états et le suivi des threads bloqués.

Cette analyse est cruciale pour détecter les problèmes de concurrence et les deadlocks.

Optimisation avec JProfiler

Une fois les problèmes identifiés, JProfiler propose des stratégies d'optimisation.

Cela peut inclure la modification du code ou l'ajustement des paramètres de la JVM pour améliorer les performances.

Utilisez les recommandations de l'outil pour atteindre une application plus performante.

Résumé sur JProfiler

JProfiler est idéal pour les développeurs souhaitant optimiser leur code Java.

Grâce à ses outils de visualisation et d'analyse, il est possible d'améliorer la performance, de détecter et résoudre des problèmes complexes, notamment liés à la concurrence.

jccmd : Introduction

jccmd est un outil fourni avec le JDK (Java Development Kit) visant à interagir avec les processus JVM en cours d'exécution.

Il permet de collecter des informations diagnostiques et de performance utiles pour le debugging et l'optimisation.

Exécuter jccmd

Pour exécuter jccmd, vous devez connaître le PID (Process ID) du processus Java.

Utilisez la commande `jccmd <PID>` pour énumérer toutes les commandes disponibles pour ce processus.

Cela vous aide à comprendre les actions possibles sur ce processus Java en cours.

Commandes communes jccmd

Quelques commandes fréquemment utilisées avec jccmd incluent :

- `help` : Liste toutes les commandes disponibles.
- `VM.system_properties` : Affiche les propriétés système.
- `GC.run` : Déclenche la collecte de garbage collection manuelle.

- `Thread.print` : Liste les threads actifs et l'état de chacun.

jcmand : Utilisation pratique

Utilisez `jcmand` pour analyser en profondeur un processus Java :

- Identifier les threads problématiques avec `Thread.print`.
- Déclencher la garbage collection lorsque les ressources sont limitées avec `GC.run`.
- Vérifier les propriétés système avec `VM.system_properties`, pour diagnostiquer des problèmes environnementaux.

Exemples pratiques jcmand

Pour voir les threads actifs d'un processus, exécutez :

```
jcmand <PID> Thread.print
```

Pour forcer une collecte de garbage, utilisez :

```
jcmand <PID> GC.run
```

Ces actions aident à comprendre l'état interne d'un programme Java et détecter les anomalies.

Avantages de jcmand

Jcmand offre des diagnostics légers et rapides sans arrêt de l'application:

- Aucune installation supplémentaire n'est requise.
- Interface en ligne de commande simple et efficace.
- Complète les autres outils comme VisualVM pour un diagnostic plus détaillé.

Conclusion : jcmand

Jcmand est essentiel pour améliorer les performances et identifier des problèmes dans les applications Java en production.

Grâce à ses commandes diverses, il permet une analyse fine des processus en cours, vous aidant à optimiser et déboguer vos applications Java efficacement.

jconsole : Introduction

JConsole est un outil graphique inclus dans le JDK de Java.

Il permet de surveiller et de gérer les performances d'applications Java. À travers une interface visuelle, vous pouvez observer les ressources utilisées par votre application en temps réel.

jconsole : Connexion

Pour utiliser jconsole, démarrez-le via la ligne de commande avec `jconsole`.

Une fois lancé, il vous permet de vous connecter à une application Java active, soit localement ou par une connexion distante.

jconsole : Surveillance

JConsole fournit plusieurs onglets de surveillance, notamment :

- **Vue d'ensemble** : Affiche l'utilisation de la mémoire, le nombre de threads, et les données sur le processeur.
- **Mémoire** : Permet de visualiser la mémoire allouée et de demander des opérations de récupération.
- **Threads** : Montre les threads actifs et leur état actuel.

jconsole : Performance

L'onglet Performance de jconsole vous permet de suivre l'utilisation CPU de votre application Java.

Vous pouvez identifier les goulets d'étranglement et les performances de votre JVM.

jconsole : Threads

Dans l'onglet Threads, jconsole vous offre des informations détaillées comme :

- Le nombre de threads vivants.
- Les threads en cours d'exécution.
- Les threads verrouillés.

Cela permet de diagnostiquer des problèmes comme les blocages éventuels ou les fuites de mémoire.

jconsole : Exercices pratiques

Pour maîtriser jconsole, essayez de :

1. Lancer une application Java simple.
2. Connecter jconsole à cette application.
3. Analyser l'utilisation de la mémoire et des threads.
4. Identifier des améliorations possibles basées sur vos observations.

Ces exercices vous offriront une expérience pratique précieuse dans le débogage et l'optimisation d'applications Java.

Java Flight Recorder

Qu'est-ce que JFR ?

Java Flight Recorder (JFR) est un outil intégré à la JVM pour surveiller et analyser les performances des applications.

- Capture des données de performance de la JVM.
- Minimal impact sur les performances.
- Aide à identifier les goulots d'étranglement.

Avantages de JFR

- **Faible Impact** : Consomme très peu de ressources.
- **Intégration Directe** : Inclus dans la JVM à partir de Java 11.
- **Données Complètes** : Capture des événements JVM détaillés.

Utilisation de JFR

Activer JFR lors du démarrage de la JVM :

```
java -XX:StartFlightRecording=name=MyRecording,duration=60s,filename=myrecording.jfr
```

- `name` : Nom de l'enregistrement.
- `duration` : Durée de l'enregistrement.
- `filename` : Fichier de sortie.

Analyser les Données JFR

Les données JFR peuvent être analysées pour :

- Déetecter des anomalies de performance.
- Identifier des points chauds du CPU.
- Analyser l'utilisation de la mémoire.

Utiliser des outils comme Java Mission Control pour visualiser les données.

Scénarios d'utilisation

- **Optimisation** : Identifier les goulets d'étranglement des performances.
- **Debugging** : Résoudre les problèmes de performance dus à la concurrence.
- **Maintenance** : Surveillance proactive des applications en production.

Exercice Pratique

Enregistrez et analysez les performances d'une application simple en utilisant Java Flight Recorder.

1. Démarrez votre application avec JFR activé.
2. Exécutez l'application pour simuler une charge.
3. Examinez les résultats avec Java Mission Control, identifiez un point chaud.

Heap dump

Qu'est-ce qu'un Heap Dump ?

Un heap dump est un snapshot de la mémoire d'exécution d'une application Java à un moment donné. Il capture l'état complet de l'espace mémoire utilisé par les objets d'une application.

Il est essentiel pour l'analyse de fuite de mémoire et le troubleshooting.

Utilisation des Heap Dumps

Les heap dumps aident à identifier :

- Les objets consommant le plus de mémoire.
- Les potentiels fuites de mémoire en repérant les objets inutilisés.
- Les chemins de références qui empêchent le GC de libérer de la mémoire.

Générer un Heap Dump

Il existe plusieurs façons de générer un heap dump :

- Utiliser l'option `-XX:+HeapDumpOnOutOfMemoryError` pour un dump automatique lors d'un OOM.
- Exécuter la commande `jmap -dump` pour créer manuellement un heap dump d'une application en cours.

Exemple de Commande avec jmap

Pour générer un heap dump avec jmap :

```
jmap -dump:format=b,file=headdump.hprof <pid>
```

- `headdump.hprof` est le fichier de sortie.
- `<pid>` est l'ID de processus de l'application Java cible.

Analyser un Heap Dump

Utilisez un outil comme **Eclipse MAT** ou **VisualVM** pour analyser le fichier hprof.

Ces outils vous aident à visualiser graphiquement la consommation mémoire.

Ils permettent de zoomer sur des objets spécifiques et d'explorer les références.

Optimisation grâce aux Heap Dumps

Analyser et comprendre un heap dump peut aider à :

- Identifier des structures de données inefficaces.

- Réduire la consommation mémoire des objets.
- Améliorer la performance en éliminant les fuites mémoire.

Meilleures pratiques

Lors de l'analyse de heap dumps :

- Sauvegardez plusieurs dumps pour observer l'usage mémoire à différents moments.
- Considérez l'utilisation de heap dumps dans un environnement de test pour minimiser les impacts sur la production.
- Complétez avec d'autres outils comme les thread dumps ou les profils de CPU pour un diagnostic complet.

Thread dump

Qu'est-ce qu'un Thread Dump?

Un thread dump est une instantané de tous les threads d'une JVM.

Il affiche l'état et la trace de la pile de chaque thread.

Il est précieux pour diagnostiquer des problèmes de performance et de concurrence dans une application Java.

Importance du Thread Dump

L'analyse d'un thread dump permet d'identifier les blocages, les deadlocks ou les performances ralenties.

C'est un outil essentiel pour le debugging des applications Java concurrentes.

Il aide à comprendre l'exécution des threads et à optimiser le code si nécessaire.

Comment Générer un Thread Dump

Il existe plusieurs méthodes pour générer un thread dump.

La méthode la plus courante sur les systèmes UNIX est d'utiliser le signal `kill -3`.

Alternativement, des outils comme `jcmd`, `jstack`, ou des IDE peuvent être utilisés pour capturer un thread dump.

Analyse du Thread Dump

Lors de l'analyse, recherchez les threads bloqués ou en attente d'une ressource.

Identifiez également les threads consommant le plus de CPU.

Les informations contenues permettent de viser des interventions précises pour optimiser l'application.

Exemple de Thread Dump

Voici un exemple simplifié d'un thread dump :

- Thread-1: en RUNNABLE
 - at com.example.MyClass.method(MyClass.java:10)
- Thread-2: en WAITING
 - at java.lang.Object.wait(Native Method)

Cet exemple montre deux threads, un actif et un en attente.

L'analyse se base sur l'état et la pile pour résoudre des problèmes spécifiques.

Deadlock detection

Qu'est-ce qu'un Deadlock ?

Un deadlock est une situation en programmation où deux ou plusieurs threads sont bloqués indéfiniment en attendant des ressources détenues par les autres.

Cela arrête le programme, rendant impossible l'exécution des tâches.

Causes des Deadlocks

Les deadlocks peuvent survenir à cause de :

- Accès concurrent à des ressources partagées.
- Mauvaise gestion des synchronisations de threads.
- Utilisation des verrous dans un ordre aléatoire.

DéTECTER UN DEADLOCK

Pour détecter un deadlock :

1. Utilisez des outils comme VisualVM ou JConsole pour analyser les threads.
2. Recherchez des threads ayant le statut "bloqué".
3. Analysez les verrous détenus et ceux en attente.

UTILISATION DE JCONSOLE

Avec JConsole :

- Connectez-vous à l'application Java.
- Accédez à l'onglet "Threads".
- Identifiez les threads bloqués et vérifiez les informations de lock.

UTILISATION DE JSTACK

jstack permet de capturer l'état des threads :

- Exécutez `jstack [pid]` pour obtenir un dump des threads.
- Recherchez les étiquettes "Found one Java-level deadlock".

PRÉVENTION DES DEADLOCKS

Pour prévenir les deadlocks :

- Ordonnez l'acquisition des verrous de manière cohérente.
- Intégrez des timeouts lors des tentatives de verrouillage.
- Évitez les verrous si possible, utilisez des structures non-bloquantes.

Exercice Pratique

Identifiez et corrigez un deadlock dans un programme Java :

1. Téléchargez le fichier `DeadlockExample.java`.
2. Exécutez et utilisez `jstack` pour localiser le deadlock.
3. Modifiez le code pour éliminer le deadlock.
4. Testez votre solution.

Exercice : Deadlock multithreadé

Objectif de l'exercice

Identifier et corriger un deadlock dans une application Java multithreadée.

Cela implique l'analyse des threads en attente et la libération des ressources bloquées.

Comprendre le Deadlock

Un deadlock survient lorsque deux ou plusieurs threads attendent indéfiniment que des ressources détenues par l'autre soient libérées.

Cela entraîne un blocage de l'exécution.

Exemple de Deadlock

Considérons deux threads T1 et T2 et deux ressources R1 et R2 :

- T1 détient R1 et attend R2.
- T2 détient R2 et attend R1.

Cela bloque l'exécution de T1 et T2.

Identifier un Deadlock

Pour identifier un deadlock, utilisez un dump de threads :

- Générer un "thread dump".
- Rechercher les threads marqués "waiting" ou "blocked".
- Identifier les ressources en attente.

Utilisation de jconsole

Utiliser l'outil `jconsole` pour visualiser les threads en attente :

- Connecter `jconsole` à votre application.
- Naviguer vers l'onglet "Threads".
- Identifier les cycles d'attente entre threads.

Résoudre le Deadlock

Approche pour résoudre un deadlock :

- Identifier les ressources partagées.
- Imposer un ordre d'acquisition de ressources pour les threads.
- S'assurer que tous les threads libèrent les ressources correctement.

Exemple de Code à Corriger

```
public class DeadlockDemo {  
    private final Object lock1 = new Object();  
    private final Object lock2 = new Object();  
  
    public void method1() {  
        synchronized (lock1) {  
            Thread.sleep(50); // Simuler un traitement  
            synchronized (lock2) {  
                // Opération sur les ressources  
            }  
        }  
    }  
  
    public void method2() {  
        synchronized (lock2) {  
            Thread.sleep(50); // Simuler un traitement  
            synchronized (lock1) {  
                // Opération sur les ressources  
            }  
        }  
    }  
}
```

Corriger l'Exemple

Pour corriger le deadlock ci-dessus :

- Standardiser l'acquisition des locks.
- Par exemple, utiliser `lock1` puis `lock2` dans les deux méthodes.

Vérifier la Solution

Après correction :

- Relancer votre application.
- Reproduire le scénario.
- Confirmer l'absence de deadlocks dans `jconsole`.

Debugging et optimisation

VisualVM

VisualVM est un outil puissant pour surveiller la performance des applications Java.

Il permet de profiler CPU et mémoire, déboguer des applications à distance et analyser les threads.

C'est un outil essentiel pour identifier les goulets d'étranglement dans une application Java.

JProfiler

JProfiler est un outil avancé de profilage Java qui aide à analyser la consommation de ressources des applications.

Il prend en charge la surveillance de la mémoire, le suivi des appels de méthode et l'analyse des threads.

JProfiler est utile pour optimiser l'efficacité des programmes Java.

jcmd

`jcmd` est un utilitaire en ligne de commande inclus dans le JDK.

Il permet d'exécuter différentes commandes sur une JVM en cours d'exécution.

On peut l'utiliser pour obtenir des informations sur l'état des threads, générer des dumps de mémoire et plus.

jconsole

jconsole est une console graphique qui permet de surveiller et gérer des applications Java en utilisant JMX (Java Management Extensions).

Elle offre une interface pour visualiser les performances de l'application et détecter les problèmes potentiels.

Java Flight Recorder

Java Flight Recorder est un outil de profilage intégré à la JVM.

Il collecte des données de performance basse-latence dans des fichiers qu'on peut analyser plus tard avec Java Mission Control.

Il est utilisé pour diagnostiquer des problèmes de performance difficiles à reproduire.

Heap dump

Un Heap dump est un snapshot de la mémoire Java à un moment donné.

Il contient des informations sur l'état de la mémoire de la JVM et peut être analysé pour trouver des fuites de mémoire ou optimiser l'utilisation de la mémoire.

Thread dump

Un Thread dump fournit un instantané de l'état des threads dans une application Java à un moment donné.

Il est utilisé pour diagnostiquer des problèmes de performance, comme des blocages de threads ou des cycles d'attente.

Deadlock detection

La détection des deadlocks est cruciale dans un environnement multithreadé.

Un deadlock se produit lorsque deux ou plusieurs threads attendent indéfiniment les ressources les uns des autres.

Outils comme VisualVM et Thread dump aident à les identifier.

Exercice : Deadlock multithreadé

Identifiez et corrigez un deadlock dans le code suivant :

1. Analysez le comportement de deux threads qui s'attendent mutuellement.
2. Utilisez jconsole ou VisualVM pour détecter le deadlock.
3. Adaptez le code pour résoudre le problème et permettre une exécution fluide.

```
class SharedResource {  
    synchronized void method1(SharedResource resource) {  
        resource.method2(this);  
    }  
    synchronized void method2(SharedResource resource) {  
        resource.method1(this);  
    }  
}
```

Programmation concurrente et réactive

Concurrence moderne

CompletableFuture

`CompletableFuture` est une classe de Java qui simplifie l'exécution asynchrone et la gestion des résultats.

Elle permet d'exécuter des tâches en parallèle et de combiner les résultats.

Avec `CompletableFuture`, vous pouvez chaîner des appels de méthode pour exécuter des actions asynchrones de manière fluide.

Chaines d'exécution asynchrone

Les chaînes d'exécution asynchrone permettent de composer des tâches asynchrones, comme une suite de promesses.

Cela permet de structurer le code sans bloquer le thread principal.

Les actions suivantes ne débutent qu'une fois la tâche précédente achevée.

ForkJoinPool

`ForkJoinPool` est un pool de threads conçu pour diviser une tâche en sous-tâches.

Il utilise une approche récursive, favorisant la parallélisation automatique.

Cela s'avère utile pour les calculs intensifs, optimisant l'utilisation des processeurs multicœurs.

ExecutorService

`ExecutorService` gère l'exécution de tâches asynchrones dans un pool de threads.

Il améliore la performance en réutilisant des threads existants.

Par conséquent, il réduit le coût de création de threads et offre une gestion efficace des ressources.

synchronized

Le mot-clé `synchronized` protège des blocs de code ou des méthodes pour un accès concurrent.

En le plaçant, vous vous assurez qu'une seule thread à la fois peut exécuter ce code, évitant ainsi les problèmes de concurrence.

ReentrantLock

`ReentrantLock` est une alternative aux blocs synchronisés.

Il offre plus de flexibilité, permettant le verrouillage explicite de sections critiques.

De plus, il permet des tentatives d'acquisition de verrou et des timeout, optimisant la gestion des threads.

Atomic variables

Les variables atomiques, comme `AtomicInteger`, fournissent des opérations thread-safe sans utiliser de verrouillage.

Elles garantissent la cohérence des données lors de mises à jour concurrentes, améliorant ainsi les performances en environnement multithreadé.

Exercice : pipeline parallèle

Implémentez un pipeline de calcul parallèle utilisant `CompletableFuture` .

Créez une série de calculs qui s'enchaînent de manière asynchrone.

Utilisez `thenApply` , `thenCombine` et d'autres méthodes pour orchestrer l'exécution.

Programmation réactive

Reactive Streams

`Reactive Streams` est une spécification pour traiter des flux de données de manière asynchrone et non bloquante.

Elle facilite le traitement de grandes quantités de données avec une consommation contrôlée, prévenant la surcharge des systèmes.

Backpressure

Le backpressure est une technique utilisée pour réguler le flux des données entre un producteur et un consommateur.

Cela garantit que le producteur ne submerge pas le consommateur, permettant une gestion efficace des ressources.

Project Reactor

`Project Reactor` est une bibliothèque pour la programmation réactive en Java.

Elle s'appuie sur les spécifications des flux réactifs pour offrir des API fonctionnelles adaptées aux applications asynchrones et non bloquantes.

RxJava

RxJava est une bibliothèque pour la programmation réactive qui permet la composition asynchrone des programmes en utilisant des Observable .

Elle simplifie le traitement de flux de données continus dans les applications Java.

Observable

Un Observable représente une source émettrice d'éléments de données au fil du temps.

Il permet d'écouter ces éléments et d'y réagir de manière non bloquante, favorisant ainsi des applications bien réactives.

map

La fonction map transforme chaque élément émis par un Observable .

Elle applique une fonction à chaque élément et retourne des nouveaux éléments, facilitant la manipulation des données en flux continu.

flatMap

La fonction flatMap transforme chaque élément d'un flux en un flux lui-même.

Cela permet de combiner plusieurs flux en un seul, enchaînant des opérations complexes sur les données de manière structurée.

filter

La fonction filter sélectionne des éléments spécifiques d'un flux selon une condition prédéfinie.

Elle favorise le tri dans les données, permettant de traiter uniquement les éléments répondant à des critères définis.

merge

`merge` combine plusieurs flux d'ensembles en un seul, tout en maintenant l'ordre de réception des éléments.

Cela permet d'unir les effets de multiples sources et d'y répondre de manière cohérente.

Exercice : système de notifications

Créez un système de notifications réactif avec RxJava .

Utilisez `Observable` pour émettre des alertes et appliquez des opérations comme `map` et `filter` afin de transformer et sélectionner les notifications à afficher.

Debugging et optimisation

VisualVM

`VisualVM` est un outil de profiling pour les applications Java.

Il offre des fonctionnalités de monitorage en temps réel, y compris l'analyse de l'utilisation de la mémoire et la performance des threads.

JProfiler

`JProfiler` est un profileur performant pour Java, dédié à l'optimisation des applications.

Il permet d'observer les appels de méthode, le chargement de classes et l'utilisation de la mémoire en temps réel.

jcmd

`jcmd` est un outil en ligne de commande pour interagir avec les process Java.

Il permet d'envoyer des commandes pour le diagnostic et la gestion des applications, efficace pour l'analyse des problèmes de performance.

jconsole

`jconsole` est un outil de monitoring qui exploite l'API JMX.

Il permet d'observer et de gérer les performances et les ressources des applications Java, affichant des graphiques en temps réel pour les données de performance.

Java Flight Recorder

`Java Flight Recorder` capture les événements et l'activité au sein de la JVM.

Il permet d'obtenir un historique détaillé des activités de l'application, facilitant la détection et l'analyse des problèmes de performance.

Heap dump

Un `heap dump` est un instantané de la mémoire utilisée par une application Java.

Il aide à identifier les fuites de mémoire, en révélant les objets présents dans le tas et leur interrelation au moment précis.

Thread dump

Un `thread dump` montre l'état de chaque thread d'une application Java à un moment donné.

Il est crucial pour diagnostiquer les blocages ou les morts, en dévoilant les points d'attente dans les threads.

Deadlock detection

La détection de deadlock est essentielle pour identifier les threads qui se bloquent mutuellement.

Elle révèle les threads qui attendent indéfiniment des ressources détenues par d'autres, ce qui peut paralyser le système.

Exercice : corriger un deadlock

Identifiez un deadlock dans une application Java simulée avec plusieurs threads.

Utilisez `jconsole` ou `VisualVM` pour localiser le blocage et modifiez le code pour permettre une exécution fluide des threads.

Importance de l'Architecture Modulaire

L'architecture modulaire permet d'organiser un projet en sous-systèmes indépendants.

- Chaque module a une responsabilité distincte.
- Favorise la maintenance et l'évolution du projet.

Avantages de la Modularity

- **Réutilisabilité** : Les modules peuvent être réutilisés dans d'autres projets.
- **Évolutivité** : Ajouter de nouvelles fonctionnalités sans affecter les modules existants.
- **Cohérence** : Promeut un code organique et bien structuré.

Structure d'un Module Java

Un module en Java est défini par un fichier `module-info.java`.

- **Déclaration** : Spécifie les dépendances et les exports.
- **Encapsulation** : Limite la visibilité des classes.

Exemple : `module-info.java`

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

- **requires** : Indique les modules nécessaires.
- **exports** : Spécifie quels packages sont accessibles.

Pratiques de Conception Modulaire

- **Responsabilité unique** : Un module ne doit remplir qu'un rôle précis.
- **Faible Couplage** : Minimise la dépendance entre modules.
- **Haute Cohésion** : Assurez que chaque module ait une fonctionnalité cohérente.

Organisation du Projet

Une bonne architecture utilise une approche modulaire pour structurer :

- Modules *core*
- Modules de *service*
- Modules *API*

Ce modèle simplifie l'interaction et la maintenance sur le long terme.

Prochaine Étape

Passez à la conception de la structure de vos modules en définissant les dépendances et exports nécessaires.

Introduction aux Modules

Comprendre les Modules

- Un **module** en Java est un groupement de packages et de ressources.
- Ils facilitent l'organisation et la gestion des dépendances.
- Les modules améliorent la sécurité du projet en définissant des limites claires.

Structure Modulaire

- Un projet modulaire se divise en plusieurs composants :
 - **Core** : cœur de l'application, logique métier principale.
 - **Service** : gestion des règles métiers spécifiques, interaction avec le Core.
 - **API** : interface d'accès aux fonctionnalités du programme.

Module Core

- Contient les classes essentielles pour la logique métier.
- Ce module est généralement stable et bien testé.
- Exemple : fonctions de calculs, gestion des utilisateurs.

Module Service

- Manipule et utilise les fonctionnalités du module Core.
- Contient des services spécifiques, comme l'envoi de notifications.
- Facile à adapter ou étendre sans toucher au Core.

Module API

- Interface entre l'utilisateur et les modules sous-jacents.
- Expose les services fournis par le module Service.
- Utilisé pour le développement de RESTful services ou interfaces graphiques.

Avantages d'une Architecture Modulaire

- Séparation claire des responsabilités.
- Facilite la maintenance et l'extension.
- Encapsulation des composants réduit les impacts en cas de changements.

Exercice Pratique

1. Créez une structure de projet avec trois modules : Core, Service, et API.
2. Implémentez une simple fonction dans le module Core, comme une addition.
3. Le module Service appelle cette fonction.
4. Exposez ce service via le module API.

Ressources d'aide : [Documentation Java](#)

Introduction aux Records

Qu'est-ce qu'un Record ?

- Introduit dans Java 14, un Record est un type spécial de classe.
- Il permet de modéliser des données immuables de manière concise.
- Les Records simplifient la création des classes avec des propriétés de lecture seule.

Avantages des Records

- **Syntaxe Simplifiée** : Moins de code à écrire pour les classes de données simples.
- **Immuabilité** : Les instances de Records sont immuables.
- **Automatisation** : Génération automatique de méthodes `equals()`, `hashCode()`, et `toString()`.

Déclaration d'un Record

- Un Record est déclaré avec le mot-clé `record`.
- Exemple de déclaration :

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

- Cela crée une classe `Point` avec deux champs : `x` et `y`.

Utilisation des Records

- Les Records sont utilisés pour représenter des objets avec un état fixe.
- Appropriés pour les classes qui contiennent principalement des données sans comportement complexe.

Exemple Pratique

- Considérons un Record pour un utilisateur :

```
public record User(String name, int age) {}
```

- Simple et efficace pour encapsuler des données, sans surcharge de code.

Records et Héritage

- Les Records ne peuvent pas étendre d'autres classes.
- Peuvent implémenter des interfaces, mais ne doivent pas redéfinir la sémantique.

Exercices Pratiques

1. Déclarez un Record `Book` avec les champs `title` et `author`.
2. Créez une instance du Record `Book` et affichez son contenu avec `toString()`.

Conclusion

- Les Records sont une avancée vers des classes de données plus simples.
- Favorisent la clarté et la réduction du code boilerplate en Java.

Pattern Matching

Introduction au Pattern Matching

Le pattern matching est une fonctionnalité moderne de Java, introduite pour simplifier le code lors des vérifications de type.

Il permet de concisément tester et extraire des valeurs d'objets via des motifs prédéfinis.

Utilisation de Pattern Matching

Le pattern matching peut être utilisé avec des instructions `switch` ou lors de l'utilisation du mot-clé `instanceof`.

Cela permet d'éviter des conversions de type explicites répétitives.

Exemple avec `instanceof`

Avec le pattern matching, vous pouvez à la fois vérifier le type et le convertir en une seule étape.

```
Object obj = "Hello, World!";
if (obj instanceof String s) {
    System.out.println(s.toUpperCase());
}
```

Pattern Matching et `switch`

Java 17 introduit des améliorations pour `switch`, rendant possible le pattern matching.

Ceci simplifie le code comparé à des structures traditionnelles.

Exemple `switch` avancé

Le pattern matching dans `switch` simplifie les opérations sur les objets.

```
Object obj = 123;
switch (obj) {
    case Integer i -> System.out.println(i);
    case String s -> System.out.println(s.toLowerCase());
    default -> throw new IllegalStateException("Unexpected value");
}
```

Conclusion

Le pattern matching améliore la lisibilité et la concision du code.

Il est important de comprendre comment et quand l'utiliser pour tirer parti des évolutions de Java.

Virtual Threads

Introduction

- Les **Virtual Threads** sont une fonctionnalité de Java 21.
- Ils permettent de gérer la concurrence de manière plus efficace.
- Facilitent la création de plusieurs threads légers.

Avantages

- **Performance:** Réduisent l'overhead lié à la gestion des threads.
- **Scalabilité:** Supportent un grand nombre de threads simultanés.
- **Facilité d'utilisation:** Simplifient la programmation concurrente.

Création d'un Virtual Thread

- Utilisation de `Thread.ofVirtual().start()`.
- Semblable à la création d'un thread traditionnel.

Exemple :

```
Thread.ofVirtual().start() -> {  
    System.out.println("Exécution d'un Virtual Thread");  
};
```

Comparaison avec les Threads Standard

- **Virtual Threads** sont légers par rapport aux threads OS.
- Réduction des ressources consommées.
- Permettent plus de threads pour des tâches I/O intensives.

Utilisation Pratique

- Appropriés pour les applications serveur à fort volume de requêtes.
- Utilisables en backend pour les microservices concurrents.
- Efficaces pour les tâches nécessitant de nombreuses connexions réseau.

Conclusion

- Les **Virtual Threads** révolutionnent la gestion de la concurrence.
- Ils s'intègrent facilement dans les projets Java modernes.
- Essentiels pour optimiser des projets à forte intensité de threads.

Structured Concurrency

Définition

La Structured Concurrency en Java 21 est une approche pour gérer les tâches asynchrones.

Elle garantit que toutes les tâches d'un programme sont facilement suivies et terminées avant que le programme ne se termine.

Cela facilite la gestion et la maintenance du code asynchrone en regroupant les tâches associées.

Avantages

- **Lisibilité Améliorée** : Facilite la lecture et la compréhension du code.
- **Gestion d'erreurs Simplifiée** : Les erreurs sont gérées de manière cohérente pour toutes les tâches d'un même groupe.
- **Ressources Libérées** : Garantit que les ressources allouées sont libérées une fois la tâche terminée.

Exemple Pratique

L'utilisation de structured concurrency permet de gérer un ensemble de tâches concurrentes comme un groupe unique.

Par exemple, une tâche parent peut lancer plusieurs tâches enfants et s'assurer qu'elles terminent avant elle.

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {  
    Future<String> task1 = scope.fork(this::task1);  
    Future<String> task2 = scope.fork(this::task2);  
    scope.join(); // Attend que toutes les tâches se terminent  
    scope.throwIfFailed(); // Relance les erreurs s'il y en a  
    String result1 = task1.resultNow();  
    String result2 = task2.resultNow();  
}
```

Comparaison avec le Passé

Avant Java 21, gérer des tâches asynchrones nécessitait souvent une gestion manuelle des threads et des ressources, ce qui pouvait être source d'erreurs.

La structured concurrency propose une approche plus claire et sécurisée.

Conclusion

La structured concurrency simplifie significativement la programmation concurrenente en Java en garantissant que toutes les tâches asynchrones sont bien gérées.

Cela permet d'écrire du code plus clair, maintenable et récupérable en cas d'erreur.

Design MVC

Qu'est-ce que le Design MVC ?

- **MVC** signifie Modèle-Vue-Contrôleur.
- C'est un pattern architectural utilisé pour séparer les préoccupations dans une application.
- Permet de diviser l'application en trois parties interconnectées.

Composants du MVC

- **Modèle** : Représente les données et la logique métier.

Il ne dépend pas des vues ou des contrôleurs.

- **Vue** : Présente les données au client.

Elle récupère les données du modèle.

- **Contrôleur** : Gère l'entrée utilisateur.

Il met à jour le modèle et décide quelle vue afficher.

Avantages du Design MVC

- **Séparation des préoccupations** : Facilite la maintenance et l'évolution du code.
- **Réutilisabilité** : Les vues peuvent être modifiées sans affecter la logique métier.
- **Testabilité** : Le modèle peut être testé indépendamment des autres composants.

Implémentation MVC : Exemple

- **Modèle** : Créer une classe `User` pour gérer les données utilisateurs.
- **Vue** : Créer une vue `UserView` pour afficher les informations utilisateurs.
- **Contrôleur** : Implémentez `UserController` pour gérer les actions de l'utilisateur.

Exemple de Code : Modèle

```
public class User {  
    private String name;  
    private int age;  
  
    // Constructeur, getters et setters  
}
```

Exemple de Code : Vue

```
public class UserView {  
    public void printUserDetails(String name, int age) {  
        System.out.println("User: " + name + ", Age: " + age);  
    }  
}
```

Exemple de Code : Contrôleur

```
public class UserController {  
    private User model;  
    private UserView view;  
  
    public UserController(User model, UserView view) {  
        this.model = model;  
        this.view = view;  
    }  
  
    public void updateView() {  
        view.printUserDetails(model.getName(), model.getAge());  
    }  
}
```

Conclusion sur le Design MVC

- MVC est crucial pour organiser une application de manière claire et modulaire.
- En séparant données, interface et logique, il offre flexibilité et simplicité.

- Essentiel pour des projets Java bien structurés et évolutifs.

Introduction à l'Architecture Hexagonale

Concept de Base

L'architecture hexagonale, introduite par Alistair Cockburn, vise à créer des systèmes logiciels où les composants sont découplés les uns des autres.

Cela facilite la maintenabilité et l'évolution du système en permettant d'interchanger des composants sans affecter l'ensemble.

Principe d'Indépendance

L'architecture hexagonale se concentre sur l'indépendance du code métier par rapport à la technologie utilisée (base de données, interface utilisateur).

L'objectif est de rendre le noyau applicatif indépendant du monde extérieur.

Ports et Adaptateurs

L'architecture est composée de ports (interfaces) et d'adaptateurs (implémentations concrètes).

Les ports définissent comment les composants interagissent, tandis que les adaptateurs transforment ces interactions en opérations concrètes.

Exemple de Ports

Par exemple, un port pourrait être une interface définissant les opérations CRUD pour un service.

Les adaptateurs seraient les implémentations de cette interface pour différents systèmes de stockage (base de données SQL, NoSQL, etc.).

Avantages de l'Architecture

- **Facilité de test** : Le découplage facilite l'écriture de tests unitaires et d'intégration.
- **Flexibilité** : Permet de changer technologiques sans affecter la logique métier.
- **Clarté** : Sépare clairement les responsabilités dans le système.

Exercice Pratique

1. Imaginez une application simple avec une logique métier et une base de données.
2. Identifiez les ports nécessaires pour cette application.
3. Définissez les adaptateurs pour deux systèmes de stockage différents.

Conclusion

L'Architecture hexagonale promeut un découplage efficace entre la logique métier et les détails d'implémentation, augmentant ainsi la maintenabilité et l'extensibilité du code.

Tests unitaires

Importance des tests unitaires

Les tests unitaires sont essentiels pour garantir la qualité du code.

Ils permettent de vérifier que chaque partie du programme fonctionne comme prévu.

Cette approche favorise également un développement plus sûr et facilite la maintenance.

Comprendre les tests unitaires

Un test unitaire évalue une petite partie du code – souvent une seule méthode.

L'objectif est de confirmer que cette méthode fonctionne correctement dans tous les scénarios possibles.

Outils de tests unitaires

En Java, JUnit est l'outil le plus couramment utilisé pour les tests unitaires.

Il permet de définir des "assertions" qui vérifient le résultat de vos méthodes.

Écrire un test JUnit

Pour créer un test JUnit, commencez par annoter une méthode avec `@Test`. À l'intérieur, utilisez des assertions comme `assertEquals` pour valider vos résultats.

Exemple :

- `assertEquals(5, additionner(2, 3));`

Bonnes pratiques de tests

- Écrire des tests clairs et compréhensibles.
- Tester un seul comportement par test.
- Nommer les tests de manière descriptive pour indiquer leur objectif.
- Exécuter régulièrement vos tests pour détecter rapidement les bugs.

Conception du projet

Architecture modulaire

L'architecture modulaire consiste à organiser une application en différents modules indépendants.

Cela facilite la maintenance et l'évolution du projet.

En Java, le système de modules introduit avec Java 9 permet de mieux structurer le code, d'éviter les conflits de classe et de limiter la visibilité.

Modules core, service et API

Dans une application modulaire, différents modules se chargent de fonctions spécifiques :

- **Core** : Gère la logique métier principale.
- **Service** : Fournit les fonctionnalités applicatives, souvent en interaction avec le core.
- **API** : Expose des fonctionnalités pour interagir avec l'extérieur, tel qu'une API REST.

Records

Les records, disponibles depuis Java 16, simplifient la création de classes immuables utilisées pour stocker des données.

Ils réduisent le besoin d'écrire du code répétitif tout en fournissant `equals()`, `hashCode()`, et `toString()` par défaut.

Exemple :

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

Pattern Matching

Le pattern matching facilite le travail avec les structures de données complexes, simplifiant le code en remplaçant les chaînes imbriquées de `instanceof` et de `cast`.

C'est un outil puissant pour écrire du code concis et lisible, notamment dans les instructions `switch`.

Exemple :

```
if (obj instanceof String s) {  
    System.out.println(s.toLowerCase());  
}
```

Virtual Threads

Les virtual threads permettent de gérer simultanément plusieurs tâches légères dans un processus tout en utilisant moins de ressources qu'avec les threads traditionnels.

Cela améliore la capacité des applications Java à traiter des opérations concurrentes à grande échelle.

Structured Concurrency

La structured concurrency aide à structurer et à gérer l'exécution concurrente des tâches, améliorant la lisibilité et la fiabilité du code.

Elle propose un ensemble d'API pour exécuter et synchroniser des tâches parallèles tout en simplifiant la gestion des erreurs.

Design MVC

Le modèle MVC (Modèle-Vue-Contrôleur) sépare une application en trois composants :

- **Modèle** : Logique des données et métier.
- **Vue** : Interface utilisateur.
- **Contrôleur** : Interactions et traitement des événements.

Architecture hexagonale

L'architecture hexagonale vise à créer des applications extensibles et testables, isolant le cœur métier de la technologie utilisée pour l'implémenter.

Elle permet de séparer les préoccupations pour une meilleure évolutivité.

Tests unitaires

Les tests unitaires vérifient le bon fonctionnement des unités de code individuelles.

Consistent en des cas de tests autonomes, ils aident à identifier les régressions rapidement.

Utilisez des frameworks comme JUnit pour les tests en Java.

Introduction à la gestion concurrente

La gestion concurrente en Java permet l'exécution simultanée de plusieurs tâches.

Cela améliore les performances des applications, surtout en présence de tâches pouvant être exécutées indépendamment.

Java offre plusieurs outils pour gérer la concurrence, comme les threads, exécutors, et les collections synchronisées.

Ces outils permettent de gérer la complexité des appels concurrents de manière efficace et sécurisée.

Les threads en Java

Un thread est une unité d'exécution distincte concurrente dans une application.

Java permet de créer et gérer des threads grâce à l'interface `Runnable` ou la classe `Thread`.

Exemple de création d'un thread :

```
public class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Thread exécuté !");  
    }  
}  
  
Thread thread = new Thread(new MyRunnable());  
thread.start();
```

Les exécutors

Le framework Executor simplifie la gestion des threads en Java.

Il permet de définir un pool de threads pour traiter des tâches asynchrones, améliorant ainsi la répartition des charges de travail.

Exemple d'utilisation d'un `ExecutorService` :

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.execute(() -> System.out.println("Tâche 1 exécutée"));
executor.shutdown();
```

Collections synchronisées

Les collections synchronisées assurent les opérations atomiques sur les collections partagées entre plusieurs threads.

Cela prévient les problèmes liés à la modification concurrente.

Utilisation de `Collections.synchronizedList` :

```
List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());
synchronizedList.add("élément");
```

Exercice pratique

Créez un programme Java qui utilise un pool de threads pour calculer la somme d'une série de nombres.

Utilisez `ExecutorService` pour gérer les threads et `List` pour stocker les résultats partiels de chaque thread.

Cet exercice mettra en pratique la création et gestion de threads, en utilisant les concepts abordés dans la gestion concurrente pour optimiser les performances du calcul.

Gestion réactive

Concepts de gestion réactive

La gestion réactive se concentre sur la gestion efficace du flux de données asynchrones entre les composants d'une application.

Elle permet de gérer les événements de manière non bloquante, permettant ainsi une meilleure utilisation des ressources CPU.

Java 21 et réactivité

Java 21 continue d'améliorer le support pour la programmation réactive introduit dans les versions précédentes.

Il inclut des améliorations sur les API réactives qui permettent de construire des applications réactives plus robustes et performantes.

Publication et souscription

Le modèle Pub/Sub est au cœur de la gestion réactive.

Les éditeurs publient des messages, et les abonnés les reçoivent sans être informés de la source du message.

Ce découplage améliore l'évolutivité et la flexibilité du système.

APIs réactives en Java

Java propose plusieurs API pour développer des applications réactives.

Parmi elles, Reactive Streams est une spécification qui permet de traiter des flux asynchrones de manière non bloquante.

Exemple avec Reactive Streams

Pour utiliser Reactive Streams, il est crucial de comprendre ses principaux composants : Publisher, Subscriber, Subscription et Processor.

Chacun joue un rôle dans la gestion du flux de données asynchrones dans une application réactive.

Publisher et Subscriber

- **Publisher** : Émet des éléments aux abonnés inscrits.
- **Subscriber** : Reçoit les éléments du publisher et gère leur traitement asynchrone.

Subscription et Processor

- **Subscription** : Assure la communication entre Publisher et Subscriber, contrôlant le flux.
- **Processor** : Agit comme à la fois Publisher et Subscriber, transformant les éléments.

Avantages de la programmation réactive

- Scalabilité améliorée
- Temps de réponse réduit
- Utilisation optimale des ressources système.

Conclusion

La gestion réactive en Java 21 permet la création d'applications résilientes et performantes, prêtes à s'adapter aux exigences modernes du développement logiciel axé sur les événements.

HTTPClient asynchrone

Introduction à HTTPClient asynchrone

L'HTTPClient asynchrone de Java 21 permet d'envoyer des requêtes HTTP sans bloquer le fil d'exécution.

Cela est particulièrement utile pour les applications nécessitant des appels réseau fréquents et performants.

Pourquoi utiliser HTTPClient asynchrone?

- **Performances** : Optimise l'utilisation des ressources en effectuant des appels non bloquants.
- **Scalabilité** : Gère plusieurs requêtes simultanément sans surcharge du système.
- **Réactivité** : Parfait pour les applications nécessitant une interaction réseau hautement réactive.

Mise en place d'un HTTPClient

Pour utiliser un HTTPClient asynchrone, commencez par créer une instance d' `HttpClient` en mode asynchrone.

```
HttpClient client = HttpClient.newHttpClient();
```

Cette instance peut être utilisée pour envoyer des requêtes sans verrouiller le thread principal.

Envoyer une requête asynchrone

Pour envoyer une requête GET de manière asynchrone, utilisez `sendAsync`.

```
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://api.exemple.com/data"))
    .build();

CompletableFuture<HttpResponse<String>> response =
    client.sendAsync(request, HttpResponse.BodyHandlers.ofString());
```

Traitement de la réponse

La méthode `sendAsync` retourne un `CompletableFuture` qui peut être utilisé pour traiter la réponse.

```
response.thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

Cette approche permet de manipuler la réponse une fois disponible sans bloquer le thread.

Utilisation des CompletableFuture

- **Combinaison** : `thenCombine` pour combiner des résultats de plusieurs futures.

- **Gestion d'erreurs** : `exceptionally` pour gérer les erreurs de manière non bloquante.
- **Transformations** : `thenApply` pour transformer les données reçues.

Ces méthodes permettent un enchaînement fluide et simplifié des actions à réaliser après la réception d'une réponse HTTP.

Exercice : Implémentez un service

Essayez d'implémenter un petit service qui envoie des requêtes API pour récupérer des données météo de différentes villes de façon asynchrone.

Utilisez `CompletableFuture` pour traiter et afficher les résultats dès leur disponibilité.

Qu'est-ce qu'un log structuré ?

Les logs structurés consistent en des entrées de journal organisées de manière à être facilement interprétées par les machines.

Contrairement aux logs textuels traditionnels, les logs structurés utilisent un format comme JSON pour inclure des champs clés-valeurs, facilitant ainsi l'analyse automatique et la recherche d'informations spécifiques.

Avantages des logs structurés

- **Facilité d'analyse** : Permet une recherche et filtrage avancés.
- **Interopérabilité** : Compatible avec de nombreux outils de monitoring.
- **Clarté** : Informations systématiquement ordonnées.

Mise en œuvre avec Java

Java permet la création de logs structurés à l'aide de bibliothèques comme SLF4J avec logback ou Log4j.

Voici un exemple simple d'utilisation de SLF4J avec une structure JSON.

Exemple :

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class StructuredLoggingExample {
    private static final Logger logger = LoggerFactory.getLogger(StructuredLoggingExample.class);

    public static void main(String[] args) {
        logger.info("{\"event\": \"UserLogin\", \"user\": \"john_doe\", \"status\": \"success\"}");
    }
}
```

Configuration de logback

Pour utiliser logback, il est nécessaire de configurer un fichier `logback.xml` qui définit le format des logs.

Exemple de configuration :

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>{"timestamp": "%date", "level": "%level", "thread": "%thread", "message": "%msg"}</pattern>
        </encoder>
    </appender>

    <root level="info">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Conclusion : Importance des logs structurés

Les logs structurés sont essentiels pour un projet Java moderne.

Ils améliorent la lisibilité des logs, facilitent l'analyse automatisée, et rendent l'intégration avec des outils de gestion de logs, comme ELK Stack, beaucoup plus aisée.

Ils aident les développeurs à maintenir et superviser leurs applications efficacement.

Monitoring des threads

Importance du Monitoring

- Surveille l'utilisation et la performance des threads.
- Identifie les goulets d'étranglement et optimise les performances.
- Prévoit les problèmes liés à la concurrence.

Outils de Monitoring

- JConsole : Interface GUI pour surveiller les performances JVM.
- VisualVM : Analyse et dépannage des applications Java.
- Commande `jstack` : Affiche les traces de pile pour déboguer.

JConsole

- Connecte-toi à une application Java en cours d'exécution.
- Affiche l'utilisation CPU, la mémoire, et les threads actifs.
- Idéal pour une surveillance en temps réel.

VisualVM

- Offre une vue d'ensemble détaillée de l'application.
- Profite des options de profiling pour analyser les performances.
- Permet l'exploration profonde des threads et la découverte des fuites de mémoire.

Utilisation de `jstack`

- Exécute `jstack <pid>` pour obtenir la trace des threads.
- Utile pour diagnostiquer les blocages ou les temps d'attente.

- Intègre l'analyse des états de threads et des verrous.

Métriques de Threads

- Threads actifs : Nombre de threads en cours d'exécution.
- Temps d'attente : Périodes passées en état de blocage.
- Consommation de CPU : Utilisation des ressources par thread.

Conseils Pratiques

- Surveille régulièrement l'utilisation des threads.
- Configure des alertes pour détecter les anomalies.
- Utilise des outils de monitoring intégrés pour automatiser l'observation.

Exercice Pratique

- Implémente un petit service Java.
- Utilise JConsole pour surveiller les threads créés.
- Analyse les résultats et propose des optimisations potentielles.

Exercice : service de calcul distribué

Objectif de l'exercice

- Créer un service de calcul distribué simple.
- Utiliser Java 21 pour tirer parti des nouvelles fonctionnalités.
- Mettre en place un système de communication asynchrone entre les services.

Étapes de base

1. Initialiser le projet Java.

- Utiliser Maven pour gérer les dépendances.
- Créer la structure de base du projet.

2. Setup de l'environnement.

- Installer et configurer Docker.
- Utiliser des conteneurs pour l'isolation du service.

Réalisation du service

- Implémenter un service qui effectue un calcul simple (e.g., addition).
- Utiliser Java 21 et ses améliorations sans tomber dans la complexité excessive.
- Assurer que ce service peut gérer plusieurs requêtes simultanément.

Communication entre services

- Mettre en place un mécanisme de communication entre différents services.
- Utiliser HttpClient asynchrone pour appeler le service de calcul depuis un autre service.
- S'assurer du traitement concurrent des requêtes.

Consignes importantes

- Le service doit être capable de gérer les erreurs et notifier ces dernières de manière structurée.
- Mettre en place un système de log structuré pour tracer l'activité du service.
- Monitorer les threads pour optimiser la réactivité et la performance.

Évaluation

- Fonctionnalité du service : chaque opération doit être correcte et efficace.
- Robustesse et gestion des erreurs.
- Qualité du code : respecte-t-il les pratiques modernes de Java 21 ?
- Utilisation efficace des ressources de Docker pour isoler et déployer.

Finalisation et Tests

- Construire le conteneur Docker contenant le service.
- Effectuer des tests de charge pour assurer que le service distribué peut effectivement gérer plusieurs requêtes.
- Tester le système avec des cas limites pour évaluer la robustesse.

Gestion concurrente

La gestion concurrente permet d'exécuter plusieurs tâches simultanément dans une application.

Elle est essentielle pour augmenter la performance et la réactivité, surtout dans les logiciels multitâches.

- Utilise les API concurrentes de Java pour gérer les threads et tâches.
- Les Executors simplifient la gestion des threads en créant un pool de threads.

Gestion réactive

La programmation réactive se concentre sur le développement d'applications non-bloquantes et réactives, améliorant la gestion des flux de données asynchrones.

- Concerne des flux de données, les événements sont traités au fur et à mesure.
- Java 21 propose `Flow`, une API pour la gestion réactive des flux de données.

HTTPClient asynchrone

HTTPClient intégré de Java permet de réaliser des requêtes HTTP de manière asynchrone, améliorant ainsi la performance des applications.

- Permet d'effectuer des requêtes sans bloquer le thread principal.
- Utilise `sendAsync` pour des requêtes non-bloquantes avec futures.

Logs structurés

Les logs structurés sont cruciaux pour le diagnostic et la maintenance des applications.

Ils permettent de suivre le comportement des applications de manière plus efficace.

- Utilise des frameworks de logs qui supportent JSON pour structurer les logs.
- Les logs structurés facilitent l'analyse et le monitoring.

Monitoring des threads

Le monitoring des threads est crucial pour détecter des problèmes liés à la concurrence comme les blocages ou les fuites de mémoire.

- Java fournit `ThreadMXBean` pour superviser l'activité des threads.
- Scrute les threads pour détecter des anomalies dans l'application.

Exercice : Service de calcul distribué

Implémentez un service de calcul distribué qui :

1. Utilise les API Java pour gérer les tâches de calcul de manière concurrente.
2. S'appuie sur un `HttpClient` asynchrone pour recevoir et envoyer des calculs.
3. Intègre des logs structurés pour suivre les calculs effectués.
4. Inclut un système de monitoring des threads pour gérer la charge de calcul.

Instructions :

- Utilisez un ensemble de données à part pour simuler plusieurs utilisateurs.
- Testez la performance sous différentes charges.
- Assurez-vous que votre code est bien optimisé et documenté.

Optimisation mémoire

Importance de l'optimisation

Optimiser la mémoire est crucial pour assurer la performance et la stabilité des applications Java.

Réduire la consommation de mémoire peut prolonger la durée de vie du système et améliorer l'expérience utilisateur.

Le but est d'utiliser efficacement les ressources tout en maintenant une exécution rapide de l'application.

Gestion de la mémoire en Java

Java gère automatiquement la mémoire via le garbage collector.

Cependant, il est essentiel de comprendre comment optimiser son utilisation.

Une gestion efficace inclut la minimisation des objets temporaires et la compréhension du fonctionnement du heap et des stacks.

Réduction des objets temporaires

Créer des objets inutiles augmente la charge du garbage collector.

Il est recommandé de réutiliser les objets existants et d'utiliser des types primitifs lorsque c'est possible.

Cela permet de réduire les allocations fréquentes et d'améliorer les performances.

Utilisation de collections adaptées

Les collections Java offrent différents types de structures de données.

Choisir la collection appropriée réduit la consommation de mémoire.

Par exemple, préférez `ArrayList` à `LinkedList` si l'accès aux éléments est plus fréquent que les modifications.

Utilisation de `StringBuilder`

Lors de la manipulation de chaînes, préférez `StringBuilder` à la concaténation répétée de `String`. `StringBuilder` est plus efficace car il modifie la chaîne directement au lieu de créer de nouveaux objets à chaque concaténation.

Nettoyage des références

Utilisez des références nulles pour les objets non utilisés afin de libérer de la mémoire.

Cela permet au garbage collector de les récupérer plus rapidement.

Toutefois, veillez à ne pas libérer prématièrement des objets encore nécessaires.

Outils de profilage de mémoire

Des outils comme VisualVM, Eclipse Memory Analyzer ou JProfiler vous aident à identifier les fuites de mémoire et les objets non nécessaires dans votre application.

Ces outils fournissent des rapports détaillés pour optimiser le code.

Exercices pratiques

- Identifiez et optimisez une partie du code Java consommant beaucoup de mémoire.
- Utilisez un outil de profilage pour détecter les fuites de mémoire.
- Comparez l'efficacité de différentes collections dans votre projet.

Optimisation CPU

Importance de l'optimisation CPU

L'optimisation CPU vise à réduire le temps de traitement et améliorer l'efficacité des ressources processeur dans une application Java.

Cela est crucial pour une meilleure performance et réactivité.

Identifier les goulots d'étranglement

- Utiliser des outils de profilage pour identifier les sections de code qui consomment le plus de CPU.
- Exemples d'outils : VisualVM, JProfiler.

Code efficace

- Éviter les opérations inutiles dans les boucles.
- Réduire les appels de méthodes redondants.

Exemple :

```
// Moins efficace
for (int i = 0; i < list.size(); i++) {
    // traitement
}

// Plus efficace
int size = list.size();
for (int i = 0; i < size; i++) {
    // traitement
}
```

Utilisation des Threads

- Utiliser des threads pour exécuter des tâches en parallèle.
- Améliore l'utilisation CPU en permettant des traitements simultanés.

Utilisation de Streams

- Les Stream API de Java 8 améliorent l'efficacité en traitant les données en parallèle.
- Utile pour des opérations sur de grandes collections.

Exemple :

```
list.parallelStream().forEach(item -> process(item));
```

Informé de l'état du CPU

- Surveiller et ajuster les configurations JVM pour un meilleur usage du CPU.
- Paramètres comme le garbage collector peuvent impacter les performances.

Évaluer les performances

- Toujours tester et évaluer les optimisations via des benchmarks.
- Cela garantit que les changements apportent réellement des améliorations.

Exercices pratiques

Implémentez un système de calcul parallèle performant en utilisant Java Streams dans un projet. Évaluez et comparez les performances avec une solution séquentielle.

Tests d'intégration

Concepts des tests d'intégration

Les tests d'intégration vérifient l'interaction entre plusieurs composants d'une application.

Ils s'assurent que les modules fonctionnent correctement ensemble après avoir été intégrés, détectant ainsi les incompatibilités ou erreurs entre les interfaces.

Importance des tests

Ces tests sont cruciaux pour identifier les problèmes qui n'apparaissent pas lors des tests unitaires.

Ils garantissent la cohérence fonctionnelle des composants intégrés, aidant à prévenir les dysfonctionnements en production.

Outils pour tests d'intégration

- **JUnit 5** : Utilisé pour les tests automatisés.
- **Testcontainers** : Fournit des environnements de test isolés basés sur Docker.
- **Mock Servers** : Simulent le comportement de services externes.

Exemple avec JUnit 5

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class IntegrationTest {
    @Test
    void testIntegration() {
        // Supposons que vous ayez deux services à intégrer
        ServiceA serviceA = new ServiceA();
        ServiceB serviceB = new ServiceB(serviceA);

        assertEquals("Expected Result", serviceB.performAction());
    }
}
```

Ce test vérifie l'interaction entre `ServiceA` et `ServiceB`, validant que les composants fonctionnent ensemble comme attendu.

Testcontainers : Exemple pratique

Les Testcontainers permettent de créer des environnements réels pour les tests.

```
import org.testcontainers.containers.PostgreSQLContainer;
import org.junit.jupiter.api.Test;

class PostgresTest {
    @Test
    void testWithPostgres() {
        try (PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:latest")) {
            postgres.start();

            // Tester votre code ici avec la base de données PostgreSQL en cours d'exécution
        }
    }
}
```

Ici, Testcontainers crée une instance PostgreSQL pour tester votre application avec une base de données réelle.

Exercice pratique

1. Implémentez un service simple et unifié.
2. Écrivez un test d'intégration utilisant JUnit 5 pour valider l'interaction entre deux modules.
3. Employez Testcontainers pour simuler une base de données ou un autre service externe.

4. Assurez-vous que votre test vérifie correctement que les modules fonctionnent en synergie.

Tests de charge

Introduction aux tests de charge

Les tests de charge évaluent comment une application se comporte sous une forte demande.

L'objectif est d'identifier les limites de performance avant la mise en production, garantissant que l'application peut supporter le volume d'utilisateurs attendu sans dégradation.

Outils pour tests de charge

Voici quelques outils populaires utilisés pour réaliser des tests de charge sur des applications Java :

- **Apache JMeter** : Offre une interface graphique pour tester facilement les performances.
- **Gatling** : Un outil basé sur Scala, connu pour ses scénarios de test flexibles.
- **Locust** : Permet de définir des tests de charge en Python de manière simple et intuitive.

Configurer un test de charge

Pour configurer un test, il est important de définir :

- **Scénarios réalistes** : Reproduire le comportement des utilisateurs.
- **Volume d'utilisateurs** : Simuler le nombre d'utilisateurs simultanés attendu.
- **Durée** : Tester sur une période adéquate pour obtenir des résultats fiables.

Analyser les résultats

Après le test, examinez les métriques clés :

- **Taux de requêtes réussies** : Vérifie l'efficacité serveur.
- **Temps de réponse moyen** : Mesure la rapidité des traitements.
- **Erreurs** : Identifie les raisons des échecs pour les corriger.

Exercice pratique

1. Choisissez un outil de tests de charge (e.g., JMeter).
2. Créez un scénario pour simuler 1000 utilisateurs simultanés accédant à votre application.
3. Lancer le test pendant 10 minutes.
4. Analysez les résultats obtenus et proposez des optimisations si nécessaire.

JUnit 5

Introduction à JUnit 5

JUnit 5 est une plateforme de tests unitaires pour Java, offrant des outils flexibles pour écrire et exécuter des tests de manière efficace.

Sa modularité facilite son intégration dans différents contextes de développement.

Structure de JUnit 5

JUnit 5 se compose de trois sous-projets :

- **JUnit Platform** : Gère l'exécution des tests.
- **JUnit Jupiter** : Fournit l'API permettant d'écrire les tests.
- **JUnit Vintage** : Maintient la compatibilité avec les tests JUnit 3 et 4.

Installation de JUnit 5

Pour utiliser JUnit 5, ajoutez les dépendances Maven suivantes à votre projet :

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.x.y</version>
    <scope>test</scope>
</dependency>
```

Remplacez `5.x.y` par la dernière version disponible.

Annotation de Test

La principale annotation de JUnit 5 est `@Test` .

Elle identifie une méthode comme un cas de test.

Les méthodes de test doivent être publiques et sans retour :

```
import org.junit.jupiter.api.Test;

class MyTests {
    @Test
    void exampleTest() {
        // code de test
    }
}
```

Assertions en JUnit 5

Les assertions vérifient les conditions dans les tests.

JUnit 5 fournit des méthodes comme `assertEquals` , `assertTrue` , et `assertThrows` :

```
import static org.junit.jupiter.api.Assertions.*;

@Test
void testAddition() {
    assertEquals(4, 2 + 2, "2 + 2 devrait être 4");
}
```

Paramétrisation des Tests

JUnit 5 supporte les tests paramétrés via l'annotation `@ParameterizedTest` , permettant d'exécuter la même logique de test avec différents ensembles de données :

```
@ParameterizedTest
@ValueSource(strings = {"racecar", "radar", "level"})
void palindromes(String candidate) {
    assertTrue(isPalindrome(candidate));
}
```

Cycle de Vie des Tests

JUnit 5 offre des méthodes d'initialisation et de nettoyage avec `@BeforeEach` et `@AfterEach`.

Elles préparent le contexte de test et libèrent les ressources après chaque test :

```
@BeforeEach  
void setUp() {  
    // set up  
}  
  
@AfterEach  
void tearDown() {  
    // clean up  
}
```

Tester des Exceptions

Pour vérifier qu'une méthode lance une exception, utilisez `assertThrows` avec JUnit 5 :

```
@Test  
void testException() {  
    Throwable exception = assertThrows(  
        IllegalArgumentException.class, () -> {  
            // code qui lance l'exception  
        }  
    );  
    assertEquals("Message d'erreur", exception.getMessage());  
}
```

Exercice Pratique

Créez un projet Java utilisant JUnit 5 pour tester une calculatrice simple.

Implémentez des tests unitaires pour additionner, soustraire, multiplier et diviser deux nombres.

Assurez-vous d'inclure des tests pour valider le comportement normal ainsi que les exceptions, par exemple la division par zéro.

Testcontainers

Introduction à Testcontainers

Testcontainers est une bibliothèque Java qui facilite les tests d'intégration avec des services basés sur des conteneurs.

Elle utilise Docker pour créer des environnements isolés et reproductibles, permettant de tester les interactions avec des bases de données, des systèmes de messagerie ou d'autres services externes.

Avantages de Testcontainers

- **Isolation complète :** Chaque test utilise un conteneur dédié, évitant toute interférence entre tests.
- **Environnements reproductibles :** Les conteneurs garantissent des conditions de test identiques à chaque exécution.
- **Simplicité :** Réduit la complexité de configuration des services pour les tests.

Exemple d'utilisation

Pour illustrer l'utilisation de Testcontainers, considérons le cas d'une base de données PostgreSQL.

Voici un exemple de code Java :

```
PostgreSQLContainer<?> postgresContainer = new PostgreSQLContainer<>("postgres:latest")
    .withDatabaseName("testdb")
    .withUsername("user")
    .withPassword("password");
postgresContainer.start();
```

Ce code crée et démarre un conteneur PostgreSQL prêt à l'emploi.

Intégration avec JUnit 5

Testcontainers s'intègre bien avec JUnit 5, permettant d'automatiser la gestion de cycles de vie des conteneurs.

Par exemple :

```
@ExtendWith(TestcontainersExtension.class)
@Testcontainers
public class SomeTest {

    @Container
    public static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:latest");

    @Test
    public void testDatabaseConnection() {
        // logique de test utilisant le conteneur
    }
}
```

Cela assure que le conteneur démarre avant les tests et s'arrête après.

Exercice pratique

Créez un test unitaire utilisant Testcontainers pour vérifier la connexion à une base de données PostgreSQL.

Configurez le conteneur pour initialiser une base de données avec des données de test, puis validez l'accès et la manipulation des données.

Objectif : assurer que chaque exécution de test offre un environnement propre et isolé.

jlink

Qu'est-ce que jlink?

jlink est un outil qui permet de créer une image Java optimisée en incluant uniquement les modules nécessaires à une application.

Cela permet de réduire la taille de l'application et d'améliorer ses performances.

Pourquoi utiliser jlink ?

- **Réduction de taille** : Exclut les modules inutiles.
- **Performance** : Moins de modules signifie démarrage plus rapide.

- **Distribution simplifiée** : Crée une image Java autonome.

Comment fonctionne jlink ?

jlink génère une image personnalisée du JRE contenant uniquement les modules requis par l'application.

Cela simplifie la distribution car l'image créée inclut le JRE et l'application ensemble.

Exemple d'utilisation de jlink

Pour utiliser jlink, exécutez la commande suivante dans un terminal :

```
jlink --module-path mods:java_home/jmods --add-modules <modules-nécessaires> --output <dossier-de-sortie>
```

Remplacez `<modules-nécessaires>` par les modules Java requis et `<dossier-de-sortie>` par le dossier de destination pour l'image.

Intégrer jlink dans le projet

Pour intégrer jlink dans un projet, identifiez d'abord les modules utilisés par votre application.

Ajoutez les options jlink à votre script de build pour générer l'image.

Bonnes pratiques avec jlink

- **Analyser les modules** : Utilisez `jdeps` pour identifier les modules nécessaires.
- **Testez l'image** : Toujours tester l'image générée sur la plateforme cible.
- **Automatisez** : Intégrez jlink dans votre processus CI/CD pour générer automatiquement des images.

Limitations de jlink

- jlink est limité aux applications modulaires.
- Nécessite une bonne connaissance des modules utilisés par l'application.

- Pas destiné pour des environnements à faible ressource sans optimisation manuelle.

jpackage

Introduction à jpackage

- **jpackage** est un outil introduit dans Java 14.
- Il permet de créer des packages installables pour des applications Java.
- C'est utile pour distribuer facilement une application Java aux utilisateurs finaux.

Pourquoi utiliser jpackage ?

- Simplifie la distribution d'applications Java.
- Génère des installateurs natifs pour différents systèmes d'exploitation.
- Permet d'inclure une JRE avec l'application, réduisant ainsi les dépendances externes.

Fonctionnalités principales

- **Création d'installateurs:** Crée des packages pour Windows, macOS, et Linux.
- **Inclusion de ressources:** Permet d'inclure des fichiers et configurations complémentaires.
- **Options de configuration:** Personnalise l'icône, le nom, et d'autres propriétés de l'application.

Exemple de commande jpackage

1. Voici un exemple de commande pour générer un package :

```
jpackage --input <input-folder> --name <app-name> --main-jar <main-jar-file> --type <installer-type>
```

- **input-folder:** Répertoire contenant l'application et ses dépendances.
- **app-name:** Nom de l'application.
- **main-jar-file:** Fichier JAR principal de l'application.
- **installer-type:** Type d'installateur (e.g., exe, msi, dmg, deb, rpm).

Avantages de jpackage

- **Portabilité:** Packaging pour plusieurs OS sans complexité.
- **Autonomie:** Les utilisateurs n'ont pas besoin d'installer une JVM séparément.
- **Standardisation:** Uniformise le processus de distribution et installation.

Exercice pratique

- **But:** Utilisez jpackage pour créer un installateur pour une application Java 21.
- **Étapes:**
 - i. Préparez votre application et ses dépendances dans un dossier.
 - ii. Choisissez le type d'installateur en fonction de votre OS.
 - iii. Exécutez la commande jpackage avec les paramètres appropriés.
 - iv. Testez l'installateur sur une machine locale pour vérifier son bon fonctionnement.

Complétez l'exercice en documentant chaque étape et en notant les éventuelles difficultés rencontrées.

Livrer une app Java 21

Objectif de l'exercice

- Livrer une application Java 21 prête pour la production.
- Intégrer les optimisations et tests effectués lors de la finalisation.

Préparation du build

- Assurez-vous que le code est exempt d'erreurs de compilation.
- Utilisez `mvn clean package` pour créer un paquet prêt à déployer.
- Résultats attendus : un fichier JAR exécutable et tout autre artefact requis.

Utilisation de jlink

- Création d'une JRE optimisée pour votre application :
 - Identifier les modules Java indispensables.
 - Utilisez `jlink` pour créer une image runtime réduite.

Utilisation de jpackage

- Emballez votre application pour une distribution facile :
 - Utilisez `jpackage` pour créer un installeur natif.
 - Incluez votre JRE personnalisé pour garantir l'environnement souhaité.

Tests finaux

- Exécutez des tests d'intégration et de charge après packaging.
- Vérifiez la performance de l'application dans votre environnement cible.
- Utilisez `Testcontainers` pour simuler les dépendances de service.

Livraison en production

- Documentez le processus de mise en œuvre pour les équipes DevOps.
- Vérifiez que tout l'environnement de production est prêt et configuré.
- Déployez et surveillez en temps réel pour détecter les problèmes éventuels.

Évaluation de l'exercice

- Les colis sont-ils auto-suffisants et exempts d'erreurs ?
- La performance respecte-t-elle les SLA définis ?
- L'application est-elle facile à installer et à utiliser ?

Terminez cet exercice en vous assurant que chaque étape a été validée avec succès avant la livraison finale.

Finalisation

Optimisation mémoire

L'optimisation mémoire implique la gestion efficace des ressources utilisées par l'application.

Java 21 offre des fonctionnalités avancées comme l'utilisation des `Records` pour des objets immuables sans surcharge mémoire.

Pensez à utiliser des collections optimisées (`Set`, `Map`) et privilégiez les références faibles (`WeakReference`) lorsque approprié pour éviter les fuites de mémoire.

Optimisation CPU

Optimiser l'utilisation du CPU consiste à améliorer l'efficacité des algorithmes et la répartition des tâches.

Utilisez les flux parallèles (`parallelStream`) pour tirer parti des architectures multi-cœurs, et profilez le code pour identifier et corriger les goulets d'étranglement dans vos algorithmes.

Tests d'intégration

Les tests d'intégration vérifient le fonctionnement des différents modules ensemble.

En Java, cela implique de s'assurer que toutes les dépendances et interactions entre classes fonctionnent comme prévu après l'assemblage complet du projet.

Utilisez des frameworks comme `JUnit 5` pour structurer et exécuter ces tests.

Tests de charge

Les tests de charge évalueront la capacité de l'application à garantir des performances acceptables sous des charges importantes.

Utilisez des outils comme `Apache JMeter` pour simuler des utilisateurs concomitants et analyser les temps de réponse et la stabilité.

JUnit 5

JUnit 5 est une version moderne du célèbre framework de tests pour Java.

Il offre une architecture modulaire qui permet des tests plus flexibles et intégrés.

Utilisez les annotations comme `@Test`, `@BeforeAll`, et `@AfterEach` pour organiser et automatiser vos tests unitaires et d'intégration.

Testcontainers

Testcontainers facilite le test des applications Java en permettant l'utilisation de conteneurs Docker pour des bases de données, des files d'attente de messages, ou d'autres services essentiels à l'application, créant ainsi des environnements de test intégrés et isolés.

jlink

`jlink` permet de créer une image d'exécution Java optimisée qui contient uniquement les modules requis par l'application.

Cela réduit la taille de l'application et améliore le temps de démarrage, idéal pour les déploiements en production.

jpackage

`jpackage` permet la création d'installateurs natifs pour vos applications Java.

Il facilite la distribution de logiciels en emballant l'application et l'environnement d'exécution Java dans un fichier d'installation unique pour les utilisateurs finaux sur diverses plateformes.

Exercice : Livrer une application

Livrez une application Java 21 prête à la production en utilisant les outils et méthodes suivants :

- Optimisez votre code pour la mémoire et le CPU.
- Écrivez des tests d'intégration et de charge.
- Utilisez JUnit 5 pour les tests unitaires.
- Configurez des conteneurs de test avec Testcontainers.
- Générez une image avec `jlink`.
- Créez un package d'installation avec `jpackage`.

Projet final

Conception du projet

Architecture modulaire

L'architecture modulaire sépare l'application en modules distincts, facilitant la maintenance, la réutilisabilité et la mise à jour indépendante.

En Java 21, l'utilisation de modules garantit des dépendances bien définies et réduit les dangers potentiels de couplage serré.

Modules core, service et API

Chaque projet Java modulaire est souvent divisé en trois types principaux de modules :

- **Core** : Fournit les fonctionnalités de base.
- **Service** : Contient la logique métier.
- **API** : Expose des interfaces consommables par d'autres logiciels ou modules.

Records

Les records simplifient la déclaration de classes immuables très utilisées, en intégrant automatiquement les méthodes telles que `equals()`, `hashCode()` et `toString()`.

Idéal pour des objets contenant simplement des données.

Pattern Matching

Le pattern matching permet de réduire les erreurs potentielles en vérifiant et extrayant les informations des objets plus efficacement.

Lors de la comparaison d'objets, il diminue la nécessité d'utiliser des `instanceof` en cascade.

Virtual Threads

Avec les virtual threads, Java 21 améliore significativement le modèle de traitement concurrent.

Très légers, ils permettent d'augmenter la scalabilité en supportant des milliers de threads sans consommer beaucoup de ressources.

Structured Concurrency

La structured concurrency offre une approche systématique pour gérer les threads.

Elle garantit que les opérations concurrentes sont organisées de manière logique et que les erreurs sont gérées centralement, améliorant ainsi la fiabilité du code.

Design MVC

Le Model-View-Controller (MVC) est un design pattern qui sépare la logique de l'application de l'interface utilisateur.

Cela facilite le développement d'applications maintenables et testables en isolant chaque composant.

Architecture hexagonale

L'architecture hexagonale met l'emphase sur la séparation nette entre le domaine métier et les mécanismes techniques.

Elle permet de rendre les applications plus adaptables en mélangeant et échangeant des couches techniques sans affecter le cœur de l'application.

Tests unitaires

Les tests unitaires permettent de vérifier le bon fonctionnement des petites unités de code, garantissant ainsi la fiabilité des fonctionnalités.

Java propose JUnit pour les implémenter facilement, testant une fonction à la fois.

Implémentation

Gestion concurrente

La gestion concurrente permet de traiter des tâches simultanément pour améliorer l'efficacité.

Java 21, avec les threads virtuels et les structures de concurrency, offre des outils puissants pour une gestion de la concurrence moderne.

Gestion réactive

La programmation réactive en Java 21 permet de concevoir des applications plus réactives et résilientes.

Elle gère les flux de données de manière asynchrone en alignant les flux d'événements avec les traitements en temps réel.

HTTPClient asynchrone

L'HTTPClient asynchrone permet de traiter des requêtes HTTP sans bloquer le thread courant, améliorant ainsi la performance des applications.

Utilisez-le pour des appels réseau efficace et non bloquants.

Logs structurés

Les logs structurés fournissent des informations détaillées et formatées, facilitant le suivi et l'analyse des erreurs.

En structurant les logs, on obtient une meilleure visibilité sur l'exécution de l'application.

Monitoring des threads

La surveillance des threads est cruciale pour optimiser et diagnostiquer les problèmes de performance.

Grâce à des outils avancés, Java 21 offre une observation fine des threads actifs et confronte les goulets d'étranglement.

Exercice : implémenter un service de calcul distribué

Développez un service de calcul distribué capable de :

- Recevoir des requêtes de calcul via une API REST.
- Gérer des calculs concurrentiels à l'aide de virtual threads.
- Implémenter des logs structurés pour le suivi des requêtes.
- Assurer une surveillance des ressources via un système de monitoring.

Finalisation

Optimisation mémoire

L'optimisation mémoire implique de réduire l'empreinte mémoire de l'application.

Utilisez des structures de données efficientes et éliminez les objets inutiles pour améliorer la performance.

Optimisation CPU

Améliorer l'utilisation du CPU permet d'accélérer l'application.

Assure-toi que le code évite les calculs redondants et tire parti des capacités de multithreading pour maximiser l'efficacité.

Tests d'intégration

Les tests d'intégration vérifient comment les différents modules de l'application interagissent ensemble.

Ils garantissent que les fonctionnalités se combinent harmonieusement et que le système fonctionne comme prévu.

Tests de charge

Les tests de charge simulent l'utilisation de l'application sous diverses charges pour évaluer sa performance.

Ces tests identifient les comportements sous contraintes lourdes, anticipant les besoins d'optimisation.

JUnit 5

JUnit 5 est un framework de test unitaire amélioré pour Java, offrant une flexibilité accrue avec une architecture modulaire.

Utilisez-le pour tester systématiquement chaque partie de votre code.

Testcontainers

Testcontainers facilite l'utilisation de conteneurs Docker pour réaliser des tests isolés, garantissant un environnement de test stable et reproductible.

Idéal pour des tests d'intégration ou d'exécution sous contraintes.

jlink

`jlink` permet de créer des images Java Runtime personnalisées, remplaçant le besoin de dépendances inutiles et réduisant la taille de l'application pour un déploiement optimisé.

jpackage

Le `jpackage` crée des installateurs natifs pour distribuer des applications Java.

Simplifiez le déploiement en fournissant une solution prête pour les utilisateurs finaux, y compris les binaires et scripts nécessaires.

Exercice : livrer une application Java 21 prête à la production

Développez et livrez une application Java 21 en suivant ces objectifs :

- Optimisez la mémoire et le CPU pour des performances accrues.
- Créez des tests d'intégration et de charge pour valider la robustesse.
- Utilisez `jlink` et `jpackage` pour créer une distribution prête à l'emploi.
- Assurez une compatibilité maximale avec les threads virtuels et l'architecture modulaire.

Java 21 – Développement

Fondations et rappels

Syntaxe

Java utilise une syntaxe orientée objet.

Quelques éléments clés incluent les classes, les méthodes et les objets.

Un programme Java commence toujours par une classe qui contient une méthode `main`.

Classes et Objets

Les classes sont les plans de construction des objets.

Un objet est une instance d'une classe.

Cela permet la modularité et la réutilisation du code.

Interfaces

Une interface définit un contrat que les classes implémentant doivent respecter.

Elles contiennent uniquement des méthodes abstraites, offrant plusieurs façons de structurer le code.

Héritage

L'héritage permet à une classe de dériver les propriétés et le comportement d'une autre classe.

Cela favorise la réutilisation du code.

Polymorphisme

Le polymorphisme permet d'utiliser une interface unique pour représenter différents types de données.

Il est souvent utilisé pour le remplacement des méthodes.

Encapsulation

L'encapsulation consiste à restreindre l'accès aux composants internes d'une classe et exposer uniquement ce qui est nécessaire.

Surcharge et Redéfinition

La surcharge permet plusieurs méthodes avec le même nom, mais des signatures différentes dans une classe.

La redéfinition permet de modifier le comportement d'une méthode héritée.

Portée des variables

La portée des variables détermine où dans le programme une variable peut être utilisée.

Les niveaux de portée incluent : local, instance, et classe.

Principes SOLID

SOLID est un ensemble de principes qui favorisent une conception logique et maintenable :

- Unique responsabilité
- Ouvert/fermé
- Substitution de Liskov
- Ségrégation des interfaces
- Inversion de dépendance

Principe DRY

DRY signifie "Don't Repeat Yourself".

Cela encourage la réduction de la duplication des informations dans un projet.

Principe KISS

KISS signifie "Keep It Simple, Stupid".

Cela suggère de garder les systèmes aussi simples que possible pour une maintenance facile.

Exercice SOLID

Refactorisez une classe pour qu'elle respecte les principes SOLID.

Identifiez les points de non-conformité et réimplémentez les parties concernées.

APIs standards

List

`List` est une collection ordonnée qui peut contenir des éléments dupliqués.

Elle offre des méthodes pour manipuler la liste à des positions spécifiques.

Set

`Set` est une collection qui ne permet pas de doublons.

Elle est idéale pour les opérations d'unicité.

Map

`Map` est une collection qui associe des clés à des valeurs.

Elle ne permet pas de clés en double, mais les valeurs peuvent être dupliquées.

Queue et Stack

`Queue` est une structure de données suivant le principe FIFO.

`Stack` est une pile, LIFO, souvent utilisée pour des opérations de backtracking.

API I/O

L'API I/O gère les entrées-sorties en Java.

Elle inclut des classes pour lire et écrire des fichiers de manière séquentielle.

API NIO et NIO.2

NIO fournit des opérations non bloquantes et des buffers.

NIO.2 améliore la gestion des fichiers et des systèmes de fichiers.

Sérialisation d'objets

La sérialisation convertit un objet en un flux de bytes, permettant son stockage ou sa transmission.

Désérialisation d'objets

La désérialisation est l'opération inverse de la sérialisation, reconstruisant un objet à partir de bytes.

Classe Optional

`Optional` est utilisé pour représenter une valeur potentiellement absente, évitant les exceptions `NullPointerException`.

Gestion d'exceptions

Les exceptions en Java permettent de gérer les erreurs de manière contrôlée.
Elles sont gérées avec les blocs `try`, `catch`, et `finally`.

Try-with-resources

Cette structure garantit la libération des ressources une fois la tâche terminée, optimisant ainsi la gestion des ressources.

Exercice I/O

Lisez et écrivez dans un fichier, tout en gérant les exceptions possibles avec `try-with-resources`.

Programmation fonctionnelle

Lambda expressions

Les expressions lambda permettent une syntaxe simplifiée pour écrire des implémentations courtes d'interfaces fonctionnelles.

Références de méthode

Les références de méthode sont des raccourcis pour appeler des méthodes ou des constructeurs, optimisant ainsi le code lambda.

Interfaces fonctionnelles

`Function` , `Predicate` , `Supplier` , `Consumer` sont des interfaces permettant une programmation fonctionnelle en Java.

Stream API

Stream API permet de traiter des données de manière déclarative en utilisant des chaînes de requêtes de méthodes.

Opérateurs map, filter, reduce

`map` , `filter` , `reduce` sont utilisés pour transformer, filtrer et réduire les éléments d'un flux de données respectivement.

Collectors

Les collecteurs sont des objets qui résument les résultats des flux, comme les listes et les ensembles.

API Date & Time

API moderne pour la manipulation de dates et d'heures (`LocalDate` , `LocalTime` , `ZonedDateTime`).

Exercice Streams

Créez un pipeline de transformation de données utilisant `Stream` , `filter` , et `map` pour manipuler des données.

Évolution du langage

De Java 8 à Java 12

Modularité (Java 9)

Project Jigsaw introduit la modularité, permettant un découpage du JDK en modules pour une meilleure gestion des dépendances.

Inférence de type (Java 10)

Avec `var`, on déduit le type des variables locales à la compile, rendant le code plus concis sans perdre la qualité de typage.

Switch expressions (Java 12)

Les switch expressions simplifient le switch en fournissant une valeur de retour, rendant le code plus lisible et moins verbeux.

Garbage Collector G1 et ZGC

G1 et ZGC sont des garbage collectors conçus pour optimiser les performances des applications en gérant efficacement la mémoire.

Exercice Java Modules

Créez un mini-module Java, gérez ses dépendances internes en utilisant les fonctionnalités de modularité de Java 9.

De Java 13 à Java 17

Text Blocks

Text Blocks (`"""`) simplifient la gestion de chaînes multilignes, rendant le code plus lisible sans concaténations ou échappements.

Records

Records sont des classes concises pour stocker des données immuables, réduisant le code standard associé aux POJOs.

Pattern Matching

Pattern Matching apporte plus de lisibilité en élaguant les vérifications de contrainte comme pour `instanceof`.

Classes scellées

Les classes scellées restreignent quelles classes peuvent les étendre, améliorant le contrôle et la maintenabilité.

Exercice Hiérarchies

Refactorisez une hiérarchie de classes en utilisant des records et des classes scellées pour améliorer la clarté et la sécurité.

De Java 18 à Java 21

Virtual Threads

Project Loom introduit les virtual threads pour une gestion simplifiée et performante de la concurrence.

Structured Concurrency

La concurrence structurée modélise les sous-tâches en tant que blocs contigus, simplifiant la gestion de leur intégration.

Scoped Values

Utilisez les valeurs délimitées pour associer des valeurs aux threads, facilitant la communication entre tâches.

Record Patterns

Record patterns permettent une déconstruction des records pour extraire directement des champs dans les expressions.

Guarded Patterns

Les guarded patterns autorisent des conditions sur les patterns, optimisant le contrôle des flux.

Foreign Function & Memory API

Cette API facilite l'accès et la manipulation de fonctions et de structures de mémoire hors JVM.

Exercice Virtual Threads

Implémentez un service performant même sous charge en utilisant les virtual threads, tout en assurant la fiabilité.

Programmation Java avancée