Sommaire

Explique-moi la notion de threads en java

En Java, un thread (fil d'exécution) est la plus petite unité d'un programme qui peut être exécutée de manière indépendante. Les threads permettent d'exécuter des tâches en parallèle, ce qui peut améliorer les performances d'une application en utilisant efficacement les processeurs multi-cœurs disponibles sur la plupart des ordinateurs modernes.

Voici quelques points clés à comprendre sur les threads en Java :

Création de threads : Vous pouvez créer des threads en Java de plusieurs manières, mais l'une des approches les plus courantes consiste à étendre la classe Thread ou à implémenter l'interface Runnable. Par exemple, vous pouvez créer un thread en étendant la classe Thread comme ceci :

```
class MonThread extends Thread {
    public void run() {
        // Code à exécuter dans le thread
    }
}
```

Ou en implémentant l'interface Runnable :

```
class MonRunnable implements Runnable {
    public void run() {
        // Code à exécuter dans le thread
    }
}
```

Démarrage d'un thread : Une fois que vous avez créé un thread, vous pouvez le démarrer en appelant **la méthode start()** sur l'instance du thread. Par exemple :

```
Thread monThread = new MonThread();
monThread.start();
```

Méthode run() : La méthode run() est la méthode principale que vous devez implémenter pour définir le comportement du thread. Le code que vous voulez exécuter dans le thread doit être placé dans cette méthode.

Multithreading : Vous pouvez créer plusieurs threads et les exécuter en parallèle pour effectuer des tâches simultanées. Cela peut être utile pour améliorer la réactivité d'une application ou pour exploiter efficacement les ressources du processeur.

Cycle de vie d'un thread : Un thread Java suit un cycle de vie, qui comprend les états suivants : Nouveau, Prêt, En cours d'exécution, En attente, En veille, et Terminé. Vous pouvez utiliser des méthodes telles que start(), sleep(), wait(), et notify() pour gérer le cycle de vie des threads.

Synchronisation : Lorsque plusieurs threads accèdent simultanément à des ressources partagées, il peut y avoir des problèmes de concurrence. Java offre des mécanismes de synchronisation tels que les mots-clés synchronized et les verrous (lock) pour garantir un accès sécurisé aux ressources partagées entre les threads.

ThreadPool : Java propose également des classes comme ExecutorService et ThreadPoolExecutor pour gérer un pool de threads réutilisables, ce qui peut être plus efficace que de créer un nouveau thread à chaque fois que vous en avez besoin.

En résumé, les threads en Java permettent d'exécuter des tâches en parallèle, améliorant ainsi les performances des applications en utilisant efficacement les capacités multi-cœurs des processeurs. Cependant, il est important de prendre en compte les problèmes de concurrence et d'utiliser des mécanismes de synchronisation appropriés pour éviter les conflits entre les threads lors de l'accès à des ressources partagées.

Threads expliqué comme à un enfant :

Imagine que ton ordinateur est comme une cuisine où tu prépares des sandwichs. Un thread est comme un petit chef cuisinier qui prépare un sandwich à la fois.

1. **Un seul chef à la fois** : Normalement, dans une cuisine, il y a un seul chef qui prépare un sandwich à la fois. C'est comme si ton ordinateur avait un seul thread pour faire une tâche à la fois. Par exemple, il peut dessiner une image, puis afficher un jeu, puis jouer de la musique, mais il le fait un par un.

2. **Plusieurs chefs (threads)** : Parfois, tu as besoin de préparer plusieurs sandwichs rapidement parce que tes amis ont faim. Pour aller plus vite, tu peux demander à tes amis de devenir des chefs cuisiniers supplémentaires. Chacun prépare un sandwich à la fois. Cela signifie que plusieurs sandwiches sont prêts en même temps, ce qui est plus rapide que d'attendre qu'un seul chef termine avant de commencer le suivant.

3. **Les chefs doivent partager** : Mais voici le problème : lorsque plusieurs chefs préparent des sandwiches en même temps, ils doivent partager la cuisine et les ingrédients. Parfois, ils peuvent se heurter ou essayer d'utiliser le même couteau en même temps. C'est ce qu'on appelle un conflit.

4. **Synchronisation** : Pour éviter les conflits, il est important de dire aux chefs quand ils peuvent prendre un ingrédient ou quand ils doivent attendre. Par exemple, si un chef utilise le couteau, il doit dire aux autres chefs d'attendre leur tour. C'est comme si tu avais une règle pour que tout se passe bien.

5. **Threads en informatique** : En informatique, c'est un peu la même chose. Les threads sont comme ces chefs qui font des tâches différentes sur l'ordinateur en même temps. Ils permettent de rendre les choses plus rapides, mais il faut faire attention à la façon dont ils se partagent les ressources pour éviter les problèmes.

En résumé, un thread en informatique est comme un petit chef cuisinier qui peut faire une tâche à la fois sur ton ordinateur. Si tu as besoin de faire plusieurs choses en même temps, tu peux avoir plusieurs threads, mais tu dois t'assurer qu'ils partagent bien les ressources et ne se gênent pas les uns les autres. Cela permet de rendre ton ordinateur plus efficace et réactif !

Exercices

Voici quelques exercices simples pour vous aider à comprendre les bases des threads en Java :

Exercice 1 : Création de threads

Créez deux threads en utilisant la classe Thread et la classe Runnable. Chacun de ces threads devrait simplement afficher un message à la console. Assurez-vous que les deux threads s'exécutent en parallèle.

Exercice 2 : Synchronisation de threads

Modifiez l'exercice 1 pour que les deux threads partagent une variable entière. Chaque thread doit incrémenter cette variable 1000 fois. Utilisez la synchronisation pour éviter les conflits et assurez-vous que la variable est correctement incrémentée à la fin de l'exécution.

Exercice 3 : Utilisation d'un ThreadPool

Utilisez un ExecutorService pour créer un pool de threads. Soumettez plusieurs tâches à ce pool pour effectuer des opérations telles que l'impression de nombres pairs et impairs jusqu'à un certain nombre donné. Assurez-vous que les threads sont correctement gérés et réutilisés par le pool.

Exercice 4 : Thread de délai

Créez un thread qui attend pendant un certain temps (par exemple, 5 secondes) et affiche un message une fois le délai écoulé. Utilisez la méthode Thread.sleep() pour mettre le thread en attente.

Exercice 5 : Communication entre threads

Créez deux threads, l'un pour produire des données et l'autre pour les consommer. Utilisez une structure de données comme une file d'attente (Queue) pour que le thread de production ajoute des éléments et que le thread de consommation les retire. Assurez-vous que les threads communiquent correctement entre eux et que le thread de consommation attend s'il n'y a pas de données disponibles.

Exercices corrigés

Exercice 1 : Création de threads

```
class MonThread1 extends Thread {
    public void run() {
        System.out.println("Thread 1 : Bonjour du premier thread !");
    }
}

class MonRunnable2 implements Runnable {
    public void run() {
        System.out.println("Thread 2 : Bonjour du deuxième thread !");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread1 = new MonThread1();
        Thread thread2 = new Thread(new MonRunnable2());

        thread1.start();
        thread2.start();
    }
}
```

Exercice 2 : Synchronisation de threads

```
class MonThread3 extends Thread {
   private static int compteur = 0;

   public void run() {
      for (int i = 0; i < 1000; i++) {
         synchronized (MonThread3.class) {
            compteur++;
         }
      }
   }
   public static void main(String[] args) {
      Thread thread1 = new MonThread3();
      Thread thread2 = new MonThread3();

      thread1.start();
      thread2.start();

      try {
         thread1.join();
         thread2.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      System.out.println("La valeur finale du compteur est : " + compteur);
   }
}
```

Exercice 3 : Utilisation d'un ThreadPool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class TacheImpairs implements Runnable {
    public void run() {
        for (int i = 1; i <= 10; i += 2) {
            System.out.println("Impair : " + i);
        }
    }
}

class TachePairs implements Runnable {
    public void run() {
        for (int i = 2; i <= 10; i += 2) {
            System.out.println("Pair : " + i);
        }
    }
}

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

        pool.submit(new TachePairs());
        pool.submit(new TacheImpairs());
        pool.shutdown();
    }
}
```

Exercice 4 : Thread de délai

```
class ThreadDélai extends Thread {
    public void run() {
        try {
            Thread.sleep(5000); // Attente de 5 secondes
            System.out.println("Le délai est écoulé !");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Thread thread = new ThreadDélai();
        thread.start();
    }
}
```

Exercice 5 : Communication entre threads

```
java
Copy code
import java.util.LinkedList;
import java.util.Queue;

class Producteur implements Runnable {
    private Queue<Integer> file;

    public Producteur(Queue<Integer> file) {
        this.file = file;
    }
```

```
    public void run() {
        for (int i = 1; i <= 5; i++) {
            synchronized (file) {
                file.offer(i);
                System.out.println("Produit : " + i);
                file.notify();
            }
        }
    }
}

class Consommateur implements Runnable {
    private Queue<Integer> file;

    public Consommateur(Queue<Integer> file) {
        this.file = file;
    }

    public void run() {
        while (true) {
            synchronized (file) {
                while (file.isEmpty()) {
                    try {
                        file.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                int valeur = file.poll();
                System.out.println("Consommé : " + valeur);
            }
```

```
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Queue<Integer> file = new LinkedList<>();

        Thread producteur = new Thread(new Producteur(file));
        Thread consommateur = new Thread(new Consommateur(file));

        producteur.start();
        consommateur.start();
    }
}
```

J'espère que ces corrigés vous aideront à mieux comprendre les concepts de base des threads en Java. N'hésitez pas à les étudier et à les exécuter pour voir comment ils fonctionnent en pratique.