Logical follies committed in multithreaded code, while trying to avoid race conditions and guarding critical sections, can lead to a host of subtle and hard to find bugs and side-effects. Some of these incorrect usage patterns have their names and are discussed below.

DeadLock

Deadlocks occur when two or more threads aren't able to make any progress because the resource required by the first thread is held by the second and the resource required by the second thread is held by the first.

Liveness

Ability of a program or an application to execute in a timely manner is called liveness. If a program experiences a deadlock then it's not exhibiting liveness.

Live-Lock

A live-lock occurs when two threads continuously react in response to the actions by the other thread without making any real progress. The best analogy is to think of two persons trying to cross each other in a hallway. John moves to the left to let Arun pass, and Arun moves to his right to let John pass. Both block each other now. John sees he's blocking Arun again and moves to his right and Arun moves to his left seeing he's blocking John. They never cross each other and keep blocking each other. This scenario is an example of a livelock. A process seems to be running and not deadlocked but in reality, isn't making any progress.

Starvation

Other than a deadlock, an application thread can also experience starvation, when it never gets CPU time or access to shared resources. Other **greedy** threads continuously hog shared system resources not letting the starving thread make any progress.

Deadlock Example

```
void increment(){

  acquire MUTEX_A
  acquire MUTEX_B
    // do work here
  release MUTEX_B
  release MUTEX_A

}


void decrement(){

  acquire MUTEX_B
  acquire MUTEX_A
    // do work here
  release MUTEX_A
  release MUTEX_B

}
```

The above code can potentially result in a deadlock. Note that deadlock may not always happen, but for certain execution sequences, deadlock can occur. Consider the below execution sequence that ends up in a deadlock:

```
T1 enters function increment

T1 acquires MUTEX_A

T1 gets context switched by the operating system

T2 enters function decrement

T2 acquires MUTEX_B

both threads are blocked now
```

Thread **T2** can't make progress as it requires **MUTEX_A** which is being held by **T1**. Now when **T1** wakes up, it can't make progress as it requires **MUTEX_B** and that is being held up by **T2**. This is a classic text-book example of a deadlock.

You can come back to the examples presented below as they require an understanding of the `synchronized` keyword that we cover in later sections. Or you can just run the examples and observe the output for now to get a high-level overview of the concepts we discussed in this lesson.

If you run the code snippet below, you'll see that the statements for acquiring locks: **lock1** and **lock2** print out but there's no progress after that and the execution times out. In this scenario, the deadlock occurs because the locks are being acquired in a nested fashion.

```java
class Demonstration {

    public static void main(String args[]) {
        Deadlock deadlock = new Deadlock();
        try {
            deadlock.runTest();
        } catch (InterruptedException ie) {
        }
    }
}

class Deadlock {

    private int counter = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    Runnable incrementer = new Runnable() {

        @Override
        public void run() {
            try {
                for (int i = 0; i < 100; i++) {
                    incrementCounter();
                    System.out.println("Incrementing " + i);
                }
            } catch (InterruptedException ie) {
            }
```

Run Save Reset

Example of a Deadlock

Reentrant Lock

Re-entrant locks allow for re-locking or re-entering of a synchronization lock. This can be best explained with an example. Consider the NonReentrant class below.

Take a minute to read the code and assure yourself that any object of this class if locked twice in succession would result in a deadlock. The same thread gets blocked on itself, and the program is unable to make any further progress. If you click run, the execution would time-out.

If a synchronization primitive doesn't allow reacquisition of itself by a thread that has already acquired it, then such a thread would block as soon as it attempts to reacquire the primitive a second time.

```java
class Demonstration {

    public static void main(String args[]) throws Exception {
        NonReentrantLock nreLock = new NonReentrantLock();

        // First locking would be successful
        nreLock.lock();
        System.out.println("Acquired first lock");

        // Second locking results in a self deadlock
        System.out.println("Trying to acquire second lock");
        nreLock.lock();
        System.out.println("Acquired second lock");
    }
}

class NonReentrantLock {

    boolean isLocked;

    public NonReentrantLock() {
        isLocked = false;
    }

    public synchronized void lock() throws InterruptedException {

        while (isLocked) {
            wait();
```

Run Save Reset

Example of Deadlock with Non-Reentrant Lock

The statement `"Acquired second lock"` is never printed