Programmation Asynchrone / Concurrence

---

1. Programmation Asynchrone

La programmation asynchrone permet d'exécuter des tâches sans bloquer le fil d'exécution principal. L'idée est de lancer des opérations (par exemple, des appels réseau) et de les traiter une fois terminées, sans attendre leur achèvement pour continuer le reste du programme.

Exemple en Java (avec `CompletableFuture`) :

```java
import java.util.concurrent.CompletableFuture;

public class AsyncExample {
    public static void main(String[] args) {
        System.out.println("Début du traitement...");

        // Création d'une tâche asynchrone
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            try {
                Thread.sleep(2000); // Simule un traitement long
                System.out.println("Traitement terminé !");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        System.out.println("Tâche asynchrone lancée, le programme continue...");

        // Attendre la fin de la tâche (optionnel)
        future.join(); // Attend que la tâche soit terminée avant de quitter
        System.out.println("Fin du programme.");
    }
}
```

Concept : Ici, nous lançons un traitement asynchrone avec `CompletableFuture.runAsync`, qui s'exécute de manière non bloquante. Le programme continue sans attendre, puis on peut éventuellement bloquer avec `join()` si nécessaire.

Exemple en JavaScript (avec `async/await`) :

```
function asyncTask() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Traitement terminé !");
            resolve();
        }, 2000); // Simule un traitement long
    });
}

async function main() {
    console.log("Début du traitement...");

    // Lancer la tâche asynchrone
    const result = asyncTask();
    console.log("Tâche asynchrone lancée, le programme continue...");

    await result; // Attendre que la tâche se termine
    console.log("Fin du programme.");
}

main();
```

Concept : Avec `async/await`, on exécute une fonction asynchrone (`asyncTask`) qui retourne une promesse. La fonction `main` continue pendant que la tâche est en cours, et `await` permet d'attendre son achèvement sans bloquer l'exécution.

---

2. Concurrence

La concurrence consiste à exécuter plusieurs tâches de manière "simultanée", que ce soit sur des threads différents ou en partageant le même thread. Cela permet de **multitâcher** dans une application, par exemple, pour gérer plusieurs requêtes en même temps.

Exemple en Java (avec les threads) :

```
public class ThreadExample {
    public static void main(String[] args) {
        // Création de deux threads concurrents
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread 1 - Compte: " + i);
                try {
                    Thread.sleep(500); // Pause de 500 ms
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
```

```
        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread 2 - Compte: " + i);
                try {
                    Thread.sleep(500); // Pause de 500 ms
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Lancement des threads
        thread1.start();
        thread2.start();
    }
}
```

Concept : Ici, deux threads sont lancés et exécutés en parallèle. Chaque thread effectue une boucle et imprime un message, démontrant ainsi la **concurrence**.

Exemple en JavaScript (avec Promises) :

```
function task1() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Tâche 1 terminée");
            resolve();
        }, 1000); // Simule un traitement de 1 seconde
    });
}

function task2() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Tâche 2 terminée");
            resolve();
        }, 1500); // Simule un traitement de 1,5 seconde
    });
}

// Exécuter deux tâches en parallèle
Promise.all([task1(), task2()]).then(() => {
    console.log("Toutes les tâches sont terminées");
});
```

Concept : Ici, `Promise.all` exécute deux tâches asynchrones en parallèle. Les deux tâches peuvent se terminer dans un ordre différent, mais le programme attend que toutes soient terminées avant d'afficher le message final.

---

3. **Différences Clés : Programmation Asynchrone vs Concurrence**

- **Asynchrone** : Permet d'exécuter des tâches en arrière-plan sans bloquer le thread principal. Elle se concentre sur le **non-blocage** et l'exécution des opérations à terme, tout en laissant le programme principal continuer son exécution.
  - Exemple : Utilisation d'`async/await` en JavaScript ou de `CompletableFuture` en Java.
- **Concurrence** : Permet d'exécuter plusieurs tâches en même temps, que ce soit avec des threads multiples ou des processus parallèles. Cela permet de **multitâcher** et d'exécuter plusieurs choses à la fois.
  - Exemple : Threads en Java ou `Promise.all()` en JavaScript pour exécuter plusieurs tâches simultanément.

Comparaison avec Commentaires :

Programmation Asynchrone (JavaScript) :

```
function asyncTask() {
    return new Promise((resolve) => {
        setTimeout(() => {
            console.log("Traitement terminé !");
            resolve();
        }, 2000); // Simule un traitement long
    });
}

async function main() {
    console.log("Début du traitement...");

    // Lancer la tâche asynchrone
    const result = asyncTask();
    console.log("Tâche asynchrone lancée, le programme continue...");

    await result; // Attendre que la tâche se termine
    console.log("Fin du programme.");
}

main();
```

Commentaire : Ici, `asyncTask` simule une opération longue avec `setTimeout`, mais la fonction principale continue à s'exécuter pendant ce temps grâce à `async/await`.

Concurrence (Java) :

```java
public class ThreadExample {
    public static void main(String[] args) {
        // Création de deux threads concurrents
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread 1 - Compte: " + i);
                try {
                    Thread.sleep(500); // Pause de 500 ms
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread 2 - Compte: " + i);
                try {
                    Thread.sleep(500); // Pause de 500 ms
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        // Lancement des threads
        thread1.start();
        thread2.start();
    }
}
```

Commentaire : Chaque thread exécute un morceau de code en parallèle, affichant le résultat sans attendre que l'autre thread ait terminé.

Conclusion :

La programmation asynchrone est idéale pour gérer des tâches longues sans bloquer l'exécution du programme principal. La concurrence, quant à elle, permet d'exécuter plusieurs tâches simultanément, souvent sur des threads différents. Ces deux concepts sont essentiels pour développer des applications performantes et réactives.
