ChatGPT

Java Threading and Concurrency Deep Dive

Core Threading Concepts

Thread Lifecycle and States

A Java thread can be in one of six states: **NEW**, **RUNNABLE**, **BLOCKED**, **WAITING**, **TIMED_WAITING**, or **TERMINATED** [1]. These are defined by the `java.lang.Thread.State` enum. A thread is **NEW** when created but not yet started. Once you call `start()`, it transitions to **RUNNABLE** (ready or running) [1] [2]. If it tries to enter a synchronized block held by another thread, it becomes **BLOCKED** [3]. A thread enters **WAITING** when waiting indefinitely (e.g. via `Object.wait()` or `Thread.join()` without timeout) [4]. It enters **TIMED_WAITING** when waiting with a timeout (e.g. `sleep(long)`, `wait(timeout)`, or `join(timeout)`) [5]. Finally, after the `run()` method finishes, the thread is **TERMINATED** [6].

```
Thread t = new Thread(() -> { /* work */ });
System.out.println(t.getState());  // NEW
t.start();
t.join();
System.out.println(t.getState());  // TERMINATED
```

Use cases: Tracking a thread's state is mainly useful for debugging and monitoring. For example, checking if a thread is BLOCKED can diagnose deadlocks. In general, avoid basing program logic on thread state. (As one expert notes, thread state is "only of interest in debugging" and should not be used for synchronization logic.)

Pros: Understanding thread states helps in debugging concurrency issues and designing thread workflows. Automatic management (e.g. blocked queue) ensures safe transitions.
Cons: Too many threads can lead to resource exhaustion. Relying on states for control flow is error-prone. Threads hold resources (stack, memory) until terminated.
Best practices: Minimize thread lifetimes; handle interruptions; never forcibly stop threads. Don't use `Thread.getState()` as program logic – only use it for monitoring or logging purposes.

Creating Threads (Thread class, Runnable, Lambda)

Threads can be created by:

- **Extending** `Thread`: subclass `Thread` and override `run()`.
- **Implementing** `Runnable`: define `run()` in a `Runnable` and pass to a `Thread`.
- **Using Lambdas** (Java 8+): since `Runnable` is a functional interface, you can write `new Thread(() -> { /* code */ }).start();` [7].

```java
// Using Runnable
class MyTask implements Runnable {
    public void run() { System.out.println("Task executed"); }
}
Thread t1 = new Thread(new MyTask());
t1.start();

// Using lambda
Thread t2 = new Thread(() -> System.out.println("Lambda task"));
t2.start();

// (Less common) Extending Thread
class MyThread extends Thread {
    public void run() { System.out.println("Extending Thread"); }
}
Thread t3 = new MyThread();
t3.start();
```

Baeldung advises *favoring* `Runnable` *(or* `Callable`*) over extending* `Thread` [8]. Implementing `Runnable` keeps the task logic separate from thread control, allows using thread pools, and works with lambdas [8]. Extending `Thread` is heavier (you can't extend another class) and often unnecessary [8].

Use cases: Direct thread creation is useful for simple tasks or one-off threads. However, in production code you typically use an `ExecutorService` (thread pool) instead of raw threads. Lambdas simplify short tasks in modern Java.

Pros: Quick to start small threads. Runnable/lambda is concise and flexible, allowing reuse of the task object.
Cons: Manual thread creation and `start()` calls are low-level and error-prone. Extending `Thread` precludes subclassing other classes [8]. Threads are expensive to create frequently.
Best practices: Prefer using `Runnable` or `Callable` tasks submitted to an `ExecutorService`. If you must create threads, use lambdas for brevity. Always handle exceptions in `run()`. Avoid bare `new Thread()` in large applications – use thread pools for scalability.

Synchronization vs Locks

Java provides two main ways to control concurrent access to shared data: the `synchronized` keyword (intrinsic locks) and the `java.util.concurrent.locks` API (e.g. `ReentrantLock`).

```java
// Using synchronized (intrinsic lock)
synchronized(this) {
    // critical section
}

// Using ReentrantLock (explicit lock)
```

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

The `synchronized` keyword is simple: it acquires the object's monitor on entry and automatically releases it (even if exceptions occur). This makes it easy and less error-prone, and modern JVMs have highly optimized synchronized locks. However, `synchronized` offers limited features: there's no way to try for a lock without blocking or to specify fairness, and you have only one wait-set per object.

The Lock API (`java.util.concurrent.locks`) offers *more extensive control* [9]. For example, a `ReentrantLock` can be tried with timeout (`tryLock(timeout)`), or acquired interruptibly (`lockInterruptibly()`), which `synchronized` cannot do [10]. You can also create multiple `Condition` objects for more complex waiting strategies [9]. On the other hand, Locks require manual `unlock()` (or try/finally), so they are more verbose and if misused can lead to deadlocks or forgotten unlocks.

Use cases: Use `synchronized` for simple mutual exclusion (quick locks, simple blocks). Use `ReentrantLock` or other locks when you need non-blocking attempts, timeouts, or fair ordering. For example, `ReentrantReadWriteLock` (from the locks API) allows multiple readers or one writer.

Pros:
- **synchronized:** Easy to use, no explicit unlock needed, modern performance optimizations. Automatically enforces memory visibility on exit.
- **Lock API:** Flexible features (tryLock, timed waits, interruptible locks, fairness options) [10].

Cons:
- **synchronized:** Can't timeout or interrupt lock acquisition. Single implicit wait set.
- **Lock API:** More boilerplate (must unlock manually) and risk of forgetting to release. Slightly higher complexity.

Best practices: Use `synchronized` for most simple cases. If you need advanced features (timed waits, multiple conditions, or fine-grained locking), use `ReentrantLock` or other `Lock` implementations [10]. Always use `try/finally` to `unlock()`. Prefer higher-level constructs (like `ConcurrentHashMap` or `Atomic*` types) over explicit locking when possible.

Volatile vs Atomic

The `volatile` keyword and the atomic classes (in `java.util.concurrent.atomic`) both deal with shared data, but in different ways.

- **volatile**: A `volatile` variable guarantees that writes by one thread are immediately visible to others (it forces reads/writes to go to main memory) [11]. It ensures visibility and ordering (happens-

before) but does *not* make compound operations atomic. For example, `volatile int count; count++;` is *not* atomic, since it is a read-modify-write sequence [12]. You can use volatile for simple flags or status variables where you only do single reads or writes.

- **Atomic**: Classes like `AtomicInteger`, `AtomicLong`, or `AtomicReference` provide atomic operations (using compare-and-set under the hood). They ensure that compound actions (e.g. incrementing) are done atomically. For example, `AtomicInteger ai = new AtomicInteger(0); ai.incrementAndGet();` is atomic and thread-safe. These atomic classes effectively have a volatile field internally and use low-level CAS operations to update without locking. They are ideal for counters or accumulators that require thread-safe increments [13].

```
volatile boolean flag = false;
flag = true;  // other threads will see this update immediately

AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();  // atomic increment
```

Use cases: Use `volatile` for simple shared flags or configuration values that can change and must be immediately visible (like a shutdown flag). Use atomic types for counters, sequence numbers, or other cases where you need atomic read-modify-write without full synchronization [13].

Pros:
- **volatile:** Lightweight, no locking overhead, ensures visibility between threads [11].
- **Atomic:** Provides rich atomic operations (increment, add, compare-and-set, etc.) without locking, suitable for high-throughput scenarios [13] [14].

Cons:
- **volatile:** Cannot guarantee atomicity of multi-step operations. Doesn't scale to multiple related variables.
- **Atomic:** Slightly more overhead than plain volatile reads/writes. Not suitable if you need to update multiple fields atomically together (use locks or `synchronized` for that).

Best practices: Prefer `AtomicInteger`/etc for counters and numeric accumulators. Use `volatile` only for simple flags or state variables. Remember that `volatile` alone cannot replace synchronization if you perform read-modify-write – use `Atomic` or synchronized blocks instead [13] [12].

ThreadLocal

A `ThreadLocal<T>` provides per-thread storage of a variable. Each thread has its own independent copy, so no synchronization is needed [15]. This is useful for thread-specific data, such as user context or non-thread-safe objects. For example, a `ThreadLocal<SimpleDateFormat>` allows each thread to use its own formatter instance without locking. Once a thread is done, it can remove its value.

```
ThreadLocal<Integer> localCount = ThreadLocal.withInitial(() -> 0);
localCount.set(localCount.get() + 1);
```

```
int value = localCount.get();
// When finished with the thread (especially in a pool), clean up:
localCount.remove();
```

Use cases: Storing per-thread data (e.g. transaction context, user ID) without passing it through method parameters [15]. Avoiding shared mutable objects; each thread gets its own instance.

Pros: Each thread has isolated data (no locking needed) [15]. Convenient for thread-specific configurations (e.g. date formatters).
Cons: Can lead to memory leaks if used in thread pools: the thread's `ThreadLocalMap` may hold references long after the request is done. It's good practice to call `remove()` when done [16]. Also, ThreadLocal should not be over-used – it can make code harder to understand.

Best practices: Only use ThreadLocal for thread-scoped context/data. Always remove values (`threadLocal.remove()`) when a thread is returned to a pool to avoid leaks [16]. Consider using `InheritableThreadLocal` if child threads should inherit a value, but be aware of its implications.

Java Concurrency Utilities

ExecutorService (Thread Pools: Fixed, Cached, Single)

Instead of manually creating threads, Java provides `ExecutorService` and factory methods in `Executors`. Common thread pool types include:

- **FixedThreadPool**: e.g. `Executors.newFixedThreadPool(n)` creates a pool with a fixed number of threads (core=max=n), an unbounded queue, and idle threads do not time out. This limits the number of concurrent threads, queueing extra tasks [17]. It's good for controlling resource use when tasks may vary in length.
- **CachedThreadPool**: e.g. `Executors.newCachedThreadPool()` creates an unbounded pool (core=0, max=∞) with a `SynchronousQueue`. Threads are created as needed and terminated after 60s idle [18]. Best for many short-lived tasks: idle threads will be reused, and if all threads are busy, new ones are made [19] [20]. However, if tasks are long or flood in, it can create very many threads (potentially exhausting memory) [21].
- **SingleThreadExecutor**: e.g. `Executors.newSingleThreadExecutor()` creates a pool with one worker thread and an unbounded queue. Tasks are executed sequentially. It's useful when order matters or to offload work but ensure only one thread runs tasks.

```
ExecutorService fixedPool = Executors.newFixedThreadPool(4);
ExecutorService cachedPool = Executors.newCachedThreadPool();
ExecutorService singlePool = Executors.newSingleThreadExecutor();

fixedPool.submit(() -> doWork());
cachedPool.submit(() -> doShortTask());
singlePool.submit(() -> doSequentialTask());
```

Use cases:
- Fixed pool: server with limited worker threads (e.g. backing microservice calls). Helps control concurrency and avoid overload.
- Cached pool: background tasks or I/O-bound short jobs (e.g. handling bursts, async logging). Good when tasks are short and you want flexibility [20].
- Single thread: scheduling or tasks that must not overlap.

Pros: Thread pools reuse threads, reducing overhead. They provide configuration (pool size, queue) to tune performance. `ExecutorService` abstracts thread management.
Cons: Fixed pool with unbounded queue can grow memory if tasks backlog. Cached pool can create too many threads under heavy load (leading to OOM or thrashing) [21]. Single thread can become a bottleneck if many tasks.

Best practices: Always shut down executors (`shutdown()` / `shutdownNow()`). Tune pool size: for CPU-bound tasks, use ~#CPU cores; for I/O-bound, more threads may be beneficial. Monitor active thread count and queue size. For critical systems, consider a bounded queue and a custom `ThreadPoolExecutor` to control rejections. For example, you can create a custom pool with a bounded queue and abort policy to prevent resource exhaustion [22]:

```
ThreadPoolExecutor custom = new ThreadPoolExecutor(
    10, 20, 60, TimeUnit.SECONDS,
    new ArrayBlockingQueue<>(1000),
    new ThreadPoolExecutor.AbortPolicy());
```

This limits threads (max 20) and tasks in queue (1000) [22].

Callable and Future

`Callable<V>` is like `Runnable` but returns a value and can throw checked exceptions. When you submit a `Callable` to an `ExecutorService`, you get a `Future<V>` back. The `Future` represents the pending result of the computation. You can call `future.get()` to block until the result is ready, or use other methods to check status.

```
Callable<Integer> task = () -> {
    // some long-running computation
    Thread.sleep(1000);
    return 42;
};
Future<Integer> future = executorService.submit(task);
try {
    Integer result = future.get(); // blocks until done
    System.out.println("Result: " + result);
} catch (InterruptedException|ExecutionException e) {
```

```
    e.printStackTrace();
}
```

A `Future` also allows `cancel()`, `isDone()`, and `isCancelled()`. Note `get()` throws `ExecutionException` if the callable threw an exception, and `InterruptedException` if the waiting thread was interrupted [23].

Use cases: Any asynchronous computation where you need a return value (e.g. data processing, remote calls). Futures are also used in `ExecutorCompletionService` or `invokeAll()`/`invokeAny()` for batch tasks.

Pros: Decouples task submission from execution. Allows retrieving results or exceptions later [24]. Supports timeouts (`get(long, TimeUnit)`), cancellation, and status checks.
Cons: `Future.get()` blocks the calling thread, potentially wasting it. There's no callback or composability (without additional handling). Handling exceptions requires try/catch on `get()`. If you forget to call `get()`, you may never observe exceptions from the task.

Best practices: Use `Future` with a timeout if you can't block indefinitely (`get(timeout)`). Check `isDone()` or catch `CancellationException` if you cancel. Always shut down the executor to avoid hanging threads. For better composition, consider `CompletableFuture` (see below).

ScheduledExecutorService

A `ScheduledExecutorService` is an `ExecutorService` that can schedule tasks to run after a delay or periodically [25]. You obtain one via `Executors.newScheduledThreadPool(n)`.

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

// Schedule a one-shot task after 5 seconds:
scheduler.schedule(() -> System.out.println("Delayed Hello"), 5,
TimeUnit.SECONDS);

// Schedule a periodic task (start after 0s, repeat every 10s):
ScheduledFuture<?> handle = scheduler.scheduleAtFixedRate(
    () -> System.out.println("Tick..."), 0, 10, TimeUnit.SECONDS);

// Optionally cancel the periodic task later:
scheduler.schedule(() -> handle.cancel(false), 60, TimeUnit.SECONDS);
```

Periodic execution methods: `scheduleAtFixedRate()` runs tasks at a fixed rate (every *period*), whereas `scheduleWithFixedDelay()` waits a fixed delay after each execution [26]. The example above (from the JDK docs) shows beeping every 10 seconds:

```
ScheduledFuture<?> beeperHandle =
    scheduler.scheduleAtFixedRate(beeper, 10, 10, SECONDS);
// ...later...
scheduler.schedule(() -> beeperHandle.cancel(true), 60*60, SECONDS);
```

[27] [26].

Use cases: Tasks that need to run in the future or repeatedly, such as timeouts, retry logic, cache cleanup, health checks, scheduled reports, etc.

Pros: Replaces the old `java.util.Timer` with a richer API. You get `Future` handles for scheduled tasks, can cancel them, and use familiar executor patterns. Supports both one-shot and recurring tasks [25].
Cons: If a periodic task execution overruns or throws an exception, scheduling behavior can change (exceptions stop future invocations, long executions delay next run). Only use for lightweight tasks or ensure tasks complete quickly.

Best practices: Use `scheduleAtFixedRate` when you want fixed-frequency execution, or `scheduleWithFixedDelay` if you want a fixed delay between runs. Always cancel scheduled tasks if they are no longer needed. Handle exceptions within scheduled tasks to avoid silent termination.

CompletableFuture (chaining, async, exception handling)

`CompletableFuture` (Java 8+) is a powerful tool for async programming. It implements `Future` and also the `CompletionStage` interface, allowing you to chain and combine asynchronous steps [28].

You can start async tasks using `CompletableFuture.runAsync()` (for `Runnable`) or `supplyAsync()` (for `Supplier`/`Callable`-like tasks):

```
CompletableFuture<String> future =
    CompletableFuture.supplyAsync(() -> {
        // simulate long-running task
        return "Hello";
    });
```

You can then attach callbacks that process the result **asynchronously**:

```
future
  .thenApply(s -> s + " World")         // transform result
  .thenAccept(result -> System.out.println("Result: " + result))
  .exceptionally(ex -> {                // handle exceptions
      System.err.println("Error: " + ex);
      return null;
  });
```

This prints `"Hello World"` when done [29]. Other chaining methods include `thenRun()` (no input, no output) and `thenCompose()` / `thenCombine()` for combining multiple futures.

```
CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(() -> "A");
CompletableFuture<String> cf2 = cf1.thenApply(s -> s + "B");
CompletableFuture<Void> cf3 = cf2.thenAccept(s -> System.out.println(s));  //
prints "AB"
```

For error handling, use `exceptionally()` or `handle()`. For example:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> { throw
new RuntimeException("Oops"); })
    .handle((res, ex) -> (ex != null) ? "Recovered" : res);
String output = future.get(); // returns "Recovered" [30]
```

Use cases: Complex async workflows: e.g. calling multiple services in parallel and combining results, building non-blocking pipelines, handling I/O-bound tasks without blocking threads. `CompletableFuture` allows writing callback-style code in a fluent way, and better error propagation than raw `Future`.

Pros: Very flexible: you can chain operations (`thenApply`, `thenCompose`), combine multiple futures (`allOf`, `anyOf`), and handle exceptions in-line [29] [30]. Can run tasks on the common `ForkJoinPool` or a custom executor. Encourages non-blocking designs.
Cons: The API is large and can be confusing. Debugging chained futures can be tricky. By default it uses the common pool, which may need tuning. If you forget to handle exceptions, they get wrapped in `CompletionException`. Blocking on them (using `get()` or `join()`) negates some benefits and needs care.

Best practices: Use `CompletableFuture` for I/O-bound or event-driven tasks. Supply a custom executor if you need dedicated threads. Always handle exceptions with `exceptionally` or `handle`. To wait for multiple futures, use `CompletableFuture.allOf(...)`. For example, launching three async lookups and waiting:

```
CompletableFuture<User> f1 = lookupService.findUser("A");
CompletableFuture<User> f2 = lookupService.findUser("B");
CompletableFuture<User> f3 = lookupService.findUser("C");
CompletableFuture.allOf(f1, f2, f3).join(); // wait for all
```

[31]. This avoids blocking threads unnecessarily while each call runs in parallel.

ForkJoinPool and RecursiveTask

The Fork/Join framework (introduced in Java 7) is designed for **divide-and-conquer** tasks on multiple cores [32]. A `ForkJoinPool` is a special thread pool where worker threads that finish their work can "steal" tasks from others (work-stealing) [33].

To use it, you extend `RecursiveTask<V>` (for tasks that return a value) or `RecursiveAction` (void tasks) and implement `compute()`. In `compute()`, if the task is small enough, do it directly; otherwise **fork** subtasks and **join** their results. For example, to sum an array:

```
class SumTask extends RecursiveTask<Integer> {
    int[] arr; int lo, hi;
    SumTask(int[] arr, int lo, int hi) { this.arr=arr; this.lo=lo; this.hi=hi; }
    protected Integer compute() {
        if (hi - lo <= THRESHOLD) {
            int sum = 0;
            for (int i = lo; i < hi; i++) sum += arr[i];
            return sum;
        } else {
            int mid = (lo + hi) >>> 1;
            SumTask t1 = new SumTask(arr, lo, mid);
            SumTask t2 = new SumTask(arr, mid, hi);
            t1.fork();
            int right = t2.compute();
            int left = t1.join();
            return left + right;
        }
    }
}
// Running the task:
ForkJoinPool pool = ForkJoinPool.commonPool(); // or new
ForkJoinPool(parallelism)
int total = pool.invoke(new SumTask(numbers, 0, numbers.length));
```

Use cases: CPU-bound, large tasks that can be broken into independent subtasks (e.g. parallel sort, recursive processing). The Fork/Join framework tries to use all CPU cores efficiently via work-stealing [34].

Pros: Automatically scales to available cores; very efficient for well-partitioned problems. `ForkJoinPool.commonPool()` makes it easy. Provides a natural parallelism model and typically outperforms manual thread management for large recursive tasks [32].
Cons: Overhead for small tasks (if tasks are too small, the overhead of forking may outweigh benefits). Not suitable for I/O-bound tasks (the worker threads may block). More complex API and must avoid shared mutable state. Fork/Join tasks should avoid external locks (which can undercut work-stealing).

Best practices: Choose a proper **threshold** so that base tasks are sizable. Keep tasks independent to allow stealing. Use `ForkJoinPool.commonPool()` or size pool to number of cores. Don't block within tasks. For Java 8+, you can also use parallel streams (which use ForkJoin under the hood) for simpler cases.

Advanced Features

Virtual Threads (Java 21+)

Java 21 finalizes Project Loom's **virtual threads** (a preview in earlier versions). Virtual threads are lightweight threads, managed by the JVM rather than the OS [35]. You can create thousands or millions of them cheaply. They enable the classic "thread-per-request" style to scale, because a blocked virtual thread (e.g. waiting on I/O) doesn't tie up an OS thread [36].

```
// Starting a virtual thread directly:
Thread vt = Thread.startVirtualThread(() -> {
    System.out.println("Running on virtual thread: " + Thread.currentThread());
});
// Or use an executor:
try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
    executor.submit(() -> {
        // Blocking I/O is fine here
        System.out.println("Virtual thread in executor: " +
Thread.currentThread());
    }).get();
}
```

Use virtual threads for highly concurrent, I/O-bound applications (e.g. web servers, async services) so you can use simple synchronous code without running out of threads. They greatly simplify threading models by avoiding the need for reactive frameworks in many cases.

Use cases: Handling large numbers of concurrent requests or tasks without complex async code. For example, a web server using a thread-per-request model can now truly have one thread per request (using virtual threads), since they are very cheap.

Pros: Extremely lightweight (each uses very little memory), easy to create and manage many. Simplifies concurrency (just write normal blocking code) and automatically maps many virtual threads onto a small pool of carrier OS threads [35]. Improves responsiveness by not blocking scarce OS threads.

Cons: Virtual threads are still relatively new; some libraries may not fully expect them. Certain operations (like thread-local or profiling) may have nuances. They do not help much for CPU-bound tasks (though they still work, they'll compete on the same cores). There is also a small scheduler overhead (though very low).

Best practices: Use virtual threads for I/O-bound and service-style tasks. Use `Executors.newVirtualThreadPerTaskExecutor()` or `Thread.Builder` to create them. Avoid holding locks while a virtual thread is blocked, to prevent "parking" (which temporarily needs an OS thread).

Profile and monitor appropriately (new tooling support is coming). Do not replace normal threads when not needed – virtual threads are beneficial mainly for high concurrency.

Thread Pools: Design, Configuration, and Tuning

Designing thread pools requires understanding the workload and resource limits. A common heuristic (from *Java Concurrency in Practice*) for CPU-bound tasks is:

```
Number of threads ≈ Number of CPU cores
```

For I/O-bound or blocking tasks, you can use:

```
threads = cores * (1 + (wait time / service time))
```

where "wait time/service time" is the ratio of idle time (e.g. waiting for I/O) to active CPU time [37]. For example, if threads spend 50ms waiting and 5ms computing on a 2-core machine: `threads = 2 * (1 + 50/5) = 22` [37]. This formula helps tune pool sizes to keep CPUs busy but not overloaded.

Configuration tips:
- **Core vs Max:** Use `ThreadPoolExecutor` if you need different core and max sizes. For example, setting `corePoolSize=10, maxPoolSize=20, queueCapacity=1000` with `AbortPolicy` will limit threads to 20 and queue to 1000 tasks [22]. If the queue is full and you're at max threads, new tasks will be rejected (which you can catch or handle).
- **Queue type:** Bounded queues (like `ArrayBlockingQueue`) can throttle task submission; unbounded queues can lead to OOM if producers are too fast.
- **Keep-Alive:** For non-core threads, adjust `keepAliveTime` so idle threads die.
- **ThreadFactory:** Give threads meaningful names and set them daemon/non-daemon as needed.
- **Rejection policy:** Decide what happens when the pool is saturated (abort, discard oldest, run on caller thread, etc.). Often a good choice is **CallerRunsPolicy** to throttle the producer.

Monitoring: Use `ThreadPoolExecutor` methods or JMX (`getActiveCount()`, `getQueue().size()`, etc.) to monitor load. Collect metrics (throughput, latency) to adjust sizes.

Pros: Well-tuned thread pools maximize resource utilization. By capping threads and queue, you prevent resource thrashing [38]. Multiple pools can be used for different tasks (e.g. separate pools for CPU-bound work vs I/O tasks) so each is tuned properly [37].

Cons: Too large a pool wastes CPU context switches and memory. Too small delays tasks and underutilizes CPUs. Misconfiguration can cause deadlocks if threads end up waiting on tasks in the same pool.

Best practices: Use the above formula [37] as a starting point. Adjust pool sizes based on profiling under realistic load. Avoid using the default cached pool in production without bounds – in high load it can spawn unbounded threads [39]. For critical systems, prefer fixed or bounded pools and handle rejections gracefully.

Concurrency Best Practices

- **Minimize shared mutable state.** Prefer immutable objects; immutable state is inherently thread-safe [40]. If shared objects must be mutable, confine them to a single thread, or guard access via synchronization or locks.
- **Use high-level concurrency utilities.** Instead of manual `wait/notify`, use higher-level constructs (e.g. `ExecutorService`, `Semaphore`, `CountDownLatch`, `ConcurrentHashMap`). The `java.util.concurrent` package offers many thread-safe collections and tools to avoid low-level bugs. For example, use `ConcurrentHashMap` instead of `HashMap` with manual sync [41]. Use atomic classes for counters, and `BlockingQueue` for producer-consumer.
- **Keep synchronized blocks small.** Hold locks for the minimal necessary time. Avoid nested locks if possible, and follow a consistent lock ordering to prevent deadlocks.
- **Prefer `volatile` / `final` for shared variables.** Mark fields `final` if possible. Use `volatile` for simple flags. For complex operations, use atomic or synchronized.
- **Handle interrupts properly.** If a thread is blocked on IO or sleep, it should catch `InterruptedException` and restore the interrupt status (`Thread.currentThread().interrupt()`) or exit as appropriate.
- **Use thread pools, not raw threads.** Manage threads via `ExecutorService` to ensure orderly shutdown and to share threads safely. Always call `shutdown()` on executors when done.
- **Avoid busy-waiting.** Use blocking waits (locks, `wait()`, `sleep()`) or timeouts. Busy loops waste CPU.
- **Consistent state:** When sharing objects, protect them via synchronization or concurrent collections. For example, avoid `Collections.synchronizedList` if many reads are expected – use `CopyOnWriteArrayList` or `ConcurrentHashMap` for better concurrency [41].
- **Document thread-safety.** Clearly indicate which classes/methods are thread-safe or not. Limit the scope of concurrency (for example, only the service layer is concurrent).
- **Testing and debugging:** Write tests for concurrent scenarios. Use thread analysis tools (like thread dumps) to find deadlocks.

Summary: Good concurrency design avoids unnecessary locks, shares minimal mutable state, and leverages the standard library's thread-safe classes [40] [41]. Always think carefully about which threads see which data, and prefer simpler designs (immutable objects, message passing, functional transformations) when possible.

Spring Framework Integrations

@Async and TaskExecutor

Spring's `@Async` annotation enables asynchronous method execution. You annotate a `@Component` or `@Service` method with `@Async`, and Spring will run it in a separate thread, returning immediately. The method must return `void`, `Future`, or `CompletableFuture` (so you can get the result later) [42]. For example:

```
@Service
public class MyService {
    @Async
```

```
    public CompletableFuture<String> doAsyncWork() {
        // runs in a thread pool
        return CompletableFuture.completedFuture("Done");
    }
}
```

To use `@Async`, you add `@EnableAsync` to a configuration. Spring will by default use a `SimpleAsyncTaskExecutor` (basically unbounded threads). Typically you define your own `TaskExecutor` bean (usually a `ThreadPoolTaskExecutor`) and give it a bean name `taskExecutor` [43]. For example:

```
@Configuration
@EnableAsync
public class AsyncConfig {
    @Bean
    public Executor taskExecutor() {
        ThreadPoolTaskExecutor exec = new ThreadPoolTaskExecutor();
        exec.setCorePoolSize(5);
        exec.setMaxPoolSize(10);
        exec.setQueueCapacity(500);
        exec.initialize();
        return exec;
    }
}
```

This limits concurrent threads and queue size (similar to earlier discussion) [43].

Use cases: Offloading work from a request thread (e.g. sending emails, calling other services). Any service method that can run independently. If the method returns a `Future` / `CompletableFuture`, the caller can wait or combine results; if `void`, it just runs in background.

Pros: Easy to use via annotation. Spring manages thread creation and injection. Helps keep controllers responsive by doing work in the background.
Cons: `@Async` is proxy-based: it only works when calling the async method from outside the class (self-invocation won't use the proxy). Errors in a `void` method are logged but not thrown to the caller. You must configure an executor to avoid an unbounded thread situation [43].

Best practices: Always specify a `TaskExecutor` bean to control pool size [43]. Use `Future` / `CompletableFuture` return types if you need to track results or exceptions [42]. Annotate only non-blocking, independent operations. Remember to enable async processing with `@EnableAsync`.

AsyncRestTemplate / WebClient with Async Processing

For non-blocking HTTP calls, Spring's newer **WebClient** (part of WebFlux) is recommended. It is asynchronous and reactive, returning `Mono` or `Flux`. For example:

```
WebClient client = WebClient.builder().baseUrl("https://
api.example.com").build();
Mono<String> mono = client.get()
    .uri("/data")
    .retrieve()
    .bodyToMono(String.class);
mono.subscribe(s -> System.out.println("Got: " + s));
```

WebClient does not block the calling thread. Under the hood it dispatches events on a small pool, allowing many concurrent calls with few threads [44]. This contrasts with the old `RestTemplate` (which blocks the thread) [44]. (The deprecated `AsyncRestTemplate` used to return `ListenableFuture`, but since Spring 5 it is superseded by WebClient.)

Use cases: Calling external APIs in a non-blocking way. For example, a microservice that needs to fetch data from other services without tying up its own threads.

Pros: WebClient (Spring WebFlux) supports reactive streams and non-blocking I/O. It can handle many simultaneous calls with a small thread pool [44]. It also integrates nicely with reactive pipelines (`Mono/Flux`).

Cons: Reactive code has a learning curve. You need to use `Mono`/`Flux` and operators (like `zip` to combine calls). If your app is otherwise synchronous, mixing paradigms can be complex. In older Spring apps, you might use `AsyncRestTemplate` (but note it's deprecated).

Best practices: For new projects, use WebClient and reactive types. Use `.subscribe()`, `.block()`, or `.toFuture()` as appropriate. When using WebClient, configure timeouts and error handling. If staying in a non-reactive app, use `.block()` carefully or switch the controller to reactive.

Real-world Example of Async Service Layering

A common pattern is to call multiple services in parallel and combine their results. For example, a user info service might fetch data from GitHub and Twitter simultaneously. Using `@Async` or `CompletableFuture`, you can do:

```
CompletableFuture<User> f1 = gitHubService.findUser("alice");
CompletableFuture<User> f2 = twitterService.findUser("alice");
CompletableFuture<Void> all = CompletableFuture.allOf(f1, f2);
all.join();  // wait for both
User git = f1.get();
User tw = f2.get();
// merge results...
```

In Spring, you might have two `@Async` methods returning `CompletableFuture`, and use `allOf(...).join()` to wait [31]. This way, the two external calls happen concurrently, reducing overall

latency. After they complete, you merge or process results in the calling thread. This kind of parallel service orchestration is a real-world use of async layering.

Pros: Improves throughput and latency by doing work in parallel. Keeps the main thread free to do other work or return to the pool.
Cons: Harder to handle partial failures (one call failing and the other succeeding). You must manage exceptions and timeouts explicitly. Debugging async flows can be tricky.

Best practices: Use `CompletableFuture` chaining (`thenCombine`, `thenCompose`) or `WebClient.zip` to merge results. Always handle exceptions for each async call. Time out external calls to prevent cascading delays. For Spring `@Async`, ensure calls are made via the proxy (e.g. via another bean). Test carefully under concurrent load.

Sources: Thread state definitions [1]; Runnable vs Thread design [8]; Lock vs synchronized [9] [10]; volatile vs atomic [13] [11]; ThreadLocal usage [15] [16]; ExecutorService types and use-cases [20] [45]; Callable/Future usage [46] [23]; ScheduledExecutorService docs [25] [26]; CompletableFuture chaining and error handling [29] [30]; ForkJoin overview [32] [47]; Virtual Threads (Project Loom) [35] [36]; Thread pool sizing [37]; Spring @Async guide [42] [43]; Spring WebClient (non-blocking) [44].

---

[1] [2] [3] [4] [5] [6] Life Cycle of a Thread in Java | Baeldung
https://www.baeldung.com/java-thread-lifecycle

[7] [8] Implementing a Runnable vs Extending a Thread | Baeldung
https://www.baeldung.com/java-runnable-vs-extending-thread

[9] [10] java - Synchronization vs Lock - Stack Overflow
https://stackoverflow.com/questions/4201713/synchronization-vs-lock

[11] [12] [13] [14] Volatile vs. Atomic Variables in Java | Baeldung
https://www.baeldung.com/java-volatile-vs-atomic

[15] How and When to Use Java's ThreadLocal Object
https://smartbear.com/blog/how-and-when-to-use-javas-threadlocal-object/

[16] java - ThreadLocal & Memory Leak - Stack Overflow
https://stackoverflow.com/questions/17968803/threadlocal-memory-leak

[17] [18] [19] [20] [21] [22] [38] [45] Executors newCachedThreadPool() vs newFixedThreadPool() | Baeldung
https://www.baeldung.com/java-executors-cached-fixed-threadpool

[23] [24] [46] Guide to java.util.concurrent.Future | Baeldung
https://www.baeldung.com/java-future

[25] [26] [27] ScheduledExecutorService (Java Platform SE 8)
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html

[28] [29] [30] Guide To CompletableFuture | Baeldung
https://www.baeldung.com/java-completablefuture

[31] [42] [43] Getting Started | Creating Asynchronous Methods
https://spring.io/guides/gs/async-method/

32 33 34 47 Guide to the Fork/Join Framework in Java | Baeldung
https://www.baeldung.com/java-fork-join

35 36 JEP 444: Virtual Threads
https://openjdk.org/jeps/444

37 39 How to set an ideal thread pool size
https://engineering.zalando.com/posts/2019/04/how-to-set-an-ideal-thread-pool-size.html

40 41 Common Concurrency Pitfalls in Java | Baeldung
https://www.baeldung.com/java-common-concurrency-pitfalls

44 Spring WebClient vs. RestTemplate | Baeldung
https://www.baeldung.com/spring-webclient-resttemplate