OpenJDK 8

Migrating OpenJDK 8 to OpenJDK 11

Last Updated: 2022-04-06

Legal Notice

Abstract

The Migrating OpenJDK 8 to OpenJDK 11 guide provides information on how to upgrade your OpenJDK 8 application to OpenJDK 11.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. To provide feedback, you can highlight the text in a document and add comments.

This section explains how to submit feedback.

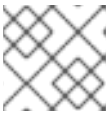
Prerequisites

- You are logged in to the Red Hat Customer Portal.
- In the Red Hat Customer Portal, view the document in **Multi-page HTML** format.

Procedure

To provide your feedback, perform the following steps:

1. Click the **Feedback** button in the top-right corner of the document to see existing feedback.



NOTE

The feedback feature is enabled only in the **Multi-page HTML** format.

2. Highlight the section of the document where you want to provide feedback.
3. Click the **Add Feedback** pop-up that appears near the highlighted text.
   A text box appears in the feedback section on the right side of the page.
4. Enter your feedback in the text box and click **Submit**.
   A documentation issue is created.
5. To view the issue, click the issue tracker link in the feedback view.

CHAPTER 1. MIGRATING OPENJDK 8 TO OPENJDK 11 OVERVIEW

The *Migrating OpenJDK 8 to OpenJDK 11* guide describes changes in the OpenJDK 11 release, including new features and deprecated or removed APIs, that might impact your migration from OpenJDK 8. You can use the information in the guide to upgrade your Java applications in OpenJDK 8 to OpenJDK 11.

The OpenJDK project is known for its conservative approach to providing updates and for providing backward compatibility. However, to guarantee the evolution, security and stability of the project, the OpenJDK project might sometimes introduces a few incompatibilities across major releases of OpenJDK. These incompatibilities are relevant for the following scenarios:

- When you use APIs that are considered obsolete or unsecure.
- When you access internals of the project that are considered implementation details and not public or supported API details.

1.1. ABOUT THE RED HAT BUILD OF OPENJDK 8U AND OPENJDK 11U

OpenJDK is the free and open source reference implementation of the Java Platform, Standard Edition (Java SE). The Red Hat builds of OpenJDK are based on the upstream OpenJDK 8u, OpenJDK 11u, and 17u OpenJDK projects. The Shenandoah Garbage Collector is included in OpenJDK 8, OpenJDK 11, and OpenJDK 17 versions.

The Red Hat builds of OpenJDK provide you with the following benefits:

- **Multi-platform** - The Red Hat build of OpenJDK is now supported on RHEL and Microsoft Windows, so you can standardize applications on a single Java platform across desktop, datacenter, and hybrid cloud environments.
- **Frequent releases** - Red Hat delivers quarterly updates of JRE and JDK for the OpenJDK 8, OpenJDK 11, and OpenJDK 17 distributions. These updates are available as archive, RPM, and Windows MSI-based installer files and container images.
- **Long-term support** - Red Hat supports the recently released OpenJDK 8, OpenJDK 11, and OpenJDK 17 distributions.

Additional resources

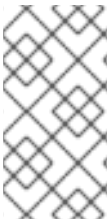
- For more information about the support lifecycle, see OpenJDK Life Cycle and Support Policy .

CHAPTER 2. MAJOR DIFFERENCES BETWEEN OPENJDK 8 AND OPENJDK 11

Before you migrate your Java applications from OpenJDK 8 to OpenJDK 11, you must understand the updates and changes in OpenJDK 11. These updates and changes might require you to configure OpenJDK 8 before you migrate your applications to OpenJDK 11.

One of the major differences between OpenJDK 8 and OpenJDK 11 is the inclusion of a module system in OpenJDK 11. If you want to migrate your OpenJDK 8 applications to OpenJDK 11, consider moving the application's libraries and modules from the OpenJDK 8 class path to the OpenJDK 11 module class. This change can improve the class-loading capabilities of your application.

OpenJDK 11 includes new features and enhancements that can improve the performance of your application, such as enhanced memory usage, improved startup speed, and increased container integration.



NOTE

Some features might differ between the Red Hat build of OpenJDK and other upstream or third-party versions of OpenJDK. For example, the Shenandoah garbage collector (GC) is available on all the Red Hat OpenJDK builds, but this feature might not be available by default in other builds of OpenJDK.

2.1. CRYPTOGRAPHY AND SECURITY

Certain minor cryptography and security differences exist between OpenJDK 8 and OpenJDK 11. However, both OpenJDK versions have many similar cryptography and security behaviors.

Red Hat builds of OpenJDK use system-wide certificates, and each build obtains its list of disabled cryptographic algorithms from a system's global configuration settings. These settings are common to all the Red Hat supported OpenJDK releases, so you can easily change from OpenJDK 8 to OpenJDK 11.

In FIPS mode, OpenJDK 8 and OpenJDK 11 releases are self-configured, so that either release uses the same security providers at startup.

The TLS stacks in OpenJDK 8 and OpenJDK 11 are similar, because the SunJSSE engine from OpenJDK 11 was backported to OpenJDK 8. Both OpenJDK versions support the TLS 1.3 protocol.

The following minor cryptography and security differences exist between OpenJDK 8 and OpenJDK 11:

- OpenJDK 8 disabled the client-side TLSv1.3 by default, so an OpenJDK 8 TLS client does not use the TLSv1.3 protocol during the communication process with the target server. You can use the **jdk.tls.client.protocols** system property to change this behavior by setting the **-Djdk.tls.client.protocols=TLSv1.3** property for OpenJDK 11.

- OpenJDK 11 supports the use of the **X25519** and **X448** EC curves in the Diffie-Hellman key exchange. This support is not available in OpenJDK 8.

- OpenJDK 11 does not support the legacy KRB5-based cipher suites. OpenJDK 8 still supports the legacy KRB5-based cipher suites, but you must enable these cipher suites by changing the **jdk.tls.client.cipherSuites** and **jdk.tls.server.cipherSuites** system properties.

- OpenJDK 11 supports the Datagram Transport Layer Security (DTLS) protocol, so TLS clients and servers can use the **max_fragment_length** extension for DTLS. OpenJDK 8 does not support this protocol.

2.2. GARBAGE COLLECTOR (GC)

OpenJDK 8 uses the Parallel GC as its default garbage collector, while OpenJDK 11 uses the G1 GC as its default garbage collector. Before you choose a garbage collector consider the following details:

- Use the Parallel GC if you want to improve throughput. The Parallel GC maximizes throughput, but at the expense of occasional pauses for **stop-the-world** collections.
- Use the G1 GC for a balance between throughput and latency. This GC can perform latencies with pause times in the **100** ms range. If you notice throughput issues when migrating applications from the OpenJDK 8 to OpenJDK 11, you can switch to the Parallel GC.
- Use the Shenandoah GC for low latency collection.

You can select the GC type by using the **-XX:+<gc_type>** JVM option, such as - **XX:+UseParallelGC** to switch to the Parallel GC.

2.3. GARBAGE COLLECTOR (GC) LOGGING OPTIONS

OpenJDK 11 includes a new logging framework that works more effectively when compared to the old logging framework. OpenJDK 11 also includes unified JVM logging options and unified GC logging options.

The logging system for OpenJDK 11 activates the -**XX:+PrintGCTimeStamps** and **-XX:+PrintGCDateStamps** options by default. Because the logging format in OpenJDK 11 is different from OpenJDK 8, you might need to update any of your code that parses GC logs.

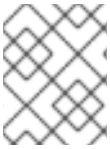
You can still access the old logging framework options in OpenJDK 11 through aliases of the new framework options. If you want to work more effectively with OpenJDK 11, use the new logging framework options.

OpenJDK 11 replaced or removed the following OpenJDK 8 options from the logging framework:

Options in OpenJDK 8	Options in OpenJDK 11
-verbose:gc	**-Xlog:gc**
-XX:+PrintGC	**-Xlog:gc**
-XX:+PrintGCDetails	**-Xlog:gc***
-Xloggc	**-Xlog:gc:file=<path_to_filename>**

When using the **-XX:+PrintGCDetails** option, pass the **-Xlog:gc*** flag.The * activates more detailed logging.

When using **-Xloggc**, append the **:file=<filename>** option to redirect log output to the specified file. For example **-Xlog:gc:file=<filename>**.



NOTE

If you specify an old tag option on the Java HotSpot VM, the VM prompts you with an available newer tag option. You can choose to use either the old or new tag option.

OpenJDK 11 does not include the following options. If you attempt to use any of the options in OpenJDK 11, you will receive startup errors.

- **-Xincgc**
- **-XX:+CMSIncrementalMode**
- **-XX:+UseCMSCompactAtFullCollection**
- **-XX:+CMSFullGCsBeforeCompaction**
- **-XX:+UseCMSCollectionPassing**

NOTE

If you want to use any of the previously listed options in OpenJDK 11, you can ignore any startup issues by issuing the **-XX:+IgnoreUnrecognizedVMOptions** option in your command-line interface.

Additional resources

- For more information about the common framework for unified JVM logging and the format of **Xlog** options, see JEP 158: Unified JVM Logging .
- For more details on removed options, see JEP 214: Remove GC Combinations Deprecated in JDK 8.
- For more information about unified GC logging, see JEP 271: Unified GC Logging .

2.4. OPENJDK GRAPHICS

OpenJDK 11 uses Marlin as its default rendering engine, as opposed to Pisces in OpenJDK 8. The Marlin rendering engine improves the handling of intensive application graphics.

The Marlin engine is available in OpenJDK 8 and you can enable the Marlin engine by issuing **-Dsun.java2d.renderer=sun.java2d.marlin.MarlinRenderingEngine** for the Java runtime.

2.5. WEBSTART AND APPLETS

You can use Java WebStart with OpenJDK 8 on RHEL 7, RHEL 8 and Microsoft Windows operating systems by using the IcedTea-Web plug-in. OpenJDK 11 and later versions of OpenJDK do not support Java Webstart.

Applets are not supported on OpenJDK.

2.6. JAVA LIBRARY CLASSES

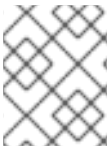
The Java Platform Module System (JPMS), which was introduced in OpenJDK 9, limits or prevents access to non-public APIs. JPMS also impacts how you can start and compile your Java application, such as using a class path or using a module path.

By default, OpenJDK 11 restricts access to JDK internal modules. As a workaround for this restriction when porting applications, you can provide access to an internal package for your application by passing options to the **java** command. These options are demonstrated in the following examples:

-

```
--add-opens <module-name>/<package-in-module>=ALL-UNNAMED

--add-opens java.base/jdk.internal.math=ALL-UNNAMED
```



NOTE

The previous workaround will not work indefinitely, because of the update behavior of OpenJDK release cycles. A future release of OpenJDK will close this workaround route.

Additionally, you can check illegal access cases by passing the **--illegal-access=warn** option to the **java** command. This option changes the default behavior of the OpenJDK.

The JPMS refactoring changes the **ClassLoader** hierarchy in OpenJDK 11. OpenJDK 11 sets the system class loader to internal classes, so Java programs cannot access the system class loader.

For example, existing OpenJDK 8 application code that invokes **ClassLoader::getSystemClassLoader** and casts the result to a **URLClassLoader** on OpenJDK 11 might not work, because **URLClassLoader** is not part of the system class loader. If you attempt to cast a result to a **URLClassLoader** on OpenJDK 11, you might receive a runtime exception message.

OpenJDK 11 includes a new class loader that can control the loading of certain classes. This improves the way that a class loader can locate all of its required classes.

When you create a class loader, you can no longer pass **null** as its parent. If you do this in OpenJDK 11, no class loader is selected as its parent, and your new class loader fails to locate platform classes. You can retrieve an instance of the platform class loader by using the **ClassLoader.getPlatformClassLoader()** API.

Additional resources

- For more information about JPMS, see JEP 261: Module System .

2.7. EXTENSION AND ENDORSED OVERRIDE MECHANISMS

In OpenJDK 11, the extension mechanism, which supported optional packages, is no longer available. OpenJDK 11 also removed the endorsed standards override mechanism.

You cannot use the libraries added to **<JAVA_HOME>/lib/ext** or **<JAVA_HOME>/lib/endorsed**. OpenJDK 11 generates an error if it detects these directories. If you want to use optional packages on OpenJDK 11, you can create a module for your optional package with the **jlink** tool, and then use the tool to include a custom runtime in your created module.

Additional resources

- For more information about the removed mechanisms, see JEP 220: Modular Run-Time Images .

2.8. DEPRECATED AND REMOVED FUNCTIONALITY FROM OPENJDK 11

Some features supported by OpenJDK 8 have been either deprecated or removed in OpenJDK 11.

COBRA

OpenJDK 11 does not support the following tools:

- **ldlj**
- **orbd**
- **servertool**
- **tnamesrv**

Logging framework

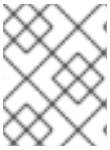
OpenJDK 11 does not support the following APIs:

- **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**

Java EE modules

OpenJDK 11 does not support the following APIs:

- **java.activation**
- **java.corba**
- **java.se.ee (aggregator)**
- **java.transaction**
- **java.xml.bind**
- **java.xml.ws**
- **java.xml.ws.annotation**



NOTE

External dependencies might provide some of the previously listed APIs, so consider using these APIs if they were provided by dependencies.

java.awt.peer

OpenJDK 11 sets the **java.awt.peer** package as internal. This means that applications cannot automatically access the package by default. Because of this change, OpenJDK 11 removed classes and methods that refer to the peer API, such as the **Component.getPeer** method.
An example use case is using the peer API to check the following criteria:

- A component can be displayed
- The component is a lightweight component.

- The component is backed by an OS native UI component.

This code can be updated with calls to **Component.isDisplayable()** and **Component.isLightweight()** to perform the same task.

javax.security and java.lang APIs

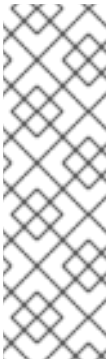
OpenJDK 11 does not support the following APIs:

- **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)**

Sun.misc

The **sun.misc package** is internal and unsupported. In OpenJDK 11, the following packages are deprecated or removed:

- **sun.misc.BASE64Encoder**
- **sun.misc.BASE64Decoder**
- **sun.reflect.Reflection**
  In OpenJDK 11, some methods were removed from the **sun.misc.Unsafe** package.



NOTE

Wherever possible, use public APIs instead of the **sun.misc package** components. For example, use the **VarHandles** package instead of the **sun.misc.Unsafe** package, or use the **StackWalker** API instead of the **sun.reflect.Reflection** API.

An application's use of classes in the **sun.misc** package is restricted in OpenJDK 11. For more information about a workaround for this issue, see Java library classes.

Additional resources

- For more information about OpenJDK 8 features, see JDK 8 Features.
- For more information about OpenJDK 11 features, see JDK 11.

- For more information about a list of all available JEPs, see JEP 0: JEP Index .
- For more information about the removed Java EE modules and COBRA modules and potential replacements for these modules, see JEP 320: Remove the Java EE and CORBA Modules .

CHAPTER 3. PREPARATION FOR MIGRATION

OpenJDK 11 includes updates and changes that might require you to re-configure your applications, which were already successfully deployed on OpenJDK 8.

You can ensure an effective migration plan by reviewing the *Major differences between OpenJDK 8 and OpenJDK 11 section* and integrating the differences into your migration plan.

Red Hat provides the following tool that you can use to help with your migration tasks:

- The Migration Toolkit for Applications (MTA), which is a specific tool that you can use to migrate Java applications from OpenJDK 8 to OpenJDK 11.

Additional resources

- For more information about the major differences between OpenJDK releases, see Major differences between OpenJDK 8 and OpenJDK 11.
- For more details about the installation of OpenJDK 11 on RHEL, see the *Installing and using OpenJDK 11 on RHEL* guide.
- For more details about the installation of OpenJDK 11 on Microsoft Windows, see the *Installing and using OpenJDK 8 for Microsoft Windows* guide.
- For more information about switching between OpenJDK versions on RHEL, see Interactively selecting a system-wide OpenJDK version of RHEL in the *Configuring OpenJDK 11 on RHEL* guide.
- For more information about switching between OpenJDK versions on Microsoft Windows, see Selecting a specific OpenJDK from the installed versions of an application in the *Configuring OpenJDK 11 on Microsoft Windows* guide.
- For more information about the MTA tool, see the *Introduction to the Migration Toolkit for Applications* guide.

CHAPTER 4. TOOLS FOR APPLICATION MIGRATION

Before you migrate your applications from OpenJDK 8 to OpenJDK 11, you can use tools to test the suitability of your application to run on OpenJDK 11.

You can use the following steps to enhance your testing process:

- Update third-party libraries.
- Compile your application code.
- Run **jdeps** on your application's code.
- Use the Migration Toolkit for Applications (MTA) to migrate Java applications from OpenJDK 8 to OpenJDK 11.

Additional resources

- For more information about the MTA tool, see the *Introduction to the Migration Toolkit for Applications* guide.