Java Concurrency Cheatsheet

Staff-Level Interview Prep

Phase 1: Core Concepts

Race Conditions

When multiple threads access shared data and outcome depends on timing.

Example Problem:

```
if (balance < 100) {
    balance++; // NOT ATOMIC
}
```

Solution: Use synchronization primitives.

synchronized Keyword

On method (locks on this):

```
public synchronized void withdraw(int amt) {
    if (balance >= amt) {
        balance -= amt;
    }
}
```

On block (explicit lock):

```
private final Object lock = new Object();
public void withdraw(int amt) {
    synchronized(lock) {
        if (balance >= amt) balance -= amt;
    }
}
```

Key Points:

- Auto-releases lock on exception
- Instance methods lock on `this`
- Static methods lock on `Class` object
- Each instance has its own lock

ReentrantLock

Manual lock/unlock:

```
private final ReentrantLock lock =
    new ReentrantLock();

public void withdraw(int amt) {
    lock.lock();
    try {
        if (balance >= amt) balance -= amt;
    } finally {
        lock.unlock(); // ALWAYS in finally
    }
}
```

Advanced features:

```
// Try without blocking
if (lock.tryLock()) {
    try { /* ... */ }
    finally { lock.unlock(); }
}

// Try with timeout
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    // ...
}

// Fair lock (FIFO ordering)
new ReentrantLock(true);
```

When to use:

- Need `tryLock()` or timeout
- Need to interrupt waiting threads
- Need fair scheduling
- Multiple locks per class (fine-grained)

Reentrant: Same thread can lock multiple times (counter-based).

volatile Keyword

Ensures visibility across threads:

```
private volatile boolean stopped = false;

public void stop() { stopped = true; }

public void run() {
    while (!stopped) { /* work */ }
}
```

Guarantees:

- Writes immediately visible to all threads
- Prevents compiler/CPU reordering
- Does NOT provide atomicity for compound operations

Use for: Simple flags, status variables

Don't use for: Compound operations like `count++`

wait/notify/notifyAll

Thread coordination (must be in synchronized):

```
// Bounded Queue Example
private final Queue<T> queue = new LinkedList
    <>();
private final int capacity = 10;

public synchronized void put(T item)
    throws InterruptedException {
    while (queue.size() >= capacity) {
        wait(); // release lock, sleep
    }
    queue.add(item);
    notifyAll(); // wake waiting threads
}

public synchronized T take()
    throws InterruptedException {
    while (queue.isEmpty()) {
        wait();
    }
    T item = queue.remove();
    notifyAll();
    return item;
}
```

Critical rules:

- ALWAYS use `while`, not `if` with `wait()`
- Must be called inside `synchronized`
- `wait()` releases lock, blocks until notified
- `notifyAll()` safer than `notify()`

Memory Visibility & Happens-Before

Happens-Before Guarantee:

Action A happens-before B if B sees effects of A.

Key happens-before edges:

- Program order: statement 1 → statement 2
- Lock release → next lock acquire (same lock)
- volatile write → volatile read (same variable)
- Thread start → first action in started thread
- Last action in thread → join() returns
- Transitivity: A→B and B→C implies A→C

synchronized provides:

- Lock release → writes visible to next lock acquirer
- Happens-before guarantee

Without synchronization: Threads may see stale cached values.

Phase 2: Java Concurrency Toolkit

Executors & Thread Pools

Factory methods:

```
// Fixed size pool
ExecutorService exec =
    Executors.newFixedThreadPool(4);

// Single thread (sequential)
ExecutorService exec =
    Executors.newSingleThreadExecutor();

// Creates threads on demand, caches idle
ExecutorService exec =
    Executors.newCachedThreadPool();

// For scheduled tasks
ScheduledExecutorService exec =
    Executors.newScheduledThreadPool(4);
```

Submit tasks:

```
// Fire and forget
exec.submit(() -> doWork());

// Get result
Future<Integer> future = exec.submit(() -> {
    return 42;
});
Integer result = future.get(); // blocks

// With timeout
result = future.get(5, TimeUnit.SECONDS);
```

Shutdown pattern:

```
exec.shutdown(); // no new tasks
try {
    if (!exec.awaitTermination(60,
            TimeUnit.SECONDS)) {
        exec.shutdownNow(); // interrupt
    }
} catch (InterruptedException e) {
    exec.shutdownNow();
}
```

Scheduled execution:

```
scheduler.scheduleAtFixedRate(
    () -> cleanup(),
    0,      // initial delay
    60,     // period
    TimeUnit.SECONDS
);
```

Future Interface

```
Future<T> future = exec.submit(callable);

T result = future.get(); // blocks
T result = future.get(5, TimeUnit.SECONDS);
boolean done = future.isDone();
boolean cancelled = future.cancel(true);
```

Exception handling:

```
try {
    result = future.get();
} catch (ExecutionException e) {
    Throwable cause = e.getCause();
}
```

Concurrent Collections

ConcurrentHashMap:

```
ConcurrentHashMap<K, V> map =
    new ConcurrentHashMap<>();

// Atomic operations
map.putIfAbsent(key, value);
map.computeIfAbsent(key, k -> compute());
map.compute(key, (k,v) -> v == null ? 1 : v+1)
    ;
map.replace(key, oldVal, newVal);
```

Benefits over synchronized map:

- Fine-grained locking (per bucket)
- Multiple readers without blocking
- Concurrent writes to different buckets

BlockingQueue implementations:

```
// Bounded, array-backed
BlockingQueue<T> q =
    new ArrayBlockingQueue<>(100);

// Optionally bounded, linked nodes
BlockingQueue<T> q =
    new LinkedBlockingQueue<>();

// Priority-based
BlockingQueue<T> q =
    new PriorityBlockingQueue<>();
```

Operations:

```
q.put(item);     // blocks if full
T item = q.take(); // blocks if empty

q.offer(item);  // returns false if full
T item = q.poll(); // returns null if empty

// With timeout
q.offer(item, 1, TimeUnit.SECONDS);
T item = q.poll(1, TimeUnit.SECONDS);
```

Atomic Variables

Lock-free thread-safe operations:

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

counter.incrementAndGet(); // ++i
counter.getAndIncrement(); // i++
counter.addAndGet(5);      // i += 5
counter.compareAndSet(10, 20); // CAS

AtomicBoolean flag = new AtomicBoolean(false);
flag.set(true);
flag.compareAndSet(false, true);

AtomicReference<T> ref =
    new AtomicReference<>(obj);
ref.set(newObj);
ref.compareAndSet(expected, newObj);
```

Use when: Single variable updates, high contention
Advantage: No locks, uses CPU CAS instructions
Limitation: Only for single variable atomicity

ReadWriteLock

Readers-writers optimization:

```
private final ReadWriteLock rwLock =
    new ReentrantReadWriteLock();
private Data data;

public Data read() {
    rwLock.readLock().lock();
    try {
        return data;
    } finally {
        rwLock.readLock().unlock();
    }
}

public void write(Data newData) {
    rwLock.writeLock().lock();
    try {
        data = newData;
    } finally {
        rwLock.writeLock().unlock();
    }
}
```

Key properties:

- Multiple readers simultaneously
- Single writer (exclusive)
- Writer blocks all readers
- Best for read-heavy workloads

Coordination Tools

CountDownLatch:

```
CountDownLatch latch = new CountDownLatch(3);

// Workers
exec.submit(() -> {
    doWork();
    latch.countDown();
});

// Main thread
latch.await(); // blocks until count = 0
```

Use case: Wait for N tasks to complete

Semaphore:

```
Semaphore sem = new Semaphore(3); // 3 permits

sem.acquire(); // get permit
try {
    // access limited resource
} finally {
    sem.release(); // return permit
}

// Non-blocking
if (sem.tryAcquire()) { /* ... */ }
```

Use case: Limit concurrent access to N threads

CyclicBarrier:

```
CyclicBarrier barrier =
    new CyclicBarrier(3, () -> merge());

// Each thread
barrier.await(); // blocks until all arrive
```

Use case: Threads wait for each other at sync point

Thread Interruption

Cooperative cancellation mechanism:

```
// Check if interrupted
if (Thread.currentThread().isInterrupted()) {
    // cleanup and exit
}

// Interruptible operations throw
try {
    Thread.sleep(1000);
    queue.take();
} catch (InterruptedException e) {
    // Restore interrupt status
    Thread.currentThread().interrupt();
    // cleanup and exit
}
```

Interrupting a thread:

```
thread.interrupt(); // sets interrupt flag
```

Best practices:

- Always restore interrupt status after catching
- Don't swallow InterruptedException
- Check interrupted() in long-running loops
- Use for graceful cancellation, not forced stop

Double-Checked Locking

Broken pattern (without volatile):

```
// DON'T DO THIS - broken!
if (instance == null) {
    synchronized(lock) {
        if (instance == null) {
            instance = new Singleton(); // NOT
                atomic
        }
    }
}
```

Fixed with volatile:

```
private volatile Singleton instance;

if (instance == null) {
    synchronized(lock) {
        if (instance == null) {
            instance = new Singleton();
        }
    }
}
return instance;
```

Better: Holder idiom (no volatile needed):

```
class Singleton {
    private static class Holder {
        static final Singleton INSTANCE = new
            Singleton();
    }
    public static Singleton get() {
        return Holder.INSTANCE;
    }
}
```

CompletableFuture

Non-blocking async composition:

```
CompletableFuture<String> future =
    CompletableFuture.supplyAsync(() ->
        fetchData());

// Chain transformations (non-blocking)
future.thenApply(data -> process(data))
      .thenAccept(result -> save(result))
      .exceptionally(ex -> handleError(ex));

// Combine multiple futures
CompletableFuture<String> f1 = asyncOp1();
CompletableFuture<String> f2 = asyncOp2();

CompletableFuture.allOf(f1, f2)
    .thenRun(() -> System.out.println("Both
        done"));
```

```
// Get first completed
CompletableFuture.anyOf(f1, f2)
    .thenAccept(result -> process(result));
```

Advantages over Future:

- Non-blocking composition
- Built-in error handling
- Combine multiple async operations
- More functional programming style

Phase 3: Classic Problems

Producer-Consumer

Pattern: Data flow between threads via shared buffer

Solution 1 - Custom with wait/notify:

```
class BoundedQueue<T> {
    private Queue<T> queue = new LinkedList
        <>();
    private int capacity;

    public synchronized void put(T item)
        throws InterruptedException {
        while (queue.size() >= capacity) {
            wait();
        }
        queue.add(item);
        notifyAll();
    }

    public synchronized T take()
        throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notifyAll();
        return item;
    }
}
```

Solution 2 - BlockingQueue:

```
BlockingQueue<T> queue =
    new ArrayBlockingQueue<>(100);

// Producer
queue.put(item);

// Consumer
T item = queue.take();
```

Priority variant:

```
PriorityBlockingQueue<Task> queue =
    new PriorityBlockingQueue<>();
```

Readers-Writers

Problem: Multiple readers safe, writers need exclusive access

Solution:

```
ReadWriteLock rwLock =
    new ReentrantReadWriteLock();

// Read: multiple concurrent
rwLock.readLock().lock();
try { /* read */ }
finally { rwLock.readLock().unlock(); }

// Write: exclusive
rwLock.writeLock().lock();
try { /* write */ }
finally { rwLock.writeLock().unlock(); }
```

Dining Philosophers

Problem: Deadlock when all grab left fork simultaneously

Solution 1 - Numbered resources:

```
// Always acquire lower-numbered fork first
int first = Math.min(left, right);
int second = Math.max(left, right);
synchronized(forks[first]) {
    synchronized(forks[second]) {
        eat();
    }
}
```

Solution 2 - Limit diners:

```
Semaphore seats = new Semaphore(4); // n-1
seats.acquire();
try {
    pickUpForks();
    eat();
} finally {
    seats.release();
}
```

Solution 3 - Asymmetry:

```
// Odd: left-right, Even: right-left
if (id % 2 == 0) {
    synchronized(leftFork) {
        synchronized(rightFork) { eat(); }
    }
} else {
    synchronized(rightFork) {
        synchronized(leftFork) { eat(); }
    }
}
```

Advanced Topics

LongAdder vs AtomicLong

Use LongAdder for high contention:

```
LongAdder counter = new LongAdder();
counter.increment(); // split across cells
long total = counter.sum(); // aggregate
```

When: Multiple threads frequently increment/decrement
Why: Reduces contention by striping updates

Lock-Free Algorithms

Compare-and-swap (CAS):

```
AtomicReference<Node> head = new
    AtomicReference<>();

// Lock-free stack push
Node newNode = new Node(value);
do {
    newNode.next = head.get();
    // CAS: atomically set head to newNode
    // only if it's still equal to newNode.
       next
} while (!head.compareAndSet(newNode.next,
    newNode));
```

Trade-offs:

- Pro: No blocking, better under contention
- Con: More complex, potential livelock, ABA problem

Red Flags to Avoid

Common mistakes in interviews:

1. Busy-wait spinning without yield

```
// BAD
while (!ready) { } // burns CPU

// BETTER
while (!ready) {
    Thread.yield(); // or use wait/notify
}
```

2. Swallowing InterruptedException

```
// BAD
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // ignore - WRONG!
}

// GOOD
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    // cleanup
}
```

3. Synchronized on public/mutable object

```
// BAD - others can lock on this
public synchronized void method() { }

// BETTER - private lock
private final Object lock = new Object();
synchronized(lock) { }
```

4. Nested locks without ordering

```
// BAD - potential deadlock
synchronized(lockA) {
    synchronized(lockB) { }
}

// BETTER - consistent lock ordering
// always acquire lower-numbered lock first
```

Common Pitfalls

1. **Using `if` instead of `while` with `wait()`**
2. **Forgetting `finally` block with ReentrantLock**
3. **Calling `wait/notify` outside synchronized**
4. **Assuming `volatile` makes compound ops atomic**
5. **Not handling InterruptedException**
6. **Forgetting to shutdown ExecutorService**
7. **Using `synchronized(this)` when object exposed**

Interview Strategy

1. Identify the pattern:

- Producer-Consumer?
- Readers-Writers?
- Resource contention?
- Coordination needed?

2. Choose primitives:

- Simple mutual exclusion → `synchronized`
- Need tryLock/timeout → `ReentrantLock`
- Read-heavy → `ReadWriteLock`
- Coordination → `wait/notify` or utilities
- Producer-consumer → `BlockingQueue`

3. Consider:

- Deadlock prevention
- Starvation
- Fairness
- Performance vs complexity

4. Discuss trade-offs:

- Lock granularity
- Built-in vs custom
- Simplicity vs optimization

5. Ask clarifying questions:

- Expected load (QPS/throughput)?
- Latency requirements (p50, p99)?
- Read-heavy or write-heavy?
- Strict ordering needed?
- Failure tolerance requirements?

Code Review Checklist

Before submitting concurrent code:

☐ All shared mutable state synchronized
☐ Locks released in finally blocks
☐ wait() always in while loop
☐ InterruptedException handled properly
☐ ExecutorService shut down in cleanup
☐ No nested locks without deadlock strategy
☐ Thread-safety documented in javadoc
☐ Atomic operations for check-then-act
☐ volatile not used for compound ops
☐ No sync on computed/mutable objects

Key Imports

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