Multi-Threading in Java

Multithreading is a process of executing multiple threads simultaneously. Threads are independent, concurrent units of execution within a program. Java provides built-in support for multithreaded programming, enabling the efficient execution of tasks in parallel.

1. What is a Thread?

A thread in Java is a lightweight process that runs within a program. It represents a single flow of control within a program and runs concurrently with other threads. Each thread shares the process's resources, such as memory and file descriptors.

In Java, threads are represented by the java.lang.Thread class.

Key Terminology:

Thread: A single, sequential flow of control within a program.

Multithreading: The ability to run multiple threads simultaneously to execute tasks concurrently.

Main Thread: Every Java program starts with a single thread, known as the main thread, which is created by the JVM.

2. Usage of Threads

Threads are used to:

Perform Multiple Tasks Concurrently: For example, downloading files, user interaction, and real-time data updates can happen simultaneously in a program.

Improve Performance: By utilizing multi-core processors, multiple tasks can be executed in parallel, improving efficiency.

Keep User Interfaces Responsive: Threads can be used to handle long-running tasks like file downloads or database operations without freezing the user interface.

3. Creating Threads in Java

There are two primary ways to create threads in Java:

a. Extending the Thread Class

A thread can be created by extending the Thread class and overriding its run() method. The run() method contains the code that will be executed by the thread.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();  // Creating a new thread
        t1.start();  // Starting the thread
    }
}
```

In the example, t1.start() starts the execution of the thread, calling the run() method internally.

b. Implementing the Runnable Interface

Another way to create a thread is by implementing the Runnable interface. This is the preferred way if the class already extends another class because Java does not support multiple inheritance.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread t1 = new Thread(myRunnable);  // Creating a thread object
        t1.start();  // Starting the thread
    }
}
```

4. Types of Threads

There are two types of threads in Java:

a. User Threads

User threads are high-priority threads that are meant to complete their task before termination.

The JVM keeps running as long as any user threads are running. When the last user thread finishes, the JVM exits.

b. Daemon Threads

Daemon threads are low-priority threads that provide background services like garbage collection.

The JVM terminates when only daemon threads are left running.

Example of Daemon Thread:

```
class MyDaemonThread extends Thread {
    public void run() {
        if (Thread.currentThread().isDaemon()) {
            System.out.println("Daemon thread is running.");
        } else {
            System.out.println("User thread is running.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyDaemonThread t1 = new MyDaemonThread();
        MyDaemonThread t2 = new MyDaemonThread();

        t1.setDaemon(true);  // Setting thread t1 as daemon

        t1.start();
        t2.start();
    }
}
```

5. Thread Lifecycle

A thread can be in one of the following states:

New: A thread has been created but not yet started.

Runnable: A thread is ready to run or is currently running.

Blocked/Waiting: A thread is blocked, waiting for a resource or another thread to finish its task.

Timed Waiting: A thread is waiting for a specified time before it becomes ready again.

Terminated: A thread has completed execution.

6. Handling Threads in Java

a. Starting and Stopping Threads

The start() method is used to start a thread. This method internally calls the run() method.

A thread stops when the run() method completes its execution, or when it is interrupted using the interrupt() method.

b. Thread Sleep

The Thread.sleep(milliseconds) method is used to pause a thread's execution for a specified duration.

Example:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(1000);  // Pauses for 1 second
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

c. Thread Join

The join() method allows one thread to wait for the completion of another. If a thread calls t1.join(), the calling thread will wait for thread t1 to finish.

Example:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();

        try {
            t1.join();  // Main thread waits for t1 to complete
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Thread t1 has finished execution.");
    }
}
```

7. Synchronization in Threads

Synchronization in Java ensures that multiple threads do not interfere with each other when accessing shared resources. Without synchronization, a situation known as race condition may occur, where multiple threads modify shared resources in an unpredictable order.

Synchronized Methods

A method can be marked as synchronized to ensure that only one thread can execute it at a time.

Example:

java

Copy code

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

class MyThread extends Thread {
    Counter counter;
    MyThread(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());  // Expected 2000
    }
}
```

Synchronized Block

A synchronized block can also be used to synchronize only a specific part of a method, reducing the overhead of synchronization.

Example:

```
class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {
            count++;
        }
    }

    public int getCount() {
        return count;
    }
}
```

8. Inter-Thread Communication

Java provides a mechanism called inter-thread communication, where threads can communicate with each other. The key methods for inter-thread communication are:

wait(): Causes the current thread to wait until another thread calls notify() or notifyAll().

notify(): Wakes up a single thread that is waiting on the object's monitor.

notifyAll(): Wakes up all threads waiting on the object's monitor.

Example:

```
class Shared {
```

```
int value;
boolean flag = false;

public synchronized void put(int value) {
    while (flag) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
    this.value = value;
    flag = true;
    System.out.println("Produced: " + value);
    notify();
}

public synchronized void get() {
    while (!flag) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
    System.out.println("Consumed: " + value);
    flag = false;
    notify();
}
```

```
}

class Producer extends Thread {
    Shared shared;

    Producer(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            shared.put(i);
        }
    }
}

class Consumer extends Thread {
    Shared shared;

    Consumer(Shared shared) {
        this.shared = shared;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            shared.get();
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Shared shared = new Shared();


        Producer producer = new Producer(shared);
        Consumer consumer = new Consumer(shared);


        producer.start();
        consumer.start();
    }
}
```

Summary of Multi-Threading in Java

Thread: Represents a separate path of execution.

Usage: Improves performance, concurrency, and user interface responsiveness.

Types: User threads and daemon threads.

Lifecycle: New, Runnable, Blocked, Timed Waiting, and Terminated.

Synchronization: Prevents race conditions when multiple threads access shared resources.

Inter-Thread Communication: Methods like wait(), notify(), and notifyAll() help manage coordination between threads.