1. CAS

Example (Java pseudo-code)

java Copy code

```java
AtomicInteger counter = new AtomicInteger(0);

void increment() {
    int oldValue, newValue;
    do {
        oldValue = counter.get();
        newValue = oldValue + 1;
        // try until success
    } while (!counter.compareAndSet(oldValue, newValue));
}
```

2. Runnable and Callable Thread

1. **Runnable:** Run the thread
2. **Callable:** return the value, usually go with Future

3. Thread Pool Executor

1. workQueue: number of jobs
2. corePoolSize: The **minimum number of threads** that the pool tries to keep alive **even if they are idle**.
3. maximumPoolSize: The **maximum number of threads** created when the length of workQueue == corePoolSize

4. Why we use ArrayBlockingQueue but not ArrayList or LinkedBlockingQueue

1. ArrayList: Insert/Detete O(N)
2. LinkedBlockingQueue: unbounded.

=> Using ArrayBlockingQueue to make sure it insert/delete O(1), FCFS and bounded elements.

5. Execute and Submit

1. Execute: run the thread function.
2. Submit: return a value when calling get().

6. LinkedBlockingQueue and SynchronousQueue

Other queues like **LinkedBlockingQueue** or **SynchronousQueue** behave differently:

- **LinkedBlockingQueue (unbounded)** → thread pool **never grows beyond corePoolSize**.
- **SynchronousQueue** → always tries to **create new threads immediately**.

7. Types of Thread Pool

1️⃣ Fixed Thread Pool

- **Created by:** `Executors.newFixedThreadPool(n)`
- **Behavior:**
  - Has a **fixed number of threads** (`n`).
  - Tasks beyond that number **wait in an unbounded queue**.
  - Threads **never die**, even if idle.
- **Use case:** Predictable number of threads, e.g., processing a fixed number of tasks concurrently.

```java
ExecutorService executor = Executors.newFixedThreadPool(3);
```

2️⃣ Cached Thread Pool

- **Created by:** `Executors.newCachedThreadPool()`
- **Behavior:**
  - **Creates new threads as needed**.
  - Threads that are **idle for 60 seconds** are terminated.
  - Good for **many short-lived tasks**.
- **Use case:** Highly dynamic workloads, tasks submitted sporadically.

```java
ExecutorService executor = Executors.newCachedThreadPool();
```

3️⃣ Single Thread Pool

- **Created by:** `Executors.newSingleThreadExecutor()`
- **Behavior:**
  - Only **one thread** in the pool.
  - Tasks are executed **sequentially** in **submission order**.
- **Use case:** Ensure **serial execution** of tasks, like logging or ordered updates.

```java
ExecutorService executor = Executors.newSingleThreadExecutor();
```

4️⃣ Scheduled Thread Pool

- **Created by:** `Executors.newScheduledThreadPool(n)`
- **Behavior:**
  - Can **schedule tasks** to run after a delay or periodically.
  - Fixed number of threads.
- **Use case:** Timers, periodic tasks, delayed execution.

```java
ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);
executor.schedule(() -> System.out.println("Delayed task"), 5, TimeUnit.SECONDS);
```

5️⃣ Work Stealing Pool

- **Created by:** `Executors.newWorkStealingPool()` (Java 8+)
- **Behavior:**
  - Uses **ForkJoinPool** internally.
  - Threads **steal tasks** from other threads' queues to balance load.
  - Ideal for **many small tasks**.
- **Use case:** Parallel algorithms, divide-and-conquer tasks.

```java
ExecutorService executor = Executors.newWorkStealingPool();
```