

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 Copy code  
  
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  
            try {  
                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.

java

 Copy code

```
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).

java

 Copy code

```
import java.util.concurrent.CountDownLatch;

public class SharedLatchExample {
    private static CountDownLatch latch = new CountDownLatch(3); // wait for 3 signals


    public static void main(String[] args) {
        Runnable worker = () -> {
            System.out.println(Thread.currentThread().getName() + " working");
            latch.countDown(); // signal finished
        };

        for (int i = 0; i < 3; i++) {
            new Thread(worker).start();
        }

        try {
            latch.await(); // wait until count reaches 0
            System.out.println("All workers finished, main thread proceeding");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output example:

mathematica

 Copy code

```
Thread-0 working
Thread-1 working
Thread-2 working
All workers finished, main thread proceeding
```

✓ Multiple threads can signal the latch, main thread waits until all done.

3. Atomic Class

1. **Atomic Class:** Using CAS => optimistic locking.

2. **Lock:** Using a lock => pessimistic 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.

5 Common Operations

Operation	Example
Get current value	<code>get()</code>
Set value	<code>set(value)</code>
Compare and set	<code>compareAndSet(expect, update)</code>
Increment	<code>incrementAndGet()</code> / <code>getAndIncrement()</code>
Decrement	<code>decrementAndGet()</code>
Add a value	<code>addAndGet(delta)</code>
Update with a function	<code>updateAndGet(x -> x*2)</code>

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

1 ThreadLocal

Purpose:

- Provides **per-thread variables** — each thread has its own independent copy.
- **Threads do not share the value**, even if they run the same code.

When to use:

1. **Thread-specific state**
 - Example: per-thread counters, session info, or database connections.
 - Avoid passing the same object through multiple method calls.
2. **Avoid synchronization**
 - Each thread has its own copy → **no locks needed**.
3. **Examples:**

```
java                                                                    Copy code

private static ThreadLocal<Integer> threadLocalCounter = ThreadLocal.withInitial(() -> 0)

threadLocalCounter.set(threadLocalCounter.get() + 1);
```

- Each thread sees its own counter value, independent of other threads.

Do NOT use ThreadLocal when:

- You need **threads to share the same data**.
- For shared counters or caches → use **Atomic variables** or **synchronized structures**.

2 Executor / ThreadPoolExecutor

Purpose:

- **Manages thread creation and task execution**.
- Allows **reuse of threads** and **limiting concurrency**.
- Handles multiple tasks efficiently.

When to use:

1. **Running multiple tasks concurrently**
 - CPU-bound or I/O-bound tasks that can run in parallel.
2. **Avoid creating too many threads manually**
 - Executors reuse threads → reduces overhead.
3. **Control concurrency and queueing**
 - `corePoolSize`, `maximumPoolSize`, `workQueue` control parallelism.

Example:

```
java                                                                    Copy code

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.submit(() -> System.out.println(Thread.currentThread().getName()));
```

- The executor decides which thread runs which task.

Do NOT use Executor when:

- You need **per-thread isolated data** unless connected with `ThreadLocal`.

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.