Java Concurrency: Complex Practical Examples

Integrating Multiple Concepts for Staff-Level Interviews

1 Example 1: Thread-Safe Cache with Expiration

Requirements:

- Multiple threads read frequently (optimize for reads)
- Occasional writes to update entries
- Entries expire after TTL (time-to-live)
- Background thread periodically removes expired entries
- Thread-safe statistics tracking (hits, misses)

Concepts integrated: ReadWriteLock, ScheduledExecutorService, AtomicLong, ConcurrentHashMap

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.concurrent.atomic.*;

class ExpiringCache<K, V> {
    private static class CacheEntry<V> {
        final V value;
        final long expiryTime;

        CacheEntry(V value, long ttlMillis) {
            this.value = value;
            this.expiryTime = System.currentTimeMillis() + ttlMillis;
        }

        boolean isExpired() {
            return System.currentTimeMillis() > expiryTime;
        }
    }

    private final ConcurrentHashMap<K, CacheEntry<V>> cache;
    private final AtomicLong hits = new AtomicLong(0);
    private final AtomicLong misses = new AtomicLong(0);
    private final ScheduledExecutorService cleaner;
    private final long ttlMillis;

    public ExpiringCache(long ttlMillis, long cleanupIntervalMillis) {
        this.cache = new ConcurrentHashMap<>();
        this.ttlMillis = ttlMillis;
        this.cleaner = Executors.newScheduledThreadPool(1);

        // Schedule periodic cleanup
        cleaner.scheduleAtFixedRate(
            this::removeExpiredEntries,
            cleanupIntervalMillis,
            cleanupIntervalMillis,
            TimeUnit.MILLISECONDS
        );
    }

    public V get(K key) {
        CacheEntry<V> entry = cache.get(key);

        if (entry == null || entry.isExpired()) {
            misses.incrementAndGet();
            if (entry != null) {
                cache.remove(key, entry); // Remove expired
            }
            return null;
        }

        hits.incrementAndGet();
        return entry.value;
```

```java
    }

    public void put(K key, V value) {
        cache.put(key, new CacheEntry<>(value, ttlMillis));
    }

    private void removeExpiredEntries() {
        cache.entrySet().removeIf(entry -> entry.getValue().isExpired());
    }

    public CacheStats getStats() {
        long hitCount = hits.get();
        long missCount = misses.get();
        long total = hitCount + missCount;
        double hitRate = total == 0 ? 0.0 : (double) hitCount / total;

        return new CacheStats(hitCount, missCount, hitRate, cache.size());
    }

    public void shutdown() {
        cleaner.shutdown();
        try {
            if (!cleaner.awaitTermination(5, TimeUnit.SECONDS)) {
                cleaner.shutdownNow();
            }
        } catch (InterruptedException e) {
            cleaner.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }

    static class CacheStats {
        final long hits, misses;
        final double hitRate;
        final int size;

        CacheStats(long hits, long misses, double hitRate, int size) {
            this.hits = hits;
            this.misses = misses;
            this.hitRate = hitRate;
            this.size = size;
        }
    }
}
```

Listing 1: Thread-Safe Cache Implementation

Key design decisions:

- **ConcurrentHashMap** for cache storage - allows concurrent reads/writes to different keys
- **AtomicLong** for hit/miss counters - lock-free increments under high contention
- **ScheduledExecutorService** for cleanup - better than manual thread + Timer
- **No ReadWriteLock on cache operations** - ConcurrentHashMap already optimized
- **Graceful shutdown** with timeout and force-stop fallback
- **Benign race in get():** Between checking isExpired() and removing, another thread might access the entry. This is acceptable - worst case is redundant removal. Alternative: `cache.computeIfPresent(key, (k,v) -> v.isExpired() ? null : v)` for atomic check-and-remove

2 Example 2: Rate-Limited API Client

Requirements:

- Limit to N requests per second across all threads
- Block threads when rate limit exceeded
- Support burst capacity (can briefly exceed rate)

- Track and report throttling statistics

Concepts integrated: Semaphore, ScheduledExecutorService, AtomicInteger, synchronized blocks

```java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

class RateLimitedApiClient {
    private final Semaphore tokens;
    private final int maxTokens;
    private final int refillRate; // tokens per second
    private final ScheduledExecutorService refiller;
    private final AtomicInteger throttledRequests = new AtomicInteger(0);

    public RateLimitedApiClient(int maxTokens, int refillRate) {
        this.maxTokens = maxTokens;
        this.refillRate = refillRate;
        this.tokens = new Semaphore(maxTokens);
        this.refiller = Executors.newScheduledThreadPool(1);

        // Refill tokens every 100ms
        long refillIntervalMs = 100;
        // Calculate tokens per interval to achieve target rate
        // refillRate tokens/second = refillRate/10 tokens per 100ms
        double tokensPerInterval = refillRate / 10.0;

        refiller.scheduleAtFixedRate(
            () -> refillTokens(tokensPerInterval),
            refillIntervalMs,
            refillIntervalMs,
            TimeUnit.MILLISECONDS
        );
    }

    public <T> T makeRequest(Callable<T> apiCall)
            throws Exception {
        // Try to acquire token (blocks if none available)
        boolean acquired = tokens.tryAcquire(5, TimeUnit.SECONDS);

        if (!acquired) {
            throttledRequests.incrementAndGet();
            throw new RateLimitException("Rate limit exceeded");
        }

        try {
            return apiCall.call();
        } finally {
            // Token not returned - consumed by rate limit
        }
    }

    private void refillTokens(double tokensToAdd) {
        // Accumulate fractional tokens across refills
        int available = tokens.availablePermits();
        int wholePart = (int) tokensToAdd;

        // Never exceed maxTokens
        int toAdd = Math.min(wholePart, maxTokens - available);

        if (toAdd > 0) {
            tokens.release(toAdd);
        }
        // Note: Race between availablePermits() and release() is acceptable
        // Worst case: slightly exceed maxTokens temporarily
    }

    public RateLimitStats getStats() {
        return new RateLimitStats(
            tokens.availablePermits(),
            throttledRequests.get()
        );
```

```java
    }

    public void shutdown() {
        refiller.shutdown();
    }

    static class RateLimitStats {
        final int availableTokens;
        final int throttledRequests;

        RateLimitStats(int availableTokens, int throttledRequests) {
            this.availableTokens = availableTokens;
            this.throttledRequests = throttledRequests;
        }
    }

    static class RateLimitException extends Exception {
        RateLimitException(String message) {
            super(message);
        }
    }
}
```

Listing 2: Token Bucket Rate Limiter

Usage example:

```java
RateLimitedApiClient client = new RateLimitedApiClient(
    100,  // max 100 tokens (burst capacity)
    50    // refill 50 tokens/second
);

// Multiple threads making requests
ExecutorService executor = Executors.newFixedThreadPool(20);
for (int i = 0; i < 1000; i++) {
    executor.submit(() -> {
        try {
            String result = client.makeRequest(() -> {
                return callExternalApi();
            });
            System.out.println("Success: " + result);
        } catch (RateLimitException e) {
            System.out.println("Throttled!");
        }
    });
}
```

Key design decisions:

- **Semaphore for token bucket** - naturally models available capacity
- **tryAcquire with timeout** - prevents indefinite blocking
- **Tokens not returned** after request - consumed by rate limit
- **Periodic refill** - scheduled task adds tokens back
- **Bounded refill** - never exceed maxTokens (prevents overflow)

3 Example 3: Parallel File Processor with Work Stealing

Requirements:

- Process large directory of files in parallel
- Some files larger than others (imbalanced workload)
- Aggregate results from all files
- Track progress and report completion
- Handle errors gracefully

Concepts integrated: ExecutorService, CountDownLatch, ConcurrentHashMap, AtomicInteger, Future

```java
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.io.*;
import java.nio.file.*;

class ParallelFileProcessor {
    private final ExecutorService executor;
    private final ConcurrentHashMap<String, ProcessingResult> results;
    private final AtomicInteger processed = new AtomicInteger(0);
    private final AtomicInteger failed = new AtomicInteger(0);

    public ParallelFileProcessor(int threadCount) {
        // Work-stealing pool for imbalanced workloads
        this.executor = Executors.newWorkStealingPool(threadCount);
        this.results = new ConcurrentHashMap<>();
    }

    public AggregateResult processDirectory(Path directory)
            throws IOException, InterruptedException {
        List<Path> files = Files.walk(directory)
            .filter(Files::isRegularFile)
            .toList();

        int totalFiles = files.size();
        CountDownLatch latch = new CountDownLatch(totalFiles);

        // Submit all file processing tasks
        List<Future<ProcessingResult>> futures = new ArrayList<>();
        for (Path file : files) {
            Future<ProcessingResult> future = executor.submit(() -> {
                try {
                    ProcessingResult result = processFile(file);
                    results.put(file.toString(), result);
                    processed.incrementAndGet();
                    return result;
                } catch (Exception e) {
                    failed.incrementAndGet();
                    System.err.println("Failed: " + file + " - " + e.getMessage());
                    return ProcessingResult.error(file.toString(), e);
                } finally {
                    latch.countDown();
                }
            });
            futures.add(future);
        }

        // Progress reporting in separate thread
        AtomicBoolean progressRunning = new AtomicBoolean(true);
        Thread progressThread = new Thread(() -> {
            while (progressRunning.get() && latch.getCount() > 0) {
                reportProgress(totalFiles);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    break;
                }
            }
        });
        progressThread.start();

        // Wait for all files to complete
        latch.await();
        progressRunning.set(false);
        progressThread.interrupt();

        // Aggregate results
        return aggregateResults(files, futures);
```

```java
    }

    private ProcessingResult processFile(Path file) throws IOException {
        // Simulate file processing
        long wordCount;
        try (var lines = Files.lines(file)) {
            wordCount = lines.count();
        }
        long byteSize = Files.size(file);
        return new ProcessingResult(file.toString(), wordCount, byteSize, true, null);
    }

    private void reportProgress(int total) {
        int done = processed.get();
        int errors = failed.get();
        double pct = (done + errors) * 100.0 / total;
        System.out.printf("Progress: %.1f%% (%d/%d, %d errors)%n",
            pct, done, total, errors);
    }

    private AggregateResult aggregateResults(List<Path> files,
            List<Future<ProcessingResult>> futures) {
        long totalWords = 0;
        long totalBytes = 0;
        int successCount = 0;
        List<String> errors = new ArrayList<>();

        for (Future<ProcessingResult> future : futures) {
            try {
                ProcessingResult result = future.get();
                if (result.success) {
                    totalWords += result.wordCount;
                    totalBytes += result.byteSize;
                    successCount++;
                } else {
                    errors.add(result.filename + ": " + result.error.getMessage());
                }
            } catch (Exception e) {
                errors.add("Future failed: " + e.getMessage());
            }
        }

        return new AggregateResult(files.size(), successCount,
            totalWords, totalBytes, errors);
    }

    public void shutdown() {
        executor.shutdown();
        try {
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }
    }

    static class ProcessingResult {
        final String filename;
        final long wordCount;
        final long byteSize;
        final boolean success;
        final Exception error;

        ProcessingResult(String filename, long wordCount, long byteSize,
                boolean success, Exception error) {
            this.filename = filename;
            this.wordCount = wordCount;
            this.byteSize = byteSize;
            this.success = success;
```

```java
            this.error = error;
        }

        static ProcessingResult error(String filename, Exception error) {
            return new ProcessingResult(filename, 0, 0, false, error);
        }
    }

    static class AggregateResult {
        final int totalFiles;
        final int successCount;
        final long totalWords;
        final long totalBytes;
        final List<String> errors;

        AggregateResult(int totalFiles, int successCount, long totalWords,
                long totalBytes, List<String> errors) {
            this.totalFiles = totalFiles;
            this.successCount = successCount;
            this.totalWords = totalWords;
            this.totalBytes = totalBytes;
            this.errors = errors;
        }
    }
}
```

Listing 3: Parallel File Processor

Key design decisions:

- **newWorkStealingPool()** - threads steal work from each other when idle (good for imbalanced tasks)
- **CountDownLatch** - wait for all files to complete before aggregating
- **ConcurrentHashMap** - store results from multiple threads
- **AtomicInteger** - track progress without locks
- **Future list** - collect results for final aggregation
- **Separate progress thread** - non-blocking progress reporting
- **Graceful error handling** - failures don't stop other tasks

4 Example 4: Connection Pool

Requirements:

- Fixed pool of N database connections
- Threads block when all connections in use
- Connections have idle timeout (close if unused)
- Health check connections periodically
- Track utilization statistics

Concepts integrated: BlockingQueue, ScheduledExecutorService, ReentrantLock, wait/notify

```java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;

class ConnectionPool {
    private final BlockingQueue<PooledConnection> available;
    private final Set<PooledConnection> inUse;
    private final Lock useLock = new ReentrantLock();
    private final int maxSize;
    private final long idleTimeoutMs;
    private final ScheduledExecutorService healthChecker;
    private volatile boolean isShutdown = false;

    public ConnectionPool(int maxSize, long idleTimeoutMs) {
        this.maxSize = maxSize;
        this.idleTimeoutMs = idleTimeoutMs;
```

```java
        this.available = new LinkedBlockingQueue<>();
        this.inUse = new HashSet<>();
        this.healthChecker = Executors.newScheduledThreadPool(1);

        // Initialize pool
        for (int i = 0; i < maxSize; i++) {
            available.offer(new PooledConnection(i));
        }

        // Periodic health check and idle timeout
        healthChecker.scheduleAtFixedRate(
            this::maintainPool,
            30, 30, TimeUnit.SECONDS
        );
    }

    public PooledConnection acquire() throws InterruptedException {
        if (isShutdown) {
            throw new IllegalStateException("Pool is shutdown");
        }

        // Block until connection available
        PooledConnection conn = available.take();

        useLock.lock();
        try {
            conn.markInUse();
            inUse.add(conn);
        } finally {
            useLock.unlock();
        }

        return conn;
    }

    public PooledConnection tryAcquire(long timeout, TimeUnit unit)
            throws InterruptedException {
        if (isShutdown) {
            throw new IllegalStateException("Pool is shutdown");
        }

        PooledConnection conn = available.poll(timeout, unit);
        if (conn == null) {
            return null; // Timeout
        }

        useLock.lock();
        try {
            conn.markInUse();
            inUse.add(conn);
        } finally {
            useLock.unlock();
        }

        return conn;
    }

    public void release(PooledConnection conn) {
        if (conn == null) return;

        useLock.lock();
        try {
            if (!inUse.remove(conn)) {
                throw new IllegalStateException("Connection not in use");
            }
            conn.markAvailable();
        } finally {
            useLock.unlock();
        }

```

```
            available.offer(conn);
        }

        private void maintainPool() {
            useLock.lock();
            try {
                // Health check in-use connections
                for (PooledConnection conn : inUse) {
                    if (!conn.isHealthy()) {
                        System.err.println("Unhealthy connection: " + conn.id);
                        // In real implementation: close and recreate
                    }
                }

                // Check idle timeout for available connections
                long now = System.currentTimeMillis();
                Iterator<PooledConnection> iter = available.iterator();
                while (iter.hasNext()) {
                    PooledConnection conn = iter.next();
                    if (now - conn.lastUsedTime > idleTimeoutMs) {
                        iter.remove();
                        conn.close();
                        // Create new connection to maintain pool size
                        available.offer(new PooledConnection(conn.id));
                    }
                }
            } finally {
                useLock.unlock();
            }
        }

        public PoolStats getStats() {
            useLock.lock();
            try {
                return new PoolStats(
                    available.size(),
                    inUse.size(),
                    maxSize
                );
            } finally {
                useLock.unlock();
            }
        }

        public void shutdown() {
            isShutdown = true;
            healthChecker.shutdown();

            useLock.lock();
            try {
                for (PooledConnection conn : available) {
                    conn.close();
                }
                available.clear();
            } finally {
                useLock.unlock();
            }
        }

        static class PooledConnection {
            final int id;
            volatile long lastUsedTime;
            volatile boolean inUse;

            PooledConnection(int id) {
                this.id = id;
                this.lastUsedTime = System.currentTimeMillis();
                this.inUse = false;
            }

```

```java
        void markInUse() {
            this.inUse = true;
            this.lastUsedTime = System.currentTimeMillis();
        }

        void markAvailable() {
            this.inUse = false;
            this.lastUsedTime = System.currentTimeMillis();
        }

        boolean isHealthy() {
            // Simulate health check
            return true;
        }

        void close() {
            // Close actual connection
        }

        public void execute(String sql) {
            // Execute query
        }
    }

    static class PoolStats {
        final int available;
        final int inUse;
        final int total;

        PoolStats(int available, int inUse, int total) {
            this.available = available;
            this.inUse = inUse;
            this.total = total;
        }

        public double getUtilization() {
            return total == 0 ? 0.0 : (double) inUse / total;
        }
    }
}
```

Listing 4: Database Connection Pool

Usage pattern:

```java
ConnectionPool pool = new ConnectionPool(10, 60000); // 10 conns, 60s idle

// Thread-safe acquire/release
PooledConnection conn = null;
try {
    conn = pool.acquire(); // blocks if none available
    conn.execute("SELECT * FROM users");
} finally {
    if (conn != null) {
        pool.release(conn); // Always return to pool
    }
}

// With timeout
conn = pool.tryAcquire(5, TimeUnit.SECONDS);
if (conn != null) {
    try {
        conn.execute("SELECT * FROM orders");
    } finally {
        pool.release(conn);
    }
} else {
    System.out.println("Could not acquire connection");
}
```

Key design decisions:

- **BlockingQueue for available connections** - natural blocking behavior when pool exhausted
- **Set for in-use tracking** - fast lookup to validate release
- **ReentrantLock for state transitions** - protects inUse set modifications
- **ScheduledExecutorService** for maintenance - health checks and idle timeout
- **Try-acquire with timeout** - prevents indefinite blocking
- **Graceful shutdown** - close all connections, reject new acquires

5 Interview Discussion Points

When presenting these examples in interviews, discuss:

Trade-offs made:

- Why BlockingQueue vs custom wait/notify?
- When to use ConcurrentHashMap vs synchronized Map?
- Lock granularity: coarse-grained vs fine-grained
- Memory overhead vs performance

Edge cases handled:

- What happens during shutdown?
- How are errors propagated?
- What about thread interruption?
- Resource cleanup in finally blocks

Scalability considerations:

- How does it perform under high contention?
- Where are the bottlenecks?
- How would you monitor/tune in production?

Alternative designs:

- Could use different primitives?
- What about lock-free approaches?
- Trade-offs between approaches