12.2-Mutliple Threads

Understanding Threads in Java

In the initial stages of learning about threads, we typically create them manually to understand the core concepts. However, as we delve into more advanced topics, we'll encounter frameworks and libraries that automate thread management, allowing developers to focus solely on building optimized business logic.

Let's start by exploring the basics using a simple example.

Example without Threads

```java
 class Hi{
    public void show() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hi, I'm from class Hi");
        }
    }
}

class Hello{
    public void show() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello, I'm from class Hello");
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        Hi obj1 = new Hi();
        Hello obj2 = new Hello();

        obj1.show(); // Call method of class Hi
        obj2.show(); // Call method of class Hello
    }
}
```

Output:

```
Hi, I'm from class Hi
Hi, I'm from class Hi
Hi, I'm from class Hi
...
Hi, I'm from class Hi
Hello, I'm from class Hello
Hello, I'm from class Hello
...
Hello, I'm from class Hello
```

Explanation

In the above example:

- Class Each class, Hi and Hello, has a show() method that prints a message ten times.
- In the Demo class, we create objects of both classes and call their show() methods sequentially.
- As a result, the messages from **Class Hi** are printed first, followed by the messages from **Class Hello**.

Observation:
Both classes are executing their tasks sequentially, one after the other. If we want them to run simultaneously and in parallel, we need to use ***threads.***

Implementing Multiple Threads

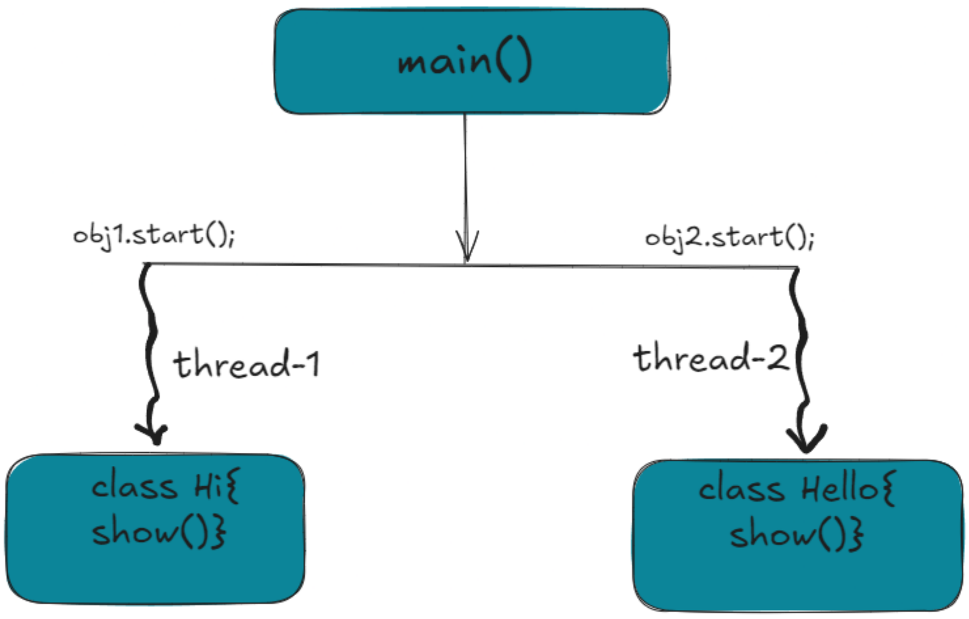
We can modify the above example to run the tasks concurrently using threads. Let's see how we can achieve this.

Example with Threads

```
class Hi extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hi, I'm from class Hi");
        }
    }
}

class Hello extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Hello, I'm from class Hello");
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        Hi obj1 = new Hi();
        Hello obj2 = new Hello ();

        obj1.start(); // Start thread for class Hi
        obj2.start(); // Start thread for class Hello
    }
}
```

Explanation:

1. **Extending the Thread Class**:
   - Instead of creating a regular class, we extend the Thread class to make Class A and Class B into threads.
2. **Implementing the run() method**:
   - Each class overrides the run() method to specify the task that each thread should complete.
3. **Starting the Threads**:
   - Instead of calling the show() method directly, we invoke the start() method on both thread objects (obj1.start() and obj2.start()).
   - Internally, the start() method calls the run() method, initiating the thread's execution.



Output with Threads:

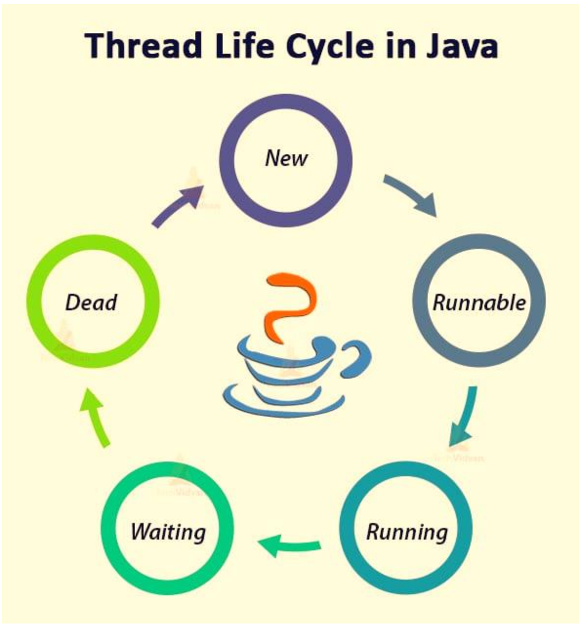
***With threading enabled, you may see an interleaved output like*:**

```
Hi, I'm from class Hi
Hello, I'm from class Hello
Hi, I'm from class Hi
Hello, I'm from class Hello
...
```

This shows that both threads are running concurrently.

Understanding the Role of the Thread Scheduler

The Thread Scheduler, a component operating in the background, determines the execution order of threads. Here's how it works:



- The **Thread Scheduler** is responsible for deciding which thread should run and when.
- If multiple threads are in the ***runnable*** state, the scheduler picks one based on various factors, such as priority and time-sharing principles.
- On a dual-core CPU, two threads can execute simultaneously. If there are

more threads than cores, the scheduler uses a time-sharing mechanism to allocate execution time for each thread.

Real-World Scenario: Multi-Core Systems

Modern computers often have multiple cores (e.g., dual-core, quad-core, or even 16-core CPUs), which allows them to run several threads in parallel. For example:

- Two threads will run simultaneously if a dual-core CPU has six threads to execute.
- Due to time-sharing, the threads will switch between cores, ensuring that each thread gets a chance to execute without remaining idle.

Advantages of Java Multithreading

1. **Improved Performance**:
   Multiple threads allow us to perform various tasks simultaneously, improving overall application performance.
2. **Better Resource Utilization**:
   Multithreading enables better CPU utilization by allowing idle threads to take over tasks when active threads are blocked or waiting.
3. **Non-Blocking Execution**:
   If an exception occurs in one thread, it won't impact the execution of other threads, ensuring a smooth and non-blocking user experience.

Conclusion

Multithreading in Java is a powerful feature that allows developers to create responsive and efficient applications. While creating and managing threads manually is useful for understanding the basics, advanced applications often rely on frameworks and tools that handle thread management automatically. By leveraging multithreading, we can ensure our programs make the most out of modern multi-core systems.