

# Getting started with Java

# La concurrence

# Threads et Multithreading


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

# Concepts de Base des Threads

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

# Mémoire Partagée et Communication

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

 center

# Tâches et Threads

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

# Comprendre la Concurrency des Threads

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

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



# Création et Gestion des Threads

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

# Différences entre `start()` et `run()`

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

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

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

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

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

# Meilleures Pratiques avec Runnable

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

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

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

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

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

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

# Types de Threads en Java

- **Threads système :**

- Créés par la JVM, exécutent des tâches en arrière-plan.
- Exemple : **Garbage Collector**.

- **Threads définis par l'utilisateur :**

- Créés par le développeur pour des tâches spécifiques.
- Par défaut, un programme a un thread utilisateur : celui qui exécute `main()`.

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

# Threads Daemon

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



# Exemple : Thread utilisateur vs daemon

```
public class Zoo {  
    public static void pause() {  
        try {  
            Thread.sleep(10_000); // Attend 10 secondes  
        } catch (InterruptedException e) {}  
        System.out.println("Thread terminé !");  
    }  
  
    public static void main(String[] args) {  
        var job = new Thread(() -> pause());  
        // job.setDaemon(true); // Activer pour un daemon  
        job.start();  
        System.out.println("Main terminé !");  
    }  
}
```

Sortie (sans daemon) :

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

Sortie (avec daemon) :

```
Main terminé !
```

# Threads Virtuels : Détails

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

# Création d'un thread virtuel

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

# Exemple : Avantages des threads virtuels

## Sans thread virtuel (classique)

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

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

# Avec threads virtuels

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

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

# Cycle de vie des Threads

center

# États des Threads

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



# Points importants sur les Threads Virtuels

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

# États d'un Thread

```
public class CheckResults {  
    private static int counter = 0;  
  
    public static void main(String[] args) {  
        new Thread(() -> {  
            for (int i = 0; i < 1_000_000; i++) counter++;  
        }).start();  
  
        while (counter < 1_000_000) {  
            System.out.println("Pas encore atteint");  
        }  
        System.out.println("Atteint : " + counter);  
    }  
}
```

# Amélioration avec sleep()

Mauvaise pratique : Boucle infinie

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

## Utiliser Thread.sleep():

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

**Avantage:** le CPU pour d'autres tâches.

## Points importants

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

# Exemple de Polling sans Sleep

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

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

## Problèmes

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

# Amélioration avec Thread.sleep()

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



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

## Explication

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

# Limites du Polling avec Sleep

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

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

# Amélioration avec Thread.interrupt()

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

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

## Explication

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

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

# Créer des Threads avec l'API de Concurrency

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



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

# Introduction à l'Executor à Thread Unique

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

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

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

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

# Arrêter un Executor de Thread

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

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

# Arrêter un Executor de Thread

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

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

# Exemple d'arrêt ordonné avec shutdown()

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

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

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

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

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

# Exemple d'arrêt immédiat avec shutdownNow()

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

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

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

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

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



# Soumettre des Tâches

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

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

## Exemple de soumission avec execute() (pour Runnable)

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

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

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

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

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

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

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

## Avec Runnable :

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

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

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

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

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

## Avec Callable (avec retour de valeur) :

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

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

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

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

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

# Gestion des Exceptions lors de la soumission des tâches

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

# Exemple avec Callable et gestion d'exception

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

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

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

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

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

# Attendre les Résultats

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

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



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

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

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

# L'interface Callable

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

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

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

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

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

# Sécurité des threads

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

# Problème d'accès concurrent aux données

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

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

public class SheepManager {
    private int sheepCount = 0;

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

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

Ce programme peut produire des sorties inattendues, comme :

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

Le problème réside dans l'incrémentation de `sheepCount`.

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



# Mot-clé volatile

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

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

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

# Classes atomiques

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

## Exemple avec `AtomicInteger` :

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

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

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

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

# Résumé des classes atomiques

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

Nom de la classe	Description
<code>AtomicBoolean</code>	Valeur booléenne modifiable de manière atomique
<code>AtomicInteger</code>	Entier modifiable de manière atomique
<code>AtomicLong</code>	Long modifiable de manière atomique

# Améliorer l'accès avec des blocs

synchronized

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

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

# Exemples de méthodes atomiques courantes

Méthode	Description
<code>get()</code>	Récupère la valeur actuelle
<code>set(newValue)</code>	Définit la nouvelle valeur
<code>getAndSet(newValue)</code>	Définit la nouvelle valeur et retourne l'ancienne



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

# Bloc `synchronized` en Java

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

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

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

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

## Exemple : Gestion des threads avec `synchronized`

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

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

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

# Solution correcte : synchronisation de l'exécution des threads

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

## Résultat

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

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

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

# Synchronisation sur une méthode

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

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

Ou de manière plus concise :

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



# Synchronisation statique

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

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

# Le framework de verrouillage (Lock)

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

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

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

## Méthodes de l'interface Lock

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

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

## Utilisation de `tryLock()`

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

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

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

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

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

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

## **tryLock(long, TimeUnit)**

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



# Utilisation des Collections Concurrentes

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

## Comprendre les erreurs de cohérence mémoire

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

## Exemple de `ConcurrentModificationException`

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

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

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

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

# Travailler avec les classes concurrentes

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

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

# Passer une référence de collection concurrente

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

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

# Classes concurrentes courantes

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



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

# Comportement des classes CopyOnWrite

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

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

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

# Classes CopyOnWrite et mémoire

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

# LinkedBlockingQueue

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

# Problèmes de threading

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

# Liveness et ses problèmes

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

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



# Exemple de Deadlock, Starvation et Livelock

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

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

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

# Flux Parallèles (Parallel Streams)

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

## 2. Création de Flux Parallèles

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

- À partir d'un flux existant :

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

- Directement depuis une collection :

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

# Vérification de l'État Parallèle

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

```
Stream<Integer> serialStream = List.of(1, 2, 3, 4).stream();  
Stream<Integer> parallelStream = List.of(1, 2, 3, 4).parallelStream();  
  
System.out.println(serialStream.isParallel()); // false  
System.out.println(parallelStream.isParallel()); // true
```

# Décomposition Parallèle

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

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

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



# Exécution en Flux Série

Exemple d'exécution en flux série:

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

Cela prend environ 5 secondes pour chaque élément.

# Exécution en Flux Parallèles

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

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

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

# Garantie d'Ordre avec `forEachOrdered()`

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

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

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

# Réductions Parallèles

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

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

# Combiner les Résultats avec reduce()

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

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

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

# Utilisation de collect() en Parallèle

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

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

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

# Connexion à une Base de Données

## Construction d'une URL JDBC

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

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



## Exemples :

```
jdbc:hsqldb:file:zoo
```

```
jdbc:postgresql://localhost/zoo
```

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

# Obtenir une Connexion à une Base de Données

Deux principales méthodes :

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

## Exemple avec DriverManager:

```
import java.sql.*;

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

## Exemple avec un utilisateur et un mot de passe:

```
import java.sql.*;

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

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

# Gestion des Exceptions et Fermeture des Ressources

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

## Exemple :

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

## Résultat attendu:

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

# Vérification des Pilotes

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

## Exemple:

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

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

# Travailler avec `PreparedStatement`

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



# Différences principales :

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

# Avantages de `PreparedStatement`

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

# Obtenir un PreparedStatement

Exemple simple :

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

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

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

# Exécuter un PreparedStatement

Modifier des données avec executeUpdate()

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

## Exemple:

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

# Lire des données avec executeQuery()

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

## Exemple:

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

# Méthode générique : `execute()`

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

## Exemple:

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



# Travailler avec des paramètres

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

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

# Récapitulatif des méthodes

Méthode	DELETE	INSERT	SELECT	UPDATE
<code>ps.execute()</code>	Oui	Oui	Oui	Oui
<code>ps.executeQuery()</code>	Non	Non	Oui	Non
<code>ps.executeUpdate()</code>	Oui	Oui	Non	Oui

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

## ## Updating Multiple Records

### Ajouter plusieurs enregistrements avec

PreparedStatement

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

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

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

# Batching Statements

Exécuter plusieurs requêtes en un seul appel réseau

## Avantages :

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

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

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

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

## Appel de la méthode:

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

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



# Récupérer des données d'un ResultSet

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

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

## Comportement du curseur :

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

## Exemple de sortie :

{1=African Elephant, 2=Zebra}

# Accès aux colonnes dans un ResultSet

Accéder aux colonnes en utilisant :

- Le nom : `rs.getInt("id");`
- L'index (commençant à 1) : `rs.getInt(1);`

## Exemple:

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

### \*\* Règles importantes : \*\*

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

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

## Exemple:

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

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

# Introduction aux procédures stockées

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

# Syntaxe pour appeler une procédure

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

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

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



## Exemple: appeler une procédure stockée:

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

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

# Passer un paramètre IN

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

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

Ici, le ? est utilisé pour indiquer un paramètre, et la méthode `setString` définit la valeur du paramètre.

# Appeler une procédure avec un paramètre OUT

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

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

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

# Travailler avec un paramètre INOUT

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

**Exemple :**

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

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

# Types de paramètres d'une procédure stockée

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

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

# Options supplémentaires pour CallableStatement

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

## Types de ResultSet :

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

## Modes de Concurrency de ResultSet :

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



# Utilisation des options dans CallableStatement

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

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

# Contrôler les données avec des transactions

# Commit et Rollback

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

# Cas particuliers avec l'Autocommit

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

# Marquer avec des Savepoints

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

# Méthodes des APIs de transaction

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

# Fermeture des ressources de base de données

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

# Écriture d'une fuite de ressources

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

**Mauvais exemple :**

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



# Éviter les Fuites de Ressources

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

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

# Gestion des Exceptions SQL

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

Exemple :

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

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

# Gestion des Connexions et des Transactions

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

## Exemple de pool de connexions:

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

# Garantir l'Isolation des Transactions

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

## Exemple :

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

# Utilisation des Prepared Statements

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

**Exemple :**

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

# Travailler avec les Transactions dans un Environnement Multi-Thread

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



# Opérations Courantes sur les Bases de Données dans les Transactions

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

# Bonnes Pratiques

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

