Java 8 Bytecode Execution on a Java 17 JVM: Feature Availability and Improvements

1. Introduction
The Java platform, known for its "write once, run anywhere" principle, achieves this through the use of bytecode, an intermediate language executed by the Java Virtual Machine (JVM). A common question among Java developers is whether code compiled for an older version of Java can leverage the features and improvements of a newer JVM simply by being run on it. This report specifically addresses the scenario of running Java code compiled with a Java 8 compiler on a JVM version 17. It will explore the fundamental aspects of Java bytecode and JVM compatibility, the extent to which Java 8 bytecode can function on a Java 17 JVM, and whether this setup allows for the utilization of new language features and API enhancements introduced in Java versions 9 through 17. Furthermore, the report will delve into the performance and security benefits that a Java 17 JVM might offer to older bytecode, as well as potential compatibility concerns. Finally, it will discuss the steps necessary to fully utilize the capabilities of Java 17.

2. Understanding Java Bytecode and JVM Compatibility
Java bytecode serves as the instruction set for the JVM, acting as the intermediary language into which Java source code, as well as code written in other JVM-compatible languages, is translated.[1] This compilation process is typically performed by the javac compiler, originally provided by Sun Microsystems.[1] The nature of bytecode ensures platform independence, meaning that a Java bytecode program can be executed on any operating system or hardware architecture that has a compatible JVM, without requiring the time-consuming process of recompiling from the original source code.[1] At runtime, the JVM either interprets this bytecode or further compiles it into native machine code using Just-In-Time (JIT) compilation for direct execution by the underlying hardware.[1] The JVM itself is a runtime environment that manages the execution of Java bytecode.[1] Internally, the JVM operates as both a stack machine and a register machine, utilizing an operand stack and local variable arrays for each method invocation.[1] The sizes of these structures are determined by the compiler and are included within the attributes of each method.[1] Notably, various implementations of the JVM are available from different vendors and are used in diverse environments.[1]
A crucial aspect of the Java ecosystem is the concept of compatibility between different versions of the Java Development Kit (JDK) and the JVM. Backward compatibility refers to the ability of a newer version of the JVM to execute bytecode that was compiled for an older version of Java.[2] This design principle is fundamental to the smooth adoption of new Java

versions, as it allows existing applications to continue running on updated runtime environments without immediate recompilation. For instance, a Java 17 JVM is generally capable of running bytecode produced by a Java 8 compiler.[3] Conversely, forward compatibility describes the ability of an older JVM to run bytecode compiled for a newer Java version. In the case of Java, bytecode is generally **not** forward compatible.[3] This is because newer Java versions may introduce new bytecode instructions or modifications to the class file format that older JVMs would not recognize or understand, potentially leading to errors such as UnsupportedClassVersionError.[5] To manage this versioning, each JDK release is associated with a specific class file version number.[2] For example, Java SE 8 uses class file version 52.0 [2], while Java 17 uses version 61.0.[8] A JVM typically checks this version number to ensure it can execute the bytecode; a newer class file version will generally not be executable by an older JVM.[13]

3. Java 8 Bytecode on a Java 17 JVM: Core Functionality

Due to the strong emphasis on backward compatibility in the Java platform, bytecode compiled for Java 8 (which has a class file version of 52.0) can, in most scenarios, be executed without any issues on a Java 17 JVM.[3] This capability allows organizations to upgrade their JVM infrastructure to newer versions like Java 17, which may offer improvements in performance, security, and manageability, without the immediate necessity of recompiling all their existing Java 8 applications.[4] This provides a significant advantage in terms of operational flexibility and cost-effectiveness during technology upgrades.
However, while backward compatibility is a core principle, there are potential exceptions and considerations to be aware of.[3] Over the evolution of Java, certain very old or internal APIs have been removed.[3] For instance, Java 8 dropped support for some of the original semantics of the invokespecial instruction from Java 1.0.[3] Similarly, the jsr and ret bytecode instructions, while removed in class file version 51.0 (corresponding to Java 7), are still generally supported for older class file versions to maintain backward compatibility, although future JVMs could potentially drop support for these older versions.[3] More recently, Java 17 saw the removal of RMI Activation.[16] Applications that depend on such removed features, even if compiled with Java 8, might encounter runtime errors when executed on a Java 17 JVM.[3] Additionally, issues can arise if Java 8 code relies on unspecified behaviors or internal implementation details of the JVM that might have changed between versions 8 and 17.[2] Libraries that make use of internal JDK APIs (such as those within the sun.* packages) are particularly susceptible to compatibility problems during JVM upgrades, as these internal implementations are not guaranteed to remain consistent across major versions.[2]

4. Accessing Java 17 Language Features and APIs

A key aspect of the user's query is whether running Java 8 compiled code on a Java 17 JVM provides access to the new language features and APIs introduced in Java versions 9 through 17. It is crucial to understand that new language features, such as records, sealed classes, pattern matching for switch statements, text blocks, and virtual threads, which were

introduced in Java versions after 8 [8], are implemented through specific bytecode instructions and class file structures that were not part of the Java 8 specification.[8] These newer features require the compiler to understand the corresponding syntax and to generate bytecode that adheres to the specifications introduced in the respective Java versions.
When Java 8 code is compiled using a Java 8 compiler, the resulting bytecode will conform to the Java 8 bytecode specification (class file version 52.0).[2] This bytecode inherently lacks the instructions and structures necessary to support language features introduced in later Java versions, which utilize a newer class file version (61.0 for Java 17).[8] Therefore, simply running this Java 8 bytecode on a Java 17 JVM does not magically enable the use of these newer language features.[8] The JVM's role is to execute the bytecode it is given; it does not retroactively add functionality based on its own version. Similarly, while a Java 17 JVM includes all the standard libraries up to version 17, code compiled against the Java 8 standard libraries will only be able to access the APIs that were available in Java 8.[3] If the Java 8 compiled code attempts to call a method or use a class that was introduced in Java 9 or later, the JVM will not be able to find it at runtime, resulting in a NoSuchMethodError or NoClassDefFoundError.[3] Even though the Java 17 JVM contains these newer APIs, the bytecode generated by the Java 8 compiler does not include the necessary references or instructions to utilize them. It is important to note that while Java 8 bytecode running on a Java 17 JVM will not directly use Java 17 features or APIs, the underlying implementation of some Java 8 APIs in JVM 17 might benefit from internal optimizations or improvements made in later Java versions.[27] However, this is not equivalent to directly using new Java 17 APIs in the Java 8 compiled code.

5. Benefiting from Java 17 JVM Improvements
Despite the inability to directly use new language features and APIs without recompilation, running Java 8 bytecode on a Java 17 JVM can still offer several advantages, particularly in terms of performance and security.[22] Newer JVM versions often include significant performance enhancements that can benefit even older applications. One notable area is garbage collection. Java 9 introduced G1 as the default garbage collector [22], and subsequent versions, including Java 17, have further refined it, along with introducing new collectors like ZGC and Shenandoah.[20] These newer garbage collectors generally offer improved throughput and lower latency compared to the garbage collectors available in Java 8.[20] For example, a study of migrating from Java 8 to Java 17 showed a significant reduction in garbage collection frequency and long request rates.[32] Running Java 8 bytecode on a JVM 17 allows the application to leverage these advanced garbage collection algorithms.[27]
The Just-In-Time (JIT) compiler is another area where Java 17 likely offers improvements over Java 8.[22] The JIT compiler translates bytecode into native machine code at runtime, and newer JVMs often incorporate more sophisticated optimization techniques. This can lead to faster execution of the same Java 8 bytecode on a Java 17 JVM.[29] Benchmarking has shown performance differences between JVM implementations, highlighting the impact of compiler optimizations.[34] Beyond garbage collection and JIT compilation, Java 17 JVMs include various

other enhancements that can contribute to better performance, such as improvements in thread management, intrinsic functions, and data structures.[22] For instance, features like elastic metaspace (introduced in JDK 16) can lead to more efficient memory management.[22] In addition to performance benefits, running on a Java 17 JVM automatically provides all the security updates and patches included in that version.[20] This is a critical advantage compared to running on an older, potentially unpatched Java 8 JVM, as newer versions address known security vulnerabilities.[22] Java 17 includes several security enhancements, such as improved deserialization filters and a default set of root Certification Authority (CA) certificates.[22]

Feature	**Java 8**	**Java 17**	**Potential Benefit for Java 8 Bytecode**	**Snippet(s)**
Default Garbage Collector	Parallel GC	G1 GC	Improved throughput and latency	20
Other Garbage Collectors	CMS, ParallelOld	ZGC, Shenandoah (available)	Lower pause times (ZGC, Shenandoah)	20
JIT Compiler	Older optimizations	More advanced optimization techniques	Potential for faster execution	22
Metaspace Management	Basic	Elastic Metaspace (JDK 16+)	Reduced memory footprint	22
String Handling	Standard	Compact Strings (JDK 9+)	Reduced memory usage for some strings	22
Code Cache	Single	Separated caches (JDK 9+)	Improved performance and memory	22
Security Updates and Enhancements	Regular patches for Java 8	Continuous updates and new features	Protection against newer vulnerabilities, improved security mechanisms	20

6. Potential Compatibility Concerns

While running Java 8 bytecode on a Java 17 JVM is generally supported, several potential compatibility concerns should be considered. APIs that were deprecated in Java 8 might have been removed in Java 17.[2] If the Java 8 bytecode uses any of these removed APIs, it could lead to runtime errors such as NoSuchMethodError.[3] Additionally, the default behaviors of certain aspects of the JVM might have changed between Java 8 and Java 17.[2] For example, the way

the invokespecial instruction is handled has evolved [3], and the order of operations in string concatenation using invokedynamic changed in later versions.[36] These changes in default behaviors could potentially affect the way older applications function.
The introduction of the Java Platform Module System (JPMS) in Java 9 [8] also presents a consideration. While Java 8 bytecode does not contain module information, running it on a modular JVM like Java 17 might expose issues related to the stronger encapsulation of internal JDK APIs.[20] Applications that relied on accessing these internal APIs using reflection might encounter restrictions.[7] Finally, the compatibility of third-party libraries used by the Java 8 application needs to be considered.[15] These libraries might rely on features or make assumptions about the runtime environment that are no longer valid in Java 17, or they might not have been tested or updated for compatibility with newer JVM versions.[15]

7. Utilizing Java 17 Features Fully

To fully leverage the new language features and APIs introduced in Java 17 (and versions 9 through 16), it is necessary to recompile the Java code using a Java 17 (or later) compiler, targeting the Java 17 bytecode version (61.0).[8] Simply running Java 8 compiled code on a Java 17 JVM will not enable the use of features like records, sealed classes, pattern matching for switch, text blocks, or virtual threads, as these require specific bytecode instructions and class file structures that a Java 8 compiler does not generate.[8]
The javac compiler provides options like -source and -target (or the more recent --release) to control the source language version and the target bytecode version during compilation.[8] To utilize Java 17 features, both the source and target (or release) should be set to 17 or higher.[28] Using -target 1.8 with a Java 17 compiler will still produce Java 8 bytecode, thus limiting the available language features and APIs to those of Java 8.[11] Build tools like Maven and Gradle provide configurations to specify the Java version for compilation.[8] For instance, in Maven, the <maven.compiler.source> and <maven.compiler.target> properties (or <maven.compiler.release>) are used [28], while Gradle uses sourceCompatibility, targetCompatibility, or release options.[8] Migrating a Java 8 codebase to Java 17 involves updating dependencies to versions compatible with Java 17 [35], addressing any deprecated APIs used in the code [31], and conducting thorough testing on the Java 17 JVM.[31] Tools like OpenRewrite can assist in automating some of these migration tasks.[38]

8. Conclusion

In conclusion, running Java code compiled for Java 8 on a Java 17 JVM does **not** automatically grant access to the language features or APIs introduced in Java versions 9 through 17. While Java 8 bytecode is generally backward compatible with Java 17, allowing it to run and potentially benefit from JVM-level improvements in performance and security, the utilization of new language features and APIs requires recompilation of the source code using a Java 17 (or later) compiler, targeting the corresponding bytecode version. Organizations can benefit from upgrading their JVM to Java 17 even when running older bytecode due to performance and security enhancements. However, to fully leverage the capabilities of the Java 17 platform and ensure long-term compatibility and maintainability, it is recommended to plan for the recompilation of applications with a Java 17 compiler, accompanied by thorough testing.

Works cited

1. Java bytecode - Wikipedia, accessed April 26, 2025, https://en.wikipedia.org/wiki/Java_bytecode
2. Compatibility Guide for JDK 8 - Oracle, accessed April 26, 2025, https://www.oracle.com/java/technologies/javase/8-compatibility-guide.html
3. Is java byte-code always forward compatible? - jvm - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/49445775/is-java-byte-code-always-forward-compatible
4. Backward compatibility -- what does it really mean? - Code Ranch, accessed April 26, 2025, https://coderanch.com/t/633908/java/compatibility
5. Can newer versions of java run on older versions? [duplicate] - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/65850230/can-newer-versions-of-java-run-on-older-versions
6. Can I run a java program built using Java 8 (.192) using JRE 17, or does everything (JRE, JVM, JDK) have to be on the correct Java 8 version first? - Reddit, accessed April 26, 2025, https://www.reddit.com/r/javahelp/comments/uclu4x/can_i_run_a_java_program_built_using_java_8_192/
7. Can I run a java program built using Java 8 (.192) using JRE 17, or does everything (JRE, JVM, JDK) have to be on the correct Java 8 version first? - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/72020186/can-i-run-a-java-program-built-using-java-8-192-using-jre-17-or-does-everyth
8. FAQ: Upgrade from Java 11 to 17 | Crownpeak Community, accessed April 26, 2025, https://community.crownpeak.com/t5/Helpful-Information/FAQ-Upgrade-from-Java-11-to-17/ta-p/57329
9. Why do companies still use older Java releases - Reddit, accessed April 26, 2025, https://www.reddit.com/r/java/comments/140tw3l/why_do_companies_still_use_older_java_releases/
10. Why is compiled java not forwards compatible - jvm - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/31713744/why-is-compiled-java-not-forwards-compatible
11. If I compiled a Java file with the newest JDK, would an older JVM be able to run the .class files? - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/4061965/if-i-compiled-a-java-file-with-the-newest-jdk-would-an-older-jvm-be-able-to-run
12. Is Java bytecode compatible within different updates of the same version of

Java?, accessed April 26, 2025, https://stackoverflow.com/questions/1476126/is-java-bytecode-compatible-within-different-updates-of-the-same-version-of-java
13. Class File Versions - javaalmanac.io, accessed April 26, 2025, https://javaalmanac.io/bytecode/versions/
14. JDK bytecode version - Question - Scala Users, accessed April 26, 2025, https://users.scala-lang.org/t/jdk-bytecode-version/10191
15. Why is Java not forwards compatible in all cases? : r/learnjava - Reddit, accessed April 26, 2025, https://www.reddit.com/r/learnjava/comments/twgx1z/why_is_java_not_forwards_compatible_in_all_cases/
16. Break backward compatibility : r/java - Reddit, accessed April 26, 2025, https://www.reddit.com/r/java/comments/nyre4k/break_backward_compatibility/
17. Yes. But in the move to java 9 they broke java's customary backward compatibilit... | Hacker News, accessed April 26, 2025, https://news.ycombinator.com/item?id=28541675
18. How much faster is Java 17? - Hacker News, accessed April 26, 2025, https://news.ycombinator.com/item?id=28540099
19. Upgrading From Java 17 To 21: All You Need To Know - nipafx.dev, accessed April 26, 2025, https://nipafx.dev/road-to-21-upgrade/
20. Significant Changes in JDK 17 Release - Oracle Help Center, accessed April 26, 2025, https://docs.oracle.com/en/java/javase/17/migrate/significant-changes-jdk-release.html
21. Comparative Analysis of Java 17 and Java 21 Features - Brainvire, accessed April 26, 2025, https://www.brainvire.com/blog/java-17-vs-java-21/
22. A categorized list of all Java and JVM features since JDK 8 to 21 - Advanced Web Machinery, accessed April 26, 2025, https://advancedweb.hu/a-categorized-list-of-all-java-and-jvm-features-since-jdk-8-to-21/
23. Top 8 Features of JDK 17 That You Must Know: LTS, Features & Performance In Java, accessed April 26, 2025, https://www.codingshuttle.com/blogs/jdk-17-new-features-lts-features-and-performance-in-java/
24. Top 7 Reasons To Migrate From Java 8 to Java 17 | GeeksforGeeks, accessed April 26, 2025, https://www.geeksforgeeks.org/top-reasons-to-migrate-from-java-8-to-java-17/
25. What is the point of all these new Java versions when Java 8 is all that seems to be supported by most apps? - Reddit, accessed April 26, 2025, https://www.reddit.com/r/java/comments/mu4sar/what_is_the_point_of_all_these_new_java_versions/
26. Java 17 features: A comparison between versions 8 and 17 - Pretius, accessed April 26, 2025, https://pretius.com/blog/java-17-features/
27. Can I use newer Java APIs when running older Java programs with a newer JRE, accessed April 26, 2025,

https://stackoverflow.com/questions/76628100/can-i-use-newer-java-apis-when-running-older-java-programs-with-a-newer-jre
28. Question: Compiling Older Versions of Java Using by Maven : r/javahelp - Reddit, accessed April 26, 2025, https://www.reddit.com/r/javahelp/comments/vux9bs/question_compiling_older_versions_of_java_using/
29. Why would I want to target JVM > 8? - Kotlin - Reddit, accessed April 26, 2025, https://www.reddit.com/r/Kotlin/comments/rsvgpr/why_would_i_want_to_target_jvm_8/
30. jvm - Java Compiler and VM Compatibility - Software Engineering ..., accessed April 26, 2025, https://softwareengineering.stackexchange.com/questions/205664/java-compiler-and-vm-compatibility
31. How to handle Java version compatibility - LabEx, accessed April 26, 2025, https://labex.io/tutorials/java-how-to-handle-java-version-compatibility-421340
32. Java Evolution: Unlocking Performance and Efficiency from Java 8 to 17, accessed April 26, 2025, https://www.verygoodsecurity.com/blog/posts/java-evolution-unlocking-performance-and-efficiency-from-java-8-to-17
33. Performance regressions migrating from java 8 to java 17 : r/javahelp - Reddit, accessed April 26, 2025, https://www.reddit.com/r/javahelp/comments/1gfwk78/performance_regressions_migrating_from_java_8_to/
34. JVM Performance Comparison for JDK 17 - Ionut Balosin, accessed April 26, 2025, https://ionutbalosin.com/2023/03/jvm-performance-comparison-for-jdk-17/
35. It's time to move your applications to Java 17. Here's why—and how. - Oracle Blogs, accessed April 26, 2025, https://blogs.oracle.com/javamagazine/post/its-time-to-move-your-applications-to-java-17-heres-why-and-heres-how
36. Consolidated JDK 17 Release Notes - Oracle, accessed April 26, 2025, https://www.oracle.com/java/technologies/javase/17all-relnotes.html
37. How to Migrate from Java 8 to Java 17: A Step-by-Step Guide - BellSoft, accessed April 26, 2025, https://bell-sw.com/blog/migration-from-java-8-to-java-17/
38. Migrate to Java 17 | OpenRewrite Docs, accessed April 26, 2025, https://docs.openrewrite.org/running-recipes/popular-recipe-guides/migrate-to-java-17
39. Is it viable to compile my project in java8 and then run it on java 17 - Stack Overflow, accessed April 26, 2025, https://stackoverflow.com/questions/69792839/is-it-viable-to-compile-my-project-in-java8-and-then-run-it-on-java-17
40. Switching from Java 8 to Java 17 as the minimum required for the core platform and modules! - OpenMRS Talk, accessed April 26, 2025, https://talk.openmrs.org/t/switching-from-java-8-to-java-17-as-the-minimum-required-for-the-core-platform-and-modules/45188

41. Question about JVM versions on Android devices : r/androiddev - Reddit, accessed April 26, 2025, https://www.reddit.com/r/androiddev/comments/1berxjn/question_about_jvm_versions_on_android_devices/