In Java, a mutex, short for "mutual exclusion," is a synchronization mechanism used to protect shared resources from concurrent access by multiple threads. It ensures that only one thread can access the shared resource at a time, preventing data corruption and race conditions.

While Java provides several mechanisms for achieving mutual exclusion, the most commonly used one is the synchronized keyword, which turns any method or block of code into a critical section, allowing only one thread to execute it at a time.

Here's an example of using the synchronized keyword to implement a mutex in Java:

java

Copy code

public class MutexExample {

private int sharedResource = 0;

// A method protected by a mutex

public synchronized void modifySharedResource() {

// Only one thread can execute this method at a time

// Ensuring mutual exclusion on the shared resource

sharedResource++;

System.out.println("Modified shared resource to: " + sharedResource);

}

}

In the example above, the modifySharedResource() method is synchronized using the synchronized keyword. When a thread calls this method, it acquires the mutex associated with the instance of MutexExample (since the method is non-static). If another thread tries to call the same method while it's already being executed by a different thread, it will be blocked until the first thread releases the mutex by exiting the method.

Note that synchronized methods can also be used with static methods or blocks to protect class-level resources.

Alternatively, you can use explicit Lock implementations like ReentrantLock from the java.util.concurrent.locks package to create a mutex:

java

Copy code

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class MutexExample {

private int sharedResource = 0;

private final Lock lock = new ReentrantLock();

public void modifySharedResource() {

lock.lock();

try {

// Only one thread can execute this block at a time

// Ensuring mutual exclusion on the shared resource

sharedResource++;

System.out.println("Modified shared resource to: " + sharedResource);

} finally {

lock.unlock();

}

}

}

Using Lock provides more flexibility than synchronized and can be useful in more complex scenarios, but it also requires you to explicitly manage the lock acquisition and release. In simple cases, synchronized is often sufficient and more convenient.