Lab Assignment: Multithreading in Java

Objective:

Understand and implement the basics of multithreading in Java, including thread creation, synchronization, and inter-thread communication.

Requirements:

1. Create a class that extends `Thread` to simulate a countdown timer.
2. Create a class that implements `Runnable` to simulate printing numbers in a sequence.
3. Demonstrate synchronization using a shared counter, ensuring thread-safe operations when multiple threads update the counter.
4. Use the `wait()` and `notify()` methods for inter-thread communication.

Code Starter:

```
// CountdownTimer.java
public class CountdownTimer extends Thread {
    private int start;

    public CountdownTimer(int start) {
        this.start = start;
    }

    @Override
    public void run() {
        for (int i = start; i >= 0; i--) {
            System.out.println("Countdown: " + i);
            try {
                Thread.sleep(1000); // Pause for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// NumberPrinter.java
public class NumberPrinter implements Runnable {
    private int limit;

    public NumberPrinter(int limit) {
        this.limit = limit;
    }

    @Override
```

```
    public void run() {
        for (int i = 1; i <= limit; i++) {
            System.out.println("Number: " + i);
            try {
                Thread.sleep(500); // Pause for 0.5 seconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// SharedCounter.java
public class SharedCounter {
    private int count = 0;

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

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

Exercises:

1. Modify the `CountdownTimer` to accept a message that gets printed alongside the countdown.
2. Create a thread-safe `Queue` class and implement `enqueue()` and `dequeue()` methods using synchronization.
3. Use `ExecutorService` to manage a fixed pool of threads for executing `Runnable` tasks.
4. Create a multithreaded application where one thread generates random numbers and another thread calculates their sum.

Bonus Tasks:

1. Implement a `Producer-Consumer` problem using `wait()` and `notify()`.
2. Demonstrate `ReentrantLock` for thread synchronization instead of the `synchronized` keyword.
3. Use `Callable` and `Future` to execute a task that returns a result, such as calculating factorial.
4. Visualize thread states (`NEW`, `RUNNABLE`, `BLOCKED`, etc.) by printing thread states during execution.

Submission:
Submit the following:

- Java source code files.
- Screenshots showing the program execution with multiple threads.
- A short document explaining the synchronization and inter-thread communication used.