1. AQS (Abstract Queue Synchronizer)

AbstractQueuedSynchronizer (AQS) is a **low-level** framework in Java concurrency

(java.util.concurrent.locks package).

It is used to build synchronization utilities like:

- ReentrantLock
- Semaphore
- CountDownLatch
- ReentrantReadWriteLock

2. Types of lock:

1. Exclusive Lock:

- Only one thread can hold at a time

ReentrantLock

```
java
import java.util.concurrent.locks.ReentrantLock;
public class ExclusiveExample {
    private static ReentrantLock lock = new ReentrantLock();
    public static void main(String[] args) {
       Runnable task = () -> {
            lock.lock(); // acquire exclusive lock
               System.out.println(Thread.currentThread().getName() + " acquired lock");
               Thread.sleep(1000); // simulate work
            } catch (InterruptedException e) {
               e.printStackTrace();
            } finally {
               System.out.println(Thread.currentThread().getName() + " released lock");
               lock.unlock(); // release lock
       };
        for (int i = 0; i < 3; i++) {
            new Thread(task).start();
}
```

ReentrantReadWriteLock

Purpose: Allow multiple threads to read concurrently, but only one thread to write.

Two locks:

- a. Read lock (readLock()): Multiple threads can hold it simultaneously.
- b. **Write lock** (writeLock()): Exclusive only one thread can hold it, blocks readers.

Reentrancy:

c. A thread holding a read or write lock can reacquire it without deadlocking itself.

Notes:

- When read can not write, when write can not write
- When read you can read

StampedLock

Limit block write when read => add validate(stamp) for optimistic read to self-validate while any threads are writing.

2. Shared Lock:

Semaphore

- Multiple threads can acquire permits simultaneously.
- The semaphore count defines the max threads allowed at the same time.

```
2 Shared Mode - Semaphore

    Multiple threads can acquire permits simultaneously.

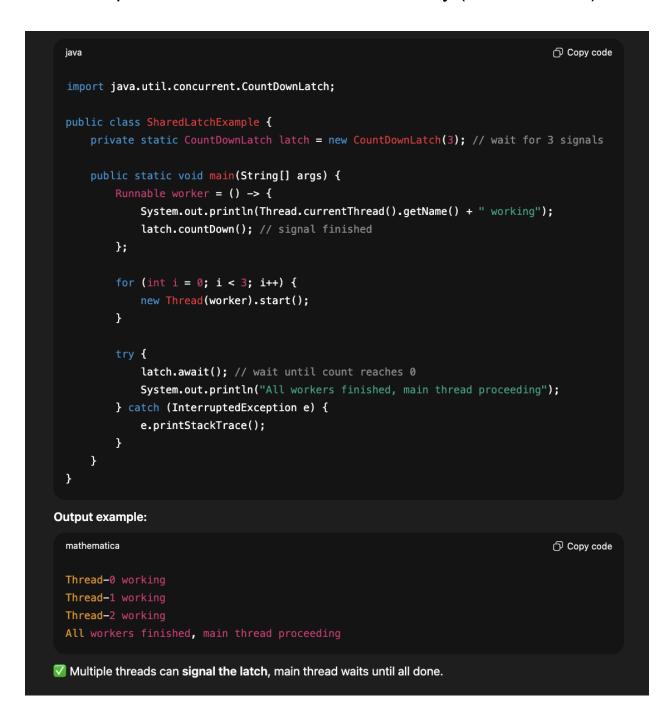
 • The semaphore count defines the max threads allowed at the same time.

    ○ Copy code

 java
  import java.util.concurrent.Semaphore;
 public class SharedSemaphoreExample {
      private static Semaphore semaphore = new Semaphore(2); // 2 permits
      public static void main(String[] args) {
          Runnable task = () -> {
              try {
                  semaphore.acquire(); // acquire a permit
                  System.out.println(Thread.currentThread().getName() + " acquired permit")
                  Thread.sleep(1000); // simulate work
              } catch (InterruptedException e) {
                  e.printStackTrace();
              } finally {
                  System.out.println(Thread.currentThread().getName() + " released permit")
                  semaphore.release(); // release permit
          };
          for (int i = 0; i < 5; i++) {
              new Thread(task).start();
      }
  }
```

CountDownLatch (Wait for 3 threads signal)

- Threads wait for a signal (latch reaches zero) before proceeding.
- Multiple threads can wait simultaneously (shared mode).

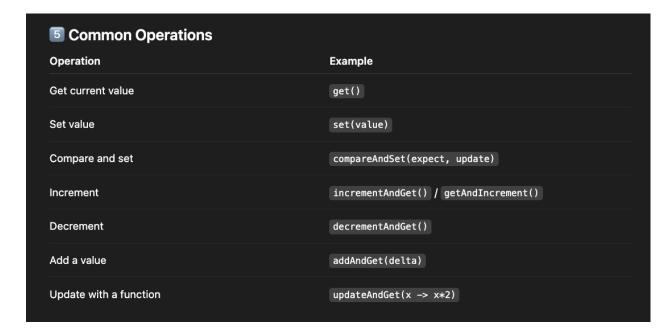


3. Atomic Class

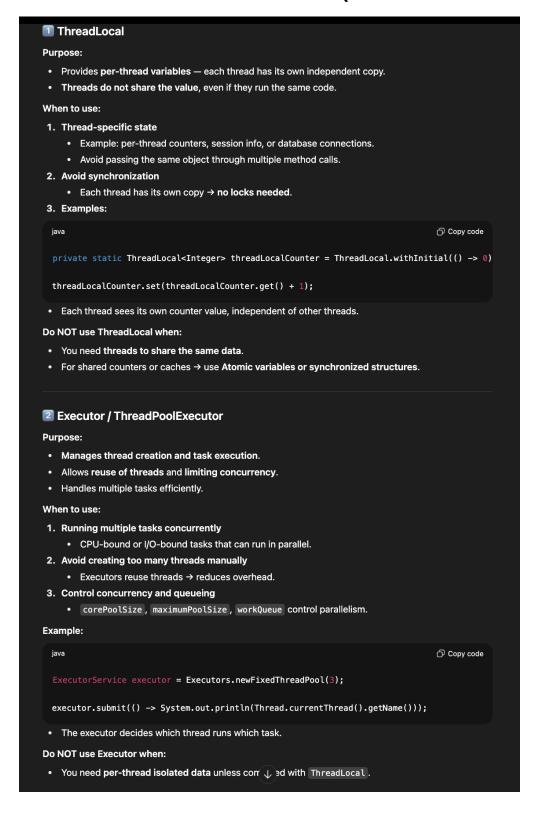
- **1. Atomic Class:** Using CAS => optimistic locking.
- **2. Lock:** Using a lock => persimistic locking.

Why we use them

- Avoid locks, so less overhead and better performance in multithreaded code.
- Ensure atomic updates even if multiple threads modify the variable concurrently.



4. When to use ThreadLocal (isolated variables), when to use Thread Pool (shared variables)



Notes:

- Thread Local do not shared variables => Do not conflict.
- RequestContext is ThreadLocal Wrapper.
- Under Java 19, ThreadLocal is OS Thread. And after Java 19, ThreadLocal is Virtual Thread.