IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 1, January 2026

DOI: 10.17148/IARJSET.2026.13123

Memory Leak Detection in JVM-Based Applications

Prof. Varshitha C

Department of Computer Science, Bangalore City University, C.B. Bhandari Jain College

Abstract: Java applications run on the Java Virtual Machine (JVM), which provides automatic memory management through garbage collection. Despite this feature, memory leaks remain a critical problem in long-running JVM-based applications. A memory leak occurs when objects that are no longer required are unintentionally retained in memory, leading to increased heap usage and eventual application failure. This research paper explores the causes of memory leaks in JVM applications, techniques for detecting them, tools used for leak analysis, and preventive best practices. The paper emphasizes monitoring and profiling approaches to identify leaks efficiently and proposes a systematic methodology for memory leak detection.

1. INTRODUCTION

The Java Virtual Machine plays a central role in executing Java programs by managing runtime resources, especially memory. One of Java's main advantages is that developers are not required to manually allocate and free memory. Garbage collectors inside the JVM reclaim unused objects automatically. However, poorly written code can still cause objects to remain referenced, preventing garbage collection. Over time, this results in memory leaks.

Memory leaks in JVM-based systems are difficult to detect because the application may appear to function normally for a long time before performance gradually degrades. Enterprise systems such as web servers, cloud services, and big data platforms rely heavily on Java, making leak detection essential. Therefore, understanding JVM memory behavior and applying leak detection mechanisms is a major research concern.

2. JVM MEMORY MANAGEMENT OVERVIEW

JVM memory is divided into several runtime areas:

- Heap Area
- Stack Area
- Method Area
- Program Counter Register
- Native Method Stack

Among these, the **heap** is where Java objects are stored. The garbage collector identifies objects without active references and removes them. Modern JVMs use generational garbage collection, separating memory into Young Generation and Old Generation.

A memory leak does not mean that the JVM fails to free memory; rather, it means that the program logic mistakenly keeps references to unused objects. As a result, the garbage collector treats them as still active.

3. CAUSES OF MEMORY LEAKS IN JVM APPLICATIONS

Common reasons for Java memory leaks include:

1. Unclosed resources such as database connections or streams
2. Static collections holding object references
3. Improper implementation of caches
4. Listener objects not deregistered
5. Infinite object growth in session data
6. Incorrect equals() and hash Code() methods
7. Thread Local variables not cleared
8. Third-party library issues

These problems are mainly related to **object retention**. Identifying which objects are retained unnecessarily is the key to detecting leaks.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 1, January 2026

DOI: 10.17148/IARJSET.2026.13123

4. SYMPTOMS OF MEMORY LEAKS

A JVM-based application with memory leaks shows the following symptoms:

- Gradual increase in heap memory usage
- Frequent garbage collection cycles
- Decreased response time
- OutOfMemoryError: Java heap space
- High CPU consumption by GC
- Application crashes after long execution

Monitoring these runtime behaviours helps in suspecting the presence of leaks.

5. TECHNIQUES FOR MEMORY LEAK DETECTION

Memory leak detection in JVM applications can be performed using several approaches:

5.1 Heap Dump Analysis

A heap dump is a snapshot of JVM memory at a particular moment. By comparing multiple heap dumps taken at different times, developers can identify object growth patterns.

Steps:

- Trigger heap dump
- Analyse retained objects
- Identify GC roots
- Find unnecessary references

5.2 Profiling

Profilers monitor memory allocation in real time. They help track:

- Number of created objects
- Lifetime of objects
- Dominator trees
- Reference chains

5.3 Monitoring GC Logs

JVM GC logs provide valuable information about memory behavior:

- Allocation failures
- Heap occupancy after GC
- Pause times

Continuous growth even after full GC is a strong indicator of memory leaks.

5.4 Object Retention Graphs

Retention graphs show which objects prevent others from being garbage collected. This graph-based technique is widely used in leak detection research.

6. TOOLS USED FOR LEAK DETECTION

Popular JVM tools include:

Tool	Purpose
VisualVM	Monitoring and profiling
JConsole	Real-time heap monitoring
Eclipse MAT	Heap dump analysis
Java Mission Control	Advanced diagnostics
YourKit Profiler	Commercial profiling
JProfiler	Memory and CPU analysis

Among them, **Eclipse Memory Analyzer Tool (MAT)** is most effective for detecting leaks because it can display retained heap and reference chains precisely.

7. PROPOSED METHODOLOGY FOR DETECTION

This research proposes the following systematic process:

1. Continuous heap monitoring using JConsole

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 1, January 2026

DOI: 10.17148/IARJSET.2026.13123

2. Enabling detailed GC logging
3. Capturing periodic heap dumps
4. Analyzing dumps with MAT
5. Identifying top retained objects
6. Tracing reference chains
7. Modifying code and retesting

This iterative approach reduces detection time and improves accuracy.

8. CASE STUDY

Consider a Java web application where user session objects are stored in a static ArrayList and never removed. Monitoring shows heap usage increasing from 500MB to 1.5GB within hours.
Using MAT heap dump comparison:

- Session objects appear as top dominators
- GC roots point to static list
- Reference chain reveals unused user data

After clearing the collection properly, heap stabilizes. This case study demonstrates the importance of profiling-based leak detection.

9. BEST PRACTICES TO PREVENT LEAKS

- Avoid unnecessary static references
- Close all I/O and JDBC resources
- Use weak references in caches
- Clear ThreadLocal variables
- Deregister listeners
- Limit object scope
- Use proper collection cleanup
- Apply JVM tuning

10. CHALLENGES IN LEAK DETECTION

- Complexity of large heaps
- Overhead of profilers
- Difficulty in reproducing leaks
- Third-party library debugging
- Analyzing distributed JVM systems
- Dynamic object creation patterns

These challenges open further research directions.

11. FUTURE SCOPE

Future research can focus on:

- AI-based JVM leak prediction
- Automated leak detection frameworks
- Real-time leak alert systems
- Cloud-native JVM monitoring
- Hybrid static + dynamic analysis
- Leak detection in Java microservices

12. CONCLUSION

Although JVM provides automatic garbage collection, memory leaks continue to affect Java applications. Detecting leaks requires deep understanding of JVM memory structure, object retention, and profiling tools. This paper analysed major causes and detection techniques and presented a structured methodology using JVM monitoring and heap dump analysis. Proper leak detection improves performance, stability, and reliability of JVM-based applications. The study highlights that early monitoring and systematic analysis are essential for preventing critical memory failures.

IARJSET ISSN (O) 2393-8021, ISSN (P) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology

Impact Factor 8.311 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 13, Issue 1, January 2026

DOI: 10.17148/IARJSET.2026.13123

REFERENCES

[1]. Lindholm, T., Yellin, F., "The Java Virtual Machine Specification," Oracle
[2]. Eclipse Foundation, "Memory Analyzer Tool Documentation"
[3]. Oracle Docs, "Java Garbage Collection Tuning Guide"
[4]. Hunt, C., John, B., "Java Performance," Addison Wesley
[5]. Java Mission Control User Guide