Java Multithreading

Let's dive into Java multithreading and how to use the `Executor` framework effectively. Multithreading allows your program to run multiple tasks concurrently, improving performance for I/O-bound or CPU-intensive operations. The `Executor` framework simplifies thread management by abstracting away low-level thread creation and handling. I'll walk you through the basics, then show you how to use `Executor` and related tools.

---

1. Basics of Multithreading in Java

Java provides two primary ways to create threads: - **Extend `Thread` class**:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
}

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

- **Implement `Runnable` interface**:

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

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

- Call `start()` (not `run()`) to launch the thread—`run()` just executes the code in the current thread.

---

2. Why Use `Executor`?

Manually creating threads works for simple cases, but it's inefficient for managing many tasks (e.g., thread overhead, resource exhaustion). The `Executor` framework (introduced in Java 5 under `java.util.concurrent`) provides a thread pool and task management system, making multithreading cleaner and more scalable.

---

3. Using `ExecutorService`

The `ExecutorService` interface (a subinterface of `Executor`) is the go-to tool. Here's how to use it:

Step 1: Create an ExecutorService Use the `Executors` utility class to create a thread pool:

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

public class Main {
    public static void main(String[] args) {
        // Fixed thread pool with 4 threads
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Submit tasks
        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                System.out.println("Task executed by: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simulate work
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        // Shut down the executor
        executor.shutdown(); // Prevents new tasks, waits for existing ones to finish
```

```
    }
}
```

- `newFixedThreadPool(4)` creates a pool with 4 threads. Excess tasks wait in a queue.
- `submit()` accepts `Runnable` or `Callable` tasks (`Callable` returns a result).

Common Executor Types

- `Executors.newSingleThreadExecutor()`: One thread, processes tasks sequentially.
- `Executors.newCachedThreadPool()`: Creates threads as needed, reuses idle ones (good for short-lived tasks).
- `Executors.newScheduledThreadPool(n)`: For scheduling tasks with delays or periodicity.

---

4. Handling Results with `Callable` and `Future`

If you need task results, use `Callable` instead of `Runnable`:

```java
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Submit a Callable task
        Future<Integer> future = executor.submit(() -> {
            Thread.sleep(1000);
            return 42;
        });

        // Do other work while task runs
        System.out.println("Task submitted...");

        // Get result (blocks until done)
        Integer result = future.get();
        System.out.println("Result: " + result);

        executor.shutdown();
    }
}
```

- `Future.get()` retrieves the result, blocking if the task isn't finished.
- Use `future.isDone()` to check completion without blocking.

---

5. Advanced: Thread Pool Tuning

For real-world apps, you might want a custom `ThreadPoolExecutor`:

```
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        // Custom thread pool: 2-4 threads, queue capacity of 10
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            2, // Core pool size
            4, // Max pool size
            60L, // Idle thread timeout
            TimeUnit.SECONDS,
            new LinkedBlockingQueue<>(10) // Task queue
        );

        for (int i = 0; i < 15; i++) {
            executor.execute(() -> {
                System.out.println("Task by: " + Thread.currentThread().getName());
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        executor.shutdown();
    }
}
```

- Core size (2): Minimum threads kept alive.
- Max size (4): Max threads if queue is full.
- Queue: Holds tasks when all threads are busy.

---

6. Best Practices

- **Shutdown Properly**: Always call `shutdown()` or `shutdownNow()` to avoid resource leaks.
- **Handle Interruptions**: Check `Thread.currentThread().isInterrupted()` in loops.
- **Avoid Overloading**: Too many threads can degrade performance—tune pool size based on CPU cores (e.g., `Runtime.getRuntime().availableProcessors()`).
- **Use `try-finally` with `ExecutorService`**:

```
ExecutorService executor = Executors.newFixedThreadPool(4);
try {
    // Submit tasks
} finally {
    executor.shutdown();
}
```

Example with Log4j (Tying It Together)

Combining multithreading with your earlier Log4j question:

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import java.util.concurrent.*;

public class Main {
    private static final Logger logger = LogManager.getLogger(Main.class);

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            final int taskId = i;
            executor.submit(() -> {
                logger.info("Task {} started by {}", taskId, Thread.currentThread().getName());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error("Task {} interrupted", taskId, e);
                    Thread.currentThread().interrupt();
                }
```

```
                logger.info("Task {} completed", taskId);
            });
        }

        executor.shutdown();
    }
}
```

With the `log4j2.xml` from earlier, this logs task progress to console and file.