JDK 18 Release Notes

All JDK Release Notes | Java Development Kit 18 Release Notes

JDK 18 Release Notes

The Java Platform, Standard Edition 18 Development Kit (JDK 18) is a feature release of the Java SE platform. It contains new features and enhancements in many functional areas. The *Release Notes* below describe the important changes, enhancements, removed APIs and features, deprecated APIs and features, and other information about JDK 18 and Java SE 18. Links to other sources of information about JDK 18 are also provided below:

- JDK 18 Read Me
- JDK 18 Guides and Reference Documentation - Displays a page containing links to the user guides, troubleshooting information, and specific information of interest to users moving from previous versions of the JDK.
- JDK 18 Specifications:
  - JDK 18 API Specification
  - Java Language and Virtual Machine Specifications
- Oracle JDK 18 Certified System Configurations
- JDK 18 Supported Locales
- Submitting a Bug Report and Available Support Options
- Copyright and License Terms for Documentation

Jump to category:

Introduction

These notes describe important changes, enhancements, removed APIs and features, deprecated APIs and features, and other information about JDK 18 and Java SE 18. In some cases, the descriptions provide links to additional detailed information about an issue or a change. This page does not duplicate the descriptions provided by the Java SE 18 (JSR 393) Platform Specification, which provides informative background for all specification changes and might also include the identification of removed or deprecated APIs and features not described here. The Java SE 18 (JSR 393) specification provides links to:

- **Annex 1:** The complete Java SE 18 API Specification.
- **Annex 2:** An annotated API specification showing the exact differences relative to Java SE 18. Informative background for these changes may be found in the list of approved Change Specification Requests for this release.
- **Annex 3:** Java SE 18 Editions of The Java Language Specification and The Java Virtual Machine Specification. The Java SE 18 Editions contain all corrections and clarifications made since the Java SE 17 Editions, as well as additions for new features.

You should be aware of the content in the Java SE 18 (JSR 393) specification as well as the items described in this page.

The descriptions on this Release Notes page also identify potential compatibility issues that you might encounter when migrating to JDK 18. The Kinds of Compatibility page on the OpenJDK wiki identifies the following three types of potential compatibility issues for Java programs that might be used in these release notes:

- **Source:** Source compatibility preserves the ability to compile existing source code without error.
- **Binary:** Binary compatibility is defined in The Java Language Specification as preserving the ability to link existing class files without error.
- **Behavioral:** Behavioral compatibility includes the semantics of the code that is executed at runtime.

See CSRs Approved for JDK 18 for the list of CSRs closed in JDK 18 and the Compatibility & Specification Review (CSR) page on the OpenJDK wiki for general information about compatibility.

The full version string for this release is build 18+36 (where "+" means "build"). The version number is 18.

IANA Data 2021e

JDK 18 contains IANA time zone data version 2021e. For more information, refer to Timezone Data Versions in the JRE Software.

TOP

What's New in JDK 18 - New Features and Enhancements

This section describes some of the enhancements in Java SE 18 and JDK 18. In some cases, the descriptions provide links to additional detailed information about an issue or a change. The APIs described here are provided with the Oracle JDK. It includes a complete implementation of the Java SE 18 Platform and additional Java APIs to support developing, debugging, and monitoring Java applications. Another source of information about important enhancements and new features in Java SE 18 and JDK 18 is the Java SE 18 (JSR 393) Platform Specification, which documents the changes to the specification made between Java SE 17 and Java SE 18. This document includes descriptions of those new features and enhancements that are also changes to the specification. The descriptions also identify potential compatibility issues that you might encounter when migrating to JDK 18.

NOTE: Release Notes for JEPs that added new features or enhancements in this release are grouped in the following categories of Core library improvements and updates, Tool improvements, and Previews and Incubators. Release Notes for JEPs that deprecated or removed APIs, features, and options are described in Deprecated Features and Options.

JEPs for Core library improvements and updates:

core-libs/java.nio.charsets

➜ **JEP 400: UTF-8 by Default**

Starting with JDK 18, UTF-8 is the default charset for the Java SE APIs. APIs that depend on the default charset now behave consistently across all JDK implementations and independently of the user's operating system, locale, and configuration. Specifically, `java.nio.charset.Charset#defaultCharset()` now returns `UTF-8` charset by default. The `file.encoding` system property is now a part of the implementation specification, which may accept `UTF-8` or `COMPAT`. The latter is a new property value that instructs the runtime to behave as previous releases. This change is significant to users who call APIs that depend on the default charset. Users can determine whether they'd be affected or not, by specifying `-Dfile.encoding=UTF-8` as the command line option with the existing Java runtime environment.

For further details, see JEP 400

See JDK-8187041

core-libs/java.net

➜ **JEP 408: Simple Web Server**

jwebserver, a command-line tool to start a minimal static web server, has been introduced. The tool and the accompanying API are located in the com.sun.net.httpserver package of the jdk.httpserver module and are designed to be used for prototyping, ad-hoc coding, and testing, particularly in educational contexts.

For further details, see JEP 408.

See JDK-8260510

core-libs/java.lang:reflect

➜ **JEP 416: Reimplement Core Reflection With Method Handles**

JEP 416 reimplements core reflection with method handles. Code that depends upon highly implementation-specific and undocumented aspects of the existing implementation might be impacted. Issues that might arise include:

- Code that inspects the internal generated reflection classes (such as, subclasses of `MagicAccessorImpl`) no longer works and must be updated.
- Code that attempts to break the encapsulation and change the value of the private final `modifiers` field of `Method`, `Field` and `Constructor` class to be different from the underlying member might cause a runtime error. Such code must be updated.

To mitigate this compatibility risk, you can enable the old implementation as a workaround by using –`Djdk.reflect.useDirectMethodHandle=false`. We will remove the old core reflection implementation in a future release. The `-Djdk.reflect.useDirectMethodHandle=false` workaround will stop working at that point.

For further details, see JEP 416.

See JDK-8271820

core-libs/java.net

➜ **JEP 418: Internet-Address Resolution SPI**

Introduce a service-provider interface (SPI) for host name and address resolution, so that java.net.InetAddress can make use of resolvers other than the platform's built-in resolver. This new SPI allows replacing the operating system's native resolver, which is typically configured to use a combination of a local hosts file and the Domain Name System (DNS).

For further details, see JEP 418.

See JDK-8263693

JEPs for Tool improvements:

tools/javadoc(tool)

➜ **JEP 413: Code Snippets in Java API Documentation**

An @snippet tag for JavaDoc's Standard Doclet has been added, to simplify the inclusion of example source code in API documentation.

For further details, see JEP 413 and A Programmmer's Guide to Snippets.

See JDK-8201533

JEPs for Previews and Incubators:

core-libs

➜ **JEP 417: Vector API (Third Incubator)**

Introduce an API 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.

For further details, see JEP 417.

See JDK-8269306

core-libs

➜ **JEP 419: Foreign Function & Memory API (Second Incubator)**

Introduce an API by which Java programs can interoperate with code and data outside of the Java runtime. By efficiently invoking foreign functions (i.e., code outside the JVM), and by safely accessing foreign memory (i.e., memory not managed by the JVM), the API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI.

For further details, see JEP 419.

See JDK-8274073

specification/language

➜ **JEP 420: Pattern Matching for switch (Second Preview)**

Enhance the Java programming language with pattern matching for switch expressions and statements, along with extensions to the language of patterns. 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.

For further details, see JEP 420.

See JDK-8273326

NOTE: The following Release Notes describe new features in this release that are not related to the JEPs described above:

hotspot/gc
➜ **ZGC Supports String Deduplication**
The Z Garbage Collector now supports string deduplication (JEP 192).

See JDK-8267186

hotspot/gc
➜ **SerialGC Supports String Deduplication**
The Serial Garbage Collector now supports string deduplication (JEP 192).

See JDK-8272609

hotspot/gc
➜ **ParallelGC Supports String Deduplication**
The Parallel Garbage Collector now supports string deduplication (JEP 192).

See JDK-8267185

tools/javac
➜ **Passing Originating Elements From Filer to JavaFileManager**
The `javax.tools.JavaFileManager` has been extended with two new methods, `getJavaFileForOutputForOriginatingFiles` and `getFileForOutputForOriginatingFiles`, that are used to create new files with specified originating files. The `Filer` uses these methods when creating new files (with `Filer.createSourceFile`, `Filer.createClassFile`, `Filer.createResource`) in order to pass along the files containing the *originating elements*.

See JDK-8272234

core-libs/java.nio.charsets
➜ **Charset.forName() Taking fallback Default Value**
A new method `Charset.forName()` that takes `fallback` charset object has been introduced in `java.nio.charset` package. The new method returns `fallback` if `charsetName` is illegal or the charset object is not available for the name. If `fallback` is passed as `null` the caller can check if the named charset was available without having to catch the exceptions thrown by the `Charset.forName(name)` method.

See JDK-8270490

core-libs/java.util
➜ **New System Property to Control the Default Date Comment Written Out by java.util.Properties::store Methods**
A new system property, `java.properties.date`, has been introduced to allow applications to control the default date comment written out by the `java.util.Properties::store` methods. This system property is expected to be set while launching `java`. Any non-empty value of this system property results in that value being used as a comment instead of the default date comment. In the absence of this system property or when the value is empty, the `Properties::store` methods continue to write the default date comment. An additional change has also been made in the implementation of the `Properties::store` methods to write out the key/value property pairs in a deterministic order. The implementation of these methods now uses the natural sort order of the property keys while writing them out. However, the implementation of these methods continues to use the iteration order when any sub-class of `java.util.Properties` overrides the `entrySet()` method to return a different `Set` than that returned by `super.entrySet()`.

The combination of the deterministic ordering of the properties that get written out with the new system property is particularly useful in environments where applications require the contents of the stored properties to be reproducible. In such cases, applications are expected to provide a fixed value of their choice to this system property.

See JDK-8231640

core-libs/javax.annotation.processing
➜ **printError, printWarning, and printNote methods on Messager**
For annotation processors, the `Messager` interface now has methods `printError`, `printWarning`, and `printNote` to directly report errors, warnings, and notes, respectively.

See JDK-8273157

core-libs/javax.lang.model
➜ **Method to Get Outermost Type Element**

In the `javax.lang.model` API, the `Elements` utility interface has a new method, `getOutermostTypeElement`, which returns the outermost class or interface syntactically enclosing an element.

See JDK-8140442

core-libs/javax.lang.model

➜ **Map from an Element to its JavaFileObject**

The method `Elements.getFileObjectOf(Element)` maps from an `Element` to the file object used to create the element.

See JDK-8224922

core-libs/javax.lang.model

➜ **Mapping between SourceVersion and Runtime.Version**

The `SourceVersion` enum class has methods to map between `SourceVersion` values and `Runtime.Version` values. In turn, `Runtime.Version` can be used to map from a string representation of a version to a `Runtime.Version` object.

See JDK-8275308

hotspot/compiler

➜ **Improve Compilation Replay**

Compilation Replay has been improved to make it more stable and catch up with new JVM features. Enhancements include:

- Best-effort support for hidden classes, allowing more cases involving invokedynamic, MethodHandles, and lambdas to replay successfully
- DumpReplay support for C1
- Incremental inlining support
- Numerous bug fixes to improve stability

Compilation Replay is a JVM feature of debug builds that is mainly used to reproduce crashes in the C1 or C2 JIT compilers.

See JDK-8254106

hotspot/gc

➜ **Configurable Card Table Card Size**

JDK-8272773 introduces the VM option `-XX:GCCardSizeInBytes` used to set a size of the area that a card table entry covers (the "card size") that is different from the previous fixed value of 512 bytes. Permissible values are now 128, 256, and 512 bytes for all platforms, and 1024 bytes for 64 bit platforms only. The default value remains 512 bytes.

The card size impacts the amount of work to be done when searching for references into an area that is to be evacuated (for example, young generation) during garbage collection. Smaller card sizes give more precise information about the location of these references, often leading to less work during garbage collection. At the same time, however, smaller card sizes can lead to more memory usage in storing this information. The increase in memory usage might result in slower performance of maintenance work during the garbage collection.

See JDK-8272773

hotspot/gc

➜ **Allow G1 Heap Regions up to 512MB**

The JDK-8275056 enhancement extends the maximum allowed heap region size from 32MB to 512MB for the G1 garbage collector. Default ergonomic heap region size selection is still limited to 32MB regions maximum. Heap region sizes beyond that must be selected manually by using the `-XX:G1HeapRegionSize` command line option.

This can be used to mitigate both inner and outer fragmentation issues with large objects on large heaps.

On very large heaps, using a larger heap region size may also decrease internal region management overhead and increase performance due to larger local allocation buffers.

See JDK-8275056

hotspot/jfr

➜ **JDK Flight Recorder Event for Finalization**

A new JDK Flight Recorder Event, `jdk.FinalizerStatistics`, identifies classes at runtime that use finalizers. The event is enabled by default in the JDK (in the `default.jfc` and `profile.jfc` JFR configuration files). When enabled, JFR will emit a `jdk.FinalizerStatistics` event for each instantiated class with a non-empty finalize() method. The event includes: the class that overrides `finalize()`, that class's `CodeSource`, the number of times the class's finalizer has run, and the number of objects still on the heap (not yet finalized). For information about using JFR, see the User Guide.

If finalization has been disabled with the `--finalization=disabled` option, no `jdk.FinalizerStatistics` events are emitted.

See JDK-8266936

security-libs

➜ **SunPKCS11 Provider Now Supports Some PKCS#11 v3.0 APIs**

PKCS#11 v3.0 adds several new APIs that support new function entry points, as well as message-based encryption for AEAD ciphers, etc. For JDK 18, the SunPKCS11 provider has been updated to support some of the new PKCS#11 v3.0 APIs. To be more specific, if the "functionList" attribute in the provider configuration file is not set, the SunPKCS11 provider will first try to locate the new PKCS#11 v3.0 C_GetInterface() method before falling back to the C_GetFunctionList() method to load the function pointers of the native PKCS#11 library. If the loaded PKCS#11 library is v3.0, then the SunPKCS11 provider will cancel crypto operations by trying the new PKCS#11 v3.0 C_SessionCancel() method instead of finishing off remaining operations and discarding the results. Support for other new PKCS#11 v3.0 APIs will be added in later releases.

See JDK-8255409

security-libs/java.security

➜ **Alternate Subject.getSubject and doAs APIs Created That Do Not Depend on Security Manager APIs**

New methods `javax.security.auth.Subject::current` and `javax.security.auth.Subject::callAs` have been created as replacements for existing methods `javax.security.auth.Subject::getSubject` and `javax.security.auth.Subject::doAs`. The `javax.security.auth.Subject::getSubject` and `javax.security.auth.Subject::doAs` methods are deprecated for removal because they depend on Security Manager APIs deprecated in JEP 411.

See JDK-8267108

security-libs/java.security

➜ **KeyStore Has a getAttributes Method**

A `KeyStore::getAttributes` method has been added that returns the attributes of an entry without having to retrieve the entry first. This is especially useful for a private key entry that has attributes that are not protected but previously could only be retrieved with a password.

See JDK-8225181

security-libs/java.security

➜ **Support for RSASSA-PSS in OCSP Response**

An OCSP response signed with the RSASSA-PSS algorithm is now supported.

See JDK-8274471

security-libs/java.security

➜ **New Option -version Added to keytool and jarsigner Commands**

A new `-version` option has been added to the `keytool` and `jarsigner` commands. This option is used to print the program version of `keytool` and `jarsigner`.

See JDK-8272163

security-libs/java.security

➜ **Migrate cacerts From JKS to Password-Less PKCS12**

The `cacerts` keystore file is now a password-less PKCS #12 file. All certificates inside are not encrypted and there is no MacData for password integrity. Since the PKCS12 and JKS keystore types are interoperable, existing code that uses a JKS `KeyStore` to load the `cacerts` file with any password (including null) continue to behave as expected and can view or extract the certificates contained within the keystore.

See JDK-8275252

security-libs/java.security

➜ **Allow Store Password to Be Null When Saving a PKCS12 KeyStore**

Calling `keyStore.store(outputStream, null)` on a PKCS12 KeyStore creates a password-less PKCS12 file. The certificates inside the file are not encrypted and the file contains no MacData. This file can then be loaded with any password (including `null`) when calling `keyStore.load(inputStream, password)`.

See JDK-8231107

security-libs/javax.crypto:pkcs11

➜ **SunPKCS11 Provider Supports AES Cipher With KW and KWP Modes if Supported by PKCS11 Library**

The SunPKCS11 provider has been enhanced to support the following crypto services and algorithms when the underlying PKCS11 library supports the corresponding PKCS#11 mechanisms:

AES/KW/NoPadding Cipher <=> CKM_AES_KEY_WRAP mechanism
AES/KW/PKCS5Padding Cipher <=> CKM_AES_KEY_WRAP_PAD mechanism
AES/KWP/NoPadding Cipher <=> CKM_AES_KEY_WRAP_KWP mechanism

See JDK-8264849

tools/javac

➜ **Expand checks of javac's serial lint warning**

The `serial` lint warning implemented in `javac` traditionally checked that a `Serializable` class declared a `serialVersionUID` field. The structural checking done by the `serial` lint warning has been expanded to warn for cases where declarations would cause the runtime serialization mechanism to silently ignore a mis-declared entity, rendering it ineffectual. Also, the checks include several compile-time patterns that could lead to runtime failures. The specific checks include:

- For `Serializable` classes and interfaces, fields and methods with names matching the special fields and methods used by the serialization mechanism are declared properly.
- Additional analagous checks are done for `Externalizable` types as some serialization-related methods are ineffectual there.
- A serializable class is checked that it has access to a no-arg constructor in the first non-serializable class up its superclass chain.
- For enum classes, since the serialization spec states the five serialization-related methods and two fields are all ignored, the presence of any of those in an enum class will generate a warning.
- For record classes, warnings are generated for the ineffectual declarations of serialization-related methods that serialization would ignore.
- For interfaces, if a public `readObject`, `readObjectNoData`, or `writeObject` method is defined, a warning is issued since a class implementing the interface will be unable to declare `private` versions of those methods and the methods must be `private` to be effective. If an interface defines default `writeReplace` or `readResolve` methods, a warning will be issued since serialization only looks up the super*class* chain for those methods and not for default methods from interfaces. Since a `serialPersistentFields` field must be `private` to be effective, the presence of such a (non-private) field an in interface generates a warning.
- For serializable classes without `serialPersistentFields`, the type of each non-transient instance field is examined and a warning issued if the type of the field cannot be serialized. Primitive types, types declared to be serializable, and arrays can be serialized. While by the JLS all arrays are considered serializable, a warning is issued if the innermost component type is not serializable.

The new warnings can be suppressed using `@SuppressWarnings("serial")`.

See JDK-8202056

tools/javadoc(tool)

➜ **Options to Include Script Files in Generated Documentation**

The Standard Doclet supports an `--add-script` option used to include a reference to an external script file in the file for each page of generated documentation.

See JDK-8275786

tools/javadoc(tool)

➜ **@SuppressWarnings for DocLint Messages**

You can now use `@SuppressWarnings` annotations to suppress messages from DocLint about issues in documentation comments, when it is not possible or practical to fix the issues that were found. For more details, see Suppressing Messages in the DocLint section of the javadoc Tool Guide.

See JDK-8189591

TOP

Removed Features and Options

This section describes the APIs, features, and options that were removed in Java SE 18 and JDK 18. The APIs described here are those that are provided with the Oracle JDK. It includes a complete implementation of the Java SE 18 Platform and additional Java APIs to support developing, debugging, and monitoring Java applications. Another source of information about important enhancements and new features in Java SE 18 and JDK 18 is the Java SE 18 (JSR 393) Platform Specification, which documents changes to the specification made between Java SE 17 and Java SE 18. This document includes the identification of removed APIs and features not described here. The descriptions below might also identify potential compatibility issues that you could encounter when migrating to JDK 18. See CSRs Approved for JDK 18 for the list of CSRs closed in JDK 18.

security-libs/java.security

➜ **Removal of IdenTrust Root Certificate**

The following root certificate from IdenTrust has been removed from the `cacerts` keystore:

```
+ alias name "identrustdstx3 [jdk]"

  Distinguished Name: CN=DST Root CA X3, O=Digital Signature Trust Co.
```

See JDK-8225082

security-libs/java.security

➜ **Removal of Google's GlobalSign Root Certificate**

The following root certificate from Google has been removed from the `cacerts` keystore:

```
+ alias name "globalsignr2ca [jdk]"

  Distinguished Name: CN=GlobalSign, O=GlobalSign, OU=GlobalSign Root CA - R2
```

See JDK-8225083

client-libs/2d

➜ **Removal of Empty finalize() Methods in java.desktop Module**

The `java.desktop` module had a few implementations of `finalize()` that did nothing. These methods were deprecated in Java 9 and terminally deprecated in Java 16. These methods have been removed in this release. See https://bugs.openjdk.java.net/browse/JDK-8273103 for details.

See JDK-8273102

core-libs/java.net

➜ **Removal of Support for Pre JDK 1.4 DatagramSocketImpl Implementations**

Support for pre JDK 1.4 `DatagramSocketImpl` implementations (DatagramSocketImpl implementations that don't support connected datagram sockets, peeking, or joining multicast groups on specific network interface) has been dropped in this release. Old implementations have not have been buildable since JDK 1.4 but implementations compiled with JDK 1.3 or older continued to be usable up to this release.

Trying to call `connect` or `disconnect` on a `DatagramSocket` or `MulticastSocket` that uses an old implementation will now throw `SocketException` or `UncheckedIOException`. Trying to call `joinGroup` or `leaveGroup` will result in an `AbstractMethodError`.

See JDK-8260428

core-libs/java.net

➜ **Removal of impl.prefix JDK System Property Usage From InetAddress**

The system property `impl.prefix` has been removed. This undocumented system property dates from early JDK releases where it was possible to add an implementation of the JDK internal and non-public `java.net.InetAddressImpl` interface to the `java.net` package, and have it be used by the `java.net.InetAddress` API.
The `InetAddressResolver` SPI introduced by JEP 418 provides a standard way to implement a name and address resolver.

See JDK-8274227

core-libs/java.net

➜ **Removal of Legacy PlainSocketImpl and PlainDatagramSocketImpl Implementations**

Legacy implementations of `java.net.SocketImpl` and `java.net.DatagramSocketImpl` have been removed from the JDK. The legacy implementation of `SocketImpl` has not been used by default since JDK 13, while the legacy implementation of `DatagramSocketImpl` has not been used by default since JDK 15. Support for system properties `jdk.net.usePlainSocketImpl` and `jdk.net.usePlainDatagramSocketImpl` used to select these implementations has also been removed. Setting these properties now has no effect.

See JDK-8253119

security-libs/org.ietf.jgss:krb5

➜ **Removal of default_checksum and safe_checksum_type From krb5.conf**

The "default_checksum" and "safe_checksum_type" settings in the `krb5.conf` configuration file are no longer supported. The checksum type in a KRB_TGS_REQ message is derived from the type of the encryption key used to generate it. The "safe_checksum_type" setting was never used in Java.

See JDK-8274656

TOP

Deprecated Features and Options

Additional sources of information about the APIs, features, and options deprecated in Java SE 18 and JDK 18 include:

- The Deprecated API page identifies all deprecated APIs including those deprecated in Java SE 18.
- The Java SE 18 (JSR 393) specification documents changes to the specification made between Java SE 17 and Java SE 18 that include the identification of deprecated APIs and features not described here.
- JEP 277: Enhanced Deprecation provides a detailed description of the deprecation policy. You should be aware of the updated policy described in this document.

You should be aware of the contents in those documents as well as the items described in this release notes page.

The descriptions of deprecated APIs might include references to the deprecation warnings of `forRemoval=true` and `forRemoval=false`. The `forRemoval=true` text indicates that a deprecated API might be removed from the next major release. The `forRemoval=false` text indicates that a deprecated API is not expected to be removed from the next major release but might be removed in some later release.

The descriptions below also identify potential compatibility issues that you might encounter when migrating to JDK 18. See CSRs Approved for JDK 18 for the list of CSRs closed in JDK 18.

NOTE: Release Notes for JEPs that deprecated or removed APIs, features, and options in this release are grouped in the following category:

JEPs for Deprecation and Removals:

core-libs/java.lang
➜ **JEP 421: Deprecated 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, but it can be disabled for testing purposes by using the command-line option `--finalization=disabled` introduced in this release. Maintainers of libraries and applications that rely upon finalization should migrate to other resource management techniques in their code, such as try-with-resources and cleaners.

For further details, see JEP 421.

See JDK-8274609

NOTE: The following Release Notes, not related to JEPs, describe deprecated or removed APIs, features, and options in this release:

security-libs/javax.security
➜ **Deprecated Subject::doAs for Removal**
Two `javax.security.auth.Subject::doAs` methods have been deprecated for removal. This is a part of the ongoing effort to remove Security Manager related APIs.

See JDK-8267108

core-libs
➜ **Deprecated sun.misc.Unsafe Methods That Return Offsets**
Developers that use the unsupported `sun.misc.Unsafe` API should be aware that the methods that return base and field offsets have been deprecated in this release. The methods `objectFieldOffset`, `staticFieldOffset`, and `staticFieldBase` methods are an impediment to future changes. A future release will eventually degrade or remove these methods along with the heap accessor methods.

The `java.lang.invoke.VarHandle` API (added in Java 9) provides a strongly typed reference to a variable that is safe and a replacement to many cases that use field offsets and the heap accessor methods. It is strongly recommended to migrate to the

`VarHandle` API where possible. Multi-Release JARs can be used by libraries and frameworks that continue to support JDK 8 or older.

See JDK-8277863

core-libs/java.lang

➜ **Terminally Deprecated Thread.stop**

`Thread.stop` is terminally deprecated in this release so that it can be degraded in a future release and eventually removed. The method is inherently unsafe and has been deprecated since Java 1.2 (1998).

See JDK-8277861

hotspot/gc

➜ **Obsoleted Product Options -XX:G1RSetRegionEntries and -XX:G1RSetSparseRegionEntries**

The options `-XX:G1RSetRegionEntries` and `-XX:G1RSetSparseRegionEntries` have been obsoleted with the changes from JDK-8017163.

JDK-8017163 implements a completely new remembered set implementation in which these two options no longer apply. In this release, neither `-XX:G1RSetRegionEntries` nor `-XX:G1RSetSparseRegionEntries` have a function, and their use will trigger an obsoletion warning.

See JDK-8017163

security-libs/javax.security

➜ **Obsoleted Biased-Locking**

The VM option `UseBiasedLocking` along with the VM options `BiasedLockingStartupDelay`, `BiasedLockingBulkRebiasThreshold`, `BiasedLockingBulkRevokeThreshold`, `BiasedLockingDecayTime` and `UseOptoBiasInlining` have been obsoleted. Use of these options will produce an obsolete option warning but will otherwise be ignored. Biased locking has been disabled by default and deprecated since JDK 15.

See JDK-8256425

TOP

Other Notes

The following notes describe additional changes and information about this release. In some cases, the following descriptions provide links to additional detailed information about an issue or a change.

install/install

➜ **Extended Delay Before JDK Executable Installer Starts From Network Drive**

On Windows 11 and Windows Server 2022, there can be some slowness with the extraction of temporary installation files when launched from a mapped network drive. The installer will still work, but there can be a temporary delay.

JDK-8274002 (not public)

security-libs/java.security

➜ **Change the java.security.manager System Property Default Value to disallow**

The default value of the `java.security.manager` system property has been changed to `disallow`. Unless the system property is set to `allow` on the command line, any invocation of `System.setSecurityManager(SecurityManager)` with a non-null argument will throw an `UnsupportedOperationException`.

See JDK-8270380

security-libs/java.security

➜ **Disabled SHA-1 Signed JARs**

JARs signed with SHA-1 algorithms are now restricted by default and treated as if they were unsigned. This applies to the algorithms used to digest, sign, and optionally timestamp the JAR. It also applies to the signature and digest algorithms of the certificates in the certificate chain of the code signer and the Timestamp Authority, and any CRLs or OCSP responses that are used to verify if those certificates have been revoked.

To reduce the compatibility risk for applications that have been previously timestamped, there is one exception to this policy:

- Any JAR signed with SHA-1 algorithms and timestamped prior to January 01, 2019 will not be restricted.

This exception might be removed in a future JDK release.

Users can, at their own risk, remove these restrictions by modifying the `java.security` configuration file (or override it by using the `java.security.properties` system property) and removing "SHA1 usage SignedJAR & denyAfter 2019-01-01" from the `jdk.certpath.disabledAlgorithms` security property and "SHA1 denyAfter 2019-01-01" from the `jdk.jar.disabledAlgorithms` security property.

See JDK-8269039

tools/javac

➜ **Enclosing Instance Fields Omitted from Inner Classes That Don't Use Them**

Prior to JDK 18, when javac compiles an inner class it always generates a private synthetic field with a name starting with `this$` to hold a reference to the enclosing instance, even if the inner class does not reference its enclosing instance and the field is unused.

Starting in JDK 18, unused `this$` fields are omitted; the field is only generated for inner classes that reference their enclosing instance.

For example, consider the following program:

```
class T {

    class I {
    }
}
```

Prior to JDK 18, the program would be translated as:

```
class T {

    class I {
        private synthetic T this$0;
        I(T this$0) {
            this.this$0 = this$0;
        }
    }
}
```

Starting in JDK 18, the unused `this$0` field is omitted:

```
class T {

    class I {
        I(T this$0) {}
    }
}
```

Note that the form of the inner class constructor is not affected by this change.

The change may cause enclosing instances to be garbage collected sooner, if previously they were only reachable from a reference in an inner class. This is typically desirable, since it avoids a source of potential memory leaks when creating inner classes that are intended to outlive their enclosing instance.

Subclasses of `java.io.Serializable` are not affected by this change.

See JDK-8271623

tools/javac

➜ **Corrected References to Overloaded Methods in Javadoc Documentation**

For a reference to a specific overload of an overloaded method, the javadoc tool might have linked to the wrong overload in the generated documentation. This fix resolves that issue, and the specified overload will now be used for the links in the generated documentation.

See JDK-8278373

core-libs/java.io

➜ **Default charset for PrintWriter That Wraps PrintStream**

The `java.io.PrintStream`, `PrintWriter`, and `OutputStreamWriter` constructors that take a `java.io.OutputStream` and no charset now inherit the charset when the output stream is a `PrintStream`. This is important for usages such as:

```
new PrintWriter(System.out)
```

where it would be problematic if `PrintStream` didn't use the same charset as that used by `System.out`. This change was needed because JEP 400 makes it is possible, especially on Windows, that the encoding of `System.out` is not UTF-8. This

would cause problems if `PrintStream` were wrapped with a `PrintWriter` that used UTF-8.

As part of this change, `java.io.PrintStream` now defines a `charset()` method to return the print stream's charset.

See JDK-8276970

core-libs/java.io:serialization

➜ **ObjectInputStream.GetField.get(name, object) Throws ClassNotFoundException**

The `java.io.ObjectInputStream.GetField.get(String name, Object val)` method now throws `ClassNotFoundException` when the class of the object is not found. Previously, null was returned, which prevented the caller from correctly handling the case where the class was not found. The signature of `GetField.get(name, val)` has been updated to throw `ClassNotFoundException`and a `ClassNotFoundException` exception is thrown when the class is not found.

The source compatibility risk is low. The addition of a throws `ClassNotFoundException` should not cause a source compatibility or a compilation warning. The `GetField` object and its methods are called within the context of the `readObject` method and include `throws ClassNotFoundException`.

To revert to the old behavior, a system property, `jdk.serialGetFieldCnfeReturnsNull`, has been added to the implementation. Setting the value to `true` reverts to the old behavior (returning null); leaving it unset or to any other value results in the throwing of `ClassNotFoundException`.

See JDK-8276665

core-libs/java.io:serialization

➜ **Deserialization Filter and Filter Factory Property Error Reporting Were Under Specified**

Invalid values of the command line and the security properties of `jdk.serialFilter` and `jdk.serialFilterFactory` are reported by throwing `java.lang.IllegalStateException` on the first use. The property values are checked when constructing `java.io.ObjectInputStream` or when calling the methods of `java.io.ObjectInputFilter.Config` including `getSerialFilter()` and `getSerialFilterFactory()`. The `IllegalStateException` indicates that the serial filter or serial filter factory is invalid and cannot be used; deserialization is disabled. Previously, the exception thrown was `ExceptionInInitializerError`.

See JDK-8278087

core-libs/javax.naming

➜ **System Property to Control Reconstruction of Reference Address Objects by JDK's Built-in JNDI LDAP Implementation**

The scope of the `com.sun.jndi.ldap.object.trustSerialData` system property has been extended to control the deserialization of java objects from the `javaReferenceAddress` LDAP attribute. This system property now controls the deserialization of java objects from the `javaSerializedData` and `javaReferenceAddress` LDAP attributes.

To prevent deserialization of java objects from these attributes, the system property can be set to `false`. By default, the deserialization of java objects from `javaSerializedData` and `javaReferenceAddress` attributes is allowed.

JDK-8267712 (not public)

core-libs/java.net

➜ **HttpURLConnection's getHeaderFields and URLConnection.getRequestProperties Methods Return Field Values in the Order They Were Added**

`URLConnection` has been fixed to return multiple header values for a given field-name in the order in which they were added.

Previously, if a `URLConnection` contained multiple header values for a given header field-name, when retrieved by using the `HttpURLConnection::getHeaderFields` and the `URLConnection::getRequestProperties` methods, they would be returned in the reverse order to which they were added.

This has been fixed to conform to RFC2616, which explicitly states that the order matters and thus, should be maintained.

See JDK-8133686

core-libs/java.net

➜ **Prohibit Null for Header Keys and Values in com.sun.net.httpserver.Headers**

In JDK 18, the handling of header names and values in `jdk.httpserver/com.sun.net.httpserver.Headers` has been reconciled. This includes the eager and consistent prohibition of `null` for names and values. The class represents header names and values as a key-value mapping of `Map<String, List <String>>`. Previously, it was possible to create a headers instance with a `null` key or value, which would cause undocumented exceptions when passed to the `HttpServer`. It was also possible to query the instance for a `null` key and `false` would be returned. With this change, all methods of the class now throw a

`NullPointerException` if the key or value arguments are `null`. For more information, see https://bugs.openjdk.java.net/browse/JDK-8269296.

See JDK-8268960

core-libs/java.nio

➜ **ReadableByteChannel::read No Longer Throws ReadOnlyBufferException**

`ReadableByteChannel::read` no longer incorrectly throws `ReadOnlyBufferException`.

The `read(ByteBuffer)` method of the `ReadableByteChannel` returned by `java.nio.channels.Channel.newChannel(InputStream)` was incorrectly throwing `ReadOnlyBufferException` when its `ByteBuffer` parameter specified a read-only buffer. `ReadableByteChannel::read` is, however, specified in this case to throw an `IllegalArgumentException`. The implementation has been changed for this case to throw an `IllegalArgumentException` instead of a `ReadOnlyBufferException`, as dictated by the specification.

See JDK-8275149

core-libs/java.nio

➜ **Zip File System Provider Throws ZipException When Entry Name Element Contains "." or ".."**

The ZIP file system provider has been changed to reject existing ZIP files that contain entries with "." or ".." in name elements. ZIP files with these entries cannot be used as a file system. Invoking the `java.nio.file.FileSystems.newFileSystem(...)` methods throw `ZipException` if the ZIP file contains these entries.

See JDK-8251329

core-libs/java.time

➜ **Update Timezone Data to 2021c**

The IANA Time Zone Database, on which the JDK's Date/Time libraries are based, has made a tweak to some of the time zone rules in 2021c.

Note that in 2021b, which is cumulatively included in this change, some of the time zone rules prior to the year 1970 have been modified according to changes introduced with 2021b. For more details, refer to the announcement of 2021b.

See JDK-8274407

core-libs/java.util.jar

➜ **Disabled JAR Index Support**

JAR Index has been disabled in this release. JAR Index was an optimization to postpone downloading of JAR files when loading applets or other classes over the network. JAR Index has many long standing issues and does not interact well with newer features (such as, Multi-Release JARs and modular JARs). If a JAR file contains an INDEX.LIST file, then its contents are ignored by the application class loader and by any URLClassLoader created by user code.

The system property, `jdk.net.URLClassPath.enableJarIndex`, can be used re-enable the feature if required. If set to an empty string or the value "true", then JAR Index will be re-enabled. This system property is temporary. A future release will remove the JAR Index feature and the system property.

The change does not impact the `jar -i` option. The `jar` tool continues to create an index when this option is used.

See JDK-8273401

core-libs/java.util.jar

➜ **Closes Out Compressor When IOException Encountered While Using Default JDK Compressor in GZIPOutputStream.finish() , ZipOutputStream.closeEntry() and DeflaterOutputStream.close()**

`DeflaterOutputStream.close()` and `GZIPOutputStream.finish()` has been changed to close out the associated default JDK compressor before propagating a Throwable up the stack. `ZIPOutputStream.closeEntry()` has been changed to close out the associated default JDK compressor before propagating an IOException, not of type ZipException, up the stack.

See JDK-8193682

hotspot/gc

➜ **ZGC: Fixed Long Process Non-Strong References Times**

A bug has been fixed that could cause long "Concurrent Process Non-Strong References" times with ZGC. The bug blocked the GC from making significant progress, and caused both latency and throughput issues for the Java application.

The long times could be seen in the GC logs when running with `-Xlog:gc*`:

```
[17606.140s][info][gc,phases ] GC(719) Concurrent Process Non-Strong References 25781.928ms
```

See JDK-8277212

security-libs/java.security

➜ **X509Certificate.get{Subject,Issuer}AlternativeNames and getExtendedKeyUsage Do Not Throw CertificateParsingException if Extension Is Unparseable**

The JDK implementation (as supplied by the SUN provider) of `X509Certificate::getSubjectAlternativeNames`, `X509Certificate::getIssuerAlternativeNames` and `X509Certificate::getExtendedKeyUsage` now throws `CertificateParsingException` instead of returning `null` when the extension is non-critical and unparseable (badly encoded or contains invalid values). This change in behavior is compliant with the specification of these methods.

See JDK-8251468

security-libs/javax.crypto

➜ **Fix Issues With the KW and KWP Modes of SunJCE Provider**

Support for AES/KW/NoPadding, AES/KW/PKCS5Padding and AES/KWP/NoPadding ciphers is added to SunJCE provider since jdk 17. The cipher block size for these transformations should be 8 instead of 16. In addition, for KWP mode, only the default IV, i.e. 0xA65959A6, is allowed to ensure maximum interoperability with other implementations. Other IV values will be rejected with exception during Cipher.init(...) calls.

See JDK-8271745

security-libs/javax.net.ssl

➜ **Call X509KeyManager.chooseClientAlias Once for All Key Types**

The (D)TLS implementation in JDK now calls `X509KeyManager.chooseClientAlias()` only once during handshaking for client authentication, even if there are multiple algorithms requested .

See JDK-8262186

security-libs/org.ietf.jgss:krb5

➜ **Removed Weak etypes From Default krb5 etype List**

Weak encryption types based on DES, 3DES, and RC4 have been removed from the default encryption types list in Kerberos. These weak encryption types can be enabled (at your own risk) by adding them to the "permitted_enctypes" property (or alternatively, the "default_tkt_enctypes" or "default_tgt_enctypes" properties) and setting "allow_weak_crypto" to "true" in the `krb5.conf` configuration file.

See JDK-8273670

tools/javac

➜ **Index Noteworthy Terms for jdk.compiler and jdk.javadoc Modules**

Various terms have been added to the index files in the JDK API documentation, so that these terms can be found in both the static A-Z index and in the interactive search index.

The terms are:
Language Model, *Annotation Processing*, *Compiler API*, and *Compiler Tree API*.

See JDK-8278318

tools/javadoc(tool)

➜ **Improved Navigation on Small Devices**

The pages generated by the Standard Doclet provide improved navigation controls when the pages are viewed on small devices.

See JDK-8273034

tools/javadoc(tool)

➜ **DocLint Reports Missing Descriptions**

DocLint detects and reports documentation comments that do not have a description about the associate declaration before any block tags that may be present. DocLint is a feature of the `javac` and `javadoc` tools that detects and reports issues in documentation comments.

See JDK-8272374

tools/javadoc(tool)

➜ **Updated Documentation for DocLint**

The documentation in the Tool Guides ("man pages") for the `javac` and `javadoc` tools has been reorganized and updated. The primary documentation is now in the Tool Guide for `javadoc`, with information about the tool-specific command-line options being given in the `Options` section of the appropriate Tool Guide.

JDK-8275146 (not public)

tools/javadoc(tool)

➜ **Indicate Invalid Input in Generated Output**

When the Standard Doclet encounters content in a documentation comment that it cannot process, it may create an element in the generated output to indicate clearly to the reader that the output at that position is not as the author intended. (This replaces the earlier behavior to show either nothing or the invalid content.) The element will contain a short summary message and may contain additional details. This is in addition to the existing behavior to report diagnostics and to return a suitable exit code.

See JDK-8276964

tools/javadoc(tool)

➜ **Merge "Exceptions" and "Errors" into "Exception Classes"**

The "Exceptions" and "Errors" tabs in documentation generated by JavaDoc have been merged into a single "Exception Classes" tab, which includes all exception classes, as defined in JLS 11.1.1.

See JDK-8269401

core-libs/java.io

➜ **file.encoding System Property Has an Incorrect Value on Windows**

The initialization of the `file.encoding` system property on non macOS platforms has been reverted to align with the behavior on or before JDK 11. This has been an issue especially on Windows where the system and user's locales are not the same.

See JDK-8275145

hotspot/runtime

➜ **OperatingSystemMXBean.getProcessCpuLoad Is Now Container Aware**

For JVMs running in a container, `OperatingSystemMXBean.getProcessCpuLoad` now considers only the CPU resources available to the container when calculating CPU load. Prior to this change, the calculation included all CPUs on a host. After this change, management agents may report higher CPU usage by JVMs in containers that are constrained to a limited set of CPUs.

See JDK-8269851