12.3 Thread Priority and Sleep in Java

Understanding the concepts of thread priority and the sleep mechanism is essential for managing how threads execute concurrently in Java. This section covers how to assign priorities to threads and how to control their execution using the sleep() method.

---

1. Introduction to Thread Priority

When Java assigns tasks to multiple threads, they often need to work simultaneously in parallel. However, the thread scheduler determines the order in which threads execute, sometimes leading to unexpected results.

Let's consider the following example:

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

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

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

        obj1.start();
        obj2.start();
    }
}
```

Output:

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

Explanation:

Here, the output shows that the obj1 thread (class A) executes first, followed by the obj2 thread (class B). The thread scheduler prioritized obj1 to finish its execution before allowing obj2 to run.However, this behavior is not guaranteed and may vary across runs

Thread priorities are only a hint to the thread scheduler. On some systems, priorities may not be strictly followed, especially in multi-threaded environments where multiple factors influence scheduling.

Example: On most modern operating systems (e.g., Windows or Linux), the JVM relies on the OS thread scheduler, which may not strictly adhere to Java thread priorities.

Best Practices you can follow:

Program logic shouldn't rely too much on thread priorities because they aren't cross-platform compatible.

For effective thread management, thread synchronization and other concurrency controls (such as locks or ExecutorService) are typically recommended.

---

2. Using Thread Priority to Control Execution

If you want to control the order of execution between threads, you can assign priorities to each thread. By default, every thread in Java has a priority of 5. The priority range is from **1** (lowest) to **10** (highest).

- **Default Priority**: 5
- **Lowest Priority**: 1

- **Highest Priority**: 10

You can get the current priority of a thread using the getPriority() method:

```
System.out.println(obj1.getPriority() + " " +
obj2.getPriority());
```

This will display the default priority for both obj1 and obj2:

```
5 5
```

Example: Setting Thread Priority

Let's see an example where we manually set the priorities for the threads:

```
public class Demo {
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        // Set thread priorities
        obj1.setPriority(1);  // Lowest priority
        obj2.setPriority(10); // Highest priority

        obj1.start();
        obj2.start();
    }
}
```

Output:

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

Explanation:

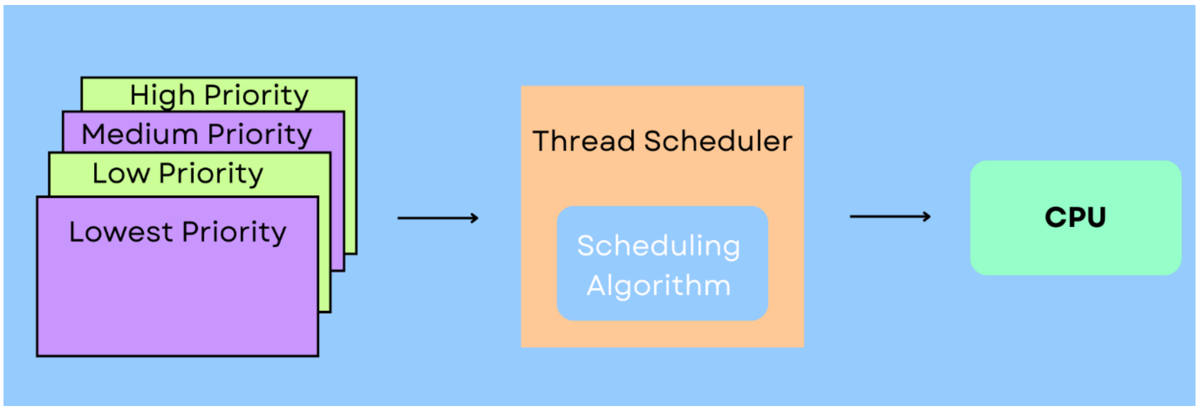
By assigning obj1 a priority of 1 (lowest) and obj2 a priority of 10 (highest), we give obj2 a better chance to execute first. However, the thread scheduler's behavior is platform-dependent, and output may not be guaranteed.

---

3. Thread Priority Constants

Instead of using numeric values for priorities, you can use the predefined constants in the Thread class:

- **Thread.MIN_PRIORITY**: 1
- **Thread.NORM_PRIORITY**: 5
- **Thread.MAX_PRIORITY**: 10



Example: Using Thread Constants

```
public class Demo {
    public static void main(String[] args) {
        A obj1 = new A();
        B obj2 = new B();

        // Using Thread priority constants
        obj1.setPriority(Thread.MIN_PRIORITY);   // 1
        obj2.setPriority(Thread.MAX_PRIORITY);   // 10

        obj1.start();
        obj2.start();
    }
}
```

4. Making Threads Sleep

The sleep() method in Java is used to pause the execution of a thread for a specified period. This can help manage thread execution and achieve consistent results.

Syntax:

```
Thread.sleep(milliseconds);
```

- **milliseconds**: The duration in milliseconds for which the current thread will sleep.

Important Points:

1. **Throws InterruptedException**: The sleep() method throws an InterruptedException, which is a checked exception that must be handled using a try-catch block.
2. **Pauses Current Thread**: As Thread.sleep() is a static method, it affects only the current executing thread by pausing the current thread without releasing any locks it holds, meaning other threads waiting for the same lock cannot proceed until the sleeping thread completes or exits the synchronized block.

Example: Using sleep() Method

```
class A extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hi, I'm from class A");

            // Make the thread sleep for 1 second
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted: " +
e);
            }
        }
    }
}

class B extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hello, I'm from class B");
            }
        }
}

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

        obj2.start();
        obj1.start();
    }
}
```

Output:

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

Explanation:

Here, obj1 prints "Hi,I'm **from** class A " and then sleeps for 1 second before printing again. Meanwhile, obj2 can execute without any delay, leading to interleaved outputs.