Multithreading in Java

Multithreading allows a program to execute multiple threads simultaneously, improving performance and resource utilization. This lab will explore creating and managing threads, understanding their lifecycle, implementing synchronization, and using thread pools for efficient task management.

Creation Of Threads

In Java, threads can be created by implementing the `Runnable` interface or by extending the Thread class.

Using Runnable Interface:

```
public class RunnableExample implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " -
Count: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        RunnableExample task = new RunnableExample();
        Thread thread1 = new Thread(task, "Thread-1");
        Thread thread2 = new Thread(task, "Thread-2");
        thread1.start();
        thread2.start();
    }
}
```

Using Thread Class:

```java
public class ThreadExample extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - 
Count: " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        ThreadExample thread1 = new ThreadExample();
        ThreadExample thread2 = new ThreadExample();
        thread1.start();
        thread2.start();
    }
}
```

- Runnable: Implement the Runnable interface and override the run method. Create a Thread object with the Runnable instance and start it.
- Thread: Extend the Thread class and override the run method directly. Create an instance of the subclass and start it.

Thread Life Cycle

Threads in Java go through various states during their lifecycle: New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated. Below is an example demonstrating some of these states:

```
public class ThreadLifeCycleExample {
    public static void main(String[] args) throws
InterruptedException {
        Thread thread = new Thread(() -> {
            try {
                Thread.sleep(1000);
                System.out.println("Running");
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        System.out.println("Thread state after creation: " +
thread.getState()); // NEW
        thread.start();
        System.out.println("Thread state after start: " +
thread.getState()); // RUNNABLE

        thread.join(); // Wait for thread to complete
        System.out.println("Thread state after completion: " +
thread.getState()); // TERMINATED
    }
}
```

- NEW: The thread has been created but not yet started.
- RUNNABLE: The thread is running or ready to run.
- TERMINATED: The thread has finished execution.

. Thread Synchronization

To avoid conflicts when multiple threads access shared resources, synchronization is used. Below is an example using synchronized blocks and methods:

```
public class ThreadSynchronizationExample {
```

```java
    private int count = 0;

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

    public void decrement() {
        synchronized (this) {
            count--;
        }
    }

    public static void main(String[] args) {
        ThreadSynchronizationExample example = new
ThreadSynchronizationExample();

        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                example.increment();
            }
        };

        Runnable decrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                example.decrement();
            }
        };

        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(decrementTask);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final count: " + example.count);
    }
}
```

- Synchronized Methods: increment() is synchronized to prevent concurrent access to the `count` variable.
- Synchronized Blocks: decrement() uses a synchronized block to achieve the same effect, allowing fine-grained control over synchronization.

Thread Pools

Thread pools manage a pool of worker threads to execute tasks efficiently, reusing threads instead of creating new ones for each task.

ExecutorService in code:

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

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

        Runnable task = () -> {
            System.out.println(Thread.currentThread().getName() + "
is executing task.");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        for (int i = 0; i < 6; i++) {
            executor.submit(task);
        }

        executor.shutdown();
    }
}
```

- Fixed Thread Pool: Created with Executors.newFixedThreadPool(3) to handle a fixed number of threads. Tasks are submitted to the pool, and the pool manages their execution.
- Shutdown: The shutdown method is called to stop accepting new tasks and to gracefully terminate existing ones.

1. Thread Creation and Management
   - Code demonstrating thread creation using Runnable and Thread` classes.
   - Examples of thread lifecycle states and synchronization.

2. Thread Pool Implementation:
   - A thread pool example using `ExecutorService` for efficient task management.