Identifying and Troubleshooting Memory Leaks in Java Applications

Lakshmi Narasimha Rohith Samudrala

Abstract

Memory leaks in Java applications is a challenge that programmers find it difficult to overcome. Although Java has an in-built memory management, its GC process fails to clear objects that are referenced. This causes programmers to rely on manual memory management techniques to prevent unintended object retention.

This paper dives into the memory segmentation of Java, while explaining the definition of memory leaks and what causes it. The paper explores few examples of codes that cause memory leaks and provides solutions to those problems. This paper explores practical methods for identifying and troubleshooting memory leaks using heap dumps, garbage collection logs, and profiling tools It also provides best practices which can help development team to prevent memory leaks. By implementing these strategies, developers can enhance application stability, optimize memory usage, and prevent performance degradation caused by excessive memory consumption.

Keywords: Java, Java Virtual Machine (JVM), Garbage Collection (GC), Memory Leak, Heap, Perm Gen, Stack Memory, Heap Dump, GC Log, JConsole, Thread Dump

I. INTRODUCTION

A core benefit of Java while using it for application development over languages like C and C++ is Java Virtual Machine (JVM). JVM is an out-of-the-box memory management [3]. Unlike languages like C and C++ where programmers would need to manually allocate and free memory, Java uses Garbage Collection (GC) to take care of the allocating and freeing up of memory. This, however, introduces the risk of memory leaks.

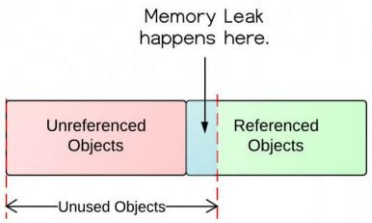

Figure 1 – Image showing memory leak concept [2]

Figure 2 – Garbage collection takes up to 16% of available CPU usage [4]

What is a Memory Leak:

Memory leak is a situation when object that are no longer being used by the application are still holding memory and GC is unable to remove these objects to free up memory [2]. This usually happens because GC can only remove the objects that are not referenced and if the code is still referencing some objects, GC will not remove them [2]. This causes the application to continuously grow its memory consumption while not releasing memory [3]. This eventually results in "OutOfMemory (OOM) Error". Figure1, visually demonstrates the concept of memory leak.

What problems do memory leaks cause:

In case of memory leaks, as unused objects are not removed and new objects are constantly created by the application, the memory consumption of the application slowly increases overtime [5]. This causes less memory to be available for active processes. This also causes JVM to trigger GC more frequently, but the triggered GC would not be successful as the objects are referenced. As the memory utilization remains high, the JVM will try to trigger GC again. This creates a loop, which in turn causes increase in CPU usage and slows down the application[5]. The application would finally get to a point where there is no more memory available and the application crashes unexpectedly. Figure 2 shows increase in suspension rate caused by GC [4].

II. UNDERSTANDING MEMORY AND MEMORY LEAKS IN JAVA.

The memory is Java is divided into different memory regions with in the JVM. The different regions are as follows:

1. Java Heap – Heap is a place where objects created by the Java applications are stored.This is where Garbage Collection takes place. Java Heap is the focus of memory leaks [1].
2. Perm Gen – Perm Gen is a place to store loaded class definition and metadata [1].
3. Stack Memory – Stack memory stores method call frames, local variables, and references to objects in the heap [1].
4. Garbage Collector (GC) – GC is a process that automatically reclaims memory occupied by objects that are no longer referenced.

Although JVM has GC, memory leaks can happen where objects are unintentionally retained in memory.This happens when an object is still reachable (i.e., referenced), but it is no longer needed. Here are some examples that can cause unused objects to still be referenced.

1. Collection Holding Reference: Objects stored in collections but never removed.

```
Map<Key, Value>myMap = newHashMap<>();
myMap.put(key, value);
// If key is not removed, it stays in memory indefinitely.
```

2. Static Field: Static fields keep objects alive if the class is loaded.

```
public class MemoryLeakExample {
    private static List<String>myList = new
ArrayList<>();
// Stays in memory permanently.
}
```

3. Improper Caching: Caches without eviction policies retain objects longer than needed.

```
public class CacheMemoryLeak {
    private static Map<String, String> cache = new
HashMap<>();

    public static void main(String[] args) {
        cache.put("user1", "data1");  // This object never
gets removed
    }
```

4. Listeners Not Removed: Event listeners and callbacks are not removed after use.

```
myButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clicked");
    }
});
```

III. IDENTIFYING MEMORY LEAKS

Memory Leaks are usually difficult to detect because they do not cause immediate failure to the application. Memory leak causes gradual increase in memory usage. The following symptoms can help identify memory leaks.

1. If the heap memory usage continuously grows without stabilizing, a memory leak is likely [4].
2. If the GC runs frequently but does not free memory, objects might still be referenced.
3. If the application frequently runs out of memory, it may be due to a memory leak.

Java provides several tools to monitor, profile, and analyze memory usage. These tools can be used to confirm memory leak on an application. Some of the tools are:

1. Heap Dumps: A heap dump is a snapshot of memory that shows all objects and references at a point in time. It helps in analyzing memory leaks.Figure 3 shows example of heap dump.
2. GC Logs: GC logs help determine if memory usage is abnormally high [4].

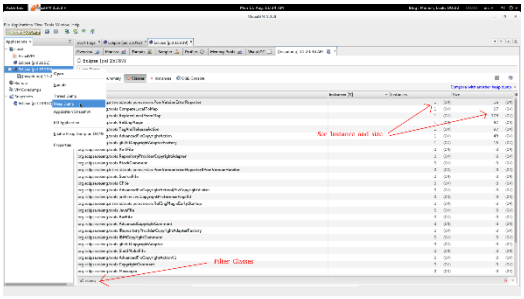

Figure 3 – Example Heap Dump

3. JConsole: JConsole (Java Monitoring & Management Console) provides real-time monitoring of heap usage [6].

4. Thread Dumps: Memory leaks sometimes originate from long-lived threads. A thread dump can show blocked or waiting threads that hold references to large objects [6].

IV. How To Prevent Memory Leaks

Memory leaks in Java can seriously impact applications, causing excessive memory usage, frequent garbage collection (GC), slow response times, and even system crashes. Although Java is good with managing its memory, poor coding practices can cause poor memory management and can still result in objects being retained longer than expected. These issues can be avoided by following some coding standards, some of these best practices are:

1. Avoid unused object references in collections, always make sure to clear the collections after use.

2. Remove unused static references, as static fields live if the application is running, leading to memory leaks if they hold large objects. To avoid this issue set static fields to null when no longer needed.

3. Always close resources, failing to close resources like database connections, file streams, and sockets causes memory leaks.

4. Remove event listeners after use, Swing/AWT applications often register ActionListeners, but if they are not removed, they retain references to UI components, preventing garbage collection.

By following these best practices, Java applications can prevent memory leaks and ensure stable memory usage.

V. Conclusion

Java's out-of-the-box memory management gives it an edge over languages like C and C++. But this also opens the risk of memory leaks. Usually, memory leaks cannot be easily identified. Some key symptoms can indicate the presence of memory leak. Upon identification of these memory leaks, tools like heap dumps, GC logs, and profiling tools can be used to identify the presence of memory leaks. These tools and techniques can also help narrow down the root-cause of the leak which can be fixed. Implementing best practices such as explicitly clearing collections, removing event listeners, and closing database connections can help avoid memory leaks from occurring.

VI. REFERENCES

[1] Y. M. Kim, "Find out your Java Heap memory size," mkyong.com, Mar. 10, 2014. https://mkyong.com/java/find-out-your-java-heap-memory-size/

[2] S. P. R. Janapati, "Understanding the Java memory model and garbage collection," dzone.com, Aug. 18, 2016. https://dzone.com/articles/understanding-the-java-memory-model-and-the-garbag

[3] "Research on memory leakage in Java application," IEEE Conference Publication | IEEE Xplore, Jul. 01, 2010. https://ieeexplore.ieee.org/document/5563623

[4] M. Goodwell, "How to identify a Java memory leak," Dynatrace News, Sep. 24, 2015. [Online]. Available: https://www.dynatrace.com/news/blog/how-to-identify-a-java-memory-leak/

[5] E. Paraschiv, "How memory leaks happen in a Java application," Stackify, Aug. 14, 2017. https://stackify.com/memory-leaks-java/

[6] L. Ufimtsev, "How to find and fix memory leaks in your Java application | Red Hat Developer," Red Hat Developer, Aug. 14, 2014. https://developers.redhat.com/blog/2014/08/14/find-fix-memory-leaks-java-application#