Java Multi-Threading Guide

Mastering Multi-Threading in Java: A Deep Dive

Multi-threading allows concurrent execution of multiple tasks for better performance.

1. What is Multi-Threading?

Multi-threading is a process of executing multiple threads simultaneously.

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

2. Thread Lifecycle in Java

A thread in Java has different states:

- **New**
- **Runnable**
- **Running**
- **Blocked**

- Waiting/Timed Waiting
- Terminated

3. Ways to Create Threads in Java

- Extending Thread class
- Implementing Runnable interface

Example:

```
class MyTask implements Runnable {
    public void run() {
        System.out.println("Task running in a thread...");
    }
}
public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyTask());
        t1.start();
    }
}
```

4. Thread Synchronization - Avoiding Race Conditions

When multiple threads access shared resources, race conditions occur.

Example using synchronized:

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

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

```
    }

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

5. Thread Communication (wait, notify, notifyAll)
Java provides inter-thread communication:

```
class SharedData {
    private boolean ready = false;

    public synchronized void produce() {
        try {
            Thread.sleep(1000);
            ready = true;
            notify();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public synchronized void consume() {
        while (!ready) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Consumed!");
```

```
    }
}
```

6. Thread Pools (Executor Framework)

Instead of creating new threads manually, Javas ExecutorService manages a pool of threads.

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

public class ThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 1; i <= 5; i++) {
            executor.execute(() -> {
                                System.out.println("Executing task by " +
Thread.currentThread().getName());
            });
        }
        executor.shutdown();
    }
}
```

7. Concurrency Utilities (java.util.concurrent)

- CountDownLatch
- CyclicBarrier
- Semaphore
- Concurrent Collections

Example using CountDownLatch:

```
import java.util.concurrent.CountDownLatch;
```

```
public class CountDownLatchExample {
    public static void main(String[] args) {
        CountDownLatch latch = new CountDownLatch(3);
        for (int i = 1; i <= 3; i++) {
            new Thread(() -> {
                    System.out.println(Thread.currentThread().getName() + "
finished task.");
                latch.countDown();
            }).start();
        }
        try {
            latch.await();
            System.out.println("All tasks are completed!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

8. Best Practices for Multi-Threading

- Use thread-safe collections (ConcurrentHashMap, BlockingQueue)

- Prefer ExecutorService over manual thread management

- Handle thread interruptions properly

Conclusion:

Mastering multi-threading is essential for building efficient, scalable applications.