

# Java Performance Optimization

## *Patterns and Anti-Patterns*

WRITTEN BY ANANT MISHRA

PRINCIPAL ENGINEER, PRAMATI TECHNOLOGIES

### CONTENTS

- Introduction
- Benefits of Performance Optimization
- Garbage Collection Performance Tuning
- Memory Leak Issues
- Java Concurrency
- Coding Practices
- Performance Monitoring & Troubleshooting Tools

The purpose of this Refcard is to help developers avoid performance issues while creating new applications and to improve performance of their existing applications. We will cover anti-patterns in Java application development and their respective patterns to follow for optimized performance. Lastly, a list of different tools for monitoring and troubleshooting application performance is provided for you to explore and, when you're ready, to begin applying some of the concepts covered in this Refcard in your own applications.

Please note that concepts of application performance optimization are advanced-level topics for the given language — in this case, Java. So intermediate knowledge of Java and its internal architecture are prerequisites for this Refcard. Here are resources to revisit the required concepts:

- [Introduction of Java](#)
- [Java Virtual Machine \(JVM\)](#)
- [JIT Compiler](#)

### BENEFITS OF PERFORMANCE OPTIMIZATION

Poor performance of applications can cause delayed responses to the end user and unoptimized use of servers and other resources. Ultimately, these issues can impact user experience and the cost of running an application.

Optimizing performance of your Java applications will improve:

- **Latency**, enhancing user experience.
- **Application adoption**, multiplying overall ROI.
- **Code quality**, resulting in reduced downtime.
- **Server efficiency**, reducing total infrastructure costs.
- **SEO**, increasing search rankings.

The following sections in this Refcard cover the main anti-patterns and coding mistakes in Java applications that can result in the degradation of application performance.



**Site24x7**

**Achieve 2x improvement in your Java application performance.**

**Learn More**

The banner features a dark blue background with a central illustration of a laptop displaying a Java logo. Surrounding the laptop are various icons: a red warning triangle, a database cylinder, a gear, a speedometer, and a group of people. The text is white and green, with a prominent green 'Learn More' button at the bottom.

# Site24x7

## Gain code-level visibility in to the performance of your Java applications.

Optimize user experience with AI-powered, real-time alerts

With Site24x7 APM Insight, you can

- Track the performance of JVM components
- Monitor custom components
- Eliminate potential bottlenecks

Get Started



## GARBAGE COLLECTION PERFORMANCE TUNING

The Java garbage collection process is one of the most important contributing factors for optimal application performance. In order to provide efficient garbage collection, the heap is essentially divided into two sub areas: young generation (nursery space) and old generation (tenured space).

[Choosing the right garbage collector](#) for your application is a determinant factor for optimal application performance, scalability, and reliability. The GC algorithms are included in the table below:

GC TYPE	DESCRIPTION
Serial Collector	<ul style="list-style-type: none"> <li>Has the smallest footprint of any collector</li> <li>Runs with a footprint that requires a small number of data structures</li> <li>Uses a single thread for minor &amp; major collections</li> </ul>
Parallel Collector	<ul style="list-style-type: none"> <li>Stops all app threads &amp; executes garbage collection</li> <li>Best suited for apps that run on multicore systems &amp; need better throughput</li> </ul>
Concurrent Mark-Sweep Collector	<ul style="list-style-type: none"> <li>Has less throughput, but smaller pauses, than the parallel collector</li> <li>Best suited for all general Java applications</li> </ul>
Garbage-First (G1) Collector	<ul style="list-style-type: none"> <li>Is an improvement from the CMS collector</li> <li>Uses entire heap, divides it into multiple regions</li> </ul>

### GC PERFORMANCE CONSIDERATIONS

When evaluating application performance, most of the time you won't have to worry about configuring garbage collection. However, if you are seeing that your application performance degrades significantly during garbage collection, you do have a lot of control over garbage collection. Concepts like generation size, throughput, pauses, footprint, and promptness can all be tuned.

For more information on performance metric types, please refer to [Oracle's Java 8 documentation](#) on GC tuning.

### GC MONITORING AND TUNING

There are two steps to monitor and troubleshoot GC issues:

#### 1. GENERATING GC LOGS

There are multiple options available for setting up and enabling GC logging. Details are present in [JEP 158: Unified JVM Logging](#).

#### Enabling GC Logging (Java 11) on Unix & Linux Systems

For Tomcat containers, GC logging can be enabled by specifying `CATALINA_OPTS` settings in the `setenv.sh` file (typically located in the `/tomcat/bin/directory`).

If this file doesn't exist, then create it in the same directory as the `catalina.sh` file (also typically located in the `/tomcat/bin/directory`).

To enable GC logging, add a line similar to the one provided below to the `setenv.sh` file and restart the web container:

```
export CATALINA_OPTS="$CATALINA_OPTS
-Xlog:gc=info:
file=/tmp/logs/gc.log:time,uptime,level,tags:
filecount=10,filesize=128m"
```

#### Enabling GC Logging (Java 8 and earlier) on Unix & Linux Systems

For Tomcat containers, you should enable GC logging by specifying `CATALINA_OPTS` settings in the `setenv.sh` file (typically located in the `/tomcat/bin/directory`).

If this file doesn't exist, then create it in the same directory as the `catalina.sh` file (also typically located in the `/tomcat/bin/directory`).

To enable GC logging, add a line similar to the one provided below to the `setenv.sh` file and restart the web container:

```
export CATALINA_OPTS="$CATALINA_OPTS -verbose:gc
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps
-XX:+PrintGCDateStamps -XX:+PrintGCCause
-Xloggc:[filename]"
```

### 2. ANALYZING GC LOGS

It is hard to analyze GC logs manually, as they are in simple text format. So it is better to use a GC analysis tool. There are multiple tools available online, such as [GCeasy](#).

## MEMORY LEAK ISSUES

A memory leak occurs when there are objects present in the heap that are no longer used, but the garbage collector cannot identify that they are unused. Thus, they are unnecessarily maintained. Memory leaks can exhaust memory resources and degrade system performance over time. It is possible for the application to terminate with a fatal `java.lang.OutOfMemoryError`.

### OVERUSE OF STATIC FIELDS

The lifetime of a static field is the same as the lifetime of the class in which that field is present. So the garbage collector does not collect static fields unless the `ClassLoader` of that class itself becomes eligible for garbage collection.

If there is a collection of objects (e.g., `List`) marked static — and throughout the execution of the application objects are added to that `List` — the application will retain all of those objects even if they are no longer in use. Such scenarios can lead to a `java.lang.OutOfMemoryError`.

Let's create a simple Java program that populates a static List:

```
import java.util.ArrayList;
import java.util.List;

public class StaticCollectionTest {
    public static List<Student> list = new
        ArrayList<>();

    public void addStudentsToList() {
        for (int i = 0; i < 1E7; i++) {
            list.add(new Student("studentName_" + i));
        }
    }

    public static void main(String[] args) {
        new StaticCollectionTest().addNumbersToList();
    }
}

class Student {
    private String name;

    public Student(String name) {
        this.name = name;
    }
}
```

In this example, all 10 million objects added to the `List` will remain in memory, even after execution of the `addStudentsToList` method finishes. Scenarios such as this can exhaust memory resources. The decision to use static keywords with collection objects should be made carefully.

## ThreadLocal OBJECTS WITH ThreadPools

[ThreadLocal](#) allows the storage of data that will be accessible only by a specific thread. Data will be hidden from other threads. [Thread pools](#) in application servers are implemented to create a pool of threads. Some threads in thread pools are never garbage collected and get reused. Normally, `ThreadLocal` objects are garbage collected after the execution of a thread. However, it is possible that threads in a thread pool will never be garbage collected.

In those cases, `ThreadLocal` objects will lead to memory leaks, as the thread object will remain referenced within the thread pool. And `ThreadLocal` objects will not be garbage collected, even after the thread execution completes. They will be returned to the thread pool. This causes memory leaks of `ThreadLocal` objects. Extending the `ThreadPoolExecutor` class and removing the `ThreadLocal` data using the `afterExecute()` method can help avoid `ThreadLocal` memory leaks:

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadLocalAwareThreadPool extends
    ThreadPoolExecutor {

    public ThreadLocalAwareThreadPool(
```

```
int corePoolSize, int maximumPoolSize,
long keepAliveTime,
TimeUnit unit,
BlockingQueue<Runnable> workQueue) {
    super(corePoolSize, maximumPoolSize,
        keepAliveTime, unit, workQueue);
}

@Override
protected void afterExecute(Runnable runnable,
    Throwable throwable) {
    /* Call remove method for each ThreadLocal
    object of given thread*/
}
}
```

## USE OF NON-STATIC INNER CLASSES

Initialization of a non-static inner class always requires an instance of the outer class. Non-static inner classes have, by default, an implicit reference to their containing classes. It is possible that only the inner class is referenced, and that the outer class object is out of scope. In this scenario, the outer class is not eligible for garbage collection because the inner class object implicitly holds a reference to the outer class, and the inner class is referenced by another object. If the outer class consists of objects with large memory footprints, those objects will not be garbage collected and will continue to hold a large memory footprint.

```
import java.util.ArrayList;
import java.util.List;

class OuterClass {

    private int[] array = new int[1E6];

    class InnerClass {
        public void printFromInnerClass(int count)
        {
            System.out.println("printFromInnerClass
                method call count: " + count);
        }
    }
}

public class NonStaticClassMemoryLeakDemo {

    public static void main(String args[]) {
        List al = new ArrayList<>();
        int counter = 1;
        while (counter <= 1000) {
            OuterClass.InnerClass innerClass =
                new OuterClass().new InnerClass();
            innerClass.printFromInnerClass(counter);
            al.add(innerClass);
            counter++;
        }
    }
}
```

In this example, we need to add 1,000 inner objects in the list. While the `InnerClass` is lightweight and 1,000 objects isn't a large number, the `OuterClass` has a heavy array object.

CODE CONTINUES IN NEXT COLUMN

The `InnerClass` maintains the reference of the `OuterClass`, so this heavy array object will not be garbage collected, which is why this program will throw a `java.lang.OutOfMemoryError`. To avoid this issue, inner classes should be static, unless access to the containing class members is required.

## UNCLOSED RESOURCES

There are database connections, input streams, and a few other objects that need a considerable amount of memory and resources at the time of creation. These types of objects should close explicitly as soon as their work completes. Otherwise, the memory that the JVM allocates to them will be blocked. This can deteriorate performance and may even result in an `OutOfMemoryError`.

To avoid this issue, resources should be closed in a `finally` block, or resources should be created using `try-with-resources` statements (introduced in Java 7, see the [Java™ Tutorials](#) for more information).

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class NonStaticClassMemoryLeakDemo {

    private static void closeInFinally() throws
        IOException {
        InputStream inputStream = null;

        try {
            inputStream = new FileInputStream(
                "file.txt");

            int data = inputStream.read();
            while(data != -1){
                System.out.print((char) data);
                data = inputStream.read();
            }
        } finally {
            if(inputStream != null){
                inputStream.close();
            }
        }
    }

    private static void closeUsingTryWithResources()
    throws IOException {

        try (InputStream inputStream = new
        FileInputStream("file.txt")) {
            int data = inputStream.read();
            while(data != -1){
                System.out.print((char) data);
                data = inputStream.read();
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) {
        try {
```

CODE CONTINUES IN NEXT COLUMN

```
        closeInFinally();
        closeUsingTryWithResources();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

## finalize() METHOD IMPLEMENTATION

An object's `finalize()` method is called during GC. The issue here is that garbage collection cannot be controlled, as it happens when the JVM determines it should happen. So if there is code to free a resource or close a connection, it will be alive until the point GC happens and `finalize()` is executed. Another issue is that the garbage collection algorithm is JVM-implementation dependent, so code may run well on one system while behaving differently and creating performance issues on another system.

Custom implementation of the `finalize()` method should be avoided unless actually required.

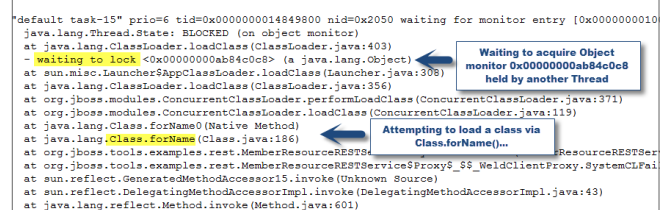
## JAVA CONCURRENCY

Java concurrency can be defined as the ability to execute several tasks of a program in parallel. For large Java EE systems, this means the capability to execute multiple user business functions concurrently while achieving optimal throughput and performance. Regardless of your hardware capacity or the health of your JVM, Java concurrency problems can bring any application to its knees and severely affect the overall application performance and availability.

### THREAD LOCK CONTENTION

Thread lock contention is by far the most common Java concurrency problem you will observe when assessing the concurrent threads health of your Java application. This problem manifests itself by the presence of 1...n BLOCKED threads (thread waiting chain) waiting to acquire a lock on a particular object monitor. Depending on the severity of the issue, lock contention can severely affect application response time and service availability.

**Example:** Thread lock contention triggered by non-stop attempts to load a missing Java class (`ClassNotFoundException`) to the default JDK 1.7 `ClassLoader`.



```

"default task-15" prio=6 tid=0x00000000014649800 nid=0x2050 waiting for monitor entry 0x00000000014649800
  java.lang.Thread.State: BLOCKED (on object monitor)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:403)
    - waiting to lock <0x000000000ab84c0c8> (a java.lang.Object)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:356)
    at org.jboss.modules.ConcurrentClassLoader.performLoadClass(ConcurrentClassLoader.java:371)
    at org.jboss.modules.ConcurrentClassLoader.loadClass(ConcurrentClassLoader.java:119)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:186)
    at org.jboss.tools.examples.rest.MemberResourceRESTServiceProxy$$_WeldClientProxy.SystemCLFail
    at org.jboss.tools.examples.rest.MemberResourceRESTServiceProxy$$_WeldClientProxy.SystemCLFail
    at sun.reflect.GeneratedMethodAccessor15.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    .....

```

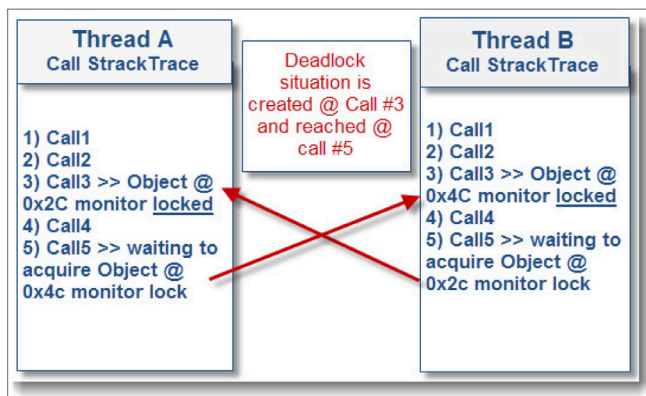


It is highly recommended that you assess the presence of such a problem in your environment via proven techniques such as thread dump analysis. Typical root causes of this issue can vary from abuse of plain old Java synchronization to legitimate IO blocking or other non-thread-safe calls.

Lock contention problems are often “symptoms” of another problem.

## JAVA-LEVEL DEADLOCKS

True Java-level deadlocks, while less common, can also greatly affect the performance and scalability of an application. They are triggered when two or more threads are blocked forever, waiting for each other. This situation is very different from other more common “day-to-day” thread problems like lock contention, threads waiting on blocking IO calls, etc. A true lock-ordering deadlock can be visualized below:



The Oracle HotSpot and IBM JVM implementations provide **deadlock detectors** for most scenarios, allowing you to quickly identify the culprit threads involved in such conditions. Similar to lock contention troubleshooting, it is recommended to use techniques like thread dump analysis as a starting point.

Found one Java-level deadlock:  
=====
"pool-1-thread-5":
waiting to lock monitor 0x04d29604 (object 0x2705cc50, a eu.javaspec:
which is held by "pool-1-thread-1"
"pool-1-thread-1":
waiting to lock monitor 0x04d2896c (object 0x2705cc58, a eu.javaspec:
which is held by "pool-1-thread-2"
"pool-1-thread-2":
waiting to lock monitor 0x0256e45c (object 0x2705cc60, a eu.javaspec:
which is held by "pool-1-thread-3"
"pool-1-thread-3":
waiting to lock monitor 0x0256e3f4 (object 0x2705cc68, a eu.javaspec:
which is held by "pool-1-thread-4"
"pool-1-thread-4":
waiting to lock monitor 0x04d2966c (object 0x2705cc70, a eu.javaspec:
which is held by "pool-1-thread-5"

**Java-level deadlock detected!**

Once the culprit code is identified, solutions involve addressing the lock-ordering conditions and/or using other available concurrency programming techniques from the JDK such as `java.util.concurrent.locks.ReentrantLock`, which provides methods like `tryLock()`. This approach gives Java developers much more flexibility and ways to prevent deadlock or thread lock “starvation.”

## CLOCK TIME AND CPU BURN

In parallel with JVM tuning, it is also essential for you to review your application's behavior — more precisely, the highest clock time and CPU burn contributors. When the Java garbage collection and thread concurrency are no longer a pressure point, it is important to drill down into your application code execution patterns and focus on the top response time contributors, referred to as clock time.

It is also crucial to review the CPU consumption of your application code and Java threads (CPU burn). High CPU utilization (> 75%) should not be assumed to be “normal” (good physical resource utilization). It is often the symptom of inefficient implementation and/or capacity problems.

For large Java EE enterprise applications, it is essential to keep a safe CPU buffer zone in order to deal with unexpected load surges.

Stay away from traditional tracing approaches such as adding response time “logging” in your code. Java profiler tools and APM solutions exist precisely to help you with this type of analysis and in a much more efficient, reliable way. For Java production environments that lack a robust APM solution, you can still rely on tools such as Java VisualVM, thread dump analysis (via multiple snapshots), and OS CPU per Thread analysis.

Finally, do not try to address all problems at the same time. Start by building a list of your top five clock time and CPU burn contributors and explore possible solutions.

## CODING PRACTICES

A few basic rules should be followed while writing Java code in order to avoid performance issues.

### IN-PROCESS CACHING

In Java applications, the caching of objects is done to avoid multiple database or network calls. These objects can be cached to the in-process memory (an in-process cache) of the application or to the external memory (a distributed cache). You can learn more about this in the article “[In-Process Caching vs. Distributed Caching](#)” on DZone.

An in-process cache uses the JVM memory to store objects. This is why storing large objects or many objects in such a cache can lead to memory consumption and an `OutOfMemoryError`.

Below are a few ways to avoid this:

- Do not store objects whose count can't be controlled at the application's runtime — for example, the number of active user sessions. In this scenario, an application may crash due to an `OutOfMemoryError` if the number of active sessions increases.

- Avoid storing heavy objects, as these objects will consume a lot of JVM memory.
- Store only those objects that need to be accessed frequently. Storing them in a distributed cache can create significant performance degradation due to multiple network calls.
- Configure the cache eviction policy carefully. Unnecessary objects should not be in the cache and should be evicted.

## EXCESSIVE LOGGING

Logging statements trigger I/O operations to log the statements to external files. Unnecessary logging statements are an overhead on I/O operations. Although time logging libraries have evolved and this is no longer a significant issue, it is still recommended to think before writing a log statement and its logging level in code.

## USING CORRECT DATA STRUCTURES

Different data structures such as `ArrayList`, `LinkedList`, `HashMap`, and `TreeMap` are optimized for specific kinds of operations. So before choosing any data structure, consider the [time complexity](#) of that data structure for a required operation.

## APP PERFORMANCE MONITORING & TROUBLESHOOTING TOOLS

TOOL	DESCRIPTION
<a href="#">Apache JMeter</a>	Apache JMeter can be used to: <ul style="list-style-type: none"> <li>• Test performance both on static &amp; dynamic resources and Web-dynamic applications.</li> <li>• Simulate a heavy load on a server, group of servers, network, or object to test its strength or to analyze overall performance under different load types.</li> </ul>
<a href="#">VisualVM</a>	Features of VisualVM: <ul style="list-style-type: none"> <li>• Display local &amp; remote Java processes</li> <li>• Monitor process performance &amp; memory</li> <li>• Visualize process threads</li> <li>• Profile performance &amp; memory usage</li> <li>• Take &amp; display thread dumps</li> <li>• Take &amp; browse heap dumps</li> <li>• Analyze core dumps</li> </ul>
<a href="#">JProfiler</a>	JProfiler's intuitive GUI helps you: <ul style="list-style-type: none"> <li>• Find performance bottlenecks.</li> <li>• Pin down memory leaks &amp; resolve threading issues.</li> </ul>
<a href="#">GCeasy</a>	GCEasy uncovers memory problems in your application. The tool can detect if your application: <ul style="list-style-type: none"> <li>• Is suffering from memory leaks.</li> <li>• Might suffer from memory leaks in the near future.</li> <li>• It experiences long GC pauses.</li> </ul>

### UPDATED BY ANANT MISHRA,

PRINCIPAL ENGINEER, PRAMATI TECHNOLOGIES

Anant is a Java-based web application developer. He has been working on SOA and Microservices architecture-based Java applications for the last seven years. He enjoys solving architectural and scalability issues in applications. He has worked on Java, Spring, JPA, RESTful web services, microservice components, publisher-subscriber models, Caching, ELK, Containerization, and CI/CD pipeline implementation.



DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.  
 600 Park Offices Drive  
 Suite 150  
 Research Triangle Park, NC 27709  
 888.678.0399 919.678.0300

Copyright © 2020 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.