Java: OutOfMemoryError :

When a Java application throws an OutOfMemoryError (OOM), it means the JVM failed to allocate memory for a new object because the heap or another memory region is exhausted. This guide explains what happens, how JVM handles it, and how developers can deal with it.

---

1. What Happens Internally

1.1 Garbage Collection First

- When JVM cannot allocate memory for a new object, it first runs Garbage Collection (GC).
- GC attempts to reclaim unused objects.
- If GC frees enough space, allocation succeeds.
- If not, OOM is thrown.

1.2 Heap and JVM Behavior

- Java Heap is divided into:
  - Young Generation (new objects, cleaned by Minor GC).
  - Old Generation (long-lived objects, cleaned by Major/Full GC).
  - Metaspace (class metadata, Java 8+).
- If both Young and Old generations cannot provide memory after GC, JVM throws:
- `java.lang.OutOfMemoryError: Java heap space`

1.3 JVM After OOM

- The application thread that requested memory fails with `OutOfMemoryError`.
- If not caught, the program may terminate.
- JVM itself usually continues running unless the OOM affects critical internals.

---

2. How Developers Handle It

2.1 Increase Heap Size (if not a leak)

```
java -Xms512m -Xmx1024m MyApp
```

- `-Xms` → Initial heap size.
- `-Xmx` → Maximum heap size.
- Example: `-Xms512m -Xmx2g` → Start with 512MB, allow up to 2GB.

2.2 Detect Memory Leaks

- Use tools such as jVisualVM, Eclipse MAT, YourKit, JProfiler.
- Look for objects that are never released (e.g., static collections, caches).

2.3 Free Object References

- Avoid unnecessary `static` references.
- Properly clear caches, maps, and lists.

2.4 Use Off-Heap Storage

- Store large data outside the JVM heap (e.g., DirectByteBuffer, Redis, Memcached).

2.5 Catch OOM (Rarely Useful)

```
try {
    // memory-intensive work
} catch (OutOfMemoryError e) {
    System.err.println("OOM occurred! Logging and cleaning up...");
}
```

- Useful only for logging or graceful shutdown.
- You cannot fully recover from OOM in most cases.

---

3. Detecting “Small Heap” vs “Memory Leak”

3.1 Check Error Message

- `Java heap space` → Heap full.
- `GC overhead limit exceeded` → JVM is stuck doing GC but not freeing memory.
- `Metaspace` → Too many loaded classes.
- `Direct buffer memory` → Off-heap allocation failed.

3.2 Enable GC Logs

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:gc.log
```

- If GC runs and frees space but demand is too high → Small heap problem.
- If GC runs but cannot free memory (objects still retained) → Memory leak.

3.3 Generate Heap Dump

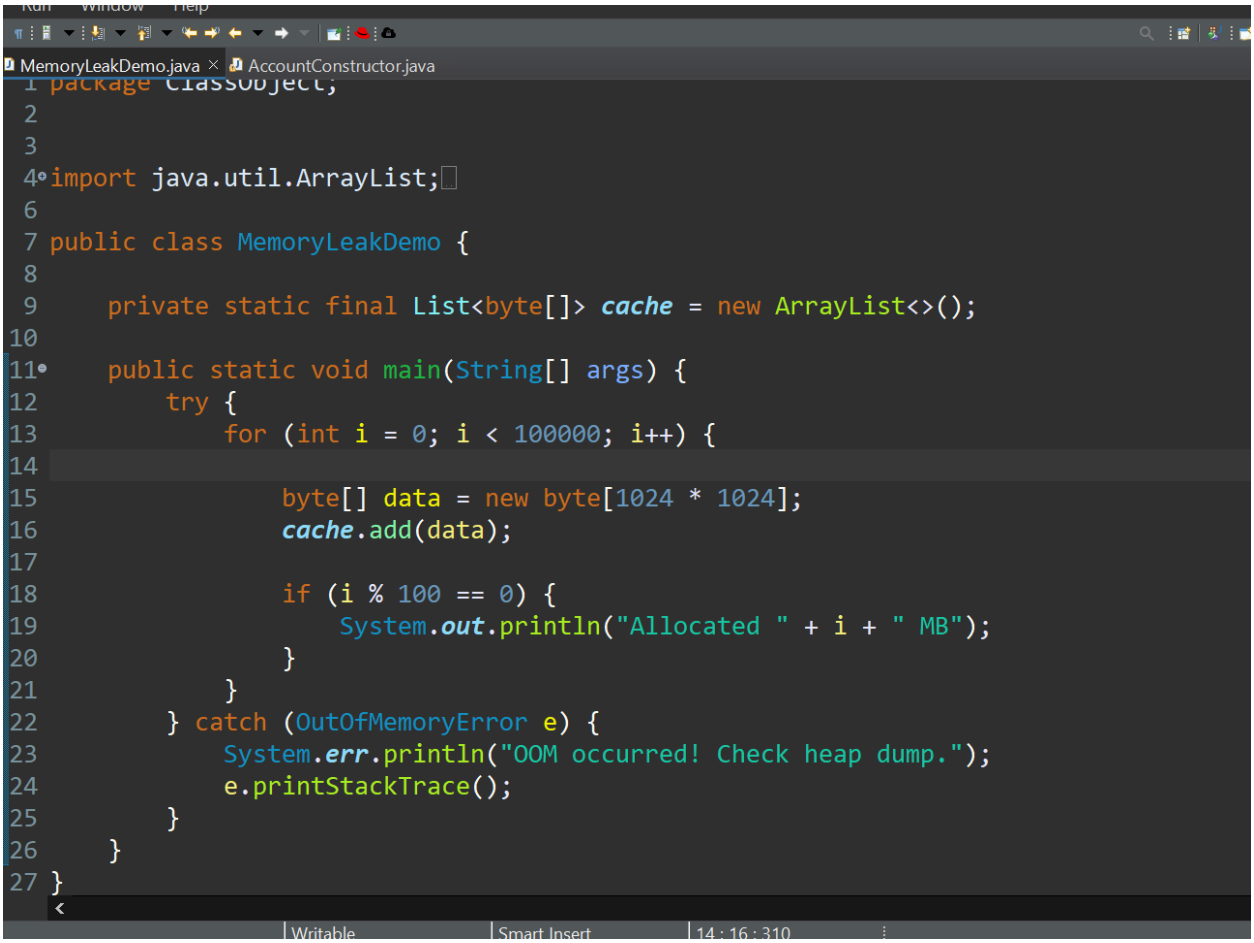
```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=heapdump.hprof
```

- Analyze with Eclipse MAT / jVisualVM.
- If one type of object dominates heap and is still referenced → Memory leak.
- If objects are diverse and GC works fine → Small heap.

---

4. Example: Intentional Memory Leak

Here is a simple Java program that creates a memory leak by holding strong references in a static list.



---

5. How to Run

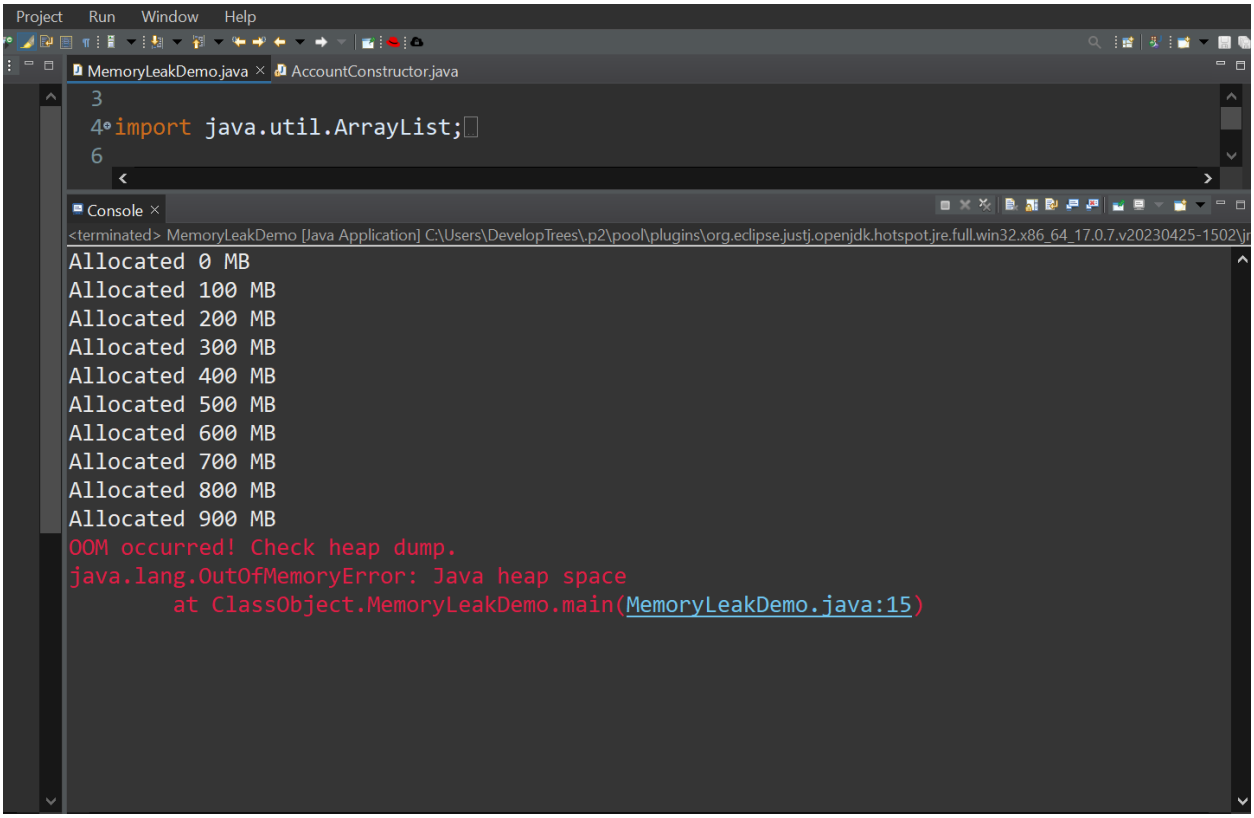
5.1 Compile

```
javac MemoryLeakDemo.java
```

5.2 Run with Small Heap

```
java -Xms64m -Xmx64m -XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=heapdump.hprof MemoryLeakDemo
```

5.3 Sample Output



- A heap dump file (`heapdump.hprof`) will be generated.

Example 2: Small Heap, No Memory Leak

```java
public class SmallHeapDemo {
    public static void main(String[] args) {
        try {
            List<byte[]> temp = new ArrayList<>();
            for (int i = 0; i < 100000; i++) {
                // Each iteration allocates 1MB
                byte[] data = new byte[1024 * 1024];
                temp.add(data);

                if (i % 100 == 0) {
                    System.out.println("Allocated " + i + " MB");
                }

                // Free memory every 500 MB
                if (i % 500 == 0) {
                    temp.clear(); // allow GC to reclaim memory
                    System.out.println("Cleared temp list, memory should be freed.");
                }
            }
        } catch (OutOfMemoryError e) {
            System.err.println("OOM occurred! Check heap dump.");
            e.printStackTrace();
        }
    }
```

Line: 21

```
Console ×
<terminated> SmallHeapDemo [Java Application] C:\Users\DevelopTrees\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1
Allocated 98400 MB
Allocated 98500 MB
Cleared temp list, memory should be freed.
Allocated 98600 MB
Allocated 98700 MB
Allocated 98800 MB
Allocated 98900 MB
Allocated 99000 MB
Cleared temp list, memory should be freed.
Allocated 99100 MB
Allocated 99200 MB
Allocated 99300 MB
Allocated 99400 MB
Allocated 99500 MB
Cleared temp list, memory should be freed.
Allocated 99600 MB
Allocated 99700 MB
Allocated 99800 MB
Allocated 99900 MB
```

6. Heap Dump Analysis (Eclipse MAT / jVisualVM)

1. Open dump in Eclipse MAT.
2. Run Leak Suspects Report.
3. You will see something like:

```
One instance of "java.util.ArrayList" occupies ~64MB (95% of total heap).
Path to GC Roots:
  -> static field MemoryLeakDemo.cache
```

Conclusion: The static cache is holding memory, which confirms a memory leak.