Java Incident Management: Full Master Runbook

Purpose and Core Functionality of Java

Java is a high-level, class-based, object-oriented programming language designed to have as few implementation dependencies as possible.
Its main purposes:

- Write Once, Run Anywhere (WORA) — compiled Java code runs on all platforms that support Java without recompilation.
- Strong performance due to Just-In-Time (JIT) compilation.
- Robust multithreading, memory management, and security model.

Core Functionality:

- Managed memory via Garbage Collection.
- Multithreading primitives (synchronized, locks, thread pools).
- Exception handling mechanisms.
- Standard libraries for networking, file I/O, concurrency, etc.
- Virtual Machine abstraction (JVM) decoupling application from hardware/OS.

Effective Java Incident Diagnosis and Troubleshooting Methods

General Best Practices:

- **Use Observability Tools** (metrics, logs, traces).
- **Thread Dumps**: Capture and analyze live JVM thread states.
- **Heap Dumps**: Analyze memory contents.
- **GC Logs**: Understand memory collection behaviors.
- **Profiling**: CPU/memory usage under real loads.
- **Structured Logging**: Correlate logs with requests/events.

Common Java Diagnostic Tools:

- jstack — Thread dump
- jmap — Heap dump
- jstat — GC and JVM statistics
- visualvm, jconsole — UI-based profiling and monitoring
- APM tools — e.g., New Relic, Dynatrace, Datadog
- JVM built-in metrics — via Micrometer, Prometheus, etc.

Table of Contents

Runbook #	Failure Mode
16	Resource Starvation (File Handles, DB Cursors)
17	Thread Starvation
18	JVM Crash Due to Native Memory Issues
19	Memory Thrashing
20	JVM CPU Thrashing (High Context Switching)
21	Application Hangs Without Errors

Full Runbooks

Runbook 1: High CPU Utilization

Failure Mode:
Java application consumes excessive CPU resources.

Symptoms:

- CPU usage > 90%
- High response times or service degradation
- Load averages spike on server

Diagnostic Steps:

- Capture thread dump: jstack <pid>
- Identify CPU-heavy threads:
  top -H -p <pid>, match thread IDs with thread dump.
- Use profilers: VisualVM, JProfiler

Common Causes:

- Infinite loops
- Busy-waiting on locks

- Bad thread synchronization
- Expensive computation

Immediate Actions:

- Restart the application to temporarily relieve pressure.
- Throttle or divert traffic if needed.

Remediation:

- Profile hotspots and fix tight loops.
- Introduce appropriate sleep/yield in polling loops.
- Improve lock granularity.

Runbook 2: Thread Deadlocks

Failure Mode:
Two or more threads hold locks that block each other.

Symptoms:

- Application hangs
- No progress despite normal CPU
- Thread dumps show “Found one Java-level deadlock”

Diagnostic Steps:

- jstack <pid>: Look for deadlock section.
- Analyze resource ordering in code.

Common Causes:

- Incorrect lock acquisition order
- Nested synchronized blocks

Immediate Actions:

- Restart application if recovery is not possible.

Remediation:

- Always acquire locks in a consistent global order.
- Minimize synchronized sections.

Runbook 3: OutOfMemoryError (Java Heap Space)

Failure Mode:
Application runs out of heap memory.

Symptoms:

- java.lang.OutOfMemoryError: Java heap space
- Application crash or instability

Diagnostic Steps:

- Capture heap dump: jmap -dump:format=b,file=heapdump.hprof <pid>
- Analyze with Eclipse MAT or VisualVM.
- Monitor GC logs.

Common Causes:

- Memory leaks
- Loading large data sets in memory
- Retained references

Immediate Actions:

- Restart application.
- Increase heap size temporarily.

Remediation:

- Fix memory leaks.
- Stream large datasets instead of loading into memory.

Runbook 4: OutOfMemoryError (GC Overhead Limit Exceeded)

Failure Mode:
Application spends too much time in GC but gains little memory back.

Symptoms:

- java.lang.OutOfMemoryError: GC overhead limit exceeded
- Extreme GC frequency

Diagnostic Steps:

- Analyze GC logs.
- Heap dump and MAT analysis.

Common Causes:

- Memory leak
- Heap too small
- Too many small objects

Immediate Actions:

- Increase heap size.
- Restart application.

Remediation:

- Tune GC settings.
- Optimize object creation.

Runbook 5: Memory Leak

Failure Mode:
Memory usage continually grows until exhaustion.

Symptoms:

- Heap usage never drops after GC
- OutOfMemoryError

Diagnostic Steps:

- Heap dump analysis
- Monitor retained objects

Common Causes:

- Collections not cleared
- Listeners not deregistered
- Static fields referencing heavy objects

Immediate Actions:

- Restart to free memory.

Remediation:

- Fix leaking references.
- Add memory pressure testing to CI/CD.

Runbook 6: Garbage Collection Pauses (Stop-the-World)

Failure Mode:
GC "Stop-The-World" pauses cause latency spikes.

Symptoms:

- Long GC pause times
- High GC logs pause durations
- Request timeouts

Diagnostic Steps:

- Analyze GC logs with GCViewer.
- Observe pause patterns.

Common Causes:

- Too large young/old generation
- Full GC triggered frequently

Immediate Actions:

- Increase heap.
- Reduce live set size if possible.

Remediation:

- Tune heap sizing and GC algorithms (G1GC recommended).
- Break up large objects.

Runbook 7: Database Connection Pool Exhaustion

Failure Mode:
All database connections are used up, blocking new queries.

Symptoms:

- Application hangs on DB operations
- Errors like Timeout waiting for connection from pool
- Connection pool metrics show zero available connections

Diagnostic Steps:

- Monitor HikariCP, Tomcat JDBC pool, or other pool metrics.
- Analyze slow queries or leaked connections.

Common Causes:

- Long-running queries
- Connections not closed properly
- Pool size too small for load

Immediate Actions:

- Increase pool size temporarily.
- Restart application if necessary.

Remediation:

- Ensure connections are closed in finally blocks.
- Tune pool sizing.
- Optimize slow queries.

Runbook 8: High GC Frequency

Failure Mode:
Garbage collector triggers extremely often.

Symptoms:

- High GC count
- High GC CPU usage
- Performance degradation

Diagnostic Steps:

- Analyze GC logs.
- Heap dump if needed.

Common Causes:

- Heap too small
- Memory churn from frequent object allocation/deallocation

Immediate Actions:

- Increase heap size.
- Reduce load.

Remediation:

- Reduce temporary object creation.
- Use object pooling.

Runbook 9: Slow Application Start-up

Failure Mode:
Application takes a very long time to initialize.

Symptoms:

- Start-up time significantly longer than normal
- High CPU or IO during boot

Diagnostic Steps:

- Analyze start-up logs.
- Use Java Flight Recorder (JFR).

Common Causes:

- Classpath scanning overhead
- Lazy GC tuning
- Expensive resource initialization

Immediate Actions:

- Restart and monitor.
- Roll back recent changes if introduced.

Remediation:

- Optimize dependency injection frameworks (e.g., Spring).
- Lazy-load non-critical components.

Runbook 10: Application Crash (SIGSEGV or Fatal Error)

Failure Mode:
Java process crashes due to JVM-level fatal error.

Symptoms:

- JVM dumps fatal error log (hs_err_pid.log)
- SIGSEGV (segmentation fault) or similar

Diagnostic Steps:

- Analyze hs_err_pid.log.
- Look for native libraries, JNI calls.

Common Causes:

- Native library bugs
- JVM bugs
- Memory corruption

Immediate Actions:

- Restart application.

Remediation:

- Upgrade JVM.
- Patch native dependencies.

Runbook 11: Uncaught Exceptions Causing Service Instability

Failure Mode:
Uncaught exceptions bubble to thread group handler, destabilizing service.

Symptoms:

- Unexpected service restarts
- Log entries of fatal exceptions

Diagnostic Steps:

- Review logs for UncaughtExceptionHandler messages.
- Identify crash triggers.

Common Causes:

- Unhandled application logic bugs
- Unexpected input

Immediate Actions:

- Restart service.

Remediation:

- Catch and handle exceptions properly.
- Validate inputs aggressively.

Runbook 12: Thread Pool Exhaustion

Failure Mode:
No available threads in a thread pool.

Symptoms:

- Requests hang
- Metrics show full thread pool and growing queue

Diagnostic Steps:

- Thread dump analysis.
- Thread pool monitoring (size, active, queue).

Common Causes:

- Too few threads configured
- Thread leakage (threads blocked indefinitely)

Immediate Actions:

- Restart to clear blocked threads.

Remediation:

- Tune thread pool sizing.
- Refactor to avoid blocking operations.

Runbook 13: Socket Timeout or Connection Reset Errors

Failure Mode:
Socket operations timeout or reset.

Symptoms:

- java.net.SocketTimeoutException
- java.net.SocketException: Connection reset

Diagnostic Steps:

- Review client/server logs.
- Analyze network behavior.

Common Causes:

- Downstream service slowness

- Network congestion

Immediate Actions:

- Retry failed requests with backoff.
- Failover to backup systems.

Remediation:

- Tune socket timeout settings.
- Implement retries and circuit breakers.

Runbook 14: High Latency in External Service Calls

Failure Mode:
External dependency response times degrade.

Symptoms:

- Increased end-to-end response times
- Backend timeouts

Diagnostic Steps:

- Analyze APM traces.
- Log timings for external calls.

Common Causes:

- External service degradation
- Latency spikes due to retries

Immediate Actions:

- Redirect traffic if possible.
- Rate-limit requests.

Remediation:

- Tune retry logic.
- Implement fallback mechanisms.

Runbook 15: Stuck Threads

Failure Mode:
Threads are indefinitely waiting on resources.

Symptoms:

- Active threads stuck in WAITING/BLOCKED states
- Thread dump analysis shows repeating patterns

Diagnostic Steps:

- Thread dumps (jstack).
- Look for locks, monitors.

Common Causes:

- Blocking I/O
- Improper thread synchronization

Immediate Actions:

- Restart application.

Remediation:

- Refactor blocking operations to async.
- Improve lock acquisition strategies.

Runbook 16: Resource Starvation (File Handles, DB Cursors)

Failure Mode:
Process hits OS or DB resource limits.

Symptoms:

- Too many open files
- Database cursor exhaustion

Diagnostic Steps:

- Check ulimit -n

- Monitor open connections/files

Common Causes:

- Resource leaks
- No proper close() on streams/connections

Immediate Actions:

- Restart service.

Remediation:

- Ensure resource closure.
- Increase system limits if justified.

Runbook 17: Thread Starvation

Failure Mode:
Application becomes unresponsive because critical tasks are blocked from running due to lack of available threads.

Symptoms:

- High request latency
- Metrics show zero idle threads in thread pool
- Long queue length for tasks

Diagnostic Steps:

- Thread dump: jstack <pid> — look for threads stuck in WAITING or BLOCKED
- Review thread pool metrics (e.g., active count, queue size)
- Check for synchronized blocks or lock() usage in stack traces

Common Causes:

- Synchronous calls within thread-limited pool
- Blocking operations inside async frameworks
- Poorly sized executor services

Immediate Actions:

- Restart service to clear congestion
- Temporarily increase thread pool size

Remediation:

- Use separate pools for blocking vs compute tasks
- Refactor blocking logic into async/queued patterns
- Monitor thread pool utilization over time

Runbook 18: JVM Crash Due to Native Memory Issues

Failure Mode:
Out-of-heap memory, often native (direct buffer leaks, metaspace).

Symptoms:

- Crash logs mentioning native memory
- OutOfMemoryError (Direct Buffer Memory)

Diagnostic Steps:

- JVM native memory tracking (-XX:NativeMemoryTracking=summary)
- Analyze crash logs

Common Causes:

- Direct ByteBuffers not cleaned
- JNI native memory leaks

Immediate Actions:

- Restart application.
- Reduce load.

Remediation:

- Tune or fix direct memory usage.
- Update native libraries.

Runbook 19: Application Memory Leak

Failure Mode:
Application memory usage continuously grows without bound.

Symptoms:

- Heap usage increases steadily over time
- OutOfMemoryError eventually thrown
- GC logs show increasing pause times and heap occupancy

Diagnostic Steps:

- Analyze heap dumps (jmap -dump:live,file=heapdump.hprof <pid>)
- Use tools like Eclipse MAT, VisualVM, or YourKit to analyze references
- Identify leaking object types and their retainers

Common Causes:

- Caching without proper eviction
- Static collections accumulating objects
- Unclosed listeners or subscriptions

Immediate Actions:

- Restart application to clear memory
- Reduce incoming load if necessary

Remediation:

- Fix the memory leak based on heap analysis
- Implement proper cache eviction policies
- Regularly unsubscribe listeners, clean up resources

Runbook 20: JVM Deadlock Detection and Recovery

Failure Mode:
Two or more threads waiting indefinitely for each other's locks.

Symptoms:

- Application hangs or becomes very slow
- Thread dumps show DEADLOCK detected messages

Diagnostic Steps:

- Thread dump analysis (jstack <pid>)
- Look for cyclic locking patterns

Common Causes:

- Poor lock ordering in code
- Nested synchronization on shared resources

Immediate Actions:

- Restart service to clear deadlock
- Terminate only the deadlocked threads (advanced JVM tooling required)

Remediation:

- Enforce consistent lock acquisition order
- Use tryLock() with timeouts instead of blocking locks
- Redesign critical sections to avoid nested locks

Runbook 21: Service Hangs without Obvious Errors

Failure Mode:
Application becomes unresponsive, no clear exceptions or crashes.

Symptoms:

- Service stops responding to requests
- No recent log entries
- CPU may be high or low

Diagnostic Steps:

- Capture thread dumps repeatedly
- Analyze stuck threads, deadlocks, infinite loops

Common Causes:

- Deadlocks
- Infinite loops
- Resource exhaustion (e.g., stuck I/O operations)

Immediate Actions:

- Restart service to recover
- Gather diagnostic artifacts for RCA (thread dumps, heap dumps, GC logs)

Remediation:

- Fix root cause from thread analysis
- Improve timeout handling and resilience patterns

List of Other Known Failure Modes (Not yet fully detailed)

These are additional common and niche Java failure types not yet associated with a full runbook:

- JVM Safepoint Bias — very long GC pause times
- ClassLoader Leaks (e.g., dynamic redeploy apps)
- Native Buffer Exhaustion (DirectByteBuffer leaks)
- Time Skew Issues impacting distributed apps (NTP misconfigs)
- High Frequency JIT Recompilation (tiered compilation thrash)
- OS-level CPU Throttling (container limits)
- Premature Finalization (WeakReference handling bugs)
- TCP Connection Pileup (FIN_WAIT2 issues)
- Stuck Futures / CompletableFutures
- Overloaded ThreadLocals causing slow GC
- GC Overhead Limit Exceeded Errors
- Incorrect Unsafe memory operations

- Failing Native Memory Tracking itself
- Incorrect JVM options leading to instability