When your Java application starts throwing java.lang.OutOfMemoryError (OOM), it generally indicates that the JVM cannot allocate enough memory for new objects. There are multiple possible causes—like memory leaks, high concurrency usage that saturates the heap, or improper memory configuration. In diagnosing these issues, **thread dumps** and **heap dumps** are two of the most important artifacts to gather and analyze. Below is a step-by-step guide on how to capture, configure, and interpret thread and heap dumps, along with some examples and best practices.

---

1. Overview of OutOfMemoryError

Types of OOM

1. **Java heap space**: The most common type. The application has exhausted the Java heap.
2. **GC overhead limit exceeded**: The JVM is spending too much time doing garbage collection with too little reclaimed.
3. **Metaspace/PermGen space**: (For Java 8+, Metaspace replaced PermGen.) Could happen if too many classes are loaded or the metaspace is undersized.
4. **Unable to create new native thread**: The system or JVM cannot create new threads, possibly due to system resource limits.

This answer focuses primarily on the classic Java heap space exhaustion scenario, but the diagnostics are similar for other types.

---

2. Capturing the Data

To diagnose OOM issues, you need both **thread dumps** and **heap dumps** around the time the OOM occurs. These dumps show the application's state and memory contents, respectively, letting you correlate which threads might be causing excessive memory usage and which objects are consuming the heap.

2.1 Configuring Automatic Dumps

You can configure the JVM to automatically generate dumps on OOM using the following JVM flags:

```
# Example 1: Generate a heap dump upon OOM
-XX:+HeapDumpOnOutOfMemoryError \
-XX:HeapDumpPath=/path/to/heapdump.hprof
```

```
# Example 2: Generate a thread dump or run a command upon OOM
-XX:OnOutOfMemoryError="jstack -l %p > /path/to/threaddump.%p.log"
```

Note:

- %p in the OnOutOfMemoryError string is replaced by the process ID of the JVM.
- Make sure you have write permissions to /path/to/ and enough disk space because heap dumps can be large.

Alternatively, you can configure the OS or container environment (e.g., Docker, Kubernetes) to capture logs or run commands when the application fails.

2.2 Manual Trigger

You may also **manually** trigger dumps if the application is still partially responsive or if you have access to the process via jps/jcmd:

```
# jmap for heap dump
jmap -dump:live,file=/path/to/heapdump.hprof <PID>

# jstack for thread dump
jstack -l <PID> > /path/to/threaddump.log
```

If using the newer jcmd utility:

```
# Heap dump
jcmd <PID> GC.heap_dump /path/to/heapdump.hprof

# Thread dump
jcmd <PID> Thread.print > /path/to/threaddump.log
```

---

3. Analyzing the Thread Dumps

A **thread dump** is a snapshot of all live threads in the JVM at a given moment. It helps you see:

- Which threads are blocked, waiting, or in deadlock.
- Which threads are performing heavy computations or extensive allocations.
- The stack traces of each thread.

3.1 Reading a Thread Dump

A typical thread dump snippet might look like this (simplified):

```
"main" #1 prio=5 os_prio=31 tid=0x00007fcb28001000 nid=0x2003 runnable [0x0000700008334000]
   java.lang.Thread.State: RUNNABLE
    at com.example.myapp.processor.DataProcessor.process(DataProcessor.java:42)
    at com.example.myapp.Main.main(Main.java:15)
```

Key sections:

- **Thread name**: "main", or something from your thread pool like "pool-1-thread-2".
- **State**: RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, etc.
- **Stack trace**: Tells you which classes and lines of code the thread is executing.

3.2 Looking for Clues

For OOM diagnosis, you may notice:

- Many threads stuck in a method that allocates large objects (e.g., building enormous collections).
- Threads blocked waiting for a resource while a few threads are busy doing memory-intensive tasks.

You can match the suspicious threads to object allocations from the **heap dump** analysis to see which code path is generating large structures.

---

4. Analyzing the Heap Dumps

A **heap dump** is a snapshot of all the Java objects in the heap at a certain moment. It contains:

- Object instances
- Class definitions
- Fields and references

4.1 Tools for Heap Dump Analysis

1. **Eclipse Memory Analyzer (MAT)** – One of the most commonly used open-source tools.
2. **VisualVM** – Comes with the JDK (in newer JDK distributions, you might have to install separately).

3. **YourKit / JProfiler** – Commercial profilers with powerful features.

4.2 Basic Steps in Eclipse Memory Analyzer (MAT)

1. **Open the heap dump**: File -> Open Heap Dump....
2. **Run a Leak Suspects Report**: MAT has a built-in "Leak Suspects" tool that points to suspicious growth areas.
3. **Inspect dominator tree**: The dominator tree shows which objects or classes are retaining the most memory. Look for classes with very large retained size.

4.3 Typical Findings in the Heap Dump

- **Large Collections** (e.g., ArrayList, HashMap, ConcurrentHashMap) with unexpectedly high entry counts.
- **Caches** that never evict, causing memory bloat.
- **Excessively large strings** or large arrays (char[], byte[], int[], etc.).
- **Class Loader leaks** or references preventing classes from unloading (in metaspace scenarios).
- **Custom objects** that are not being cleaned up (listeners, resource handles, etc.).

For example, if you see something like:

```
Class Name                | Shallow Heap | Retained Heap |  Number of Objects
-------------------------------------------------------------------------------
com.example.myapp.LargeMap |  1,024,432 |  912,345,678 |          1
java.util.HashMap$Node[]   |        ... |          ...
```

You might deduce that LargeMap is holding onto nearly 900 MB of objects, which is likely the culprit.

---

5. Correlating Thread Dumps and Heap Dumps

1. **Who is filling up the memory?**
   - From the thread dump, identify if there's a suspicious thread stack trace repeatedly creating large objects or data structures.
   - From the heap dump, check if those objects appear in large quantities or with large memory footprints.
2. **Is the memory usage truly a leak or just usage spikes?**
   - If the memory usage is legitimate but the heap is too small for the workload, you may need to scale up the heap with -Xmx.

- If it's a leak, you'll see objects that are never released and keep growing over time (e.g., a static collection that never shrinks).

3. **Look for repeated patterns**
   - Repeated thread states indicating frequent synchronous operations.
   - The same data structure dominating memory in multiple heap dumps.

---

6. Common Root Causes of OOM and How to Fix Them

1. **Insufficient Heap Size**
   - Increase the maximum heap: `-Xmx2g` (example).
   - Monitor GC logs to confirm if you're consistently near the heap limit.
2. **Memory Leak**
   - Identify the leaking data structure (cache, queue, static field).
   - Fix code to properly remove references or reduce size.
3. **Unbounded Queues or Caches**
   - Implement size limits, eviction policies, or soft references.
4. **Excessive Number of Threads**
   - Check if too many threads lead to overhead or memory usage for each thread's stack.
   - Use a thread pool with a controlled maximum size.
5. **Incompatible GC Configuration**
   - Check if the GC collector choice or GC parameters are hurting performance.
   - Example: If using G1 GC, ensure you have appropriate region sizes for your heap.

---

7. Example Diagnostic Workflow

Let's walk through a hypothetical scenario:

1. **Symptom**: Your web application repeatedly crashes with `java.lang.OutOfMemoryError: Java heap space`.
2. **Setup**: You enable automatic heap dump generation using:

   ```
   -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/logs/myapp-oom.hprof
   ```

3. **Occurrence**: The application crashes again, generating `/logs/myapp-oom.hprof`. You also add:

   ```
   -XX:OnOutOfMemoryError="jstack -l %p > /logs/threaddump_%p.log"
   ```

So you get /logs/threaddump_12345.log (where 12345 is the PID).

4. **Analysis**:
    - Open myapp-oom.hprof in Eclipse MAT.
    - Run the "Leak Suspects" report. It points you to com.example.cache.InMemoryCache retaining ~75% of your total heap.
    - Examine the thread dump (threaddump_12345.log) and notice that multiple threads named CacheRefresher-X are always adding new data to InMemoryCache.
5. **Resolution**: Limit the size of InMemoryCache or add an eviction strategy. Redeploy, watch memory usage, and confirm that OOM no longer occurs.

---

8. Configuration Best Practices

- Always **enable GC logging** (in newer JDKs, use -Xlog:gc* or -verbose:gc in older versions). This helps correlate how memory usage grows over time:

  -Xlog:gc*:file=/logs/myapp-gc.log:time,uptimemillis

- Keep an eye on **system-level resources** as well (CPU usage, swap usage, container memory limits).
- For production, **automate** capturing and archiving logs/dumps so you don't have to rely solely on manual triggers.

---

9. Summary

- **Thread dumps** tell you **what** your application is doing at the moment of trouble (which methods, how many threads, any deadlocks).
- **Heap dumps** tell you **how** memory is being used (which objects occupy large portions of the heap, potential leaks).
- By carefully **correlating** the data from thread dumps (suspect code paths) and heap dumps (memory usage patterns), you can pinpoint root causes for frequent OutOfMemoryError.
- **Proper configuration** (e.g., -XX:+HeapDumpOnOutOfMemoryError) automates capture, ensuring you have the data you need when OOM occurs.
- **Tools** like Eclipse MAT, VisualVM, and jcmd/jstack/jmap are crucial to a robust, repeatable diagnostic process.

With these steps and strategies, you should be well-positioned to investigate, identify, and resolve the root cause of frequent OutOfMemoryError in your Java applications.