The Java Class Loader: Unveiling the JVM's Dynamic Core

I. Introduction: The Unseen Engine of Java

The Java Virtual Machine (JVM) is renowned for its "Write Once, Run Anywhere" (WORA) capability, a promise underpinned by several sophisticated internal mechanisms. Among the most critical of these is the Java Class Loader. This report delves into the intricacies of the Java Class Loader, elucidating its fundamental purpose, its operational mechanics, and its pivotal role within the broader Java ecosystem. For a comprehensive understanding, it is essential to first distinguish between the core components of the Java environment: the Java Development Kit (JDK), the Java Runtime Environment (JRE), and the Java Virtual Machine (JVM).

What is a Java Class Loader?

At its essence, a Java Class Loader is an integral part of the Java Runtime Environment (JRE) responsible for dynamically loading Java classes into the Java Virtual Machine (JVM).[1] This dynamic loading paradigm dictates that classes are typically loaded only when they are required for program execution, rather than being loaded all at once at startup.[1] This on-demand approach significantly contributes to Java's efficiency and responsiveness.

The class loader serves as a crucial intermediary, abstracting away the complexities associated with file systems and binary representations of classes. This delegation allows the JVM to concentrate solely on the execution of bytecode.[1] In the Java language, software libraries—collections of related object code—are commonly packaged within JAR (Java Archive) files. The class loader's responsibilities extend to locating these JAR files, reading their contents, and extracting the necessary

.class files for loading.[1] A fundamental principle governing class loading is that a class

with a given name can only be loaded once by a specific class loader.[1] Every Java class, whether it originates from the core Java libraries or from user-defined applications, must be loaded by a class loader.[1]

Why Class Loaders are Essential for Java's Dynamic Nature

The existence and design of Java Class Loaders are not arbitrary; they are a fundamental design choice that enables Java's core promise of "Write Once, Run Anywhere." Without the class loader handling the dynamic loading of bytecode into the JVM's memory, the JRE would be unable to execute programs, and the JVM would function as a static, less flexible engine. The class loader is the mechanism that translates the abstract concept of running bytecode on a virtual machine into a concrete reality. This mechanism provides several profound benefits:

- **Dynamic Loading:** Class loaders enable classes to be loaded on demand, which substantially reduces the initial memory footprint and improves application startup time.[2] This dynamic capability is central to Java's WORA philosophy.
- **Modularity:** They facilitate the loading of classes from different modules, thereby supporting and enhancing modular application design.[2]
- **Security & Isolation:** Class loaders play a vital role in Java's security model. They allow for sandboxing, preventing untrusted code from interfering with core system classes or other application components.[2]
- **Extensibility:** They form the backbone for advanced features such as hot deployment, plugin architectures, and dynamic code generation, allowing applications to extend or modify their functionality at runtime without requiring a full restart.[1]

Briefly Setting the Stage: JVM, JRE, and JDK – A Quick Distinction

To fully appreciate the role of the Class Loader, it is helpful to clarify the relationship between the JDK, JRE, and JVM. These components work in concert to enable Java application development and execution.

- **JDK (Java Development Kit):** This is the comprehensive software development environment for Java developers. It includes the JRE along with essential

development tools such as the Java compiler (javac), a debugger, and various other utilities necessary for writing and compiling Java code.[9] The JDK is platform-dependent, meaning specific installers are required for different operating systems (e.g., Windows, macOS, Linux).[9]

- **JRE (Java Runtime Environment):** The JRE is designed for end-users who primarily need to run Java applications. It provides the necessary environment to execute Java programs and comprises the JVM, Java binaries, and core libraries (such as rt.jar in older versions).[9] Like the JDK, the JRE is also platform-dependent.[9]
- **JVM (Java Virtual Machine):** The JVM is the core component responsible for executing Java bytecode. It is the "virtual" machine that provides the platform independence for Java bytecode. While the JVM *implementation* itself is platform-dependent (requiring different versions for Windows, Linux, macOS), the bytecode (.class files) generated by the Java compiler is platform-independent, allowing it to run on any machine equipped with a compatible JVM.[9] The Class Loader is an integral and indispensable part of the JVM.[9]

The deep understanding of Java's runtime behavior, particularly concerning performance or memory issues, necessitates a clear grasp of the Class Loader, as it directly influences which code is available and how it is managed within the JVM.

Table 1: JDK vs. JRE vs. JVM Comparison

Aspect	JDK (Java Development Kit)	JRE (Java Runtime Environment)	JVM (Java Virtual Machine)
Purpose	Used to develop Java applications	Used to run Java applications	Executes Java bytecode
Platform Dependency	Platform-dependent (OS specific)	Platform-dependent (OS specific)	JVM implementation is OS-specific; bytecode is platform-independent [9]
Includes	JRE + Development tools (javac, debugger)	JVM + Libraries (e.g., rt.jar)	Class Loader, JIT Compiler, Garbage Collector [9]
Use Case	Writing and compiling Java code	Running a Java application on a system	Converts bytecode into native machine code [9]

II. The Core Trio: Built-in Class Loaders

Within the Java ecosystem, three primary built-in Class Loaders operate in a hierarchical fashion, each with distinct responsibilities crucial for the proper functioning and security of Java applications. Understanding this hierarchy and the role of each loader is fundamental to comprehending how Java manages its runtime environment.

The Bootstrap Class Loader: The JVM's Foundation

The Bootstrap Class Loader, often referred to as the primordial or root class loader, is foundational to the JVM itself.[2] It is responsible for loading the absolute core Java libraries, including fundamental classes like

java.lang.Object and other essential components of the Java standard library.[2] In older JDK versions, these classes were typically found in

rt.jar; in Java 9 and later, they reside within jmods directories.[1]

A unique characteristic of the Bootstrap Class Loader is its implementation: it is written in native code, not Java, and is an intrinsic part of the core JVM.[1] Consequently, it is not associated with any

java.lang.ClassLoader object. This means that invoking getClassLoader() on a class loaded by the Bootstrap loader will return null.[1]

The Extension/Platform Class Loader: Expanding Core Capabilities

The role of this class loader has evolved with Java versions.

- **Pre-Java 9 (Extension Class Loader):** Historically, this loader was known as the

Extension Class Loader. Its primary function was to load classes from the JDK's extension directories, typically located at $JAVA_HOME/jre/lib/ext, or any other directory specified by the java.ext.dirs system property.[1]

- **Java 9+ (Platform Class Loader):** With the introduction of the Java Platform Module System (JPMS) in Java 9, the Extension Class Loader was superseded by the Platform Class Loader.[1] This loader is specifically tasked with loading "platform classes," which encompass Java SE platform APIs, their implementation classes, and JDK-specific runtime classes.[12] The Platform Class Loader acts as a child of the Bootstrap Class Loader and can serve as a parent for other ClassLoader instances.[3]

The System/Application Class Loader: Your Code's Gateway

Also known as the Application Class Loader, this is the class loader that handles the loading of application-specific classes.[2] It loads classes from the classpath, which is typically specified when launching a Java application using command-line options like

-cp or -classpath, or via the CLASSPATH environment variable.[1] This classpath can include various resources such as directories and JAR files.[3] The Platform Class Loader (or Extension Class Loader in older Java versions) is either the direct parent or an ancestor of the System Class Loader.[3] When a Java application begins execution, the class containing the

main method is typically loaded by the System Class Loader.[3]

Understanding Their Hierarchy and Responsibilities

The three built-in class loaders operate within a strict hierarchical relationship: the Bootstrap Class Loader is the parent of the Extension/Platform Class Loader, which in turn is the parent of the System/Application Class Loader.[2] This hierarchy is not merely for organizational purposes; it is a fundamental security and stability mechanism. By delegating class loading requests upwards, the JVM ensures that core Java classes (like

java.lang.String or java.util.List) are always loaded by the highly trusted Bootstrap or

Extension/Platform class loaders. This design prevents malicious or poorly written application code, which is loaded by the System Class Loader, from overriding critical Java API classes. Such an override could lead to severe security vulnerabilities or runtime instability. This structure establishes a "trust chain" where higher-level loaders are responsible for more critical, immutable components of the Java runtime. This design directly contributes to Java's robustness and security, making it particularly suitable for enterprise applications and environments where code from various sources, some potentially untrusted, might coexist. It also explains why attempting to place a custom java.lang.String class on the application classpath will not result in it being used by the core JVM.

Table 2: Built-in Class Loaders: Hierarchy and Responsibilities

Class Loader Name	Parent	Child	Primary Responsibility	Typical Path/Source	Example Class
Bootstrap	None (Root)	Extension/Platform	Loads core Java libraries	<JAVA_HOME>/jre/lib or <JAVA_HOME>/jmods [1]	java.lang.Object, java.util.HashMap [11]
Extension/Platform	Bootstrap	System/Application	Loads JDK extension/platform classes	$JAVA_HOME/jre/lib/ext (pre-Java 9), Platform APIs (Java 9+) [1]	DNSNameService [11]
System/Application	Extension/Platform	None (Application-specific)	Loads application-specific classes	Application classpath (-cp, CLASSPATH) [1]	Your Main class, JDBC drivers [11]

III. The Delegation Model: How Classes Are Found

The parent-first delegation model is a cornerstone of Java's class loading mechanism, dictating how class loaders interact to locate and load classes. This model is

fundamental to ensuring consistency, security, and efficient resource management within the JVM.

Parent-First Principle: Ensuring Consistency and Security

When a class loader receives a request to load a class or resource, its default behavior is to first delegate this request to its parent class loader.[2] This delegation process continues recursively up the hierarchy, from the System Class Loader to the Extension/Platform Class Loader, and finally to the Bootstrap Class Loader.[5] Only if the parent (and its ancestors) cannot find or load the requested class does the current class loader attempt to find and load it from its own defined locations.[5]

This model is critical for maintaining consistency and security across the JVM.[2] By prioritizing core Java API classes to be loaded by the Bootstrap/Platform loaders, it effectively prevents application-level classes from inadvertently or maliciously overriding these fundamental components. This ensures that the JVM operates with a consistent set of core functionalities, safeguarding against potential runtime instability or security vulnerabilities that could arise from class conflicts.

The loadClass() Method: The Heart of Delegation

The java.lang.ClassLoader is an abstract class that provides the blueprint for all class loaders in Java.[12] The

loadClass(String name) method within this class serves as the primary entry point for all class loading requests.[5]

The default implementation of loadClass() adheres to a specific sequence of operations:

1. **Check for Already Loaded Class:** It first verifies if the class has already been loaded by invoking findLoadedClass(String name). If a Class object for the given name is found, it is immediately returned, preventing redundant loading.[5]
2. **Delegate to Parent:** If the class is not already loaded, the request is then delegated to its parent class loader via a call to parent.loadClass(name). If the

parent is null (indicating the Bootstrap loader), the JVM's built-in mechanism handles the search.[5]

3. **Find and Load Locally:** Only if the parent (and its ancestors) are unable to find the class, the current class loader's findClass(String name) method is invoked. This method is where the specific logic for locating the binary data of the class (e.g., from a JAR file, a directory, or a network location) resides.[5]
4. **Define Class:** Once the binary data (typically as a byte array) is obtained, the defineClass() method is used to convert this byte array into a java.lang.Class object. This step also involves the linking phase (verification, preparation, and resolution).[11]
5. **Resolve Class (Optional):** Optionally, resolveClass() can be called to explicitly link the class immediately. However, for performance reasons, this linking step is often delayed until the class is actively used.[15]

Uniqueness in the JVM: Class Identity (ClassLoader + Class Name)

A critical concept in Java's class loading model is how classes are uniquely identified within the JVM. A class is not merely identified by its fully qualified name (e.g., java.lang.String); rather, its identity is uniquely defined by the combination of its fully qualified name *and* the specific ClassLoader instance that loaded it.[3] This implies that

Class A loaded by ClassLoader X is considered a distinct class from Class A loaded by ClassLoader Y, even if their bytecode is absolutely identical.[2]

This "uniqueness" feature is not a limitation but a deliberate design choice that underpins Java's advanced modularity and isolation capabilities. If class identity were solely based on name, running multiple web applications within the same application server (e.g., Tomcat) that both depend on different versions of the same library (e.g., Log4j 1.x vs. Log4j 2.x) would lead to severe conflicts, commonly known as "dependency hell." By associating the class with its loader, the JVM effectively creates distinct "namespaces" for classes. This allows applications to maintain their own isolated sets of dependencies, even if those dependencies share identical class names, thereby preventing version conflicts.[2] This concept is vital for understanding how application servers, OSGi frameworks, and modern module systems manage complex deployments and achieve features like hot-swapping or dynamic plugin loading. It also provides the explanation for why a

ClassCastException might occur even when two objects appear to be of the "same" class, if they were loaded by different class loaders.

IV. The Class Loading Lifecycle: From Bytes to Objects

The process of making a Java type (which can be a class or an interface) available to a running program is a meticulously choreographed sequence of events involving three main phases: Loading, Linking, and Initialization.[3] While the overall order of these phases is fixed, the resolution sub-phase of linking can be strategically delayed for performance benefits.[18]

Phase 1: Loading (Reading the Bytecode)

Loading is the initial step where the class loader reads the binary representation of a class—typically a .class file—and brings it into the JVM's memory.[2] During this phase, the JVM undertakes three primary activities:

1. **Locating Binary Data:** Given the fully qualified name of the class, the class loader must produce a stream of binary data that constitutes the definition for that class. This data can originate from diverse sources, including local file systems, network locations (e.g., HTTP, FTP), JAR or ZIP archives, proprietary databases, or even dynamically generated bytecode.[19]
2. **Parsing Data:** The acquired binary data stream is then parsed and transformed into internal, implementation-dependent data structures within the JVM's method area.[19]
3. **Creating java.lang.Class Instance:** An instance of the java.lang.Class class is created to represent the loaded type.[2] This
   Class object serves as a programmatic interface to the loaded class, enabling runtime inspection and manipulation through Java's reflection API.

Phase 2: Linking (Verification, Preparation, Resolution)

Linking is the crucial process of integrating the binary class data into the runtime state of the virtual machine. It ensures that the class is properly formed, adheres to Java's semantic rules, and is ready for execution.[2] This phase is subdivided into three sequential steps:

- Bytecode Verification: Ensuring Integrity
  This sub-step is paramount for security and stability. It rigorously ensures that the binary representation of the class is structurally correct and does not violate any Java language rules, thereby preventing corrupted or malicious code from compromising the JVM's integrity.2 Checks performed include: verifying that final classes are not subclassed, final methods are not overridden, constant pool entries are consistent, and the bytecode itself is well-formed (e.g., jump instructions do not lead outside a method's boundaries).19 If verification fails, a VerifyError is thrown.[3] While verification can be disabled using the -noverify JVM option, this practice significantly undermines Java's inherent safety and security guarantees.[3]
- Preparation: Allocating Memory for Statics
  In this phase, the JVM allocates memory for all static fields (also known as class variables) declared within the class and initializes them with their default values.2 For example, numeric types are initialized to 0, booleans to false, and object references to null.[19] No Java code is executed during preparation; it is purely a memory allocation and default value assignment step. The JVM may also allocate memory for performance-enhancing data structures, such as method tables, during this phase.[19]
- Resolution: Connecting Symbolic References
  Java bytecode uses symbolic references (e.g., the textual name and signature of a method or field) rather than direct memory addresses. Resolution is the process of translating these symbolic references, which are stored in the class's constant pool, into direct references that the JVM can use to locate the actual memory locations of classes, interfaces, fields, and methods.2 This step can be performed "eagerly" (all at once during linking) or "lazily" (on demand, when a symbolic reference is first encountered and used by the running program).3 Modern JVMs often employ lazy resolution for performance optimization. If a class or member cannot be resolved (e.g., a referenced class is missing from the classpath or an incompatible version is found), a LinkageError or one of its subtypes, such as NoClassDefFoundError, can occur.[20]

Phase 3: Initialization (Executing Static Blocks)

Initialization is the final phase, where the class variables are assigned their "proper" initial values as defined by the programmer.[2] These values are specified in Java code through static field initializers (e.g.,

static int count = 10;) or static initializer blocks (static {... }).[19] The Java compiler collects all static initializers for a type and places them into a special method named

<clinit>, which is invoked by the JVM during this initialization phase.[19]

A crucial rule for class initialization is that the direct superclass of a class must be initialized before the class itself can be initialized. This rule applies recursively, ensuring that all superclasses in the inheritance hierarchy are initialized, starting from java.lang.Object, before the class currently being actively used.[19] However, superinterfaces are treated differently; they are only initialized if a non-constant static field declared by them is actively used.[19] The JVM ensures that the initialization process is properly synchronized, allowing only one thread to perform initialization for a given class at any time, which prevents race conditions.[19]

Active vs. Passive Use: When Initialization Happens

Classes are initialized by the JVM only upon their *first active use*.[19] This distinction is important for understanding application startup and runtime behavior.

Active Uses (trigger initialization):

- Creating a new instance of a class (e.g., using the new keyword, reflection, cloning, or deserialization).
- Invoking a static method declared by the class.
- Using or assigning a static field declared by a class or interface, *unless* it is a final static field initialized by a compile-time constant expression.
- Invoking certain reflective methods in the Java API (e.g., methods in java.lang.Class or java.lang.reflect package).
- Initializing a subclass of a class (which, by the recursive rule, requires prior

initialization of its superclass).
- Designating a class as the initial class (the one containing the main() method) when a Java virtual machine starts up.[19]

Passive Uses (do NOT trigger initialization):

- Referencing a static final field initialized by a compile-time constant. These values are inlined by the compiler, so no class loading or initialization of the declaring class is needed at runtime.
- Declaring a field or variable of a class type without creating an instance or accessing any static members of that class.
- Using a non-constant static field declared in a superclass via a subclass. This is considered an active use of the superclass but a passive use of the subclass.[19]

The JVM's strategy of lazy loading and lazy resolution (as part of linking) represents a direct performance optimization. By deferring the loading, linking, and especially the initialization of classes until they are actively needed, the JVM minimizes startup time and reduces memory consumption for applications that might have a large number of classes but only utilize a subset in a given execution path. This is a trade-off: initial startup is often faster, but there might be small, intermittent pauses later in the application's lifecycle when a new class is first accessed and subsequently undergoes its full loading and initialization process. For developers, this implies that the mere presence of a class in the classpath does not guarantee it is loaded and initialized. Performance profiling tools often reveal "spikes" in CPU usage during the first access to certain parts of an application, which can be attributed to this on-demand class loading and initialization. Understanding this behavior is crucial for optimizing application startup and identifying runtime performance bottlenecks.

Table 3: Class Loading Lifecycle Phases

Phase	Sub-phases (if any)	What Happens	Key Outcomes /Errors
Loading	- Locating Binary Data - Parsing Data - Creating java.lang.Class	Reads binary data (.class file) into memory; creates Class	NoClassDefFoundError (if binary data not found/accessible) [21]

	Instance	object for runtime representation.				
Linking	**Verification**	Ensures bytecode is structurally correct and adheres to Java language rules; checks for integrity.	VerifyError (if bytecode is corrupted or invalid) [3]			
	Preparation	Allocates memory for static fields and initializes them with default values (e.g., 0, false, null).	Memory allocation for static data [3]			
	Resolution	Translates symbolic references in the constant pool into direct references. Can be eager or lazy.	LinkageError (general linking problem) [20],	NoClassDefFoundError (if referenced class is missing) [21],	IncompatibleClassChangeError (if referenced class changed incompatibly) [23],	UnsatisfiedLinkError (for native methods) [23]
Initialization	- Initializing Superclass - Executing	Assigns "proper" initial values to static fields;	Static variables get their programmer-defined values [19]			

	<clinit> method	executes static initializer blocks.	

V. Beyond the Basics: Custom Class Loaders

While the default class loaders are sufficient for the vast majority of Java applications, the ability to create custom class loaders offers powerful capabilities for specialized and advanced scenarios. These custom implementations extend the JVM's dynamic loading capabilities, enabling sophisticated architectural patterns.

Why Create Your Own? Use Cases and Advantages

Custom class loaders provide a level of control over the class loading process that is indispensable in certain complex environments.[2] Their advantages include:

- **Dynamic Loading/Unloading:** They enable the loading and, crucially, the unloading of classes at runtime. This capability is vital for "hot deployment" scenarios, such as deploying new versions of plugins or modules in an application server without requiring a full restart of the entire application.[1]
- **Modularity and Isolation:** Custom class loaders allow different applications or components within the same JVM to load and use different versions of the same library without conflicts.[1] This capability effectively creates isolated "namespaces" for classes, preventing "dependency hell" where conflicting library versions cause runtime issues.
- **Loading from Non-Standard Sources:** They facilitate loading classes from unconventional locations that are not part of the standard classpath. This includes sources such as databases, network resources (e.g., HTTP, FTP servers), encrypted files, or even dynamically generated bytecode.[1]
- **Bytecode Manipulation/Instrumentation:** Custom class loaders can modify the bytecode of classes *before* they are loaded into the JVM. This technique is leveraged in Aspect-Oriented Programming (AOP) for "load-time weaving" (injecting cross-cutting concerns) or by persistence frameworks (e.g., Hibernate,

JPA) to enhance entities with persistence-related behavior.[1]
- **Security Policies:** They allow for the implementation of custom security checks. For instance, a custom class loader could verify digital signatures of untrusted code before permitting its execution, enhancing the application's security posture.[2]
- **Resource Management:** Beyond classes, custom class loaders can also manage the loading of associated resources, such as configuration files or images, from non-standard locations.[2]

The immense power of custom class loaders for modularity, hot deployment, and loading from diverse sources is undeniable. However, this power comes with significant complexity. The ability to dynamically load and *unload* classes (by allowing their class loader to be garbage collected) is a key advantage for hot deployment. Yet, this introduces a new challenge: if any object loaded by a custom class loader (or any object referencing such an object) remains reachable after the "application" or "plugin" it belongs to is conceptually unloaded, the class loader itself cannot be garbage collected. This leads to a "class loader leak," where the class definitions and static data loaded by that class loader persist in memory, consuming PermGen/Metaspace (in older JVMs) or heap space, eventually leading to an OutOfMemoryError. This is a direct consequence of the "ClassLoader + Class Name" uniqueness rule: if the ClassLoader cannot be garbage collected, neither can its classes.

How to Implement a Custom Class Loader (Key Methods: findClass, defineClass)

To create a custom class loader, a developer typically extends the abstract java.lang.ClassLoader class.[2] The most common method to override is

findClass(String name). This method is invoked by the loadClass() method *after* delegation to the parent class loader has failed. The findClass() implementation is where the developer defines the specific logic for how the custom class loader *finds* the raw bytecode (as a byte array) for a given class name.[5]

Once the findClass() method has successfully obtained the class's binary data, the inherited defineClass() method is used to convert this byte array into a java.lang.Class instance.[11] The

defineClass() method is responsible for performing the linking phase (verification, preparation, and resolution) of the class loading lifecycle. It is crucial to maintain the parent-delegation model when implementing a custom class loader. This is typically achieved by calling super.loadClass(name) within a custom loadClass() method (if it is overridden) or by relying on the default ClassLoader.loadClass() implementation, which already incorporates this delegation logic.[5]

For many common custom class loading scenarios, the java.net.URLClassLoader class provides a convenient starting point. URLClassLoader is a concrete subclass of ClassLoader that can load classes and resources from a search path of URLs, which can refer to directories or JAR files. This simplifies the implementation of many custom class loaders by handling the underlying resource location logic.[5]

Practical Example: A Simple Custom Class Loader

Consider a scenario where an application needs to load a class from a non-standard location, such as a specific directory not included in the application's default classpath. A custom class loader can be implemented to achieve this:

Java

```
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class CustomFileSystemClassLoader extends ClassLoader {
    private String classPath;

    public CustomFileSystemClassLoader(String classPath, ClassLoader parent) {
        super(parent); // Delegate to parent ClassLoader
```

```
        this.classPath = classPath;
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        Path classFilePath = Paths.get(classPath, name.replace('.', File.separatorChar) +
".class");
        if (!Files.exists(classFilePath)) {
            throw new ClassNotFoundException("Class " + name + " not found in custom path: " +
classPath);
        }

        byte classBytes = null;
        try (InputStream is = Files.newInputStream(classFilePath);
             ByteArrayOutputStream bos = new ByteArrayOutputStream()) {
            byte buffer = new byte;
            int length;
            while ((length = is.read(buffer))!= -1) {
                bos.write(buffer, 0, length);
            }
            classBytes = bos.toByteArray();
        } catch (IOException e) {
            throw new ClassNotFoundException("Error reading class file: " + name, e);
        }

        // Define the class using the byte array
        return defineClass(name, classBytes, 0, classBytes.length);
    }

    // Main method to demonstrate usage
    public static void main(String args) throws Exception {
        // Assume 'MyCustomClass.class' is located in '/tmp/custom_classes'
        // MyCustomClass.java:
        // package com.example;
        // public class MyCustomClass {
        //     public void printMessage() {
        //         System.out.println("Hello from MyCustomClass loaded by custom loader!");
        //     }
        // }
```

```
        String customDirPath = "/tmp/custom_classes"; // Or any other path
        String className = "com.example.MyCustomClass";

        // Create an instance of our custom class loader
        // We pass the System ClassLoader as its parent to maintain delegation
        CustomFileSystemClassLoader customLoader =
            new CustomFileSystemClassLoader(customDirPath,
ClassLoader.getSystemClassLoader());

        try {
            // Load the class using the custom class loader
            Class<?> myClass = customLoader.loadClass(className);
            Object instance = myClass.getDeclaredConstructor().newInstance();
            myClass.getMethod("printMessage").invoke(instance);

            // Verify which class loader loaded it
            System.out.println("Class loaded by: " + myClass.getClassLoader());
            System.out.println("System ClassLoader: " +
ClassLoader.getSystemClassLoader());

        } catch (ClassNotFoundException e) {
            System.err.println("Failed to load class: " + e.getMessage());
        }
    }
}
```

This example demonstrates how findClass is overridden to locate the .class file and how defineClass is used to turn the bytecode into a Class object. It also highlights the importance of setting a parent class loader to maintain the delegation model.

VI. Advanced Class Loader Concepts

Beyond the fundamental built-in class loaders and the basics of custom implementations, Java's class loading mechanism extends to more specialized and complex scenarios, particularly in multi-threaded environments, application servers,

and modern modularity frameworks.

The Thread Context Class Loader: Breaking the Hierarchy

Each Thread in Java has an associated "context ClassLoader".[13] This class loader can be accessed and modified using the

Thread.currentThread().getContextClassLoader() and setContextClassLoader() methods, respectively.[14] By default, a thread's context class loader is inherited from its parent thread. The primordial thread, which is the initial thread created by the operating system when the application starts, typically has its context class loader set to the System Class Loader.[13]

The primary purpose of the context class loader is to provide a mechanism that can deviate from the strict parent-first delegation model. It is predominantly used in scenarios where a framework or service (often loaded by a parent class loader, higher in the hierarchy) needs to load classes provided by an application or plugin (which are typically loaded by a child class loader, lower in the hierarchy).[14] For instance, Java Naming and Directory Interface (JNDI), JDBC drivers, and Java Remote Method Invocation (RMI), along with other Service Provider Interfaces (SPIs), frequently employ the context class loader. This allows them to load specific implementation classes that are usually found on the application's classpath, which would otherwise be invisible to the core framework's class loader due to the parent-first rule.[13] In essence, the context class loader provides a mechanism for "reverse delegation" or "child-first" lookup in specific contexts, enabling components to find resources that are "lower" in the class loader hierarchy.

Class Loaders in Application Servers (e.g., Tomcat, Jakarta EE)

Application servers like Apache Tomcat or those adhering to Jakarta EE (formerly Java EE) specifications heavily rely on intricate class loader hierarchies.[1] This sophisticated architecture enables crucial features such as application isolation and efficient sharing of common libraries among multiple deployed applications.[1]

- **Isolation:** Each deployed web application (typically packaged as a WAR file) is

assigned its own dedicated WebappClassLoader (or a similar implementation).[1] This class loader is usually a child of the server's common class loader. This design effectively isolates applications from one another, preventing class conflicts even if different applications depend on different versions of the same library (e.g.,
Spring Framework 4.x in one app and Spring Framework 5.x in another).[1]

- **Sharing:** To optimize memory usage and simplify management, common libraries (such as JDBC drivers, logging frameworks, or utility libraries) can be placed in a location accessible by a "common" or "shared" class loader.[26] This allows all deployed applications to share a single instance of these classes, reducing the overall memory footprint.
- **Delegation Inversion:** While the default Java class loading model is parent-first, web application class loaders in servlet containers often implement a "delegation inversion" model for web application-specific classes.[26] This means they look in their local
/WEB-INF/classes and /WEB-INF/lib directories *first* before delegating to their parent class loader. This allows applications to bundle their own versions of libraries, even if those versions might conflict with server-provided ones, giving the application more control over its dependencies. This behavior is typically configurable (e.g., via a delegate="false" setting in configuration files like sun-web.xml for GlassFish or similar settings in Tomcat).[26]

OSGi Framework: A Different Approach to Modularity

The OSGi (Open Services Gateway initiative) framework offers a highly dynamic module system for Java that diverges significantly from traditional Java class loading.[1] Unlike the flat global classpath model, OSGi operates on a more granular level.

- **Bundles:** The fundamental unit of deployment in OSGi is a "bundle," which is essentially a JAR file augmented with specific manifest metadata.[29] Each bundle within the OSGi framework is assigned its own dedicated class loader.[29]
- **Explicit Dependencies:** OSGi bundles explicitly declare which Java packages they export (make available to others) and which they import (depend on from other bundles), along with precise version information.[29] The OSGi framework processes this metadata to resolve all inter-bundle dependencies and calculates an
independent required classpath for each bundle. This contrasts sharply with the

implicit classpath of standard Java applications.[29]

- **Benefits:** This explicit and granular approach offers several significant advantages: it allows multiple versions of the same package to coexist concurrently within the same JVM, enables dynamic installation, starting, stopping, and uninstallation of bundles without restarting the JVM ("hot deployment"), and dramatically mitigates "dependency hell" issues.[1] OSGi's class loading model is thus more sophisticated than the standard delegation model, prioritizing explicit package imports/exports over a strict parent-first hierarchy for all classes.[1]

Java Platform Module System (JPMS) and Class Loaders (Java 9+)

Introduced in Java 9, the Java Platform Module System (JPMS) represents a significant evolution in Java's modularity story.[1] JPMS defines a standardized distribution format for collections of Java code (modules) and a repository for storing them. It also specifies how these modules are discovered, loaded, and checked for integrity.[1]

The primary goal of JPMS is to address long-standing shortcomings of the existing JAR format and classpath issues by providing strong encapsulation and reliable configuration.[1] In the context of class loaders, JPMS refines their roles. It introduces the Platform Class Loader (as discussed in Section II) and redefines the responsibilities of the System Class Loader in loading application modules.[12] While JPMS provides modularity at a higher level, dealing with a module graph, it still leverages and interacts with the underlying class loader mechanism. It follows a different philosophy than OSGi, aiming for backwards compatibility with the JRE's default class loading behavior.[1]

The evolution of modularity and class loading in Java, from the traditional classpath to application server hierarchies, then to OSGi, and finally to JPMS, reflects a continuous effort to address the challenges of managing increasingly complex applications with numerous dependencies. The "dependency hell" problem, where conflicting versions of libraries cause runtime issues, is a direct consequence of the flat classpath model. Application servers introduced a degree of isolation, but OSGi and JPMS represent more fundamental shifts towards *explicit* modularity and dependency management, which inherently relies on sophisticated class loading mechanisms. Each iteration attempts to provide better isolation, dynamic capabilities, and clearer dependency

resolution. For developers, understanding this evolution is key to choosing the right architecture for complex systems. While JPMS is the standard for modern Java, OSGi remains highly relevant for highly dynamic plugin-based systems. The underlying principle of isolating class paths via class loaders remains central to all these solutions.

VII. Troubleshooting Class Loading Issues

For any Java programmer aspiring to mastery, understanding and troubleshooting common runtime errors related to class loading is indispensable. These issues can often be perplexing, but a clear grasp of their underlying causes can significantly streamline the debugging process.

ClassNotFoundException vs. NoClassDefFoundError: Understanding the Nuances

ClassNotFoundException and NoClassDefFoundError are two of the most common and frequently confused runtime issues related to Java's class loading mechanism.[21] While both indicate a problem with a class definition, their origins and implications differ significantly.

- **ClassNotFoundException:**
  - **Type:** This is a checked Exception, meaning it is a subclass of java.lang.Exception and must either be caught or declared to be thrown by the method.[21]
  - **When it occurs:** It is thrown when the JVM or an application attempts to *dynamically load* a class by its string name at runtime using methods like Class.forName(), ClassLoader.loadClass(), or ClassLoader.findSystemClass().[21] The core issue is that the class definition cannot be found on the classpath at the moment of this explicit loading attempt.[21]
  - **Implication:** The class was *not found* on the classpath during the runtime execution. It might have been present during compilation, but is missing or inaccessible at runtime.[21]
- **NoClassDefFoundError:**
  - **Type:** This is an Error, specifically a subclass of java.lang.LinkageError.[20]

Errors typically indicate severe problems that an application should not attempt to catch or recover from.
    - **When it occurs:** It is thrown when the JVM or an application tries to load the definition of a class, and that class definition was *present at compile time* but is *no longer available or compatible* at runtime.[21] This often happens when a class references another class that was available during compilation but is missing, corrupted, or has an incompatible version at runtime.[21]
    - **Implication:** The class was *expected* to be present based on the compiled code, but the JVM failed to find or link its definition *implicitly* when it was needed. This points to a fundamental problem with the application's deployment or environment, such as a missing JAR file, a corrupted class file, or an incompatible version of a dependency.[21]
    - **Handling:** As an Error, it is usually catastrophic and not typically caught, as it signifies a deeper configuration or deployment issue that needs to be addressed at the system level.[21]

LinkageError and Its Subtypes: When Things Go Wrong

LinkageError is a superclass of Error that specifically indicates a problem encountered during the linking phase of class loading.[20] It signifies that a class has a dependency on another class, but the latter has incompatibly changed or become unavailable after the former was compiled.[20]

Common subtypes of LinkageError include:

- **NoClassDefFoundError:** As discussed, this occurs when a class definition, present at compile time, is missing or inaccessible at runtime.[21]
- **IncompatibleClassChangeError:** This error arises when the JVM finds a class at runtime whose definition has changed incompatibly from the time it was compiled or verified. Examples include a method signature changing, a field being removed, or a class being converted to an interface.[23]
- **UnsatisfiedLinkError:** This error is typically thrown when a native method (a method implemented in a language other than Java, often C or C++ via JNI) cannot be found or loaded during runtime.[23] It often points to issues with missing or incorrectly configured native libraries, or mismatches between native method signatures and their Java declarations.[23]
- **VerifyError:** This occurs during the verification sub-phase of linking if the

bytecode is structurally incorrect or violates Java language rules.[3]
- **ClassFormatError:** This indicates that the .class file itself is malformed, corrupted, or does not adhere to the Java class file format specification.[20]

Common Causes and Debugging Tips

Troubleshooting class loading issues often boils down to a few common culprits:

- **Incorrect Classpath:** The most frequent cause of both ClassNotFoundException and NoClassDefFoundError. Ensure that all necessary JAR files and directories containing .class files are correctly included in the application's classpath.[3] This involves checking command-line options (-cp or -classpath), environment variables (CLASSPATH), and build tool configurations (Maven, Gradle).
- **Missing Dependencies:** A required library (JAR) that a class depends on is not present or accessible at runtime.[21]
- **Version Conflicts (Dependency Hell):** Different parts of an application or different deployed applications depend on incompatible versions of the same library.[6] This is precisely where custom class loaders and modularity systems (like OSGi or JPMS) provide robust solutions.
- **Corrupted Class Files:** The .class file itself might be damaged or incomplete, leading to ClassFormatError or VerifyError.[21]
- **Native Library Issues:** For UnsatisfiedLinkError, verify that native libraries are correctly installed, their paths are configured in system environment variables (e.g., LD_LIBRARY_PATH on Linux, PATH on Windows), and they are compatible with the JVM's architecture (32-bit vs. 64-bit).[23]

Debugging Tools:

- **JVM Arguments:** Utilize JVM command-line arguments like -XX:+TraceClassLoading to obtain detailed logs of which classes are being loaded and by which class loader.[18] This can provide invaluable insight into the loading sequence.
- **IDE Tools:** Leverage your Integrated Development Environment's (IDE) built-in dependency analysis and classpath configuration tools. Modern IDEs can often highlight missing or conflicting dependencies.
- **Profiling Tools:** Advanced tools such as VisualVM, JProfiler, or YourKit can monitor class loading activity, track memory consumption patterns, and help

identify potential class loader leaks.[31]

Class Loader Leaks: A Hidden Memory Drain

A particularly insidious issue in long-running Java applications, especially those using dynamic deployment (like application servers or plugin frameworks), is the "class loader leak".[8] This occurs when a

ClassLoader instance, along with all the classes it loaded and their static fields, cannot be garbage collected, even after the application or module it belongs to has been conceptually "unloaded" or "redeployed".[8]

The uniqueness rule, where a class is identified by the combination of its name and its class loader, is both a source of power and a potential peril. While it allows for isolation and multiple versions of the same class, this very rule is the root cause of class loader leaks. If a class loader cannot be garbage collected, then none of the classes it loaded can be unloaded, and their static fields (which are allocated during the preparation phase of linking) persist in memory.

Cause: Class loader leaks typically happen when objects loaded by a custom class loader (or objects that hold references to them) are inadvertently held by objects loaded by a *parent* or *sibling* class loader. Common culprits include static fields in parent classes, ThreadLocal variables that are not properly cleared, or caches that maintain strong references to objects from dynamically loaded classes that outlive the custom class loader's intended lifecycle.[8] For instance, a thread started by a parent class loader might hold a reference to an object from a child-loaded plugin, preventing the child's class loader from being garbage collected even after the plugin is "unloaded."

Symptom: The primary symptom of a class loader leak is a gradual, continuous increase in memory usage over time, particularly noticeable after multiple redeployments or prolonged application uptime. This often leads to OutOfMemoryError messages, especially in the PermGen or Metaspace regions in older JVMs, or the main heap in newer ones.[32]

Prevention/Debugging:

- **Proper Resource Cleanup:** Ensure that all resources (e.g., threads, event

listeners, caches, database connections) created by a dynamically loaded module are properly shut down, deregistered, and released upon module unloading.[31]

- **Avoid Strong References:** Be extremely cautious about holding strong references to objects loaded by child class loaders within objects or static fields loaded by parent class loaders.
- **Weak References:** For caches or situations where references to dynamically loaded objects might persist, consider using WeakReference or WeakHashMap. These allow objects to be garbage collected if no other strong references exist, even if the weak reference itself still exists.[31]
- **Heap Dump Analysis:** The most effective way to diagnose class loader leaks is by analyzing heap dumps (e.g., generated using jmap -histo or jcmd <pid> GC.heap_dump). Tools like Eclipse Memory Analyzer (MAT) can then be used to identify unreachable objects and trace the GC roots that are preventing the class loader from being collected.[32]

For master-level Java programmers, understanding this duality of the "Class Identity" rule—its power for isolation versus its peril in causing memory leaks—is paramount. It means that advanced features like hot deployment or plugin architectures, while powerful, introduce a new layer of memory management complexity that goes beyond typical object garbage collection. It requires a deep understanding of reference chains and the class loader hierarchy to prevent subtle, hard-to-diagnose memory issues in long-running or dynamically evolving applications.

Table 4: ClassNotFoundException vs. NoClassDefFoundError vs. LinkageError

Feature	ClassNotFoundException	NoClassDefFoundError	LinkageError (General)
Type	Checked Exception (subclass of java.lang.Exception) [21]	Error (subclass of java.lang.LinkageError) [20]	Error (base class for linking issues) [20]
When it Occurs	Runtime, during *explicit* dynamic loading (e.g., Class.forName()) [21]	Runtime, when JVM tries to *implicitly* load a class definition that was present at compile time but is now missing/incompatible [21]	Runtime, during the linking phase, due to incompatible class changes or issues with dependencies [20]

Primary Cause	Class not found on classpath at the moment of explicit loading attempt [21]	Class definition missing or changed incompatibly *after* compilation [21]	Inconsistency between dependent classes after compilation [20]
Handling/Implication	Must be caught or declared; indicates missing class file at runtime [21]	Usually catastrophic; indicates fundamental deployment/environment issue [21]	Catastrophic; indicates deep structural problem with class dependencies [23]
Example	Class.forName("com.missing.MyClass"); where MyClass.class is not on classpath [21]	new MyClass(); where MyClass.class was compiled but later removed from classpath [21]	IncompatibleClassChangeError, UnsatisfiedLinkError, VerifyError, ClassFormatError [20]

VIII. Conclusion: Mastering the JVM's Dynamic Core

The Java Class Loader, often operating silently in the background, is undeniably one of the most critical and powerful components of the Java Virtual Machine. Its sophisticated design enables Java's core promise of platform independence and dynamic behavior, making it a cornerstone of modern application development.

Recap of Key Takeaways

- **The Unsung Hero:** Java Class Loaders are the fundamental mechanism responsible for locating, loading, linking, and initializing classes on demand, enabling Java's dynamic nature and efficient resource utilization.
- **Hierarchical Structure:** The JVM's built-in class loaders (Bootstrap, Platform/Extension, and System/Application) operate in a strict parent-first delegation hierarchy. This structure is not merely organizational; it is a vital security and consistency mechanism, ensuring that core Java APIs are loaded by trusted components and preventing malicious overrides.
- **Precise Lifecycle:** Every class undergoes a well-defined lifecycle within the JVM,

progressing through Loading, Linking (Verification, Preparation, Resolution), and Initialization. Understanding these phases, particularly the performance implications of lazy loading and resolution, is crucial for optimizing application behavior.
- **Power of Customization:** Custom Class Loaders offer unparalleled flexibility for dynamic loading, application isolation, bytecode manipulation, and loading from non-standard sources. This power is essential for building complex, modular systems like application servers and plugin frameworks.
- **Complexity and Peril:** While powerful, custom class loaders introduce significant complexity, especially concerning memory management. The unique identity of a class (defined by its name and its loading class loader) can lead to subtle but severe "class loader leaks" if object references are not meticulously managed, resulting in OutOfMemoryError in long-running applications.
- **Diagnostic Acumen:** Distinguishing between common runtime errors like ClassNotFoundException, NoClassDefFoundError, and various LinkageError subtypes is paramount for effective troubleshooting. Each error signals a specific problem in the class loading or linking process, guiding developers to the correct diagnostic path.

Best Practices for Java Developers

To truly master the JVM's dynamic core and leverage class loaders effectively, developers should adhere to the following best practices:

- **Understand Your Classpath Intimately:** Many class loading issues stem from incorrect, incomplete, or conflicting classpath configurations. A clear understanding of how your application's classpath is constructed is fundamental.
- **Prefer Standard Mechanisms:** For most applications, the default class loading mechanisms provided by the JVM are robust, optimized, and sufficient. Avoid implementing custom class loaders unless there is a clear, compelling architectural requirement (e.g., building a plugin system, an application server, or performing advanced bytecode instrumentation).
- **Exercise Extreme Caution with Custom Class Loaders:** If custom class loaders are necessary, invest significant effort in understanding object lifecycles, reference management, and the potential for class loader leaks. Implement robust cleanup mechanisms to ensure that dynamically loaded components, and their associated class loaders, can be fully unloaded when no longer needed.

- **Monitor and Profile Aggressively:** Regularly use JVM monitoring and profiling tools (such as VisualVM, JProfiler, or YourKit) to observe class loading patterns, analyze memory consumption, and proactively detect potential class loader leaks before they manifest as critical runtime failures.
- **Embrace Modern Modularity:** For complex applications with numerous dependencies, leverage modern Java modularity features like the Java Platform Module System (JPMS) introduced in Java 9, or established frameworks like OSGi. These systems provide robust solutions for managing dependencies, preventing classpath conflicts, and enabling dynamic application evolution.
- **Cultivate Error Recognition:** Familiarize yourself thoroughly with the precise triggers and implications of ClassNotFoundException, NoClassDefFoundError, and the various LinkageError types. The ability to quickly identify the specific nature of a class loading error will dramatically reduce debugging time and lead to more resilient applications.

By internalizing these principles and practices, Java programmers can move beyond merely writing functional code to truly understanding and mastering the dynamic, powerful, and sometimes challenging world of Java Class Loaders.

Works cited

1. Java class loader - Wikipedia, accessed June 16, 2025, https://en.wikipedia.org/wiki/Java_class_loader
2. What Is Java Class Loader? - ITU Online IT Training, accessed June 16, 2025, https://www.ituonline.com/tech-definitions/what-is-java-class-loader/
3. Classloaders in JVM: An Overview - DZone, accessed June 16, 2025, https://dzone.com/articles/classloaders-in-jvm-an-overview
4. Understanding the Java Class Loading Mechanism - JustAcademy, accessed June 16, 2025, https://www.justacademy.co/blog-detail/java-class-loading-mechanism
5. Understanding Network Class Loaders - Oracle, accessed June 16, 2025, https://www.oracle.com/technical-resources/articles/javase/classloaders.html
6. What is the use of Custom Class Loader - java - Stack Overflow, accessed June 16, 2025, https://stackoverflow.com/questions/10828863/what-is-the-use-of-custom-class-loader
7. Use case of Java custom class loader [duplicate] - Stack Overflow, accessed June 16, 2025, https://stackoverflow.com/questions/50368092/use-case-of-java-custom-class-loader
8. Java plugins with isolating class loaders - Adevinta, accessed June 16, 2025, https://adevinta.com/techblog/java-plugins-with-isolating-class-loaders/
9. Differences Between JDK, JRE and JVM - GeeksforGeeks, accessed June 16,

2025, https://www.geeksforgeeks.org/java/differences-jdk-jre-jvm/
10. JDK vs JRE vs JVM in Java: Key Differences Explained - DigitalOcean, accessed June 16, 2025, https://www.digitalocean.com/community/tutorials/difference-jdk-vs-jre-vs-jvm
11. Java ClassLoader | DigitalOcean, accessed June 16, 2025, https://www.digitalocean.com/community/tutorials/java-classloader
12. ClassLoader (Java SE 21 & JDK 21 [ad-hoc build]) - cr, accessed June 16, 2025, https://cr.openjdk.org/~jlaskey/templates/docs/api/java.base/java/lang/ClassLoader.html
13. Class Loading, accessed June 16, 2025, https://docs.oracle.com/javase/jndi/tutorial/beyond/misc/classloader.html
14. Difference Between Thread's Context Class Loader and Normal Class Loader | Baeldung, accessed June 16, 2025, https://www.baeldung.com/java-class-loader-thread-context-vs-normal
15. ClassLoader (Java SE 24 & JDK 24) - Oracle Help Center, accessed June 16, 2025, https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/lang/ClassLoader.html
16. jdk/src/java.base/share/classes/java/lang/ClassLoader.java at master · openjdk/jdk · GitHub, accessed June 16, 2025, https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/lang/ClassLoader.java
17. ClassLoader Class (Java.Lang) - Learn Microsoft, accessed June 16, 2025, https://learn.microsoft.com/en-us/dotnet/api/java.lang.classloader?view=net-android-35.0
18. java - Loading, Linking, and Initializing - When does a class get loaded? - Stack Overflow, accessed June 16, 2025, https://stackoverflow.com/questions/41656506/loading-linking-and-initializing-when-does-a-class-get-loaded
19. Java Type Loading, Linking, and Initialization - Artima, accessed June 16, 2025, https://www.artima.com/insidejvm/ed2/lifetypeP.html
20. LinkageError Class (Java.Lang) | Microsoft Learn, accessed June 16, 2025, https://learn.microsoft.com/en-us/dotnet/api/java.lang.linkageerror?view=net-android-35.0
21. What causes and what are the differences between NoClassDefFoundError and ClassNotFoundException? - Codemia, accessed June 16, 2025, https://codemia.io/knowledge-hub/path/what_causes_and_what_are_the_differences_between_noclassdeffounderror_and_classnotfoundexception
22. ClassNotFoundException vs NoClassDefFoundError - Stack Overflow, accessed June 16, 2025, https://stackoverflow.com/questions/28322833/classnotfoundexception-vs-noclassdeffounderror
23. Understanding LinkageError in Java: A Comprehensive Guide for Developers, accessed June 16, 2025, https://exceptiondecoded.com/posts/java-linkageerror/
24. Unloading classes at runtime (Java in General forum at Coderanch), accessed June 16, 2025, https://coderanch.com/t/329554/java/Unloading-classes-runtime

25. Java Thread setContextClassLoader() Method - Tutorialspoint, accessed June 16, 2025, https://www.tutorialspoint.com/java/lang/thread_setcontextclassloader.htm
26. The Class Loader Hierarchy (Sun Java System Application Server Platform Edition 9 Developer's Guide), accessed June 16, 2025, https://docs.oracle.com/cd/E19501-01/819-3659/beadf/index.html
27. Apache Tomcat 10 (10.1.42) - Class Loader How-To, accessed June 16, 2025, https://tomcat.apache.org/tomcat-10.1-doc/class-loader-howto.html
28. Apache Tomcat 9 (9.0.106) - Class Loader How-To, accessed June 16, 2025, https://tomcat.apache.org/tomcat-9.0-doc/class-loader-howto.html
29. The OSGi Service Platform - IBM, accessed June 16, 2025, https://www.ibm.com/docs/en/cics-ts/6.x?topic=java-osgi-service-platform
30. OSGI DEMYSTIFIED, UNRAVELING JAVA AND OSGI CLASS LOADER - Cogent Infotech, accessed June 16, 2025, https://www.cogentinfo.com/resources/osgi-demystified-unraveling-java-and-osgi-class-loader
31. Understanding Java Memory Leaks and How to Prevent Them - DEV Community, accessed June 16, 2025, https://dev.to/isaactony/understanding-java-memory-leaks-and-how-to-prevent-them-2gpa
32. Classloader-Releated Memory Issues - Dynatrace, accessed June 16, 2025, https://www.dynatrace.com/resources/ebooks/javabook/class-loader-issues/