SYNCHRONIZATION

- A **synchronized block** limits synchronization to a certain block of code, allowing finer control over concurrency.
- A synchronized method locks the entire method, preventing multiple threads from accessing it simultaneously.

Example:

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

    // Synchronized method
    public synchronized void increment() {
        count++;
    }

    // Synchronized block
    public void synchronizedBlockIncrement() {
        synchronized (this) {
            count++;
        }
    }

    public int getCount() {
        return count;
    }

    public static void main(String[] args) throws
InterruptedException {
        SynchronizedExample example = new SynchronizedExample();

        Thread t1 = new Thread(example::increment);
        Thread t2 = new Thread(example::synchronizedBlockIncrement);
```

```
        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + example.getCount());
    }
}
```

- The method **increment()** is synchronized, meaning no two threads can access it at the same time.
- The **synchronizedBlockIncrement()** demonstrates synchronizing just a section of the code.

Deadlock Scenarios and Prevention

- **Deadlock** occurs when two or more threads are blocked forever, waiting for each other.
- Preventing deadlock involves avoiding cyclic dependencies or enforcing an order in resource locking.

Example of Deadlock:

```
class DeadlockExample {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() {
        synchronized (lock1) {
            System.out.println("Thread 1: Holding lock 1...");
            try { Thread.sleep(100); } catch (InterruptedException e)
{}
            synchronized (lock2) {
                System.out.println("Thread 1: Holding lock 2...");
```

```java
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            System.out.println("Thread 2: Holding lock 2...");
            try { Thread.sleep(100); } catch (InterruptedException e)
{}
            synchronized (lock1) {
                System.out.println("Thread 2: Holding lock 1...");
            }
        }
    }

    public static void main(String[] args) {
        DeadlockExample example = new DeadlockExample();
        new Thread(example::method1).start();
        new Thread(example::method2).start();
    }
}
```

Deadlock Prevention:
To avoid deadlock, ensure that locks are always acquired in the same order.

```java
public void methodSafe() {
    synchronized (lock1) {
        System.out.println("Thread: Holding lock 1...");
        synchronized (lock2) {
            System.out.println("Thread: Holding lock 2...");
        }
    }
}
```

In this case, methodSafe() avoids the deadlock by always acquiring `lock1` before `lock2`.

Locks and Condition Variables

- Locks (i.e, **ReentrantLock**) provide more flexibility than `synchronized` blocks, allowing finer control over locking and unlocking.
- Condition variables (via Condition) allow threads to wait and be signaled, enabling more advanced thread coordination.

Example:

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class LockConditionExample {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private boolean ready = false;

    public void awaitSignal() throws InterruptedException {
        lock.lock();
        try {
            while (!ready) {
                System.out.println("Waiting...");
                condition.await(); // Wait for signal
            }
            System.out.println("Received signal, proceeding...");
        } finally {
            lock.unlock();
        }
    }

    public void signal() {
        lock.lock();
        try {
            ready = true;
            condition.signal(); // Send signal
        } finally {
            lock.unlock();
        }
    }
```

```
    public static void main(String[] args) throws
InterruptedException {
        LockConditionExample example = new LockConditionExample();

        new Thread(() -> {
            try {
                example.awaitSignal();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        Thread.sleep(2000); // Simulate some work
        new Thread(example::signal).start(); // Send signal
    }
```
}

In the above code thread waits on a condition until signaled by another thread, demonstrating coordination between threads using Lock and Condition.

Atomic Variables

- Atomic variables (e.g., AtomicInteger, AtomicLong) provide lock-free, thread-safe operations on single variables.

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicExample {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.getAndIncrement();
    }

    public int getCount() {
        return count.get();
```

```java
    }

    public static void main(String[] args) throws
InterruptedException {
        AtomicExample example = new AtomicExample();

        Thread t1 = new Thread(example::increment);
        Thread t2 = new Thread(example::increment);

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + example.getCount());
    }
}
```

- The AtomicInteger class is used to perform thread-safe increments without needing explicit synchronization.