Java Platform, Standard Edition

Oracle JDK Migration Guide

Release 25
G35926-01
September 2025

Java Platform, Standard Edition Oracle JDK Migration Guide, Release 25

G35926-01

Contents

Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documents i
Conventions i

1 Getting Started

2 Significant Changes in the JDK

Significant Changes in JDK 25 Release 1
Significant Changes in JDK 24 Release 3
Significant Changes in JDK 23 Release 7
Significant Changes in JDK 22 Release 9
Significant Changes in JDK 21 Release 11
Significant Changes in JDK 20 Release 13
Significant Changes in JDK 19 Release 14
Significant Changes in JDK 18 Release 15
Significant Changes in JDK 17 Release 16
Significant Changes in JDK 16 Release 18
Significant Changes in JDK 15 Release 19
Significant Changes in JDK 14 Release 20
Significant Changes in JDK 13 Release 21
Significant Changes in JDK 12 Release 21
Significant Changes in JDK 11 Release 22

3 Security Updates

Security Updates in JDK 25 1
Security Updates in JDK 24 1
Security Updates in JDK 23 1
Security Updates in JDK 22 2

Security Updates in JDK 21 2
Security Updates in JDK 20 2
Security Updates in JDK 19 2
Security Updates in JDK 18 3
Security Updates in JDK 17 3
Security Updates in JDK 16 3
Security Updates in JDK 15 4
Security Updates in JDK 14 4
Security Updates in JDK 13 4
Security Updates in JDK 11 and JDK 12 4
Security Updates in JDK 9 and JDK 10 5
JCE Jurisdiction Policy File Default is Unlimited 5
Create PKCS12 Keystores 5

4 Removed APIs

APIs Removed in Java SE 25 1
APIs Removed in Java SE 24 1
APIs Removed in Java SE 23 1
APIs Removed in Java SE 22 2
APIs Removed in Java SE 21 2
APIs Removed in Java SE 20 2
APIs Removed in Java SE 19 2
APIs Removed in Java SE 18 2
APIs Removed in Java SE 17 2
API Removed in Java SE 16 3
APIs Removed in Java SE 15 3
APIs Removed in Java SE 14 3
APIs Removed in Java SE 13 4
APIs Removed in Java SE 12 4
APIs Removed in JDK 11 4
APIs Removed in JDK 10 5
APIs Removed JDK 9 5
Removed java.* APIs 5
Removal and Future Removal of sun.misc and sun.reflect APIs 5
java.awt.peer Not Accessible 6
Removed com.sun.image.codec.jpeg Package 6
Removed Tools Support for Compact Profiles 6

5 Removed Tools and Components

Features and Options Removed and Deprecated in JDK 25 1

Features and Options Removed and Deprecated in JDK 24 1
Features and Options Removed and Deprecated in JDK 23 2
Features and Options Removed and Deprecated in JDK 22 3
Features and Options Removed and Deprecated in JDK 21 3
Features and Options Deprecated in JDK 20 3
Features and Options Deprecated in JDK 19 4
Tools and Components Removed and Deprecated in JDK 18 4
Tools and Components Removed and Deprecated in JDK 17 4
Tools and Components Removed and Deprecated in JDK 16 4
Tools and Components Removed and Deprecated in JDK 15 5
Features and Components Removed in JDK 14 6
Tools and Components Removed in JDK 13 6
Tools and Components Removed in JDK 12 6
Tools and Components Removed in JDK 11 6
Tools and Components Removed in JDK 9 and JDK 10 8
Removed Native-Header Generation Tool (javah) 8
Removed JavaDB 8
Removed the JVM TI hprof Agent 9
Removed the jhat Tool 9
Removed java-rmi.exe and java-rmi.cgi Launchers 9
Removed Support for the IIOP Transport from the JMX RMIConnector 9
Dropped Windows 32–bit Client VM 10
Removed Java VisualVM 10
Removed native2ascii Tool 10

6 Preparing for Migration

Download the Latest JDK 1
Run Your Program Before Recompiling 1
Update Third-Party Libraries 2
Compile Your Application if Needed 2
Run jdeps on Your Code 3

7 Migrating from JDK 8 to Later JDK Releases

Changes to Internationalization 1
Be Aware of the Default Charset 1
Be Aware of Changes to Locale Data 2
Strong Encapsulation in the JDK 5
--add-exports Option 6
--add-opens Option 7
New Version-String Scheme 8

Changes to the Installed JDK/JRE Image 8
Changed JDK and JRE Layout 8
New Class Loader Implementations 9
Removed rt.jar and tools.jar 10
Removed Extension Mechanism 10
Removed Endorsed Standards Override Mechanism 11
Removed macOS-Specific Features 11
Platform-Specific Desktop Features 11
Removed AppleScript Engine 12
Windows Registry Key Changes 12
Deployment 12
Removed Launch-Time JRE Version Selection 12
Removed Support for Serialized Applets 13
JNLP Specification Update 13
Changes to Garbage Collection 14
Make G1 the Default Garbage Collector 14
Removed GC Options 14
Changes to GC Log Output 15
Running Java Applets 15
Behavior Change in Regular Expression Matching 15
Plan for Removal of the Security Manager 16
Finalization Deprecated for Removal 16
Restrictions on and Warnings from Accessing Native Code 16

8 Next Steps

Preface

The purpose of this guide is to help you identify potential issues and give you suggestions on how to proceed as you migrate your existing Java application to JDK 25.

Audience

This guide is intended for experienced users of the Java language, such as systems administrators and software developers, for whom the performance of the Java platform and their applications is of vital importance.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

See JDK 25 Documentation for other JDK 25 guides.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.

Convention	Meaning
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Getting Started

The guide highlights the significant changes and enhancements done in JDK 25.

This guide contains the following sections:

- Significant Changes in the JDK
- Security Updates
- Removed APIs
- Removed Tools and Components
- Preparing for Migration
- Migrating from JDK 8 to Later JDK Releases
- Next Steps

Note

- Check the Oracle JDK Certified System Configurations page for the latest supported platforms and operating system versions.
- See Removed APIs, Tools, and Components before you start the migration process.
- See Be Aware of the Default Charset if you are upgrading to JDK 17 or later.

2

Significant Changes in the JDK

Before migrating your application to the latest JDK release, you must understand what the updates and changes are between it and the previous JDK release. If you are migrating from JDK 8, you should also be familiar with the differences between JDK 8 and later releases that are described in Migrating from JDK 8 to Later JDK Releases.

See the following sections to learn about some of the significant changes in JDK releases.

Significant Changes in JDK 25 Release

See JDK 25 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 25.

The following are some of the updates in Java SE 25 and JDK 25:

Language Preview Features

- Pattern Matching is enhanced by allowing primitive types in all pattern contexts. The `instanceof` operator and `switch` expressions and statements are extended to work with all primitive types.
  First previewed in Java SE 23, this feature is re-previewed for this release. There are no significant changes between Java SE 23 and this release.

  See JEP 507: Primitive Types in Patterns, instanceof, and switch (Third Preview) and the following sections in *Java Platform, Standard Edition Java Language Updates*:

  – When Clauses
  – Type Patterns with Primitive Types
  – Primitive Types in Record Patterns
  – Primitive Values in switch Expressions and Statements
  – Safe casting with instanceof and switch

- Module Import Declarations enable you to succinctly import all of the packages exported by a module. This simplifies the reuse of modular libraries without requiring the importing code to be in a module itself.
  First previewed in Java SE 23, this feature is re-previewed for this release. There are no significant changes between Java SE 23 and this release.

  See JEP 511: Module Import Declarations and Module Import Declarations in *Java Platform, Standard Edition Java Language Updates*.

- Compact source files and instance `main` methods enable students to write their first programs without needing to understand the full set of language features designed for large programs.
  First previewed in Java SE 21 as JEP 445: Unnamed Classes and Instance Main Methods (Preview) and previewed again in Java SE 22, 23, and 24, this feature is permanent in this release with a revised title. In this release:

  – The new `IO` class for basic console I/O is now in the `java.lang` package rather than the `java.io` package. Thus, it is implicitly imported by every source file.

  - The `static` methods of the `IO` class are no longer implicitly imported into compact source files. Thus invocations of these methods must name the class, for example, `IO.println("Hello, world!")`, unless the methods are explicitly imported.
  - The implementation of the `IO` class is now based upon `System.out` and `System.in` rather than the `java.io.Console` class.

  See JEP 512: Compact Source Files and Instance Main Methods and Compact Source Files and Instance main Methods in *Java Platform, Standard Edition Java Language Updates*.

- Flexible Constructor Bodies allow statements in a constructor to appear before an explicit constructor invocation, such as `super(..)` or `this(..)`. These statements cannot reference the instance under construction, but they can initialize its fields. Initializing fields before invoking another constructor makes a class more reliable when methods are overridden.
  First previewed in Java SE 22 as JEP 447: Statements before super(...) (Preview) and previewed again in Java SE 23 and Java SE 24, this feature is permanent in this release without any significant changes.

  See JEP 513: Flexible Constructor Bodies and Flexible Constructor Bodies in *Java Platform, Standard Edition Java Language Updates*.

See JEP 12: Preview Features and Preview Language and VM Features in *Java Platform, Standard Edition Java Language Updates*.

Library Improvements, Previews, and Incubator

- Structured Concurrency is an API that simplifies concurrent programming. It treats groups of related tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
  See JEP 505: Structured Concurrency (Fifth Preview) and Structured Concurrency in *Java Platform, Standard Edition Core Libraries*.
- Scoped Values enable a method to share immutable data both with its callees within a thread and with child threads. They are easier to reason about than thread-local variables. They also have lower space and time costs, especially when used together with virtual threads and structured concurrency.
  See JEP 506: Scoped Values and the ScopedValue class in the Java API Specification.
- Stable Values API hold immutable data. They are treated as constants by the JVM, enabling the same performance optimizations that are enabled by declaring a field `final`. Compared to `final` fields, stable values offer greater flexibility as to the timing of their initialization.
  See JEP 502: Stable Values (Preview), StableValue in the Java API Specification, and the Stable Values in *Java Platform, Standard Edition Core Libraries*.
- The Vector API expresses vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations.
  See JEP 508: Vector API (Tenth Incubator) and JEP 11: Incubator Modules.

Security Libraries

- An API has been introduced for encoding objects that represent cryptographic keys, certificates, and certificate revocation lists into the widely-used Privacy-Enhanced Mail (PEM) transport format, and for decoding from that format back into objects.
  See JEP 470: PEM Encodings of Cryptographic Objects (Preview) and The DEREncodable Interface in *Java Platform, Standard Edition Security Developer's Guide*.
- An API has been introduced for Key Derivation Functions (KDFs), which are cryptographic algorithms for deriving additional keys from a secret key and other data.

See JEP 510: Key Derivation Function API and The KDF Class in *Java Platform, Standard Edition Security Developer's Guide*.

Performance and Runtime Improvements

- The size of object headers in the HotSpot JVM is reduced from 96 and 128 bits to 64 bits on 64-bit architectures. This will reduce heap size, improve deployment density, and increase data locality. This experimental feature is now changed to a product feature.
  See JEP 519: Compact Object Headers.
- Ahead-of-Time (AOT) Command-Line Ergonomics simplifies the process of creating ahead-of-time (AOT) caches by reducing the complexity of the commands required for common use cases. AOT caches help accelerate the startup of Java applications, making them more efficient and responsive.
  First introduced in JEP 483: Ahead-of-Time Class Loading and Linking, this enhancement brings new AOT-related optimizations to the HotSpot JVM.

  See JEP 514: Ahead-of-Time Command-Line Ergonomics.
- Ahead-of-Time Method Profiling improves the startup time by making the method-execution profiles from a previous run of an application instantly available when the HotSpot Java Virtual Machine starts. This allows the JIT compiler to generate the native code right at application startup, eliminating the need to wait for profile data to be collected during the current run.
  See JEP 515: Ahead-of-Time Method Profiling.

Monitoring

- JFR CPU-Time Profiling : JDK Flight Recorder (JFR) has been enhanced to collect more accurate CPU-time profiling data on Linux. This feature is currently experimental.
  See JEP 509: JFR CPU-Time Profiling (Experimental) and JDK Flight Recorder.
- JFR Cooperative Sampling: Stability of the JDK Flight Recorder (JFR) has been enhanced when it asynchronously samples Java thread stacks by restricting call stack walking to safepoints, while reducing safepoint bias.
  See JEP 518: JFR Cooperative Sampling.
- JFR Method Timing and Tracing: JDK Flight Recorder (JFR) has been extended to support method timing and tracing using bytecode instrumentation. Timing and tracing method invocations can help to identify performance bottlenecks, optimize code, and find the root causes of bugs.
  See JEP 520: JFR Method Timing & Tracing.

Removals and Warnings for Future Changes

Removal of Experimental Features Graal JIT: The optional experimental Graal JIT compiler has been removed.

For details on removals and deprecations, see Features and Options Removed and Deprecated in JDK 25.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 25.

Significant Changes in JDK 24 Release

See JDK 24 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 24.

The following are some of the updates in Java SE 24 and JDK 24:

Language Preview Features

- Pattern Matching is enhanced by allowing primitive types in all pattern contexts. The `instanceof` operator and `switch` expressions and statements are extended to work with all primitive types.
  First previewed in Java SE 23, this feature is re-previewed for this release. It is unchanged between Java SE 23 and this release.

  See JEP 488: Primitive Types in Patterns, instanceof, and switch (Second Preview) and the following sections in *Java Platform, Standard Edition Java Language Updates*:

  - When Clauses
  - Type Patterns with Primitive Types
  - Primitive Types in Record Patterns
  - Primitive Values in switch Expressions and Statements
  - Safe casting with instanceof and switch
- Module Import Declarations enable you to succinctly import all of the packages exported by a module. This simplifies the reuse of modular libraries without requiring the importing code to be in a module itself.
  First previewed in Java SE 23, this feature is re-previewed for this release. In this release:

  - Type-import-on-demand declarations shadow module import declarations.
  - Modules may declare a transitive dependence on the `java.base` module.
  - The `java.se` module transitively requires the `java.base` module. Consequently, importing the `java.se` module imports the entire Java SE API.

  See JEP 494: Module Import Declarations (Second Preview) and Module Import Declarations in *Java Platform, Standard Edition Java Language Updates*.
- Flexible Constructor Bodies allow statements in a constructor to appear before an explicit constructor invocation, such as `super(..)` or `this(..)`. These statements cannot reference the instance under construction, but they can initialize its fields. Initializing fields before invoking another constructor makes a class more reliable when methods are overridden.
  First previewed in Java SE 22 as JEP 447: Statements before super(...) (Preview) and previewed again in Java SE 23 as JEP 482: Flexible Constructor Bodies (Second Preview). This feature is re-previewed for this release without any significant changes.

  See JEP 492: Flexible Constructor Bodies (Third Preview) and Flexible Constructor Bodies in *Java Platform, Standard Edition Java Language Updates*.
- Simple Source Files and Instance Main Methods enable beginners to write their first programs without needing to understand language features designed for large programs. Far from using a separate dialect of the language, beginners can write streamlined declarations for single-class programs and then seamlessly expand their programs to use more advanced features as their skills grow. Experienced developers can likewise enjoy writing small programs succinctly, without the need for constructs intended for programming in the large.
  First previewed in Java SE 21 as JEP 445: Unnamed Classes and Instance Main Methods (Preview) and previewed again in Java SE 23. This feature is re-previewed for this release with new terminology and a revised title but otherwise unchanged.

  See JEP 495: Simple Source Files and Instance Main Methods (Fourth Preview) and Simple Source Files and Instance Main Methods in *Java Platform, Standard Edition Java Language Updates*.

See JEP 12: Preview Features and Preview Language and VM Features in *Java Platform, Standard Edition Java Language Updates*.

Performance and Runtime Improvements

- The size of object headers in the HotSpot JVM is reduced from 96 and 128 bits to 64 bits on 64-bit architectures. This will reduce heap size, improve deployment density, and increase data locality. This is an experimental feature and can be enabled by passing the command line options:

  ```
  +UnlockExperimentalVMOptions -XX:+UseCompactObjectHeaders
  ```

  See JEP 450: Compact Object Headers (Experimental).
- The implementation of the G1 garbage collector's barriers has been simplified, which record information about application memory accesses, by shifting their expansion from early in the C2 JIT's compilation pipeline to later.
  See JEP 475: Late Barrier Expansion for G1.
- Ahead-of-Time Class Loading and Linking improves the startup time by making the classes of an application instantly available, in a loaded and linked state, when the HotSpot Java Virtual Machine starts. This is achieved by monitoring the application during one run, and storing the loaded and linked forms of all classes in a cache for subsequent runs.
  See JEP 483: Ahead-of-Time Class Loading & Linking
- The ZGC garbage collector runs in the generational mode by default. The non-generational mode of ZGC is removed.
  See JEP 490: ZGC: Remove the Non-Generational Mode and the section The Z Garbage Collector in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.
- Synchronize Virtual Threads without Pinning improves the scalability of Java code that uses `synchronized` methods and statements by arranging for virtual threads that block in such constructs to release their underlying platform threads for use by other virtual threads.
  See JEP 491: Synchronize Virtual Threads without Pinning.

Library Improvements, Previews, and Incubator

- The Class-File API is a standard API for parsing, generating, and transforming Java class files.
  See JEP 484: Class-File API and Class-File API in *Java Platform, Standard Edition Java Virtual Machine Guide*.
- Stream Gatherers enable you to create custom intermediate operations, which allow stream pipelines to transform data in ways that are not easily achievable with the existing built-in intermediate operations.
  See JEP 485: Stream Gatherers and Stream Gatherers in *Java Platform, Standard Edition Core Libraries*.
- Structured Concurrency is an API that simplifies concurrent programming. It treats groups of related tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
  See JEP 499: Structured Concurrency (Fourth Preview) and Structured Concurrency in *Java Platform, Standard Edition Core Libraries*.
- Scoped Values enable a method to share immutable data both with its callees within a thread and with child threads. They are easier to reason about than thread-local variables. They also have lower space and time costs, especially when used together with virtual threads and structured concurrency.

See JEP 487: Scoped Values (Fourth Preview) and the ScopedValue class in the Java API Specification.

- The Vector API expresses vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations.
  See JEP 489: Vector API (Ninth Incubator) and JEP 11: Incubator Modules.

Security Libraries

- An API has been introduced for Key Derivation Functions (KDFs), which are cryptographic algorithms for deriving additional keys from a secret key and other data. This is a preview API.
  See JEP 478: Key Derivation Function API (Preview) and The KDF Class in *Java Platform, Standard Edition Security Developer's Guide*.
- The security of Java applications is enhanced by providing an implementation of the quantum-resistant Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). Key encapsulation mechanisms (KEMs) are used to secure symmetric keys over insecure communication channels using public key cryptography.
  See JEP 496: Quantum-Resistant Module-Lattice-Based Key Encapsulation Mechanism.
- The security of Java applications is enhanced by providing an implementation of the quantum-resistant Module-Lattice-Based Digital Signature Algorithm (ML-DSA). Digital signatures are used to detect unauthorized modifications to data and to authenticate the identity of signatories.
  See JEP 497: Quantum-Resistant Module-Lattice-Based Digital Signature Algorithm.

Removals and Warnings for Future Changes

- Prepare to Restrict the Use of JNI: Warnings are issued about uses of the Java Native Interface (JNI), and the Foreign Function and Memory (FFM) API is adjusted to issue warnings in a consistent manner. All such warnings aim to prepare developers for a future release that ensures integrity by default by uniformly restricting JNI and the FFM API.
  See JEP 472: Prepare to Restrict the Use of JNI and Restrictions on and Warnings from Accessing Native Code.
- Permanently Disable the Security Manager: The Security Manager is permanently disabled, and the Security Manager API will be removed in a future release.
  See JEP 486: Permanently Disable the Security Manager and The Security Manager Is Permanently Disabled in *Java Platform, Standard Edition Security Developer's Guide*.
- ZGC: Remove the Non-Generational Mode: The ZGC garbage collector runs in the generational mode by default. The non-generational mode of ZGC is removed.
  See JEP 490: ZGC: Remove the Non-Generational Mode and the section The Z Garbage Collector in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.
- Warn upon Use of Memory-Access Methods in sun.misc.Unsafe: A warning is issued at run time on the first occasion that any memory-access method in `sun.misc.Unsafe` is invoked. All of these unsupported methods were terminally deprecated in JDK 23. They have been superseded by standard APIs, namely the `VarHandle` API and the Foreign Function & Memory API.
  See JEP 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe.

For more details on removals and deprecations, see Features and Options Removed and Deprecated in JDK 24.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 24.

Significant Changes in JDK 23 Release

See JDK 23 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 23.

The following are some of the updates in Java SE 23 and JDK 23:

Language Preview Features

- Pattern matching is now enhanced by allowing primitive type patterns in all pattern contexts. The `instanceof` and `switch` operators, expressions, and statements work with all primitive types.
  See JEP 455: Primitive Types in Patterns, instanceof, and switch (Preview) and the following sections in *Java Platform, Standard Edition Java Language Updates*:
  - When Clauses
  - Type Patterns with Primitive Types
  - Primitive Types in Record Patterns
  - Primitive Values in switch Expressions and Statements
  - Safe casting with instanceof and switch
- Module Import Declarations enable you to succinctly import all of the packages exported by a module with a module import declaration. It simplifies the reuse of modular libraries without requiring the imported code to be in a module.
  See JEP 476: Module Import Declarations (Preview) and Module Import Declarations in *Java Platform, Standard Edition Java Language Updates*.
- Flexible Constructor Bodies enable you to add statements that don't reference the instance being created before an explicit constructor invocation. This feature helps you to prepare arguments for a superclass constructor by performing nontrivial computations or to validate arguments you want to pass to a superclass constructor.
  See JEP 482: Flexible Constructor Bodies (Second Preview) and Flexible Constructor Bodies in *Java Platform, Standard Edition Java Language Updates*.
- Implicitly Declared Classes and Instance Main Methods enable students to write their first programs even without learning the full set of language features designed for large programs. Students can write streamlined declarations for single-class programs and then seamlessly expand their programs to use more advanced features as their skills grow.
  See JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview) and Implicitly Declared Classes and Instance Main Methods in *Java Platform, Standard Edition Java Language Updates*.
- String Templates were first previewed in JDK 21 (JEP 430) and in JDK 22 (JEP 459). String templates were intended to be re-previewed in JDK 23 (JEP 465). However, after extensive feedback and discussion, we concluded that the feature is unsuitable in its current form. There is no consensus on what a better design will be; therefore, we have withdrawn the feature for now, and JDK 23 will not include it.
  See March 2024 Archives by thread and Update on String Templates (JEP 459) from the Project Amber amber-spec-experts mailing list for further discussion.

See JEP 12: Preview Features and Preview Language and VM Features in *Java Platform, Standard Edition Java Language Updates*.

Library Improvements Previews and Incubator

- The Class-File API is used for parsing, generating, and transforming Java class files.

See JEP 466: Class-File API (Second Preview) and Class-File API in *Java Platform, Standard Edition Java Virtual Machine Guide*.

- Stream gatherers enable you to create custom intermediate operations, which allow stream pipelines to transform data in ways that aren't easily achievable with existing built-in intermediate operations.
  See JEP 473: Stream Gatherers (Second Preview) and Stream Gatherers in *Java Platform, Standard Edition Core Libraries*.
- Structured concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
  See JEP 480: Structured Concurrency (Third Preview) and Structured Concurrency in *Java Platform, Standard Edition Core Libraries*.
- Scoped values enables sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using large numbers of virtual threads.
  See JEP 481: Scoped Values (Third Preview) and the ScopedValue class in the Java API Specification.
- The Vector API is introduced to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations. See JEP 469: Vector API (Eighth Incubator).

See JEP 11: Incubator Modules.

Runtime Improvements

- The ZGC garbage collector now runs in the generational mode by default. The non-generational mode of ZGC has been deprecated for removal. This will generally improve application performance, although a small number of workloads that are non-generational in nature may be impacted negatively.
  See JEP 474: ZGC: Generational Mode by Default and the section The Z Garbage Collector in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.
- The Oracle GraalVM JIT compiler (Graal JIT) is now available as an experimental option among the JITs available as part of the Oracle JDK. This integration offers innovations previously made available through Oracle GraalVM, such as novel JIT code optimization techniques (see Compiler Advantages in the GraalVM documentation). This provides developers and system administrators more options to help fine tune and improve peak performance of their applications. The Graal JIT is enabled by passing the command line options to the Java executable:

  ```
  -XX:+UnlockExperimentalVMOptions -XX:+UseGraalJIT
  ```

  If you do not pass these flags at JVM startup, the Oracle JDK default JIT (C2) will run as usual.

Tooling

JavaDoc documentation comments can now be written in Markdown syntax along with HTML elements and JavaDoc tags.

See JEP 467: Markdown Documentation Comments and Markdown in Documentation Comments in *Java Platform, Standard Edition JavaDoc Guide*.

Removed APIs, Tools, and Components

For more details on removals and deprecations, see Features and Options Removed and Deprecated in JDK 23.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 23.

Significant Changes in JDK 22 Release

See JDK 22 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 22.

The following are some of the updates in Java SE 22 and JDK 22:

Language Features

- Unnamed variables and unnamed patterns can be used when variable declarations or nested patterns are required but never used. Unnamed patterns can appear in a pattern list of a record pattern and can be used instead of a type pattern. You denote both with the underscore character (`_`).
  See JEP 456: Unnamed Variables & Patterns and Unnamed Variables and Patterns in *Java Platform, Standard Edition Java Language Updates*.

Preview Language Features

- In constructors, you can add statements that don't reference the instance being created before an explicit constructor invocation. You can use this feature to prepare arguments for a superclass constructor by performing nontrivial computations or validate arguments you want to pass to a superclass constructor.
  See JEP 447: Statements before super(...) and Statements Before super(...) in *Java Platform, Standard Edition Java Language Updates*.
- String templates complement Java's existing string literals and text blocks by coupling literal text with embedded expressions and template processors to produce specialized results. Embedded expressions are Java expressions with additional syntax that differentiates them from the literal text in the string template. A template processor combines the literal text in the template with the values of the embedded expressions to produce a result.
  See JEP 459: String Templates (Second Preview) and String Templates in *Java Platform, Standard Edition Java Language Updates*.
- The preview features Instance Main Methods and Implicitly Declared Classes enable students to write their first programs without needing to understand the full set of language features designed for large programs. They can write streamlined declarations for single-class programs and then seamlessly expand their programs to use more advanced features as their skills grow.
  See JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview) and Implicitly Declared Classes and Instance Main Methods in *Java Platform, Standard Edition Java Language Updates*.

See Preview Language and VM Features for more information about preview features.

Library Improvements

- The Foreign Function and Memory (FFM) API enables Java programs to interoperate with code and data outside the Java runtime. This API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI. The API

invokes foreign functions, code outside the JVM, and safely accesses foreign memory, memory not managed by the JVM.
See JEP 454: Foreign Function & Memory API and Foreign Function and Memory API in *Java Platform, Standard Edition Core Libraries*.

Library Improvements Previews and Incubator

- Stream gatherers enable you to create custom intermediate operations, which allow stream pipelines to transform data in ways that aren't easily achievable with existing built-in intermediate operations.
  See JEP 461: Stream Gatherers (Preview) and Stream Gatherers in *Java Platform, Standard Edition Core Libraries*.
- The Class-File API is used for parsing, generating, and transforming Java class files.
  See JEP 457: Class-File API (Preview) and Class-File API in *Java Platform, Standard Edition Java Virtual Machine Guide*.
- Structured concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
  See JEP 462: Structured Concurrency (Second Preview) and Structured Concurrency in *Java Platform, Standard Edition Core Libraries*.
- Scoped values enables sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using large numbers of virtual threads.
  See JEP 464: Scoped Values (Second Preview) and the ScopedValue class in the Java API Specification.
- The Vector API is introduced to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations. See JEP 460: Vector API (Seventh Incubator).

See JEP 12: Preview Features and JEP 11: Incubator Modules for more information about preview features and incubating APIs.

Performance Improvements

Region pinning in G1 reduces latency so that garbage collection need not be disabled during Java Native Interface (JNI) critical regions.

See JEP 423: Region Pinning for G1 and the section Evacuation Failure in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.

Tooling

The Java application launcher is enhanced to run a program supplied as multiple files of Java source code. This will enable gradual transition from small programs to larger ones.

See JEP 458: Launch Multi-File Source-Code Programs.

Removed APIs, Tools, and Components

For more details on removals and deprecations, see Features and Options Removed and Deprecated in JDK 22.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 22.

Significant Changes in JDK 21 Release

See JDK 21 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 21.

The following are some of the updates in Java SE 21 and JDK 21:

Language Features

- Record patterns and type patterns can be nested to enable a powerful, declarative, and composable form of data navigation and processing. You can use a record pattern to test whether a value is an instance of a record class type and, if it is, to recursively perform pattern matching on its component values. In this release, support for record patterns appearing in the header of an enhanced `for` statement has been removed.
  See JEP 440: Record Patterns and Record Patterns in *Java Platform, Standard Edition Java Language Updates*.
- A `switch` statement transfers control to one of several statements or expressions, depending on the value of its selector expression. Extending pattern matching to `switch` allows an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely. In this release:
  - Parenthesized patterns have been removed.
  - Qualified `enum` constants as `case` constants in `switch` expressions and statements are allowed.

  See JEP 441: Pattern Matching for switch and Pattern Matching for switch Expressions and Statements in *Java Platform, Standard Edition Java Language Updates*.

Preview Language Features

- String templates complement Java's existing string literals and text blocks by coupling literal text with embedded expressions and template processors to produce specialized results.
  See JEP 430: String Templates (Preview) and String Templates in *Java Platform, Standard Edition Java Language Updates*.
- Unnamed patterns match a record component without stating the component's name or type. Unnamed variables are variables that can be initialized but not used. You denote both with the underscore character (_).
  See JEP 443: Unnamed Patterns and Variables (Preview) and Unnamed Patterns and Variables in *Java Platform, Standard Edition Java Language Updates*.
- Unnamed classes and instance main methods enable students to write streamlined declarations for single-class programs and then seamlessly expand their programs later to use more advanced features as their skills grow.
  See JEP 445: Unnamed Classes and Instance Main Methods (Preview) and Unnamed Classes and Instance Main Methods in *Java Platform, Standard Edition Java Language Updates*.

See Preview Language and VM Features for more information about preview features.

Library Improvements

- Virtual threads are lightweight threads that reduce the effort of writing, maintaining, and debugging high-throughput concurrent applications.
  See JEP 444: Virtual Threads and Virtual Threads in *Java Platform, Standard Edition Core Libraries*.

- Sequenced collections are collections with a defined encounter order. Each such collection has a well-defined first element, second element, and so forth, up to the last element. It is an interface that represents a sequenced collection, provides uniform APIs for accessing its first and last elements, and for processing its elements in reverse order.
  See JEP 431: Sequenced Collections and Creating Sequenced Collections, Sets, and Maps in *Java Platform, Standard Edition Core Libraries*.
- The Key Encapsulation Mechanism API is introduced for key encapsulation mechanisms (KEMs), an encryption technique for securing symmetric keys using public key cryptography.
  See JEP 452: Key Encapsulation Mechanism API and The KEM Class in *Java Platform, Standard Edition Security Developer's Guide*.

Library Improvements Previews and Incubator

- The preview API Foreign Function and Memory API has been further refined as follows:
  - Centralized the management of the lifetimes of native segments in the `Arena` interface
  - Enhanced layout paths with a new element to dereference address layouts
  - Provided a linker option to optimize calls to functions that are short-lived and will not upcall to Java (e.g., `clock_gettime`)
  - Provided a fallback native linker implementation, based on `libffi`, to facilitate porting
  - Removed the `VaList` class

  See JEP 442: Foreign Function & Memory API (Third Preview) and Foreign Function and Memory API in *Java Platform, Standard Edition Core Libraries*.
- Structured concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability.
  See JEP 453: Structured Concurrency (Preview) and Structured Concurrency in *Java Platform, Standard Edition Core Libraries*.
- Scoped values enables sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using large numbers of virtual threads.
  See JEP 446: Scoped Values (Preview) and the ScopedValue class in the Java API Specification.
- The Vector API is introduced to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations. See JEP 448: Vector API (Sixth Incubator).

See JEP 12: Preview Features and JEP 11: Incubator Modules for more information about preview features and incubating APIs.

Performance Improvements

Application performance has been improved by extending the Z Garbage Collector (ZGC) to maintain separate generations for young and old objects. This will allow ZGC to collect young objects, which tend to die young, more frequently.

See JEP 439: Generational ZGC and The Z Garbage Collector in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.

Stewardship

Warnings are issued when agents are loaded dynamically into a running JVM. These warnings aim to prepare users for a future release which disallows the dynamic loading of agents by default in order to improve integrity by default.

See JEP 451: Prepare to Disallow the Dynamic Loading of Agents.

Removed APIs, Tools, and Components

For more details on removals and deprecations, see Features and Options Removed and Deprecated in JDK 21.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 21.

Significant Changes in JDK 20 Release

See JDK 20 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 20.

The following are some of the updates in Java SE 20 and JDK 20:

Concurrency Model Previews and Incubators

- Virtual threads are lightweight threads that reduce the effort of writing, maintaining, and debugging high-throughput concurrent applications. This is a preview API. Minor changes have been made to this API since the last release. See JEP 436: Virtual Threads (Second Preview) and Virtual Threads in *Java Platform, Standard Edition Core Libraries*.
- Structured Concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability. See JEP 437: Structured Concurrency (Second Incubator). See JEP 11: Incubator Modules for more information about incubating APIs.
- Scoped values enables sharing of immutable data within and across threads. They are preferred to thread-local variables, especially when using large numbers of virtual threads. See JEP 429: Scoped Values (Incubator) and JEP 11: Incubator Modules.
- An API is introduced to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations. See JEP 438: Vector API (Fifth Incubator).

Language Changes

- The preview feature Pattern Matching for `switch` Expressions and Statements has been further refined as follows:
  - An exhaustive switch (that is, a `switch` expression or a pattern `switch` statement) over an `enum` class throws a `MatchException` instead of an `IncompatibleClassChangeError` if no `switch` label applies at run time.
  - The grammar for `switch` labels is simpler.
  - The compiler can infer the type of the type arguments for generic record patterns in all constructs that accept patterns: `switch` statements and expressions, `instanceof` expressions, and enhanced `for` statements.

See JEP 433: Pattern Matching for switch (Fourth Preview) and Pattern Matching for switch Expressions and Statements in *Java Platform, Standard Edition Java Language Updates*.

- The preview feature Record Patterns has been further refined as follows:
  - The compiler can infer the type of the type arguments for generic record patterns.
  - Record patterns can appear in an enhanced `for` statement.
  - Named record patterns are no longer supported.

  See JEP 432: Record Patterns (Second Preview) and Record Patterns in *Java Platform, Standard Edition Java Language Updates*.

See Preview Language and VM Features for more information about preview features.

Library Changes

- The preview API Foreign Function and Memory API has been further refined as follows:
  - The `MemorySegment` and `MemoryAddress` abstractions are unified (memory addresses are now modeled by zero-length memory segments).
  - The sealed `MemoryLayout` hierarchy is enhanced to facilitate usage with pattern matching in switch expressions and statements. See Pattern Matching for switch Expressions and Statements.
  - `MemorySession` has been split into Arena and `SegmentScope` to facilitate sharing segments across maintenance boundaries

  See JEP 434: Foreign Function & Memory API (Second Preview) and Foreign Function and Memory API in *Java Platform, Standard Edition Core Libraries*.

Removed APIs, Tools, and Components

For more details on removals and deprecations, see Features and Options Deprecated in JDK 20.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 20.

Significant Changes in JDK 19 Release

See JDK 19 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 19.

The following are some of the updates in Java SE 19 and JDK 19:

Concurrency Model Update Previews

- Virtual threads are lightweight threads that reduce the effort of writing, maintaining, and debugging high-throughput concurrent applications. This is a preview API. See JEP 425: Virtual Threads (Preview) and Virtual Threads in *Java Platform, Standard Edition Core Libraries*.
- An API is introduced to simplify multithreaded programming for structured concurrency. Structured concurrency treats multiple tasks running in different threads as a single unit of work, thereby streamlining error handling and cancellation, improving reliability, and enhancing observability. See JEP 428: Structured Concurrency (Incubator).

Language Changes

- Record Patterns is introduced as a preview feature for this release. A record pattern consists of a type, a record component pattern list used to match against the corresponding record components, and an optional identifier. You can nest record patterns and type patterns to enable a powerful, declarative, and composable form of data navigation and processing. See JEP 405: Record Patterns (Preview) and Record Patterns in *Java Platform, Standard Edition Java Language Updates*.
- The preview feature Pattern Matching for `switch` Expressions and Statements has been re-previewed in this release. This feature enables an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely. See JEP 427: Pattern Matching for switch (Third Preview) and Pattern Matching for switch Expressions and Statements in *Java Platform, Standard Edition Java Language Updates*.

Library Changes

- The Foreign Function and Memory API enables Java programs to interoperate with code and data outside the Java runtime. This API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI. The API invokes foreign functions, code outside the JVM, and safely accesses foreign memory that is not managed by the JVM. This is a preview API. See JEP 424: Foreign Function & Memory API (Preview) and Foreign Function and Memory API in *Java Platform, Standard Edition Core Libraries*.
- The Vector API is introduced to express vector computations that reliably compile at runtime to optimal vector instructions on supported CPU architectures, thus achieving performance superior to equivalent scalar computations. See JEP 426: Vector API (Fourth Incubator).

Removed APIs, Tools, and Components

For more details on removals and deprecations, see Features and Options Deprecated in JDK 19.

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 19.

Significant Changes in JDK 18 Release

See JDK 18 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 18.

The following are some of the updates in Java SE 18 and JDK 18:

Tools

- The new command-line tool `jwebserver` enables you to start a minimal web server that serves static files. This tool is useful for prototyping and testing. See JEP 408: Simple Web Server.
- A `@snippet` tag is added to JavaDoc's Standard Doclet, which simplifies the inclusion of example source code in API documentation. See JEP 413: Code Snippets in Java API Documentation and Programmer's Guide to Snippets.

Library Changes

- UTF-8 is now the default charset for the Java SE APIs. With this change, APIs that depend on the default charset will behave consistently across all implementations, operating systems, locales, and configurations. See JEP 400: UTF-8 by Default and Be Aware of the Default Charset.
- A service-provider interface (SPI) is defined for the host name and address resolution, so that java.net.InetAddress can make use of resolvers other than the platform's built-in resolver. See JEP 418: Internet-Address Resolution SPI. See also the section Specify Mappings from Host Names to IP Addresses in *Java Platform, Standard Edition Core Libraries* for information about the `jdk.net.hosts.file` system property, which enables you to configure `InetAddress` to use a specific `hosts` file, rather than the system-wide resolver, to map host names to IP addresses.
- Core reflection with method handles has been reimplemented. This will reduce the maintenance and development cost of both the `java.lang.reflect` and `java.lang.invoke` APIs. See JEP 416: Reimplement Core Reflection with Method Handles.

Preview Features and Incubator Modules

See JEP 12: Preview Features for more information about preview features and JEP 11: Incubator Modules for more information about incubator modules.

- The preview feature Pattern Matching for `switch` Expressions and Statements has been re-previewed in this release. This feature allows an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely. See JEP 420: Pattern Matching for switch (Second Preview) and Pattern Matching for switch Expressions and Statements in *Java Platform, Standard Edition Java Language Updates*.
- The Foreign Function and Memory API, which was introduced in Java SE 17, allows Java programs to interoperate with code and data outside of the Java runtime. The API is re-incubated in this release along with enhancements. See JEP 419: Foreign Function & Memory API (Second Incubator).
- The Vector API was introduced in Java SE 16 as an incubating API. In this release the API is re-incubated with enhancements and performance improvements. See JEP 417: Vector API (Third Incubator).

Removed APIs, Tools, and Components

For more details on removals and deprecations, see:

- Finalization Deprecated for Removal
- APIs Removed in Java SE 18

In addition, there are security related updates that you need to be aware of. See Security Updates in JDK 18.

Significant Changes in JDK 17 Release

See JDK 17 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 17.

The following are some of the updates in Java SE 17 and JDK 17:

New Language Feature

- Sealed Classes, first previewed in Java SE 15, is a permanent feature in this release. Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them. See JEP 409: Sealed Classes and Sealed Classes in *Java Platform, Standard Edition Java Language Updates* guide.

Library Changes

- Applications can now configure context-specific and dynamically selected deserialization filters using a JVM-wide filter factory, which is invoked to select a filter for each individual deserialization operation. See JEP 415: Context-Specific Deserialization Filters and Serialization Filtering in *Java Platform, Standard Edition Core Libraries* guide.
- New interface types and implementations for pseudo-random number generators (PRNGs) are now available, including jumpable PRNGs and an additional class of splittable PRNG algorithms (LXM). See JEP 356: Enhanced Pseudo-Random Number Generators and Pseudorandom Number Generators in *Java Platform, Standard Edition Core Libraries* guide.
- New Java 2D internal rendering pipeline for macOS is implemented using the Apple Metal API. This is an alternative to the existing pipeline, which uses the deprecated Apple OpenGL API. See JEP 382: New macOS Rendering Pipeline.

Other Changes

- By default, all internal elements of the JDK are strongly encapsulated, except for critical internal APIs such as `sun.misc.Unsafe`. However, it will no longer be possible to relax the strong encapsulation of internal elements using a single command-line option, as it was possible in JDK 9 through JDK 16. See JEP 403: Strongly Encapsulate JDK Internals by Default.
- The floating-point operations are now consistently strict, rather than having both strict floating-point semantics (`strictfp`) and subtly different default floating-point semantics. See JEP 306: Restore Always-Strict Floating-Point Semantics.

Deprecations

- The Security Manager and APIs related to it have been deprecated for removal in a future release. See JEP 411: Deprecate the Security Manager for Removal.

Preview Features and Incubator Modules

See JEP 12: Preview Features for more information about preview features and JEP 11: Incubator Modules for more information about incubator modules.

- Pattern matching for `switch` expressions and statements is introduced in this release. This feature allows an expression to be tested against a number of patterns, each with a specific action, so that complex data-oriented queries can be expressed concisely and safely. See JEP 406: Pattern Matching for switch (Preview) and Pattern Matching for Switch Expressions and Statements in *Java Platform, Standard Edition Java Language Updates* guide.
- Foreign Function & Memory API allows Java programs to interoperate with code and data outside of the Java runtime. See JEP 412: Foreign Function & Memory API (Incubator).
- The Vector API was introduced in Java SE 16 as an incubating API. In this release, enhancements have been incorporated along with performance improvements. See JEP 414: Vector API (Second Incubator).

Removed APIs, Tools, and Components

See:

- APIs Removed in Java SE 17
- Tools and Components Removed and Deprecated in JDK 17

In addition, there are security related updates that you need to be aware of. See: Security Updates in JDK 17.

Significant Changes in JDK 16 Release

See JDK 16 Release Notes for additional descriptions of the new features and enhancements, and API specification in JDK 16.

The following are some of the updates in Java SE 16 and JDK 16:

- The Java programming language is enhanced with pattern matching for the `instanceof` operator. This feature allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely. See JEP 394: Pattern Matching for instanceof and Pattern Matching for instanceof in *Java Platform, Standard Edition Java Language Updates* guide.
- Records, first previewed in Java SE 14, is a permanent feature in this release. The earlier restrictions have been relaxed whereby the inner classes can declare members that are either explicitly or implicitly static. This includes record class members, which are implicitly static. See JEP 395: Records and Record Classes in *Java Platform, Standard Edition Java Language Updates* guide.
- By default, all internal elements of the JDK are strongly encapsulated, except for critical internal APIs such as `sun.misc.Unsafe`. You can choose the relaxed strong encapsulation that has been the default since JDK 9. See JEP 396: Strongly Encapsulate JDK Internals by Default and Strong Encapsulation in the JDK.
- UNIX domain socket channels have been integrated into JDK 16. See JEP 380: Unix-Domain Socket Channels and Internet Protocol and UNIX Domain Sockets NIO Example in *Java Platform, Standard Edition Core Libraries*.
- The Z Garbage Collector processes the thread stacks concurrently. This allows all roots in the JVM to be processed by ZGC in a concurrent phase. See JEP 376: ZGC: Concurrent Thread-Stack Processing and The Z Garbage Collector in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.
- The `jpackage` tool, which was incubated in JDK 14, is now a permanent feature. The tool packages a Java application into a platform-specific package that includes the necessary dependencies. See JEP 392: Packaging Tool and *Java Platform, Standard Edition Packaging Tool User's Guide*.
- Elastic metaspace overhauls the VM-internal metaspace- and class-space-implementation. The unused HotSpot class-metadata (that is *metaspace*) memory is returned to the operating system. It reduces the metaspace footprint and simplify the metaspace code in order to reduce maintenance costs. See JEP 387: Elastic Metaspace.

Preview Features and Incubator Modules

See Java Language Preview Feature for more information about preview features.

- Sealed classes, a preview feature from JDK 15, is re-previewed in this release. Sealed classes and interfaces restrict which other classes or interfaces may extend or implement

them. There has been several refinements in this release, including the introduction of character sequences `sealed`, `non-sealed`, and `permits` as contextual keywords. See JEP 397: Sealed Classes (Second Preview) and Sealed Classes in *Java Platform, Standard Edition Java Language Updates* guide.

- Initial iteration of an incubator module, `jdk.incubator.vector`, is provided to express vector computations. It reliably compiles at runtime to optimal vector hardware instructions on supported CPU architectures and thus achieve superior performance to equivalent scalar computations. See JEP 338: Vector API (Incubator).
- Foreign Linker API is introduced that offers statically-typed, pure-Java access to native code. This API, along with the Foreign-Memory Access API (JEP 393), will simplify the otherwise error-prone process of binding to a native library. See JEP 389: Foreign Linker API (Incubator).
- Foreign-Memory Access API allows Java programs to safely and efficiently access foreign memory outside of the Java heap. See JEP 393: Foreign-Memory Access API (Third Incubator).

Removed APIs, Tools, and Components

See:

- API Removed in Java SE 16
- Tools and Components Removed and Deprecated in JDK 16

In addition, there are security related updates that you need to be aware of. See: Security Updates in JDK 16.

Significant Changes in JDK 15 Release

See JDK 15 Release Notes for the complete list of new features and enhancements in JDK 15.

The following are some of the updates in Java SE 15 and JDK 15:

- Text Blocks, first previewed in Java SE 13, is a permanent feature in this release and can be used without enabling preview features.
  Text blocks are multiline string literals that avoid the need for most escape sequences, automatically format the string in a predictable way, and give the developer control over the format when desired. See JEP 378: Text Blocks and Programmer's Guide to Text Blocks.
- The Z Garbage Collector (ZGC) is ready to use in production and no longer an experimental feature. Enable ZGC by using the command-line option `-XX:+UseZGC`. See JEP 377: ZGC: A Scalable Low-Latency Garbage Collector (Production).
- Hidden classes are classes that cannot be used directly by the bytecode of other classes. Hidden classes are intended for use by frameworks that generate classes at run time and use them indirectly through reflection. See JEP 371: Hidden Classes.

Preview and Incubator Features

See Java Language Preview Feature for more information about preview features.

- Sealed Classes is a Java language preview feature. Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them. See JEP 360: Sealed Classes (Preview) and Sealed Classes in *Java Platform, Standard Edition Java Language Updates* guide.
- Pattern Matching for `instanceof`, a preview feature from Java SE 14, is re-previewed for this release. This feature allows common logic in a program, namely the conditional

extraction of components from objects, to be expressed more concisely and safely. See JEP 375: Pattern Matching for instanceof (Second Preview) and Pattern Matching for the instanceof in *Java Platform, Standard Edition Java Language Updates* guide.

- Records, a preview feature from Java SE 14, is re-previewed for this release. Records are classes that act as transparent carriers for immutable data. See JEP 384: Records (Second Preview) and Record Classes in *Java Platform, Standard Edition Java Language Updates* guide.
- The Foreign Memory Access API allows Java programs to efficiently and safely access foreign memory outside of the Java heap. See JEP 383: Foreign-Memory Access API (Second Incubator).

Removed APIs, Tools, and Components

See:

- APIs Removed in Java SE 15
- Tools and Components Removed and Deprecated in JDK 15

In addition, there are security related updates that you need to be aware of. See: Security Updates in JDK 15.

Significant Changes in JDK 14 Release

The following are some of the changes in Java SE 14 and JDK 14:

- Switch is extended so it can be used as either a statement or an expression, so that both forms can use either traditional `case ... :` labels (with fall through) or new `case ... ->` labels (with no fall through), with a further new statement for yielding a value from a switch expression. See JEP 361: Switch Expressions (Standard) and Java Language Changes.
- G1 is enhanced to improve allocation performance on non-uniform memory access (NUMA) memory systems. See JEP 345: NUMA-Aware Memory Allocation for G1.
- JDK Flight Recorder data is now available as a data stream allowing for continuous monitoring. See JEP 349: JFR Event Streaming.
- New JDK-specific file mapping modes have been added so that the `FileChannel` API can be used to create `MappedByteBuffer` instances that refer to non-volatile (NVM) memory. See JEP 352: Non-Volatile Mapped Byte Buffers.
- Allows currencies to be formatted with locale-specific accounting formats, for example, ($3.27) instead of -$3.27. See Accounting Currency Format Support.
- Enhanced `com.sun.management.OperatingSystemMXBean` to ensure that it reports values based on the current operating environment, such as a container environment. The MXBean for tools to get information on the operating system has been improved for container environments. See OperatingSystemMXBean made container aware.

Experimental, Preview, and Incubator Features

- `Records` is a Java language preview feature, which provides a compact syntax for declaring classes that are transparent holders for shallowly immutable data. See JEP 359: Records (Preview).
- Pattern Matching for `instanceof` is a Java language preview feature that simplifies the `instanceof-and-cast` idiom. See JEP 305: Pattern Matching for instanceof (Preview).
- Text blocks are multi-line string literals that avoids the need for most escape sequences, automatically formats the string in a predictable way, and gives the developer control over

the format when desired. Text Blocks were introduced in JDK 13 as a Preview Feature. Text Blocks is being previewed again in JDK 14 with the addition of two new escape sequences. See JEP 368: Text Blocks (Second Preview).

- `jpackage`, a simple tool for packaging self-contained Java applications. See JEP 343: Packaging Tool (Incubator).
- An API that allows Java programs to efficiently access foreign memory outside of the Java heap is introduced. See JEP 370: Foreign-Memory Access API (Incubator).
- The Z Garbage Collector (ZGC), previously available only for Linux, is introduced as an experimental feature on Windows and macOS. See JEP 364: ZGC on macOS and JEP 365: ZGC on Windows.

Removed APIs, Tools, and Components

See:

- APIs Removed in Java SE 14
- Features and Components Removed in JDK 14

In addition, there are security related updates that you need to be aware of. See: Security Updates in JDK 14.

Significant Changes in JDK 13 Release

The following were some of the important enhancements in Java SE 13 and JDK 13:

- Dynamic CDS Archiving extends application class-data sharing (ApsCDS), which allows dynamic archiving of classes when the Java application exits. See JEP 350: Dynamic CDS Archives.
- Text blocks were added to Java language, which provide developers with control over the format when desired. This is a preview language feature. See JEP 355 Text Blocks (Preview) and JEP 12: Preview Language and VM Features.
- The `switch` expression, a preview language feature, was extended to be used as either a statement or an expression, so that both forms can use either traditional labels (with fall through) or new labels (with no fall through). It is used with a further new statement for yielding a value from a `switch` expression. See JEP 354: Switch Expressions (Preview) and JEP 12: Preview Language and VM Features.
- The implementation used by `java.net.Socket` and `java.net.ServerSocket` APIs was replaced with a simpler and more modern implementation that is easy to maintain and debug. See JEP 353: Reimplement the Legacy Socket API.
- Support for Unicode 12.1. See Unicode 12.1.
- ZGC was enhanced to return unused heap memory to the operating system, which enhances the memory footprint of the applications. See JEP 351 ZGC Uncommit Unused Memory.

Significant Changes in JDK 12 Release

The following were some of the important additions and updates in Java SE 12 and JDK 12:

- JVM Constants API was introduced to model nominal descriptions of key class-file and run-time artifacts, in particular constants that were loadable from the constant pool. See JVM Constant API.

- The `switch` statement was extended so that it can be used either as a statement or an expression. This is a preview language feature. See JEP 325: Switch Expressions (Preview) and JEP 12: Preview Language and VM Features.
- Support for Unicode 11.0. See Unicode 11.0.
- Square character support was provided for the Japanese Reiwa Era, which began on May, 2019. See Square character support.
- The `NumberFormat` added support for formatting a number in its compact form. See Compact Number Formatting Support .

Significant Changes in JDK 11 Release

JDK 11 had some significant changes too. As JDK 11 is a long term support (LTS) release, you should be familiar with the following important changes in JDK 11 release:

- Oracle no longer offers JRE and Server JRE downloads; consequently, Auto-Update is not available anymore.
- Java Web Start, Java Plugin, and Java Control Panel are not available in JDK. See Removal of the Deployment Stack.
- JavaFX is no longer included in the JDK. It is now available as a separate download from https://openjfx.io/.
- JAXB and JAX-WS are no longer bundled with JDK. See Removal of Java EE and CORBA Modules.

3
Security Updates

This section provides details on the security updates in JDK releases.

Security Updates in JDK 25

The following are the notable security updates in JDK 25:

- SHAKE128-256 and SHAKE256-512 as MessageDigest Algorithms
  See The SUN Provider.
- Support for HKDF in SunPKCS11
  See SinPKCS11 Provider Supported Algorithms.
- Mechanism to Disable Signature Schemes Based on Their TLS Scope
- Add Support for TLS Keying Material Exporters to JSSE and SunJSSE Provider
  See TLS Keying Material Exporters.
- Update XML Security for Java to 3.0.5
- Enhanced jar File Validation

See Release Notes for additional information on security-related changes.

Security Updates in JDK 24

The following are the notable security updates in JDK 24:

- Support for Including Security Properties Files
  See Including a Security Properties File.
- Document Standard Hash and MGF Algorithms for RSASSA-PSS Signature
  See Java Security Standard Algorithm Names.
- SunPKCS11 Provider Is Enhanced to Use CKM_AES_CTS Mechanism If Supported by Native PKCS11 Library
  See SunPKCS11 Configuration and SunPKCS11 Provider Supported Algorithms.
- Configurable New Session Tickets Count for TLSv1.3
  See the system property `jdk.tls.server.newSessionTicket` in Customizing JSSE.
- Mechanism to Disable TLS Cipher Suites by Pattern Matching

See Release Notes for additional information on security-related changes.

Security Updates in JDK 23

The following are the notable security updates in JDK 23:

- Thread and Timestamp Options for java.security.debug System Property
- Enable Case-Sensitive Check in ccache and keytab Kerberos Entry Lookup
- Support for KeychainStore-ROOT Keystore

See Release Notes for additional information on security-related changes.

Security Updates in JDK 22

The following are the notable security updates in JDK 22:

- New Security Category for -XshowSettings Launcher Option
- HSS/LMS: keytool and jarsigner Changes
- Update XML Security for Java to 3.0.3

See Release Notes for additional information on security-related changes.

Security Updates in JDK 21

The following are the notable security updates in JDK 21:

- New System Property to Toggle XML Signature Secure Validation Mode
- Enhanced OCSP, Certificate, and CRL Fetch Timeouts
- Support for HSS/LMS Signature Verification
- SunJCE Provider Now Supports SHA-512/224 and SHA-512/256 As Digests for the PBES2 Algorithms
- Support for Password-Based Cryptography in SunPKCS11
- Update XML Security for Java to 3.0.2

See Release Notes for additional information on security-related changes.

Security Updates in JDK 20

The following are the notable security updates in JDK 20:

- New JFR Event: jdk.InitialSecurityProperty
- New JFR Event: jdk.SecurityProviderService
- Provide Poly1305 Intrinsic on x86_64 platforms with AVX512 instructions
- Provide ChaCha20 Intrinsics on x86_64 and aarch64 Platforms

See Release Notes for additional information on security-related changes.

Security Updates in JDK 19

The following are the notable security updates in JDK 19:

- Windows KeyStore Updated to Include Access to the Local Machine Location
- Break Up SEQUENCE in X509Certificate::getSubjectAlternativeNames and X509Certificate::getIssuerAlternativeNames in otherName
- (D)TLS Signature Schemes
- New Options for ktab to Provide Non-default Salt

Removed Certificates

The following certificates or options have been removed from Java SE 19:

- Finalizer Implementation in SSLSocketImpl
- Alternate ThreadLocal Implementation of the Subject::current and Subject::callAs APIs

See Release Notes for additional information on security-related changes.

Security Updates in JDK 18

The following are the notable security updates in JDK 18:

- Disabled SHA-1 Signed JARs
- Change the java.security.manager System Property Default Value to disallow
- X509Certificate.get{Subject,Issuer}AlternativeNames and getExtendedKeyUsage Do Not Throw CertificateParsingException if Extension Is Unparseable
- Fix Issues With the KW and KWP Modes of SunJCE Provider
- Removed Weak etypes From Default krb5 etype List

Removed Certificates

The following certificates or options have been removed from Java SE 18:

- IdenTrust Root Certificate
- Google's GlobalSign Root Certificate
- default_checksum and safe_checksum_type From krb5.conf

See Release Notes for additional information on security-related changes.

Security Updates in JDK 17

The following are the notable security updates in JDK 17:

- Support for specifying a signer in keytool -genkeypair command
- SunJCE provider now supports KW and KWP modes with AES cipher
- Configurable extensions with system properties
- Removal of Telia Company's Sonera Class2 CA Certificate

See Release Notes for additional information on security-related changes.

Security Updates in JDK 16

The following are the notable security updates in JDK 16:

- Signed JAR support for RSASSA-PSS and EdDSA
- SUN, SunRsaSign, and SunEC providers support SHA-3-based signature algorithms
- The SunPKCS11 provider now supports SHA-3-related algorithms
- TLS support for the EdDSA signature algorithm

See Release Notes for additional information on security-related changes.

Security Updates in JDK 15

The following are the noteable security updates in JDK 15:

- A new signature scheme Edwards-Curve Digital Signature Algorithm (EdDSA) is implemented, which is a modern elliptic curve signature scheme that has several advantages over the existing signature schemes in the JDK. This new signature scheme does not replace ECDSA. See JEP 339: Edwards-Curve Digital Signature Algorithm (EdDSA).
- SunJCE provider now supports SHA-3 based Hmac algorithms
- New System Properties to Configure the TLS Signature Schemes
- Support the certificate_authorities extension

See Release Notes for additional information on security related changes.

Security Updates in JDK 14

The following are the noteable security updates in JDK 14:

- Exact Match Required for Trusted TLS Server Certificate
- New Checks on Trust Anchor Certificates

See Release Notes for additional information on security related changes.

Security Updates in JDK 13

The following were removed from JDK 13:

- Experimental FIPS 140 compliant mode from SunJSSE provider
- Duplicated RSA services no longer supported by SunJSSE provider

Removal of Security Certificates

The following root certificates were removed from the keystore in JDK 13:

- T-Systems Deutsche Telekom Root CA 2 certificate
- Two DocuSign Root CA certificates
- Two Comodo Root CA certificates

Security Updates in JDK 11 and JDK 12

The following security updates were made in JDK 11 and JDK 12:

The JDK 11 release included an implementation of the Transport Layer Security (TLS) 1.3 specification (RFC 8446).

TLS 1.3 is the latest iteration (August 2018) of the Transport Layer Security (TLS) protocol and is enabled by default in JDK 11. This version focuses not only on speed improvements, but also updates the overall security of the protocol by emphasizing modern cryptography practices, and disallows outdated or weak crypto algorithms. (For example, RSA key exchange and plain DSA signatures are no longer allowed.)

Several features were added to the TLS 1.3 protocol to improve backwards compatibility, but there are several issues of which you need to be aware of. For details, see JEP 332.

Removal of Security Certificates

The following root certificate was removed from the keystore in JDK 12:

- Removal of GTE CyberTrust Global Root

The following root certificates were removed from the truststore in JDK 11:

- Several Symantec Root CAs
- Baltimore Cybertrust Code Signing CA
- SECOM Root Certificate
- AOL and Swisscom root certificates

Products that use certificates that have been removed may no longer work. If these certificates are required, then you must configure and populate the cacerts with the missing certs. To add certs to the truststore, see keytool in *Java Development Kit Tool Specifications* guide.

Security Updates in JDK 9 and JDK 10

Some security-related defaults have changed, starting from JDK 9.

JCE Jurisdiction Policy File Default is Unlimited

If your application previously required the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files, then you no longer need to download or install them. They are included in the JDK and are activated by default.

If your country or usage requires a more restrictive policy, the limited Java cryptographic policy files are still available.

If you have requirements that are not met by either of the policy files provided by default, then you can customize these policy files to meet your needs.

See the `crypto.policy` Security property in the `<java-home>/conf/security/java.security` file, or Cryptographic Strength Configuration in the *Java Platform, Standard Edition Security Developer's Guide*.

You are advised to consult your export/import control counsel or attorney to determine the exact requirements.

Create PKCS12 Keystores

We recommend that you use the PKCS12 format for your keystores. This format, which is the default keystore type, is based on the RSA PKCS12 Personal Information Exchange Syntax Standard.

See Creating a Keystore to Use with JSSE in *Java Platform, Standard Edition Security Developer's Guide* and keytool in *Java Development Kit Tool Specifications*.

4

Removed APIs

This section provides details about Java SE APIs that were removed in releases JDK 11 though JDK 25.

Run `jdeprscan --release 25 -l --for-removal` to get the list of APIs that are marked for removal in JDK 25.

> **Note**
>
> The jdeprscan tool is available since JDK 9. If you want to print the list of deprecated APIs from earlier JDK versions, then replace the release number with the corresponding release version.

APIs Removed in Java SE 25

Methods

```
javax.swing.plaf.synth.SynthLookAndFeel.load
```

Constructors

```
javax.swing.plaf.basic.BasicSliderUI.BasicSliderUI.<init>()
```

APIs Removed in Java SE 24

Fields

```
java.awt.Window.warningString
javax.naming.Context.APPLET
```

APIs Removed in Java SE 23

Classes

```
javax.management.loading.MLet
javax.management.loading.MLetContent
javax.management.loading.MLetMBean
javax.management.loading.PrivateMLet
```

Methods

```
java.lang.Thread.resume
java.lang.Thread.suspend
```

```
java.lang.ThreadGroup.resume
java.lang.ThreadGroup.stop
java.lang.ThreadGroup.suspend
```

APIs Removed in Java SE 22

Methods

```
java.lang.Thread.countStackFrames
```

APIs Removed in Java SE 21

Classes

```
java.lang.Compiler
javax.management.remote.rmi.RMIIIOPServerImpl
```

Methods

```
java.lang.ThreadGroup.allowThreadSuspension(boolean)
```

APIs Removed in Java SE 20

No APIs were removed in this release.

APIs Removed in Java SE 19

No APIs were removed in this release.

APIs Removed in Java SE 18

Methods

```
java.awt.color.ICC_Profile.finalize()
java.awt.image.ColorModel.finalize()
java.awt.image.IndexColorModel.finalize()
```

APIs Removed in Java SE 17

Packages

```
java.rmi.activation ()
```

Classes

```
java.rmi.activation.Activatable ()
java.rmi.activation.ActivationDesc ()
java.rmi.activation.ActivationGroup ()
java.rmi.activation.ActivationGroup_Stub ()
java.rmi.activation.ActivationGroupDesc ()
java.rmi.activation.ActivationGroupID ()
java.rmi.activation.ActivationID ()
java.rmi.activation.ActivationInstantiator ()
java.rmi.activation.ActivationMonitor ()
java.rmi.activation.ActivationSystem ()
java.rmi.activation.Activator ()
```

API Removed in Java SE 16

Constructor

```
javax.tools.ToolProvider.<init>()
```

APIs Removed in Java SE 15

The following APIs have been removed in Java SE 15.

Fields

```
java.management.rmi.RMIConnectorServer.CREDENTIAL_TYPES
```

Constructors

```
java.lang.invoke.ConstantBootstraps.<init>
    java.lang.reflect.Modifier.<init>
```

APIs Removed in Java SE 14

The following APIs have been removed in Java SE 14.

Packages

```
java.security.acl
```

Interfaces

```
java.security.acl.Acl
java.security.acl.AclEntry
java.security.acl.Group
java.security.acl.Owner
java.security.acl.Permission
```

```
java.util.jar.Pack200.Packer
java.util.jar.Pack200.Unpacker
```

Classes

```
java.util.jar.Pack200
```

APIs Removed in Java SE 13

The following APIs were removed in Java SE 13. Both of these APIs were deprecated and marked for removal with JDK 9. Both have been superseded by JVM-specific tracing mechanisms. See JVMTM Tool Interface specification.

```
java.lang.Runtime.traceInstructions(boolean)
java.lang.Runtime.traceMethodCalls(boolean)
```

APIs Removed in Java SE 12

The following APIs were removed in Java SE 12.

```
java.io.FileInputStream.finalize()
java.io.FileOutputStream.finalize()
java.util.zip.Deflater.finalize()
java.util.zip.Inflater.finalize()
java.util.zip.ZipFile.finalize()
```

APIs Removed in JDK 11

The following APIs were removed in JDK 11. Many of these APIs were deprecated in previous releases and have been replaced by newer APIs.

```
javax.security.auth.Policy
java.lang.Runtime.runFinalizersOnExit(boolean)
java.lang.SecurityManager.checkAwtEventQueueAccess()
java.lang.SecurityManager.checkMemberAccess(java.lang.Class,int)
java.lang.SecurityManager.checkSystemClipboardAccess()
java.lang.SecurityManager.checkTopLevelWindow(java.lang.Object)
java.lang.System.runFinalizersOnExit(boolean)
java.lang.Thread.destroy()
java.lang.Thread.stop(java.lang.Throwable)
```

APIs Removed in JDK 10

The following common DOM APIs were removed in JDK 10.

```
com.sun.java.browser.plugin2.DOM
```

```
sun.plugin.dom.DOMObject
```

APIs Removed JDK 9

The following are some important APIs that have been removed from JDK 10 and JDK 9 releases.

Removed java.* APIs

The Java team is committed to backward compatibility. If an application runs in JDK 8, then it will run on JDK 9 and later releases as long as it uses APIs that are supported and intended for external use.

These include:

- JCP standard, `java.*`, `javax.*`
- JDK-specific APIs, some `com.sun.*`, some `jdk.*`

Supported APIs can be removed from the JDK, but only with notice. Find out if your code is using deprecated APIs by running the static analysis tool jdeprscan.

`java.*` APIs that were removed in JDK 9 include the previously deprecated methods from the `java.util.logging.LogManager` and `java.util.jar.Pack200` packages:

```
java.util.logging.LogManager.addPropertyChangeListener
java.util.logging.LogManager.removePropertyChangeListener
java.util.jar.Pack200.Packer.addPropertyChangeListener
java.util.jar.Pack200.Packer.removePropertyChangeListener
java.util.jar.Pack200.Unpacker.addPropertyChangeListener
java.util.jar.Pack200.Unpacker.removePropertyChangeListener
```

Removal and Future Removal of sun.misc and sun.reflect APIs

Unlike the `java.*` APIs, almost all of the `sun.*` APIs are unsupported, JDK-internal APIs, and may go away at any time.

A few `sun.*` APIs were removed in JDK 9. Notably, `sun.misc.BASE64Encoder` and `sun.misc.BASE64Decoder` were removed. Instead, use the supported `java.util.Base64` class, which was added in JDK 8.

If you use these APIs, you may wish to migrate to their supported replacements:

- `sun.misc.Unsafe`
  The functionality of many of the methods in this class is available by using variable handles, see JEP 193: Variable Handles.
- `sun.reflect.Reflection::getCallerClass(int)`

Instead, use the stack-walking API, see JEP 259: Stack-Walking API.

See JEP 260: Encapsulate Most Internal APIs.

java.awt.peer Not Accessible

The `java.awt.peer` and `java.awt.dnd.peer` packages aren't accessible, starting in JDK 9. The packages were never part of the Java SE API, despite being in the `java.*` namespace.

All methods in the Java SE API that refer to types defined in these packages were removed from JDK 9. Code that calls a method that previously accepted or returned a type defined in these packages no longer compiles or runs.

There are two common uses of the `java.awt.peer` classes. You should replace them as follows:

- To see if a peer has been set yet:

  ```
  if (component.getPeer() != null) { .. }
  ```

  Replace this with `Component.isDisplayable()` from the JDK 1.1 API:

  ```
  public boolean isDisplayable() {
      return getPeer() != null;
  ```

- To test if a component is lightweight:

  ```
  if (component.getPeer() instanceof LightweightPeer) ..
  ```

  Replace this with `Component.isLightweight()` from the JDK 1.2 API:

  ```
  public boolean isLightweight() {
      return getPeer() instanceof LightweightPeer;
  ```

Removed com.sun.image.codec.jpeg Package

The nonstandard package `com.sun.image.codec.jpeg` has been removed. Use the Java Image I/O API instead.

The `com.sun.image.codec.jpeg` package was added in JDK 1.2 as a nonstandard way of controlling the loading and saving of JPEG format image files. It has never been part of the platform specification.

In JDK 1.4, the Java Image I/O API was added as a standard API, residing in the `javax.imageio` package. It provides a standard mechanism for controlling the loading and saving of sampled image formats and requires all compliant Java SE implementations to support JPEG based on the Java Image I/O specification.

Removed Tools Support for Compact Profiles

Starting in JDK 9, you can choose to build and run your application against any subset of the modules in the Java runtime image, without needing to rely on predefined profiles.

Profiles, introduced in Java SE 8, define subsets of the Java SE Platform API that can reduce the static size of the Java runtime on devices that have limited storage capacity. The tools in JDK 8 support three profiles, `compact1`, `compact2`, and `compact3`. For the API composition of each profile, see Detailed Profile Composition and API Reference in the JDK 8 documentation.

In JDK 8, you use the `-profile` option to specify the profile when running the `javac` and `java` commands. Starting in JDK 9, the `-profile` option is supported by `javac` only in conjunction with the `--release 8` option, and isn't supported by `java`.

JDK 9 and later releases let you choose the modules that are used at compile and run time. By specifying modules with the new `--limit-modules` option, you can obtain the same APIs that are in the compact profiles. This option is supported by both the `javac` and `java` commands, as shown in the following examples:

```
javac --limit-modules java.base,java.logging MyApp.java
```

```
java --limit-modules java.base,java.logging MyApp
```

The packages specified for each profile in Java SE 8 are exported, collectively, by the following sets of modules:

- For the `compact1` profile: `java.base`, `java.logging`, `java.scripting`
- For the `compact2` profile: `java.base`, `java.logging`, `java.scripting`, `java.rmi`, `java.sql`, `java.xml`
- For the `compact3` profile: `java.base`, `java.logging`, `java.scripting`, `java.rmi`, `java.sql`, `java.xml`, `java.compiler`, `java.instrument`, `java.management`, `java.naming`, `java.prefs`, `java.security.jgss`, `java.security.sasl`, `java.sql.rowset`, `java.xml.crypto`

You can use the `jdeps` tool to do a static analysis of the Java packages that are being used in your source code. This gives you the set of modules that you need to execute your application. If you had been using the `compact3` profile, for example, then you may see that you don't need to include that entire set of modules when you build your application. See `jdeps` in *Java Development Kit Tool Specifications*.

See JEP 200: The Modular JDK.

5

Removed Tools and Components

The following section lists tools and components that have been removed or deprecated in JDK.

Features and Options Removed and Deprecated in JDK 25

The following features and options are removed in JDK 25:

- java.net.Socket Constructors Can No Longer Be Used to Create a Datagram Socket
- Removal of Old JMX System Properties
- Removal of PerfData Sampling
- Removal of sun.rt._sync* Performance Counters
- Removed Baltimore CyberTrust Root Certificate After Expiry Date
- Removed Two Camerfirma Root Certificates
- Removal of SunPKCS11 Provider's PBE-related SecretKeyFactory Implementations

The following features and options are deprecated for removal, which might cause compatibility issues while migrating:

- Deprecate VFORK Launch Mechanism from Process Implementation (linux)
- Deprecate the Use of java.locale.useOldISOCodes System Property
- Deprecate XML Interchange in JMX DescriptorSupport for Removal
- The UseCompressedClassPointers Option is Deprecated
- Various Permission Classes Deprecated for Removal

Features and Options Removed and Deprecated in JDK 24

The following features and options are removed in JDK 24:

- Remove the Non-Generational Mode
- Linux Desktop GTK2 Support Is Removed
- Remove JDK1.1 Compatible Behavior for "EST", "MST", and "HST" Time Zones
- The javax.naming.Context.APPLET Constant Is Removed
- The JNDI java.naming.rmi.security.manager Environment Property Is Removed
- JNDI Remote Code Downloading Is Permanently Disabled
- Removal of serialVersionUID Compatibility Logic from JMX
- The java Command Line Options -t, -tm, -Xfuture, -checksource, -cs, and -noasyncgc are Removed

The following features and options are deprecated for removal, which might cause compatibility issues while migrating:

- java.util.zip.ZipError Is Deprecated for Removal
- jstatd Is Deprecated for Removal
- jrunscript Tool Is Deprecated for Removal
- The jdk.jsobject Module Is Deprecated for Removal
- The LockingMode Flag and the LM_LEGACY And LM_MONITOR Modes Are Deprecated
- jhsdb debugd Is Deprecated for Removal
- The java Command Line Options -verbosegc, -noclassgc, -verify, -verifyremote, -ss, -ms and -mx Are Deprecated for Removal

Features and Options Removed and Deprecated in JDK 23

The following features and options are removed in JDK 23:

- Legacy Locale Data: The legacy `JRE` locale data is removed in JDK 23. If your applications are using `JRE` or `COMPAT` in the `java.locale.providers` system property, then migrate them to use CLDR locale data based on the Unicode Consortium's Common Locale Data Registry. See Be Aware of Changes to Locale Data and JEP 252: Use CLDR Locale Data by Default.
- Thread.suspend/resume and ThreadGroup.suspend/resume
- ThreadGroup.stop
- Aligned Access Modes for MethodHandles::byteArrayViewVarHandle, byteBufferViewVarHandle, and Related Methods
- Module jdk.random
- JMX Subject Delegation
- JMX Management Applet (m-let) Feature
- -Xnoagent Option of the java Launcher
- RegisterFinalizersAtInit Has Been Obsoleted
- Desktop Integration from Linux Installers

The following features and options are deprecated for removal, which might cause compatibility issues while migrating:

- Memory-Access Methods in `sun.misc.Unsafe`: the memory-access methods in `sun.misc.Unsafe` have been deprecated for removal in future releases. It is recommended to migrate from `sun.misc.Unsafe` to the standard APIs such as VarHandle API (JEP 193, JDK 9) and the Foreign Function & Memory API (JEP 454, JDK 22).
  See JEP 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal.
- `java.beans.beancontext` Package
- The JVM TI `GetObjectMonitorUsage` Function No Longer Supports Virtual Threads
- `PreserveAllAnnotations` VM Option
- `DontYieldALot` Flag
- `-XX:+UseEmptySlotsInSupers`
- `UseNotificationThread` VM Option

Features and Options Removed and Deprecated in JDK 22

The following features and options are removed in JDK 22:

- sun.misc.Unsafe.shouldBeInitialized and ensureClassInitialized
- Thread.countStackFrames
- The Old Core Reflection Implementation
- Jdeps -profile and -P Option

The following features and options are deprecated for removal, which might cause compatibility issues while migrating:

- sun.misc.Unsafe park/unpark, getLoadAverage, xxxFence methods
- -Xnoagent Option
- Deprecation of the jdk.crypto.ec module
- -Xdebug and -debug Options

Features and Options Removed and Deprecated in JDK 21

The following features and options are removed in JDK 21:

- SECOM Trust System's RootCA1 Root Certificate
- java.io.File's Canonical Path Cache
- ThreadGroup.allowThreadSuspension
- java.compiler System Property
- java.lang.Compiler Class
- JAR Index Feature
- javax.management.remote.rmi.RMIIIOPServerImpl
- G1 Hot Card Cache

The following features and options are deprecated for removal, which might cause compatibility issues while migrating:

- GTK2
- com.sun.nio.file.SensitivityWatchEventModifier
- Emit Warning for Removal of COMPAT Provider
- JMX Subject Delegation and the JMXConnector.getMBeanServerConnection(Subject) Method

Features and Options Deprecated in JDK 20

The following features and options are deprecated in JDK 20, which might cause compatibility issues while migrating:

- Thread.suspend/resume Changed to Throw UnsupportedOperationException
- Thread.Stop Changed to Throw UnsupportedOperationException
- java.net.URL Constructors Are Deprecated

- Deprecate JMX Management Applets for Removal

Features and Options Deprecated in JDK 19

The following features and options are deprecated in JDK 19, which might cause compatibility issues while migrating:

- java.lang.ThreadGroup Is Degraded
- Deprecation of Locale Class Constructors
- PSSParameterSpec(int) Constructor and DEFAULT Static Constant Are Deprecated
- OAEPParameterSpec.DEFAULT Static Constant Is Deprecated

Tools and Components Removed and Deprecated in JDK 18

Deprecate Finalization for Removal

The finalization mechanism has been deprecated for removal in a future release. The `finalize` methods in standard Java APIs, such as `Object.finalize()` and `Enum.finalize()`, have also been deprecated for removal in a future release, along with methods related to finalization, such as `Runtime.runFinalization()` and `System.runFinalization()`.

Finalization remains enabled by default for now, but can be disabled to facilitate early testing. In a future release it will be disabled by default, and in a later release it will be removed. See JEP 421: Deprecate Finalization for Removal and Finalization Deprecated for Removal.

Tools and Components Removed and Deprecated in JDK 17

Deprecate the Applet API for Removal

The Applet API has been deprecated for removal as all web-browser vendors have either removed support for Java browser plug-ins or announced plans to do so. See JEP 398: Deprecate the Applet API for Removal.

Removed RMI Activation

The Remote Method Invocation (RMI) Activation mechanism has been removed. However, the rest of RMI is preserved. See JEP 407: Remove RMI Activation.

Tools and Components Removed and Deprecated in JDK 16

Removal of Experimental Features AOT and Graal JIT

The Java Ahead-of-Time compilation experimental tool `jaotc` and Java-based Graal JIT compiler have been removed. See Removal of experimental features AOT and Graal JIT.

Removed Root Certificates with 1024-bit Keys

The root certificates with weak 1024-bit RSA public keys have been removed from the `cacerts` keystore. For details, see Remove root certificates with 1024-bit keys.

Removal of Legacy Elliptic Curves

The SunEC provider no longer supports the following elliptic curves that were deprecated in JDK 14.

```
secp112r1, secp112r2, secp128r1, secp128r2, secp160k1, secp160r1, secp160r2,
secp192k1, secp192r1, secp224k1, secp224r1, secp256k1, sect1
```

To continue using any of these curves, find third-party alternatives. See Remove the legacy elliptic curves.

Deprecated Tracing Flags Are Obsolete and Must Be Replaced With Unified Logging Equivalents

When Unified Logging was added in Java 9, several tracing flags were deprecated and mapped to their unified logging equivalent. These flags are now obsolete and you must explicitly replace the use of these flags with their unified logging equivalent.

Obsolete Tracing Flag	Unified Logging Replacement
`-XX:+TraceClassLoading`	`-Xlog:class+load=info`
`-XX:+TraceClassUnloading`	`-Xlog:class+unload=info`
`-XX:+TraceExceptions`	`-Xlog:exceptions=info`

See Obsolete the long term deprecated and aliased Trace flags.

Tools and Components Removed and Deprecated in JDK 15

Removal of Nashorn JavaScript Engine

Nashorn JavaScript script engine and APIs, and the `jjs` tool have been removed in JDK 15. The engine, APIs, and tool were deprecated for removal in Java 11. See JEP 372: Remove the Nashorn JavaScript Engine.

Removal of RMI Static Stub Compiler (rmic) Tool

The RMI static stub compiler (rmic) tool has been removed. The rmic tool was deprecated for removal in JDK 13. See Remove rmic from the set of supported tools.

Disable and Deprecate Biased Locking

The biased locking is disabled by default and all related command-line options have been deprecated. See JEP 374: Disable and Deprecate Biased Locking.

Deprecate RMI Activation for Removal

The RMI Activation mechanism has been deprecated and may be removed in a future version of the platform. See JEP 385: Deprecate RMI Activation for Removal.

Features and Components Removed in JDK 14

Remove the Concurrent Mark Sweep (CMS) Garbage Collector

The CMS garbage collector has been removed. See JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector.

Removal of Pack200 Tools and API

The Pack200 tools and API were deprecated in JDK 11 and have been removed in JDK 14.

The `pack200` and `unpack200` tools, and `Pack200` in `java.util.jar.Pack200` package have been removed .

See JEP 367: Remove the Pack200 Tools and API.

Tools and Components Removed in JDK 13

Removal of Old Features from `javadoc` Tool

The following four features have been removed from the *javadoc* tool:

- Support for generating API documentation using HTML 4
- Support for the old *javadoc* API
- Support for generating documentation using HTML frames
- Support for the `--no-module-directories` options

For details about removed `javadoc` features, see JDK-8215608 : Remove old javadoc features.

See Removed Features and Options of JDK 13 Release Notes for list of removed tools and components.

Tools and Components Removed in JDK 12

To know more about the tools and components that are removed in JDK 12, see Removed Features and Options in JDK 12.

Tools and Components Removed in JDK 11

Removal of the Deployment Stack

Java deployment technologies were deprecated in JDK 9 and removed in JDK 11.

Java applet and Web Start functionality, including the Java plug-in, the Java Applet Viewer, Java Control Panel, and Java Web Start, along with `javaws` tool, have been removed in JDK 11.

See Remove Java Deployment Technologies.

Removal of Java EE and CORBA Modules

In JDK 11, the Java EE and CORBA modules were removed. These modules were deprecated for removal in JDK 9.

The removed modules were:

- `java.xml.ws`: Java API for XML Web Services (JAX-WS), Web Services Metadata for the Java Platform, and SOAP with Attachments for Java (SAAJ)
- `java.xml.bind`: Java Architecture for XML Binding (JAXB)
- `java.xml.ws.annotation`: The subset of the JSR-250 Common Annotations defined by Java SE to support web services
- `java.corba`: CORBA
- `java.transaction`: The subset of the Java Transaction API defined by Java SE to support CORBA Object Transaction Services
- `java.activation`: JavaBeans Activation Framework
- `java.se.ee`: Aggregator module for the six modules above
- `jdk.xml.ws`: Tools for JAX-WS
- `jdk.xml.bind`: Tools for JAXB

Existing code with references to classes in these APIs will not compile without changes to the build. Similarly, code on the class path with references to classes in these APIs will fail with `NoDefClassFoundError` or `ClassNotFoundException` unless changes are made in how the application is deployed.

See JEP 320: Remove the Java EE and CORBA Modules to get more information about possible replacements for the modules.

> **Note**
>
> You can download JAXB and JAX-WS from Maven.

Removal of Tools and Components

Main Tools

- `appletviewer`
  See JDK-8200146 : Remove the appletviewer launcher.

CORBA Tools

- `idlj`
- `orbd`
- `servertool`
- `tnamesrv`

In addition, the `rmic` (the RMI compiler) no longer supports the `-idl` or `-iiop` options. See JDK 11 Release Notes.

Java Web Services Tools

- `schemagen`
- `wsgen`
- `wsimport`

- `xjc`

Java Deployment Tools

- `javapackager`
- `javaws`
  See Removal of JavaFX from JDK.

Monitoring Tools

- `jmc`: In JDK 11, JMC is available as a standalone package and not bundled in the JDK.

See Removal of JMC from JDK and Java Mission Control.

JVM-MANAGEMENT-MIB.mib

The specification for JVM monitoring and management through SNMP `JVM-MANAGEMENT-MIB.mib` has been removed. See Removal of JVM-MANAGEMENT-MIB.mib.

SNMP Agent

The `jdk.snmp` module has been removed. See Removal of SNMP Agent.

Oracle Desktop Specific Removals

- Oracle JDK T2K font rasterizer has been removed.
- Lucida Fonts: Oracle JDK no longer ships any fonts and relies entirely on fonts installed on the operating system. See Removal of Lucida Fonts from Oracle JDK.

Tools and Components Removed in JDK 9 and JDK 10

This list includes tools and components that are no longer bundled with the JDK.

Removed Native-Header Generation Tool (javah)

The `javah` tool has been superseded by superior functionality in `javac`. It was removed in JDK 10.

Since JDK 8, `javac` provides the ability to write native header files at the time that Java source code is compiled, thereby eliminating the need for a separate tool.

Instead of `javah`, use

```
javac -h
```

Removed JavaDB

JavaDB, which was a rebranding of Apache Derby, is no longer included in the JDK.

JavaDB was bundled with JDK 7 and JDK 8. It was found in the `db` directory of the JDK installation directory.

You can download and install Apache Derby from Apache Derby Downloads.

Removed the JVM TI hprof Agent

The `hprof` agent library has been removed.

The `hprof` agent was written as demonstration code for the JVM Tool Interface and wasn't intended to be a production tool. The useful features of the `hprof` agent have been superseded by better alternatives, including some that are included in the JDK.

For creating heap dumps in the `hprof` format, use a diagnostic command (`jcmd`) or the `jmap` tool:

- Diagnostic command: `jcmd <pid> GC.heap_dump`. See jcmd.
- jmap: `jmap -dump`. See jmap.

For CPU profiler capabilities, use the Java Flight Recorder, which is bundled with the JDK.

See JEP 240: Remove the JVM TI hprof Agent.

Removed the jhat Tool

The `jhat` tool was an experimental, unsupported heap visualization tool added in JDK 6. Superior heap visualizers and analyzers have been available for many years.

Removed java-rmi.exe and java-rmi.cgi Launchers

The launchers `java-rmi.exe` from Windows and `java-rmi.cgi` from Linux and Solaris have been removed.

`java-rmi.cgi` was in `$JAVA_HOME/bin` on Linux.

`java-rmi.exe` was in `$JAVA_HOME/bin` on Windows.

These launchers were added to the JDK to facilitate use of the RMI CGI proxy mechanism, which was deprecated in JDK 8.

The alternative of using a servlet to proxy RMI over HTTP has been available, and even preferred, for several years. See Java RMI and Object Serialization.

Removed Support for the IIOP Transport from the JMX RMIConnector

The IIOP transport support from the JMX RMI Connector along with its supporting classes have been removed from the JDK.

In JDK 8, support for the IIOP transport was downgraded from required to optional. This was the first step in a multirelease effort to remove support for the IIOP transport from the JMX Remote API. In JDK 9, support for IIOP was removed completely.

Public API changes include:

- The `javax.management.remote.rmi.RMIIIOPServerImpl` class has been deprecated. Upon invocation, all its methods and constructors throw `java.lang.UnsupportedOperationException` with an explanatory message.
- Two classes, `org.omg.stub.javax.management.rmi._RMIConnection_Stub`, and `org.omg.stub.javax.management.rmi._RMIConnection_Tie`, aren't generated.

Dropped Windows 32–bit Client VM

The Windows 32–bit client VM is no longer available. Only a server VM is offered.

JDK 8 and earlier releases offered both a client JVM and a server JVM for Windows 32-bit systems. JDK 9 and later releases offer only a server JVM, which is tuned to maximize peak operating speed.

Removed Java VisualVM

Java VisualVM is a tool that provides information about code running on a Java Virtual Machine. The `jvisualvm` tool was provided with JDK 6, JDK 7, and JDK 8.

Java VisualVM is no longer bundled with the JDK, but you can get it from the VisualVM open source project site.

Removed native2ascii Tool

The `native2ascii` tool has been removed from the JDK. Because JDK 9 and later releases support UTF-8 based properties resource bundles, the conversion tool for UTF-8 based properties resource bundles to ISO-8859-1 is no longer needed.

See UTF-8 Properties Files in *Java Platform, Standard Edition Internationalization Guide*.

6

Preparing for Migration

> **Tip**
>
> **Evaluate the Effort of Migration with Java Migration Analysis**
> Oracle offers Java SE Subscribers and Oracle Cloud Infrastructure (OCI) users the Java Migration Analysis feature in the Java Management Service. This feature performs a comprehensive analysis of the effort involved in migrating to a new JDK version. It generates detailed reports highlighting the effort required and areas needing modification for migrating from an older JDK version to a newer one. During the migration process, you can utilize the analysis reports from Migration Analysis to identify classes and APIs that require changes. The reports specify line numbers in the source code where modifications are needed and highlight both mandatory and recommended changes.

The following sections will help you successfully migrate your application:

- Download the Latest JDK
- Run Your Program Before Recompiling
- Update Third-Party Libraries
- Compile Your Application if Needed
- Run jdeps on Your Code

Download the Latest JDK

Download and install the latest JDK release from Java SE Downloads.

Run Your Program Before Recompiling

Try running your application on the latest JDK release (JDK 25). Most code and libraries should work on JDK 25 without any changes, but there may be some libraries that need to be upgraded.

> **Note**
>
> Migrating is an iterative process. You'll probably find it best to try running your program (this task) first, then complete these three tasks in parallel:
>
> - Update Third-Party Libraries
> - Compile Your Application if Needed
> - Run jdeps on Your Code

When you run your application, look for warnings from the JVM about obsolete VM options. If the VM fails to start, then look for Removed GC Options.

If your application starts successfully, look carefully at your tests and ensure that the behavior is the same as on the JDK version you have been using. For example, a few early adopters have noticed that their dates and currencies are formatted differently. See Be Aware of Changes to Locale Data.

To make your code work on the latest JDK release, understand the new features and changes in each of the JDK release. See Significant Changes in the JDK .

Even if your program appears to run successfully, you should complete the rest of the steps in this guide and review the list of issues.

Update Third-Party Libraries

For every tool and third-party library that you use, you may need to have an updated version that supports the latest JDK release.

Check the websites for your third-party libraries and your tool vendors for a version of each library or tool that's designed to work on the latest JDK. If one exists, then download and install the new version.

If you use Maven or Gradle to build your application, then make sure to upgrade to a recent version that supports the latest JDK version.

If you use an IDE to develop your applications, then it might help in migrating the existing code. The NetBeans, Eclipse, and IntelliJ IDEs all have versions available that include support for the latest JDK.

You can see the status of the testing of many Free Open Source Software (FOSS) projects with OpenJDK builds at Quality Outreach on the OpenJDK wiki.

Compile Your Application if Needed

Compiling your code with the latest JDK compiler will ease migration to future releases since the code may depend on APIs and features, which have been identified as problematic. However, it is not strictly necessary.

If you need to compile your code with JDK 11 and later compilers, then take note of the following:

- Use the new `--release` flag instead of the `-source` and `-target` options. See javac in *Java Development Kit Tool Specifications*.

  The supported values of `--release` are the current Java SE release and a limited number of previous releases, detailed in the command-line help.

  The `javac` can recognize and process class files of all previous JDKs, going all the way back to JDK 1.0.2 class files.

  See JEP 182: Policy for Retiring javac -source and -target Options.

- If you use the `-source` and `-target` options with `javac`, then check the values that you use.

  The supported `-source/-target` values are 25 (the default) till 9. The value 8 is deprecated and causes a warning.

In JDK 8, `-source` and `-target` values of 1.7/7 and earlier were deprecated, and caused a warning. In JDK 9 and above, those values cause an error.

```
>javac -source 7 -target 7 Sample.java
warning: [options] bootstrap class path is not set in conjunction with -
source 7
  not setting the bootstrap class path may lead to class files that cannot
run on JDK 8
    --release 7 is recommended instead of -source 7 -target 7 because it
sets the bootstrap class path automatically
error: Source option 7 is no longer supported. Use 8 or later.
error: Target option 7 is no longer supported. Use 8 or later.
```

> **Note**
>
> When using `--release`, you cannot also use the `--source`/`-source` or `--target`/ `-target` options.

- If you use the underscore character (`_`) as a one-character identifier in source code, then your code won't compile in JDK 11 and later releases. It generates a warning in JDK 8, and an error, starting from JDK 9.

  As an example:

  ```
  static Object _ = new Object();
  ```

  This code generates the following error message from the compiler:

  ```
  MyClass.java:2: error: as of release 9, '_' is a keyword, and may not be
  used as a legal identifier.
  ```

- Critical internal JDK APIs such as `sun.misc.Unsafe` are still accessible in JDK 11 and later, but most of the JDK's internal APIs are not accessible at compile time. You may get compilation errors that indicate that your application or its libraries are dependent on internal APIs.

  To identify the dependencies, run the Java Dependency Analysis tool. See Run jdeps on Your Code. If possible, update your code to use the supported replacement APIs.

  You may use the --add-exports Option and --add-opens Option options as a temporary workaround to compile source code with references to JDK internal classes. See JEP 261: Module System and Strong Encapsulation in the JDK for more information about these options.
- You may see more deprecation warnings than previously.

Run jdeps on Your Code

Run the `jdeps` tool on your application to see what packages and classes your applications and libraries depend on. If you use internal APIs, then `jdeps` may suggest replacements to help you to update your code.

To look for dependencies on internal JDK APIs, run `jdeps` with the `-jdkinternals` option. For example, if you run `jdeps` on a class that calls `sun.misc.BASE64Encoder`, you'll see:

```
>jdeps -jdkinternals Sample.class
Sample.class -> JDK removed internal API
   Sample  -> sun.misc.BASE64Encoder  JDK internal API (JDK removed internal
API)

Warning: JDK internal APIs are unsupported and private to JDK implementation
that are
subject to be removed or changed incompatibly and could break your
application.
Please modify your code to eliminate dependency on any JDK internal APIs.
For the most recent update on JDK internal API replacements, please check:
https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

JDK Internal API                        Suggested Replacement
----------------                        ---------------------
sun.misc.BASE64Encoder                  Use java.util.Base64 @since 1.8
```

If you use Maven, there's a `jdeps` plugin available.

For `jdeps` syntax, see jdeps in the *Java Development Kit Tool Specifications*.

Keep in mind that `jdeps` is a static analysis tool, and static analysis of code might not provide a complete list of dependencies. If the code uses reflection to call an internal API, then `jdeps` doesn't warn you.

7

Migrating from JDK 8 to Later JDK Releases

There were significant changes made between the JDK 8 and later JDK releases.

Every new Java SE release introduces some binary, source, and behavioral incompatibilities with previous releases. The modularization of the Java SE Platform that happened in JDK 9 and later brought many benefits, but also many changes. Code that uses only official Java SE Platform APIs and supported JDK-specific APIs should continue to work without change. Code that uses JDK-internal APIs should continue to run but should be migrated to use the supported APIs.

Some APIs that have been made inaccessible, removed, or altered in their default behavior. You might encounter issues when compiling or running your application. See Removed Tools and Components and Security Updates.

The following sections describe the changes in the JDK package that you should be aware of when migrating your JDK 8 applications to later JDK releases.

Look at the list of changes that you may encounter as you run your application.

- Changes to Internationalization
- Strong Encapsulation in the JDK
- New Version-String Scheme
- Changes to the Installed JDK/JRE Image
- Deployment
- Changes to Garbage Collection
- Running Java Applets
- Behavior Change in Regular Expression Matching
- Plan for Removal of the Security Manager
- Finalization Deprecated for Removal
- Restrictions on and Warnings from Accessing Native Code

When your application is running successfully on the latest version of JDK, review Next Steps, which will help you avoid problems with future releases.

Changes to Internationalization

See *Java Platform, Standard Edition Internationalization Guide* for more information about internationalization APIs and features of the Java SE Platform.

Be Aware of the Default Charset

In JDK 18 and later, UTF-8 is the default charset used by Java SE APIs on all operating systems. See JEP 400: UTF-8 by Default. In contrast, in JDK 17 and earlier releases, the default charset is determined when the Java runtime starts, that is, on macOS, the default

charset used to be UTF-8 except in the POSIX C locale. On other operating systems, it used to depend on the user's locale and the default encoding.

For example, on Windows, it's a codepage-based charset such as `windows-1252` or `windows-31j`. The method `java.nio.charsets.Charset.defaultCharset()` returns the default charset.

Run the following command to determine the charset that used to be the default charset in JDK 17 or earlier releases:

```
java -XshowSettings:properties -version 2>&1 | grep native.encoding
```

If the encoding detected is different from UTF-8, then the application running in your environment may be affected.

Changing the JDK's Default Charset

If your environment is affected, then use the system property `file.encoding` to investigate further. Set its value on the command line to one of the following:

- `UTF-8`: The default charset is UTF-8.
- `COMPAT`: The default charset is determined as in JDK 17 and earlier releases.

Other values for `file.encoding` are not supported.

> **Note**
>
> - Run the command `java -Dfile.encoding=UTF-8 <your application>` with the existing JDK. This will provide the same environment as JDK 18 and later. Check if there are any differences.
> - Run the command `-Dfile.encoding=COMPAT <your application>` on JDK 18 and later to get the previous behaviour, and check if there are any differences.

For more details, see Default Charset in *Java Platform, Standard Edition Internationalization Guide*.

Be Aware of Changes to Locale Data

In JDK 9 and later, the JDK uses locale data in the Common Locale Data Repository (CLDR) to format dates, times, currencies, languages, countries, and time zones in the standard Java APIs. CLDR, which is maintained by the Unicode Consortium, provides locale data of higher quality than the legacy locale data in JDK 8.

Applications that use locale-sensitive APIs, such as `java.time.LocalDate` and `java.util.Currency`, may see different results, and possibly exceptions, when formatting and parsing dates, times, currencies, and related data on JDK 9 and later.

Prior to deploying on JDK 9 or later where CLDR locale data is used by default, you are strongly encouraged to check for compatibility issues by running your applications on JDK 8 with the selected CLDR provider. Do this by starting the Java 8 runtime as follows so that CLDR locale data has priority over legacy locale data:

```
$ java -Djava.locale.providers=CLDR,JRE ...
```

Legacy locale data is included in JDK releases 9 through 22 despite CLDR locale data being the default. In these releases, you can force locale-sensitive APIs to use legacy locale data by specifying `JRE` before `CLDR` in the value of `java.locale.providers` as follows:

```
$ java -Djava.locale.providers=JRE,CLDR ...
```

The JDK has been gradually removing support for legacy locale data:

- In JDK 21, if the value `JRE` (or its alias, `COMPAT`) is specified in the `java.locale.providers` system property at startup, the Java runtime issues a warning indicating that legacy locale data will be removed in a future release.
- In JDK 23, legacy locale data is no longer included. As a result, specifying `JRE` (or its alias, `COMPAT`) in the `java.locale.providers` system property has no effect.

Projects that are still using legacy locale data are *highly encouraged* to switch to Unicode CLDR as soon as possible. See CLDR Locale Data Enabled by Default in the *Java Platform, Standard Edition Internationalization Guide*.

If this is not feasible, use one of the following alternatives:

- Modify Your Code to Always Format and Parse Strings With the Same Patterns as Those in Legacy Locale Data
- Create a Custom Locale Data Provider and Include It in the Application

Modify Your Code to Always Format and Parse Strings With the Same Patterns as Those in Legacy Locale Data

For example, suppose your code uses the locale-sensitive `SimpleDateFormat` API to format `Date` objects. On JDK 8, the code might have obtained a `SimpleDateFormat` as follows:

```
SimpleDateFormat fmt
    = (SimpleDateFormat)DateFormat.getDateInstance(DateFormat.MEDIUM,
Locale.UK);
// prints "19-Mar-2024" on JDK 8 but "19 Mar 2024" on JDK 9
System.out.println(fmt.format(new Date()));
```

You could modify the code to create a `SimpleDateFormat` directly, passing the desired pattern (date components separated by hyphens) to the constructor of `SimpleDateFormat`:

```
SimpleDateFormat fmt = new SimpleDateFormat("dd-MMM-yyyy", Locale.UK);
// prints "19-Mar-2024", even on JDK 9
System.out.println(fmt.format(new Date()));
```

This solution can work well for small applications, or for large applications that store formats in singleton variables whose use is rigorously enforced across the codebase.

Create a Custom Locale Data Provider and Include It in the Application

A custom locale provider can override the `CLDR` provider so that locale-sensitive APIs, when formatting and parsing strings, give priority to the patterns defined by the custom locale provider.

> **Note**
>
> This option is complex because it involves changes to how the application is packaged and deployed. Before considering this option, investigate the previous option, Modify Your Code to Always Format and Parse Strings With the Same Patterns as Those in Legacy Locale Data. This option is more localized.

For example, here is a custom locale data provider that can be used on JDK 9 and later to reinstate the hyphen-separated pattern for UK dates from JDK 8:

```
package com.example.localization;
import java.text.*;
import java.text.spi.*;
import java.util.*;

public class HyphenatedUKDates extends DateFormatProvider {

     @Override
     public Locale[] getAvailableLocales() {
         return new Locale[]{Locale.UK};
     }

     @Override
     public DateFormat getDateInstance(int style, Locale locale) {
         assert locale.equals(Locale.UK);
         switch (style) {
             case DateFormat.FULL:
                 return new SimpleDateFormat("EEEE, d MMMM yyyy");
             case DateFormat.LONG:
                 return new SimpleDateFormat("dd MMMM yyyy");
             case DateFormat.MEDIUM:
                 return new SimpleDateFormat("dd-MMM-yyyy");
             case DateFormat.SHORT:
                 return new SimpleDateFormat("dd/MM/yy");
             default:
                 throw new IllegalArgumentException("style not supported");
         }
     }

     @Override
     public DateFormat getDateTimeInstance(int dateStyle, int timeStyle,
                                           Locale locale)
     {
         ...
     }

     @Override
     public DateFormat getTimeInstance(int style, Locale locale) {
         ...
     }

}
```

The following is another example. The short month name for September differs between `CLDR` and `COMPAT` in the UK locale. The following SPI implementation addresses this incompatibility:

```
package spi;

import java.text.DateFormatSymbols;
import java.text.spi.DateFormatSymbolsProvider;
import java.util.Locale;

public class ShortMonthModifier extends DateFormatSymbolsProvider {

    @Override
    public DateFormatSymbols getInstance(Locale locale) {
        assert locale.equals(Locale.UK);
        return new DateFormatSymbols() {
            @Override
            public String[] getShortMonths() {
                var ret = new
DateFormatSymbols(Locale.UK).getShortMonths().clone();
                ret[Calendar.SEPTEMBER] = "Sep";
                return ret;
            }
        };
    }

    @Override
    public Locale[] getAvailableLocales() {
        return new Locale[]{Locale.UK};
    }
}
```

Once you have implemented a custom locale data provider, package it as described in the section Packaging of Locale Sensitive Service Provider Implementations in the `LocaleServiceProvider` JavaDoc API documentation. Afterward, place it on the classpath and then run your applications with the `-Djava.locale.providers=SPI,CLDR` command-line option.

See JEP 252: Use CLDR Locale Data by Default.

Strong Encapsulation in the JDK

Some tools and libraries use reflection to access parts of the JDK that are meant for internal use only. This use of reflection negatively impacts the security and maintainability of the JDK. To aid migration, JDK 9 through JDK 16 allowed this reflection to continue, but emitted warnings about *illegal reflective access*. However, JDK 17 and later is *strongly encapsulated*, so this reflection is no longer permitted by default. Code that accesses non-public fields and methods of `java.*` APIs will throw an `InaccessibleObjectException`.

> **Note**
>
> Strong encapsulation applies to `java.*` APIs in the JDK. The `sun.misc` and `sun.reflect` packages are **not** strongly encapsulated. They are available for reflection by libraries and tools in all JDK releases, including JDK 25. This includes the `sun.misc.Unsafe` class.
> Although the `sun.misc.Unsafe` class is still available, its methods are unsupported and most have been superseded by standard APIs. As a result, the superseded methods were deprecated for removal in JDK 23. Invoking them in JDK 24 or above causes a warning at run time. See JEP 498: Warn upon Use of Memory-Access Methods in sun.misc.Unsafe.

The `java` launcher option `--illegal-access` allowed reflection to JDK internals in JDK 9 through JDK 16. You could specify the following parameters:

- `--illegal-access=permit`: Allows code on the class path to reflect over the internals of `java.*` packages that existed in JDK 8. The first reflective-access operation to any such element causes a warning to be issued, but no warnings are issued after that point.
- `--illegal-access=warn`: Causes a warning message to be issued for each illegal reflective-access operation.
- `--illegal-access=debug`: Causes both a warning message and a stack trace to be shown for each illegal reflective-access operation.
- `--illegal-access=deny`: Disables all illegal reflective-access operations except for those enabled by other command-line options, such as `--add-opens`.

Many tools and libraries have been updated to avoid relying on JDK internals and instead use standard Java APIs that were introduced between JDK 8 and 17. This means the `--illegal-access` launcher option is obsolete in JDK 17. Any use of this launcher option in JDK 17, whether with `permit`, `warn`, `debug`, or `deny`, will have no effect other than to issue a warning message.

If you cannot obtain or deploy newer versions of tools and libraries, then there are two command-line options that enable you to grant access to specific internal APIs for older versions of tools and libraries:

- --add-exports Option: If you have an older tool or library that needs to use an internal API that has been strongly encapsulated, then use the `--add-exports` runtime option. You can also use `--add-exports` at compile time to access the internal APIs.
- --add-opens Option: If you have an older tool or library that needs to access non-public fields and methods of `java.*` APIs by reflection, then use the `--add-opens` option.

See JEP 403: Strongly Encapsulate JDK Internals by Default.

--add-exports Option

If you have an older tool or library that needs to use an internal API that has been strongly encapsulated, then use the `--add-exports` runtime option. You can also use `--add-exports` at compile time to access the internal APIs.

The syntax of the `--add-exports` option is:

```
--add-exports <source-module>/<package>=<target-module>(,<target-module>)*
```

where <*source-module*> and <*target-module*> are module names and <*package*> is the name of a package.

The `--add-exports` option allows code in the target module to access types in the named package of the source module if the target module reads the source module.

As a special case, if the <*target-module*> is `ALL-UNNAMED`, then the source package is exported to all unnamed modules, whether they exist initially or are created later on. For example:

```
--add-exports java.management/sun.management=ALL-UNNAMED
```

This example allows code in all unnamed modules (code on the class path) to access the public members of public types in `java.management/sun.management`.

> **Note**
>
> If the code on the class path uses the reflection API (`setAccessible(true)`) to attempt to access non-public fields and methods of `java.*` APIs, then the code will fail. JDK 17 doesn't allow this by default. However, you can use the `--add-opens` option to allow this. See the section --add-opens Option for more information.

If an application `oldApp` that runs on the classpath must use the unexported `com.sun.jmx.remote.internal` package of the `java.management` module, then the access that it requires can be granted in this way:

```
--add-exports java.management/com.sun.jmx.remote.internal=ALL-UNNAMED
```

You can also use the `Add-Exports` JAR file manifest attribute:

```
Add-Exports:java.management/sun.management
```

Use the `--add-exports` option carefully. You can use it to gain access to an internal API of a library module, or even of the JDK itself, but you do so at your own risk. If that internal API changes or is removed, then your library or application fails.

See JEP 261: Module System.

--add-opens Option

Some tools and libraries use the reflection API (`setAccessible(true)`) to attempt to access non-public fields and methods of `java.*` APIs. This is no longer possible by default on JDK 17, but you can use the `--add-opens` option on the command line to enable it for specific tools and libraries.

The syntax for `--add-opens` is:

```
--add-opens <module>/<package>=<target-module>(,<target-module>)*
```

This option allows <*module*> to open <*package*> to <*target-module*>, regardless of the module declaration.

As a special case, if the <*target-module*> is `ALL-UNNAMED`, then the source package is exported to all unnamed modules, whether they exist initially or are created later on. For example:

```
--add-opens java.management/sun.management=ALL-UNNAMED
```

This example allows all of the code on the class path to access nonpublic members of public types in the `java.management/sun.management` package.

> **Note**
>
> In a JNLP file for Java Web Start, you must include an equals sign between `--add-opens` and its value.
>
> ```
> <j2se version="10" java-vm-args="--add-opens=<module>/<package>=ALL-
> UNNAMED"  />
> ```
>
> The equals sign between `--add-opens` and its value is optional on the command line.

New Version-String Scheme

JDK 10 introduced some minor changes, to better accommodate the time-based release model, to the version-string scheme introduced in JDK 9. JDK 11 and later retains the version string format that was introduced in JDK 10.

If your code relies on the version-string format to distinguish major, minor, security, and patch update releases, then you may need to update it.

The format of the new version-string is:

```
$FEATURE.$INTERIM.$UPDATE.$PATCH
```

A simple Java API to parse, validate, and compare version strings has been added. See `java.lang.Runtime.Version`.

See Version String Format in *Java Platform, Standard Edition Installation Guide* .

For the changes to the version string introduced in JDK 9, see JEP 223: New Version-String Scheme .

For the version string changes introduced in JDK 10, see JEP 322: Time-Based Release Versioning.

Changes to the Installed JDK/JRE Image

Significant changes have been made to the JDK and JRE.

Changed JDK and JRE Layout

After you install the JDK, if you look at the file system, you'll notice that the directory layout is different from that of releases before JDK 9.

JDK 11 and Later

JDK 11 and later does not have the JRE image. See Installed Directory Structure of JDK in *Java Platform, Standard Edition Installation Guide*.

JDK 9 and JDK 10

Prior releases had two types of runtime images: the JRE, which was a complete implementation of the Java SE Platform, and the JDK, which included the entire JRE in a `jre/` directory, plus development tools and libraries.

In JDK 9 and and JDK 10, the JDK and JRE are two types of modular runtime images containing the following directories:

- **`bin`**: contains binary executables.
- **`conf`**: contains `.properties`, `.policy`, and other kinds of files intended to be edited by developers, deployers, and end users. These files were formerly found in the `lib` directory or its subdirectories.
- **`lib`**: contains dynamically linked libraries and the complete internal implementation of the JDK.

In JDK 9 and JDK 10, there are still separate JDK and JRE downloads, but each has the same directory structure. The JDK image contains the extra tools and libraries that have historically been found in the JDK. There are no `jdk/` versus `jre/` wrapper directories, and binaries (such as the `java` command) aren't duplicated.

See JEP 220: Modular Run-Time Images.

New Class Loader Implementations

JDK 9 and later releases maintain the hierarchy of class loaders that existed since the 1.2 release. However, the following changes have been made to implement the module system:

- The application class loader is no longer an instance of `URLClassLoader` but, rather, of an internal class. It is the default loader for classes in modules that are neither Java SE nor JDK modules.
- The extension class loader has been renamed; it is now the platform class loader. All classes in the Java SE Platform are guaranteed to be visible through the platform class loader.

  Just because a class is visible through the platform class loader does not mean the class is actually defined by the platform class loader. Some classes in the Java SE Platform are defined by the platform class loader while others are defined by the bootstrap class loader. Applications should not depend on which class loader defines which platform class.

  The changes that were implemented in JDK 9 may impact code that creates class loaders with `null` (that is, the bootstrap class loader) as the parent class loader and assumes that all platform classes are visible to the parent. Such code may need to be changed to use the platform class loader as the parent (see `ClassLoader.getPlatformClassLoader`).

  The platform class loader is not an instance of `URLClassLoader`, but, rather, of an internal class.
- The bootstrap class loader is still built-in to the Java Virtual Machine and represented by `null` in the `ClassLoader` API. It defines the classes in a handful of critical modules, such as `java.base`. As a result, it defines far fewer classes than in JDK 8, so applications that are deployed with `-Xbootclasspath/a` or that create class loaders with `null` as the parent may need to change as described previously.

Removed rt.jar and tools.jar

Class and resource files previously stored in `lib/rt.jar`, `lib/tools.jar`, `lib/dt.jar` and various other internal JAR files are stored in a more efficient format in implementation-specific files in the `lib` directory.

The removal of `rt.jar` and similar files leads to issues in these areas:

- Starting from JDK 9, `ClassLoader.getSystemResource` doesn't return a URL pointing to a JAR file (because there are no JAR files). Instead, it returns a `jrt` URL, which names the modules, classes, and resources stored in a runtime image without revealing the internal structure or format of the image.

  For example:

  ```
  ClassLoader.getSystemResource("java/lang/Class.class");
  ```

  When run on JDK 8, this method returns a JAR URL of the form:

  ```
  jar:file:/usr/local/jdk8/jre/lib/rt.jar!/java/lang/Class.class
  ```

  which embeds a file URL to name the actual JAR file within the runtime image.

  A modular image doesn't contain any JAR files, so URLs of this form make no sense. On JDK 9 and later releases, this method returns:

  ```
  jrt:/java.base/java/lang/Class.class
  ```

- The `java.security.CodeSource` API and security policy files use URLs to name the locations of code bases that are to be granted specific permissions. See Policy File Syntax in *Java Platform, Standard Edition Security Developer's Guide*. Components of the runtime system that require specific permissions are currently identified in the `conf/security/java.policy` file by using file URLs.
- Older versions of IDEs and other development tools require the ability to enumerate the class and resource files stored in a runtime image, and to read their contents directly by opening and reading `rt.jar` and similar files. This isn't possible with a modular image.

Removed Extension Mechanism

In JDK 8 and earlier, the extension mechanism made it possible for the runtime environment to find and load extension classes without specifically naming them on the class path. Starting from JDK 9, if you need to use the extension classes, ensure that the JAR files are on the class path.

In JDK 9 and JDK 10, the `javac` compiler and `java` launcher will exit if the `java.ext.dirs` system property is set, or if the `lib/ext` directory exists. To additionally check the platform-specific systemwide directory, specify the `-XX:+CheckEndorsedAndExtDirs` command-line option. This causes the same exit behavior to occur if the directory exists and isn't empty. The extension class loader is retained in JDK 9 (and later releases) and is specified as the platform class loader (see `getPlatformClassLoader`.) However, in JDK 11, this option is obsolete and a warning is issued when it is used.

The following error means that your system is configured to use the extension mechanism:

```
<JAVA_HOME>/lib/ext exists, extensions mechanism no longer supported; Use -
classpath instead.
.Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

You'll see a similar error if the `java.ext.dirs` system property is set.

To fix this error, remove the `ext/` directory or the `java.ext.dirs` system property.

See JEP 220: Modular Run-Time Images.

Removed Endorsed Standards Override Mechanism

The `java.endorsed.dirs` system property and the `lib/endorsed` directory are no longer present. The `javac` compiler and `java` launcher will exit if either one is detected.

Starting from JDK 9, you can use upgradeable modules or put the JAR files on the class path.

This mechanism was intended for application servers to override components used in the JDK. Packages to be updated would be placed into JAR files, and the system property `java.endorsed.dirs` would tell the Java runtime environment where to find them. If a value for this property wasn't specified, then the default of `$JAVA_HOME/lib/endorsed` was used.

In JDK 8, you can use the `-XX:+CheckEndorsedAndExtDirs` command-line argument to check for such directories anywhere on the system.

In JDK 9 and later releases, the `javac` compiler and `java` launcher will exit if the `java.endorsed.dirs` system property is set, or if the `lib/endorsed` directory exists.

The following error means that your system is configured to use the endorsed standards override mechanism:

```
<JAVA_HOME>/lib/endorsed is not supported. Endorsed standards and standalone
APIs
in modular form will be supported via the concept of upgradeable modules.
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

You'll see a similar error if the `java.endorsed.dirs` system property is set.

To fix this error, remove the `lib/endorsed` directory, or unset the `java.endorsed.dirs` system property.

See JEP 220: Modular Run-Time Images.

Removed macOS-Specific Features

This section includes macOS-specific features that have been removed, starting in JDK 9.

Platform-Specific Desktop Features

The `java.awt.Desktop` class contains replacements for the APIs in the Apple–specific `com.apple.eawt` and `com.apple.eio` packages. The new APIs supersede the macOS APIs and are platform-independent.

The APIs in the `com.apple.eawt` and `com.apple.eio` packages are encapsulated, so you won't be able to compile against them in JDK 9 or later releases. However, they remain accessible at runtime, so existing code that is compiled to old versions continues to run. Eventually, libraries or applications that use the internal classes in the `apple` and `com.apple` packages and their subpackages will need to migrate to the new API.

The `com.apple.concurrent` and `apple.applescript` packages are removed without any replacement.

See JEP 272: Platform-Specific Desktop Features.

Removed AppleScript Engine

The AppleScript engine, a platform-specific `javax.script` implementation, has been removed without any replacement in the JDK.

The AppleScript engine has been mostly unusable in recent releases. The functionality worked only in JDK 7 or JDK 8 on systems that already had Apple's version of the `AppleScriptEngine.jar` file on the system.

Windows Registry Key Changes

The Java 11 and later installer creates Windows registry keys when installing the JDK. For JDK 18, the installer creates the following Windows registry keys:

- `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JDK`
- `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JDK\18`

If two versions of the JDK are installed, then two different Windows registry keys are created. For example, if JDK 17.0.1 is installed with JDK 18, then the installer creates the another Windows registry key as shown:

- `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JDK`
- `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JDK\18`
- `HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JDK\17.0.1`

Deployment

Java deployment technologies were deprecated in JDK 9 and removed in JDK 11.

Use the `jlink` tool introduced with JDK 9 to package and deploy dedicated runtimes rather than relying on a pre-installed system JRE.

Removed Launch-Time JRE Version Selection

The ability to request a version of the JRE that isn't the JRE being launched at launch time is removed, starting in JDK 9.

Modern applications are typically deployed using Java Web Start (JNLP), native OS packaging systems, or active installers. These technologies have their own methods to manage the JREs needed, by finding or downloading and updating the required JRE, as needed. This makes the launcher's launch-time JRE version selection obsolete.

In the previous releases, you could specify what JRE version (or range of versions) to use when starting an application. Version selection was possible through both a command-line option and manifest entry in the application's JAR file.

Starting in JDK 9, the `java` launcher is modified as follows:

- Emits an error message and exits if the `-version:` option is given on the command line.
- Emits a warning message and continues if the `JRE-Version` manifest entry is found in a JAR file.

See JEP 231: Remove Launch-Time JRE Version Selection.

Removed Support for Serialized Applets

Starting in JDK 9, the ability to deploy an applet as a serialized object isn't supported. With modern compression and JVM performance, there's no benefit to deploying an applet in this way.

The `object` attribute of the `applet` tag and the `object` and `java object` applet parameter tags are ignored when starting applet.

Instead of serializing applets, use standard deployment strategies.

JNLP Specification Update

JNLP (Java Network Launch Protocol) has been updated to remove inconsistencies, make code maintenance easier, and enhance security.

In JDK 9, JNLP has been updated as follows:

1. `&` instead of `&` in JNLP files.
   The JNLP file syntax conforms to the XML specification and all JNLP files should be able to be parsed by standard XML parsers.

   JNLP files let you specify complex comparisons. Previously, this was done by using the ampersand (`&`), but this isn't supported in standard XML. If you're using `&` to create complex comparisons, then replace it with `&` in your JNLP file. `&` is compatible with all versions of JNLP.
2. Comparing numeric version element types against nonnumeric version element types.

   Previously, when an `int` version element was compared with another version element that couldn't be parsed as an `int`, the version elements were compared lexicographically by ASCII value.

   If the element that can be parsed as an `int` is a shorter string than the other element, it will be padded with leading zeros before being compared lexicographically by ASCII value. This ensures there can be no circularity.

   In the case where both version comparisons and a JNLP servlet are used, you should use only numeric values to represent versions.
3. Component extensions with nested resources in `java` (or `j2se`) elements.
   This is permitted in the specification. It was previously supported, but this support wasn't reflected in the specification.
4. FX XML extension.
   The JNLP specification has been enhanced to add a `type` attribute to `application-desc` element, and add the subelement `param` in `application-desc` (as it already is in `applet-desc`).

This doesn't cause problems with existing applications because the previous way of specifying a JavaFX application is still supported.

See the JNLP specification updates at JSR-056.

Changes to Garbage Collection

This section describes changes to garbage collection starting in JDK 9.

Make G1 the Default Garbage Collector

The Garbage-First Garbage Collector (G1 GC) is the default garbage collector in JDK 9 and later releases.

A low-pause collector such as G1 GC should provide a better overall experience, for most users, than a throughput-oriented collector such as the Parallel GC, which is the JDK 8 default.

See Ergonomic Defaults for G1 GC and Tunable Defaults in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* for more information about tuning G1 GC.

Removed GC Options

The following GC combinations will cause your application to fail to start in JDK 9 and later releases:

- `DefNew + CMS`
- `ParNew + SerialOld`
- `Incremental CMS`

The foreground mode for CMS has also been removed. The command-line flags that were removed are `-Xincgc`, `-XX:+CMSIncrementalMode`, `-XX:+UseCMSCompactAtFullCollection`, `-XX:+CMSFullGCsBeforeCompaction`, and `-XX:+UseCMSCollectionPassing`.

The command-line flag `-XX:+UseParNewGC` no longer has an effect. The `ParNew` flag can be used only with CMS and CMS requires `ParNew`. Thus, the `-XX:+UseParNewGC` flag has been deprecated and is eligible for removal in a future release.

See JEP 214: Remove GC Combinations Deprecated in JDK 8.

> **Note**
>
> The CMS garbage collector has been removed. See JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector.

Removed Permanent Generation

The permanent generation was removed in JDK 8, and the related VM options cause a warning to be printed. You should remove these options from your scripts:

- `-XX:MaxPermSize=`*`size`*
- `-XX:PermSize=`*`size`*

In JDK 9 and later releases, the JVM displays a warning like this:

```
Java HotSpot(TM) 64-Bit Server VM warning: Ignoring option MaxPermSize;
support was removed in 8.0
```

Tools that are aware of the permanent generation may have to be updated.

See JEP 122: Remove the Permanent Generation and JDK 9 Release Notes - Removed APIs, Features, and Options .

Changes to GC Log Output

Garbage collection (GC) logging uses the JVM unified logging framework, and there are some differences between the new and the old logs. Any GC log parsers that you're working with will probably need to change.

You may also need to update your JVM logging options. All GC-related logging should use the `gc` tag (for example, `–Xlog:gc`), usually in combination with other tags. The `–XX:+PrintGCDetails` and `-XX:+PrintGC` options have been deprecated.

See Enable Logging with the JVM Unified Logging Framework in the *Java Development Kit Tool Specifications* and JEP 271: Unified GC Logging.

Running Java Applets

If you still rely on applets, it might be possible to launch them on Windows systems by using JRE 8 with Microsoft Edge in Internet Explorer mode. See Microsoft Edge + Internet Explorer mode: Getting Started guide.

As of September 2021, the Java Plugin required to launch Applets, remains updated on Windows in Java 8 but may be removed at any time in a future update release.

Oracle Customers can find more information at My.Oracle.Support Note 251148.1 - Java SE 8 End of Java Plugin Support (requires login).

Behavior Change in Regular Expression Matching

The `java.util.regex.Pattern` class defines character classes in regular expressions with square brackets. For example, `[abc]` matches `a`, `b`, or `c`. Negated character classes are defined with a caret immediately following the opening square brace. For example, `[^abc]` matches any character except `a`, `b`, or `c`.

In JDK 8 and earlier, negated character classes did not negate nested character classes. For example, `[^a-b[c-d]e-f]` matches `c` but does not match `a` or `e` as they are not within the nested class. The operators are applied one after another. In this example, the negation operator `^` is applied before nesting. In JDK 8 and earlier, the operator `^` was applied only to the outermost characters in the character class but *not* to nested character classes. This behaviour was confusing and difficult to understand.

However, in JDK 9 and later, the negation operator was applied to all nested character classes. For example, `[^a-b[c-d]e-f]` does not match `c`.

To explain further, consider the following regular expression:

```
[^a-d&&c-f]
```

In JDK 8, the `^` operator is applied first, hence this example is interpreted as `[^a-d]` intersected with `[c-f]`. This matches `e` and `f` but not `a`, `b`, `c`, or `d`.

In JDK 9 and later, the `&&` operator is applied first, hence this example is interpreted as the complement of `[a-d]&&[c-f]`. This matches `a`, `b`, `e`, and `f` but not `c` or `d`.

As a best practice, look for regular expressions that use character classes with some combination of negation, intersection, and nested classes. These regular expressions may need to be adjusted to account for the changed behavior.

Plan for Removal of the Security Manager

In JDK 17, the Security Manager and APIs related to it have been deprecated and are subject to removal in a future release. There is no replacement for the Security Manager. See JEP 411: Deprecate the Security Manager for Removal for discussion and alternatives.

In JDK 24, the Security Manager has been permanently disabled. The Security Manager API will be removed in a future release. See JEP 486: Permanently Disable the Security Manager and The Security Manager Is Permanently Disabled in *Java Platform, Standard Edition Security Developer's Guide*.

Finalization Deprecated for Removal

Finalization has been deprecated for removal in JDK 18.

The use of finalization is discouraged. It can lead to problems with security, performance, and reliability. See JEP 421: Deprecate Finalization for Removal.

For information about detecting and migrating from finalization, see:

- Finalization and Weak, Soft, and Phantom References in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*
- Monitoring the Objects Pending Finalization in *Java Platform, Standard Edition Troubleshooting Guide*

Restrictions on and Warnings from Accessing Native Code

The Java platform contains two interfaces that enable Java code and native code to interoperate with each other:

- The Java Native Interface (JNI) enables Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.
- The Foreign Function and Memory (FFM) API enables Java code to invoke foreign functions and safely access foreign memory. A foreign function is code outside the JVM. Foreign memory is memory not managed by the JVM. The FFM API is the preferred alternative to the JNI because it better mitigates the risks from interacting with native code.

In both the JNI and the FFM API, to call native code, you must first load a native library and then link a Java construct to a function in the library. The methods and functions involved in these loading and linking steps are restricted. If they are used incorrectly, they can crash the JVM and may silently result in memory corruption. These restrictions are called *native access restrictions*. The FFM API requires you to enable native access by specifying a command-line option if an application calls any of its methods that have native access restrictions. In JDK 24

and later, the JNI will also require you to specify the same command-line option if you call any of its functions or methods that have native access restrictions.

Code that uses JNI is affected by native access restrictions if any of the following are true:

- It calls `System::loadLibrary`, `System::load`, `Runtime::loadLibrary`, or `Runtime::load`.

  These methods load native libraries, which is risky because they can cause native code to be run. If a native library defines initialization functions, then the operating system runs them when loading the library. These functions contain arbitrary native code. In addition, if a native library defines a `JNI_OnLoad` function, then the Java runtime invokes it when loading the library. This function also contains arbitrary native code.

- It declares a `native` method.

  When a `native` method is first called, it is automatically linked to a corresponding function in a native library. (See Resolving Native Method Names.) This linking step, which is called *binding*, is a restricted operation. The JNI can't verify that the native method is compatible with the bound function in the native library.

Code that uses the FFM API is affected by native access restrictions if it uses a restricted method. See Restricted Methods.

Enabling Native Access

To enable native access for specific modules on the module path, specify a comma-separated list of module names:

```
java --enable-native-access=M1,M2,... MyApp
```

To enable native access for all code on the class path, which enables access to restricted JNI and FFM API methods and functions, use the following command-line option:

```
java --enable-native-access=ALL-UNNAMED MyApp
```

You can also specify the `--enable-native-access` option as follows:

- Set it in the environment variable `JDK_JAVA_OPTIONS`. See Using the JDK_JAVA_OPTIONS Launcher Environment Variable.
- Specify it in a command-line argument file. See java Command-Line Argument Files.
- Add `Enable-Native-Access: ALL-UNNAMED` to the manifest of an executable JAR file. See JAR Manifest.
- If you have created a custom runtime for your application, specify it in the `jlink` command through the `--add-options` plugin. Run the command `jlink –list-plugins` for a list of available plugins.
- If your code creates modules dynamically, enable native access for them with the `ModuleLayer.Controller::enableNativeAccess` method, which is itself a FFM API restricted method. Code can dynamically check if its module has native access with the `Module::isNativeAccessEnabled` method.
- The JNI Invocation API allows a native application to embed a JVM in its own process. A native application that uses the JNI Invocation API can enable native access for modules in the embedded JVM by specifying the `--enable-native-access` option when creating the JVM. See the function `JNI_CreateJavaVM`.

Enabling Native Access More Selectively

The `--enable-native-access=ALL-UNNAMED` option lifts native access restrictions for all classes on the class path. It's recommended that you enable native access more selectively by moving JAR files that use the JNI or the FFM API to the module path. This allows native access to be enabled for those JAR files specifically, not for the entire class path. You can move a JAR file from the class path to the module path without it being modularized. The Java runtime will treat it as an automatic module whose name is based on its file name. See Incremental Modularization with Automatic Modules.

Controlling the Effect of Native Access Restrictions

If native access is not enabled for a module, then it is illegal for code in that module to perform a restricted operation, in particular, call any restricted function or method from the JNI or the FFM API. In JDK 24 and later, you can specify what happens when such a module performs a restricted operation by setting the `--illegal-native-access` command-line option to one of the following values:

- `allow`: Allows the restricted operation to proceed.
- `warn`: Allows the restricted operation to proceed and issues a warning the first time that an illegal native access occurs in a particular module. At most one warning per module is issued. This is the default value in JDK 24 and later.
- `deny`: Throws an `IllegalCallerException` for every illegal native access operation. This will be the default value in a future release of the JDK.

It's recommended that you run existing code with `--illegal-native-access=deny` to identify code that requires native access.

Identifying the Use of Native Code

The JFR events `jdk.NativeLibraryLoad` and `jdk.NativeLibraryUnload` track the loading and unloading of native libraries.

The `jnativescan` tool helps identify libraries that use the JNI. It statically scans code in a provided module path or class path and reports uses of restricted methods and declarations of `native` methods.

8
Next Steps

After you have your application working on JDK 25, here are some suggestions that can help you get the most from the Java SE Platform:

- Use the `--release` flag to compile for your preferred JDK version. See Compile Your Application if Needed for details.
- Take advantage of your IDE's suggestions for updating your code with the latest features.
- Find out if your code is using deprecated APIs by running the static analysis tool jdeprscan. As already mentioned in this guide, APIs can be removed from the JDK, but only with advance notice.
- Get familiar with new features like multi-release JAR files (see jar).
- If you are a macOS user, then enable the new rendering pipeline (introduced in JDK 17), which uses the Apple Metal API. See JEP 382: New macOS Rendering Pipeline. This option is not enabled by default.