Red Hat build of OpenJDK 21

Migrating to Red Hat build of OpenJDK 21 from earlier versions

Last Updated: 2024-10-02

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

The Migrating to Red Hat build of OpenJDK 21 from earlier versions guide provides information on how to upgrade your Red Hat build of OpenJDK version 8, 11, or 17 applications to Red Hat build of OpenJDK 21.

Table of Contents

PROVIDING FEEDBACK ON RED HAT BUILD OF OPENJDK DOCUMENTATION 4

MAKING OPEN SOURCE MORE INCLUSIVE 5

CHAPTER 1. MIGRATION TO RED HAT BUILD OF OPENJDK 21 6
1.1. ABOUT RED HAT BUILD OF OPENJDK DISTRIBUTIONS 6

CHAPTER 2. MAJOR DIFFERENCES BETWEEN RED HAT BUILD OF OPENJDK 8 AND RED HAT BUILD OF OPENJDK 11 8
2.1. CRYPTOGRAPHY AND SECURITY 8
2.2. GARBAGE COLLECTOR 9
2.3. GARBAGE COLLECTOR LOGGING OPTIONS 10
2.4. OPENJDK GRAPHICS 11
2.5. WEBSTART AND APPLETS 11
2.6. JPMS 12
2.7. EXTENSION AND ENDORSED OVERRIDE MECHANISMS 12
2.8. JFR FUNCTIONALITY 13
2.9. JRE AND HEADLESS PACKAGES 14
2.10. JMODS 14
2.11. DEPRECATED AND REMOVED FUNCTIONALITY IN RED HAT BUILD OF OPENJDK 11 15
2.12. ADDITIONAL RESOURCES (OR NEXT STEPS) 17

CHAPTER 3. MAJOR DIFFERENCES BETWEEN RED HAT BUILD OF OPENJDK 11 AND RED HAT BUILD OF OPENJDK 17 18
3.1. REMOVAL OF CONCURRENT MARK SWEEP GARBAGE COLLECTOR 18
3.2. REMOVAL OF PACK200 TOOLS AND API 18
3.3. REMOVAL OF NASHORN JAVASCRIPT ENGINE 19
3.4. STRONG ENCAPSULATION OF JDK INTERNAL ELEMENTS 19
3.5. BIASED LOCKING DISABLED BY DEFAULT 19
3.6. REMOVAL OF RMI ACTIVATION 19
3.7. REMOVAL OF THE GRAAL COMPILER 19
3.8. ADDITIONAL RESOURCES (OR NEXT STEPS) 20

CHAPTER 4. MAJOR DIFFERENCES BETWEEN RED HAT BUILD OF OPENJDK 17 AND RED HAT BUILD OF OPENJDK 21 21
4.1. UTF-8 CHARACTER SET USED BY DEFAULT 21
4.2. Z GARBAGE COLLECTOR IMPROVEMENTS 21
4.3. WARNINGS ABOUT THE DYNAMIC LOADING OF AGENTS 21
4.4. FINALIZATION DEPRECATED FOR FUTURE REMOVAL 22
4.5. INTERNET ADDRESS RESOLUTION SPI 22
4.6. SIMPLE WEB SERVER 22
4.7. SEQUENCED COLLECTIONS 23
4.8. PATTERN MATCHING FOR SWITCH STATEMENTS 23
4.9. KEY ENCAPSULATION MECHANISM API 24
4.10. CODE SNIPPETS IN JAVA API DOCUMENTATION 24
4.11. VIRTUAL THREADS 25
4.12. VECTOR API 26
4.13. REIMPLEMENTATION OF THE CORE REFLECTION FUNCTIONALITY WITH METHOD HANDLES 26
4.14. FOREIGN FUNCTION AND MEMORY API (THIRD PREVIEW) 27
4.15. RECORD PATTERNS (PREVIEW) 27
4.16. UNNAMED PATTERNS AND VARIABLES (PREVIEW) 28
4.17. STRING TEMPLATES (PREVIEW) 29
4.18. UNNAMED CLASSES AND INSTANCE MAIN() METHODS (PREVIEW) 29

4.19. SCOPED VALUES (PREVIEW) 30
4.20. STRUCTURED CONCURRENCY (PREVIEW) 31
4.21. ADDITIONAL RESOURCES (OR NEXT STEPS) 31

CHAPTER 5. PREPARATION FOR MIGRATION 33

CHAPTER 6. TOOLS FOR APPLICATION MIGRATION 34

PROVIDING FEEDBACK ON RED HAT BUILD OF OPENJDK DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to create a ticket.
2. Enter a brief description of the issue in the **Summary**.
3. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
4. Clicking **Create** creates and routes the issue to the appropriate documentation team.

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 .

CHAPTER 1. MIGRATION TO RED HAT BUILD OF OPENJDK 21

The *Migrating to Red Hat build of OpenJDK 21 from earlier versions* guide provides information on how to upgrade your Java applications in Red Hat build of OpenJDK version 8, 11, or 17 to Red Hat build of OpenJDK 21.

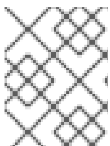
This guide describes changes in the Red Hat build of OpenJDK 21 release, including new features and deprecated or removed APIs, that might impact your migration from an earlier version of Red Hat build of OpenJDK.

If you are migrating from Red Hat build of OpenJDK 11 or earlier, this guide also describes the changes that were introduced in Red Hat build of OpenJDK 17. If you want to migrate from Red Hat build of OpenJDK 11 or earlier, ensure that you also familiarize yourself with the major differences between versions 11 and 17.

If you are migrating from Red Hat build of OpenJDK 8, this guide also describes the changes that were introduced in Red Hat build of OpenJDK 11. If you want to migrate from Red Hat build of OpenJDK 8, ensure that you also familiarize yourself with the major differences between versions 8 and 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 Red Hat build of OpenJDK project might sometimes introduce a few incompatibilities across major releases of Red Hat build 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



NOTE

Red Hat recommends that you migrate to the latest supported Red Hat build of OpenJDK distribution.

1.1. ABOUT RED HAT BUILD OF OPENJDK DISTRIBUTIONS

OpenJDK is the free and open source reference implementation of the Java Platform, Standard Edition (Java SE). Red Hat build of OpenJDK distributions are based on the upstream OpenJDK 8u, OpenJDK 11u, OpenJDK 17u, and OpenJDK 21u projects. The Shenandoah Garbage Collector is included in all supported versions of Red Hat build of OpenJDK.

All versions of Red Hat build of OpenJDK provide the following benefits:

Multi-platform

Red Hat build of OpenJDK is supported on RHEL and Microsoft Windows, so you can standardize applications on a single Java platform across desktop, data center, and hybrid cloud environments.

Frequent releases

Red Hat delivers quarterly updates of JRE and JDK for Red Hat build of OpenJDK 8, Red Hat build of OpenJDK 11, Red Hat build of OpenJDK 17, and Red Hat build of OpenJDK 21 distributions. These updates are available as archive, RPM, and Windows MSI-based installer files and container images.

Long-term support

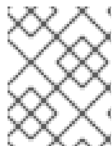
Red Hat supports the recently released Red Hat build of OpenJDK 8, Red Hat build of OpenJDK 11, Red Hat build of OpenJDK 17, and Red Hat build of OpenJDK 21 distributions.

Additional resources

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

CHAPTER 2. MAJOR DIFFERENCES BETWEEN RED HAT BUILD OF OPENJDK 8 AND RED HAT BUILD OF OPENJDK 11

If you are migrating your Java applications from Red Hat build of OpenJDK 8, first ensure that you familiarize yourself with the changes that were introduced in Red Hat build of OpenJDK 11. These changes might require that you reconfigure your existing Red Hat build of OpenJDK installation before you migrate to Red Hat build of OpenJDK 21.

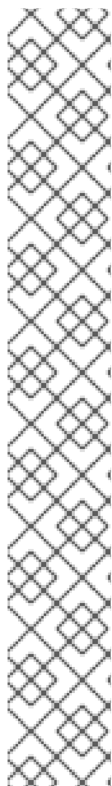


NOTE

This chapter is relevant only if you currently use Red Hat build of OpenJDK 8. You can ignore this chapter if you already use Red Hat build of OpenJDK 11 or later.

One of the major differences between Red Hat build of OpenJDK 8 and later versions is the inclusion of a module system in Red Hat build of OpenJDK 11 or later. If you are migrating from Red Hat build of OpenJDK 8, consider moving your application's libraries and modules from the Red Hat build of OpenJDK 8 class path to the module class in Red Hat build of OpenJDK 11 or later. This change can improve the class-loading capabilities of your application.

Red Hat build of OpenJDK 11 and later versions include 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 Red Hat build of OpenJDK and other upstream community or third-party versions of OpenJDK. For example:

- The Shenandoah garbage collector is available in all versions of Red Hat build of OpenJDK, but this feature might not be available by default in other builds of OpenJDK.
- JDK Flight Recorder (JFR) support in OpenJDK 8 has been available from version 8u262 onward and enabled by default from version 8u272 onward, but JFR might be disabled in certain builds. Because JFR functionality was backported from the open source version of JFR in OpenJDK 11, the JFR implementation in Red Hat build of OpenJDK 8 is largely similar to JFR in Red Hat build of OpenJDK 11 or later. This JFR implementation is different from JFR in Oracle JDK 8, so users who want to migrate from Oracle JDK to Red Hat build of OpenJDK 8 or later need to be aware of the command-line options for using JFR.
- 32-bit builds of OpenJDK are generally unsupported in OpenJDK 8 or later, and they might not be available in later versions. 32-bit builds are unsupported in all versions of Red Hat build of OpenJDK.

2.1. CRYPTOGRAPHY AND SECURITY

Certain minor cryptography and security differences exist between Red Hat build of OpenJDK 8 and Red Hat build of OpenJDK 11. However, both versions of Red Hat build of OpenJDK 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 versions of Red Hat build of OpenJDK, so you can easily change from Red Hat build of OpenJDK 8 to Red Hat build of OpenJDK 11 or later.

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

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

The following minor cryptography and security differences exist between Red Hat build of OpenJDK 8 and Red Hat build of OpenJDK 11:

Red Hat build of OpenJDK 8	Red Hat build of OpenJDK 11
TLS clients do not use TLSv1.3 for communication with the target server by default. You can change this behavior by setting the **jdk.tls.client.protocols** system property to **-Djdk.tls.client.protocols=TLSv1.3**.	TLS clients use TLSv.1.3 by default.
This release does not support the use of the **X25519** and **X448** elliptic curves in the Diffie-Hellman key exchange.	This release supports the use of the **X25519** and **X448** elliptic curves in the Diffie-Hellman key exchange.
This release still supports the legacy KRB5-based cipher suites, which are disabled for security reasons. You can enable these cipher suites by changing the **jdk.tls.client.cipherSuites** and **jdk.tls.server.cipherSuites** system properties.	This release does not support the legacy KRB5-based cipher suites.
This release does not support the Datagram Transport Layer Security (DTLS) protocol.	This release supports the DTLS protocol.
The **max_fragment_length** extension, which is used by DTLS, is not available for TLS clients.	The **max_fragment_length** extension is available for both clients and servers.

2.2. GARBAGE COLLECTOR

For garbage collection, Red Hat build of OpenJDK 8 uses the Parallel collector by default, whereas Red Hat build of OpenJDK 11 uses the Garbage-First (G1) collector by default.

Before you choose a garbage collector, consider the following details:

- If you want to improve throughput, use the Parallel collector. The Parallel collector maximizes throughput but ignores latency, which means that garbage collection pauses could become an issue if you want your application to have reasonable response times. However, if your application is performing batch processing and you are not concerned about pause times, the Parallel collector is the best choice. You can switch to the Parallel collector by setting the **-XX:+UseParallelGC** JVM option.

- If you want a balance between throughput and latency, use the G1 collector. The G1 collector can achieve great throughput while providing reasonable latencies with pause times of a few

hundred milliseconds. If you notice throughput issues when migrating applications from Red Hat build of OpenJDK 8 to Red Hat build of OpenJDK 11, you can switch to the Parallel collector as described above.

- If you want low-latency garbage collection, use the Shenandoah collector.

You can select the garbage collector type that you want to use by specifying the **-XX:+<gc_type>** JVM option at startup. For example, the **-XX:+UseParallelGC** option switches to the Parallel collector.

2.3. GARBAGE COLLECTOR LOGGING OPTIONS

Red Hat build of OpenJDK 11 includes a new and more powerful logging framework that works more effectively than the old logging framework. Red Hat build of OpenJDK 11 also includes unified JVM logging options and unified GC logging options.

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

Modified options in Red Hat build of OpenJDK 11

The old logging framework options are deprecated in Red Hat build of OpenJDK 11. These old options are still available only as aliases for the new logging framework options. If you want to work more effectively with Red Hat build of OpenJDK 11 or later, use the new logging framework options.

The following table outlines the changes in garbage collector logging options between Red Hat build of OpenJDK versions 8 and 11:

Options in Red Hat build of OpenJDK 8	Options in Red Hat build of OpenJDK 11
-verbose:gc	**-Xlog:gc**
-XX:+PrintGC	**-Xlog:gc**
-XX:+PrintGCDetails	**-Xlog:gc*** or **-Xlog:gc+$tags**
-Xloggc:$FILE	**-Xlog:gc:file=$FILE**

When using the **-XX:+PrintGCDetails** option, pass the **-Xlog:gc*** flag, where the asterisk (*) activates more detailed logging. Alternatively, you can pass the **-Xlog:gc+$tags** flag.

When using the **-Xloggc** option, append the **:file=$FILE** suffix to redirect log output to the specified file. For example **-Xlog:gc:file=$FILE**.

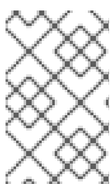
Removed options in Red Hat build of OpenJDK 11

Red Hat build of OpenJDK 11 does not include the following options, which were deprecated in Red Hat build of OpenJDK 8:

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

Red Hat build of OpenJDK 11 also removes the following options because the printing of timestamps and datestamps is automatically enabled:

- **-XX:+PrintGCTimeStamps**
- **-XX:+PrintGCDateStamps**



NOTE

In Red Hat build of OpenJDK 11, unless you specify the **-XX:+IgnoreUnrecognizedVMOptions** option, the use of any of the preceding removed options results in a startup failure.

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 information about deprecated and 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

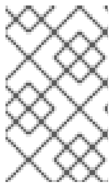
Before version 8u252, Red Hat build of OpenJDK 8 used Pisces as the default rendering engine. From version 8u252 onward, Red Hat build of OpenJDK 8 uses Marlin as the new default rendering engine. Red Hat build of OpenJDK 11 and later releases also use Marlin by default. Marlin improves the handling of intensive application graphics.

Because the rendering engines produce the same results, users should not observe any changes apart from improved performance.

2.5. WEBSTART AND APPLETS

You can use Java WebStart by using the IcedTea-Web plug-in with Red Hat build of OpenJDK 8 or Red Hat build of OpenJDK 11 on RHEL 7, RHEL 8, and Microsoft Windows operating systems. The IcedTea-Web plug-in requires that Red Hat build of OpenJDK 8 is installed as a dependency on the system.

Applets are not supported on any version of Red Hat build of OpenJDK. Even though some applets can be run on RHEL 7 by using the IcedTea-web plug-in with OpenJDK 8 on a Netscape Plugin Application Programming Interface (NPAPI) browser, Red Hat build of OpenJDK does not support this behavior.



NOTE

The upstream community version of OpenJDK does not support applets or Java Webstart. Support for these technologies is deprecated and they are not recommended for use.

2.6. JPMS

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 (for example, whether you use a class path or a module path).

Internal modules

By default, Red Hat build of OpenJDK 11 restricts but still permits access to JDK internal modules. This means that most applications can continue to work without requiring changes, but these applications will emit a warning. As a workaround for this restriction, you can enable your application to access an internal package by passing a **--add-opens <module-name>/<package-in-module>=ALL-UNNAMED** option to the **java** command.

For example:

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

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 Red Hat build of OpenJDK.

ClassLoader

The JPMS refactoring changes the **ClassLoader** hierarchy in Red Hat build of OpenJDK 11.

In Red Hat build of OpenJDK 11, the system class loader is no longer an instance of **URLClassLoader**. Existing application code that invokes **ClassLoader::getSystemClassLoader** and casts the result to a **URLClassLoader** in Red Hat build of OpenJDK 11 will result in a runtime exception.

In Red Hat build of OpenJDK 8, when you create a class loader, you can pass **null** as the parent of this class loader instance. However, in Red Hat build of OpenJDK 11, applications that pass **null** as the parent of a class loader might prevent the class loader from locating platform classes.

Red Hat build of 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. In Red Hat build of OpenJDK 11, when you create a class loader instance, you can set the platform class loader as its parent 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 Red Hat build of OpenJDK 11, both the extension mechanism, which supported optional packages, and the endorsed standards override mechanism are no longer available.

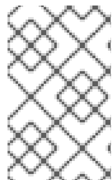
These changes mean that any libraries that are added to the **<JAVA_HOME>/lib/ext** or **<JAVA_HOME>/lib/endorsed** directory are no longer used, and Red Hat build of OpenJDK 11 generates an error if these directories exist.

Additional resources

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

2.8. JFR FUNCTIONALITY

JDK Flight Recorder (JFR) support was backported to Red Hat build of OpenJDK 8 starting from version 8u262. JFR support was subsequently enabled by default from Red Hat build of OpenJDK 8u272 onward.



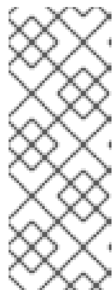
NOTE

The term *backporting* describes when Red Hat takes an update from a more recent version of upstream software and applies that update to an older version of the software that Red Hat distributes.

Backported JFR features

The JFR backport to Red Hat build of OpenJDK 8 included all of the following features:

- A large number of events that are also available in Red Hat build of OpenJDK 11
- Command-line tools such as **jfr** and the Java diagnostic command (**jcmd**) that behave consistently across Red Hat build of OpenJDK versions 8 and 11
- The Java Management Extensions (JMX) API that you can use to enable JFR by using the JMX beans interfaces either programmatically or through **jcmd**
- The **jdk.jfr** namespace



NOTE

The JFR APIs in the **jdk.jfr** namespace are not considered part of the Java specification in Red Hat build of OpenJDK 8, but these APIs are part of the Java specification in Red Hat build of OpenJDK 11. Because the JFR API is available in all supported Red Hat build of OpenJDK versions, applications that use JFR do not require any special configuration to use the JFR APIs in Red Hat build of OpenJDK 8 and later versions.

JDK Mission Control, which is distributed separately, was also updated to be compatible with Red Hat build of OpenJDK 8.

Applications that need to be compatible with other OpenJDK versions

If your applications need to be compatible with any of the following OpenJDK versions, you might need to adapt these applications:

- OpenJDK versions earlier than 8u262
- OpenJDK versions from other vendors that do not support JFR
- Oracle JDK

To aid this effort, Red Hat has developed a special compatibility layer that provides an empty implementation of JFR, which behaves as if JFR was disabled at runtime. For more information about the JFR compatibility API, see openjdk8-jfr-compat. You can install the resulting **.jar** file in the

jre/lib/ext directory of an OpenJDK 8 distribution.

Some applications might need to be updated if these applications were filtering out OpenJDK 8 by checking only for the version number instead of querying the MBeans interface.

2.9. JRE AND HEADLESS PACKAGES

All Red Hat build of OpenJDK versions for RHEL platforms are separated into the following types of packages. The following list of package types is sorted in order of minimality, starting with the most minimal.

Java Runtime Environment (JRE) headless

Provides the library only without support for graphical user interface but supports offline rendering of images

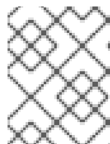
JRE

Adds the necessary libraries to run for full graphical clients

JDK

Includes tooling and compilers

Red Hat build of OpenJDK versions for Windows platforms do not support headless packages. However, the Red Hat build of OpenJDK packages for Windows platforms are also divided into JRE and JDK components, similar to the packages for RHEL platforms.



NOTE

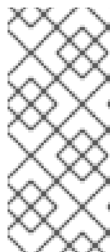
The upstream community version of OpenJDK 11 or later does not separate packages in this way and instead provides one monolithic JDK installation.

OpenJDK 9 introduced a modularised version of the JDK class libraries divided by their namespaces. From Red Hat build of OpenJDK 11 onward, these libraries are packaged into **jmods** modules. For more information, see Jmods.

2.10. JMODS

OpenJDK 9 introduced **jmods**, which is a modularized version of the JDK class libraries, where each module groups classes from a set of related packages. You can use the **jlink** tool to create derivative runtimes that include only some subset of the modules that are needed to run selected applications.

From Red Hat build of OpenJDK 11 onward, Red Hat build of OpenJDK versions for RHEL platforms place the **jmods** files into a separate RPM package that is not installed by default. If you want to create standalone OpenJDK images for your applications by using **jlink**, you must manually install the **jmods** package (for example, **java-11-openjdk-jmods**).



NOTE

On RHEL platforms, OpenJDK is dynamically linked against system libraries, which means the resulting **jlink** images are not portable across different versions of RHEL or other systems. If you want to ensure portability, you must use the portable builds of Red Hat build of OpenJDK that are released through the Red Hat Customer Portal. For more information, see Installing Red Hat build of OpenJDK on RHEL by using an archive .

2.11. DEPRECATED AND REMOVED FUNCTIONALITY IN RED HAT BUILD OF OPENJDK 11

Red Hat build of OpenJDK 11 has either deprecated or removed some features that Red Hat build of OpenJDK 8 supports.

CORBA

Red Hat build of OpenJDK 11 does not support the following Common Object Request Broker Architecture (CORBA) tools:

- **Idlj**
- **orbd**
- **servertool**
- **tnamesrv**

Logging framework

Red Hat build of 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

Red Hat build of 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**

java.awt.peer

Red Hat build of OpenJDK 11 sets the **java.awt.peer** package as internal, which means that applications cannot automatically access this package by default. Because of this change, Red Hat build of OpenJDK 11 removed a number of classes and methods that refer to the peer API, such as the **Component.getPeer** method.

The following list outlines the most common use cases for the peer API:

- Writing of new graphics ports
- Checking if a component can be displayed
- Checking if a component is either lightweight or backed by an operating system native UI component resource such as an Xlib XWindow

From Java 1.1 onward, the **Component.isDisplayable()** method provides the functionality to check whether a component can be displayed. From Java 1.2 onward, the **Component.isLightweight()** method provides the functionality to check whether a component is lightweight.

javax.security and java.lang APIs

Red Hat build of 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** has always been considered internal and unsupported. In Red Hat build of OpenJDK 11, the following packages are deprecated or removed:

- **sun.misc.BASE64Encoder**
- **sun.misc.BASE64Decoder**
- **sun.misc.Unsafe**
- **sun.reflect.Reflection**

Consider the following information:

- Red Hat build of OpenJDK 8 added the **java.util.Base64** package as a replacement for the **sun.misc.BASE64Encoder** and **sun.misc.BASE64Decoder** APIs. You can use the **java.util.Base64** package rather than these APIs, which have been removed from Red Hat build of OpenJDK 11.
- Red Hat build of OpenJDK 11 deprecates the **sun.misc.Unsafe** package, which is scheduled for removal. For more information about a new set of APIs that you can use as a replacement for **sun.misc.Unsafe**, see JEP 193.

- Red Hat build of OpenJDK 11 removes the **sun.reflect.Reflection** package. For more information about new functionality for stack walking that replaces the **sun.reflect.Reflection.getCallerClass** method, see JEP 259.

Additional resources

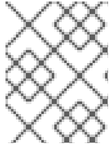
- For more information about the removed Java EE modules and CORBA modules and potential replacements for these modules, see JEP 320: Remove the Java EE and CORBA Modules .

2.12. ADDITIONAL RESOURCES (OR NEXT STEPS)

- For more information about Red Hat build of OpenJDK 8 features, see JDK 8 Features.
- For more information about OpenJDK 9 features inherited by Red Hat build of OpenJDK 11, see JDK 9.
- For more information about OpenJDK 10 features inherited by Red Hat build of OpenJDK 11, see JDK 10.
- For more information about Red Hat build of 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 changes introduced in version 17, see Major differences between Red Hat build of OpenJDK 11 and Red Hat build of OpenJDK 17.
- For more information about the changes introduced in version 21, see Major differences between Red Hat build of OpenJDK 17 and Red Hat build of OpenJDK 21.

CHAPTER 3. MAJOR DIFFERENCES BETWEEN RED HAT BUILD OF OPENJDK 11 AND RED HAT BUILD OF OPENJDK 17

If you are migrating your Java applications from Red Hat build of OpenJDK 11 or earlier, first ensure that you familiarize yourself with the changes that were introduced in Red Hat build of OpenJDK 17. These changes might require that you reconfigure your existing Red Hat build of OpenJDK installation before you migrate to Red Hat build of OpenJDK 21.



NOTE

This chapter is relevant only if you currently use Red Hat build of OpenJDK 11 or earlier. You can ignore this chapter if you already use Red Hat build of OpenJDK 17.

3.1. REMOVAL OF CONCURRENT MARK SWEEP GARBAGE COLLECTOR

Red Hat build of OpenJDK 17 no longer includes the Concurrent Mark Sweep (CMS) garbage collector, which was commonly used in earlier releases for workloads sensitive to pause times and latency.

If you have been using the CMS collector, switch to one of the following collectors based on your workload before migrating to Red Hat build of OpenJDK 17 or later.

- The Garbage-First (G1) collector balances performance and latency. G1 is a generational collector that offers a high ephemeral object allocation rate with typical pause times of a few hundred milliseconds. G1 is enabled by default, but you can manually enable this collector by setting the **-XX:+UseG1GC** JVM option.

- The Shenandoah collector is a low-latency collector with typical pause times of a few milliseconds. Shenandoah is not a generational collector and might exhibit worse ephemeral object allocation rates than the G1 collector. If you want to enable the Shenandoah collector, set the **-XX:+UseShenandoahGC** JVM option.

- The Z Garbage Collector (ZGC) is another low-latency collector. Unlike the Shenandoah collector, ZGC does not support compressed ordinary object pointers (OOPs) (that is, heap references). Compressed OOPs help to save heap memory and improve performance for heap sizes up to 32 GB. This means that ZGC might exhibit worse resident memory sizes than the Shenandoah collector, especially on small heap sizes. If you want to enable the ZGC collector, set the **-XX:+UseZGC** JVM option.

For more information, see JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector .

3.2. REMOVAL OF PACK200 TOOLS AND API

Red Hat build of OpenJDK 17 no longer includes any of the following features:

- The **pack200** tool
- The **unpack200** tool
- The **java.util.jar.Pack200** API
- The **java.util.jar.Pack200.Packer** API
- The **java.util.jar.Pack200.Unpacker** API

The use of these tools and APIs has been limited since the introduction of the JMOD module format in OpenJDK 9.

For more information, see JEP 367: Remove the Pack200 Tools and API .

3.3. REMOVAL OF NASHORN JAVASCRIPT ENGINE

Red Hat build of OpenJDK 17 no longer includes any of the following features:

- The Nashorn JavaScript engine
- The **jjs** command-line tool
- The **jdk.scripting.nashorn** module
- The **jdk.scripting.nashorn.shell** module

The scripting API, **javax.script**, is still available in Red Hat build of OpenJDK 17 or later. Similar to releases before OpenJDK 8, you can use the **javax.script** API with a JavaScript engine of your choice, such as Rhino or the now externally maintained Nashorn JavaScript engine.

For more information, see JEP 372: Remove the Nashorn JavaScript Engine .

3.4. STRONG ENCAPSULATION OF JDK INTERNAL ELEMENTS

Red Hat build of OpenJDK 17 introduces strong encapsulation of all internal elements of the JDK, apart from critical internal APIs such as **sun.misc.Unsafe**. From Red Hat build of OpenJDK 17 onward, you cannot relax the strong encapsulation of internal elements by using a single command-line option. This means that Red Hat build of OpenJDK 17 and later versions prevent reflective access to JDK internal types apart from critical internal APIs.

For more information, see JEP 403: Strongly Encapsulate JDK Internals .

3.5. BIASED LOCKING DISABLED BY DEFAULT

Red Hat build of OpenJDK 17 disables biased locking by default. In Red Hat build of OpenJDK 17, you can enable biased locking by setting the **-XX:+UseBiasedLocking** JVM option at startup. However, the **-XX:+UseBiasedLocking** option is deprecated in Red Hat build of OpenJDK 17 and planned for removal in OpenJDK 18.

For more information, see JEP 374: Deprecate and Disable Biased Locking .

3.6. REMOVAL OF RMI ACTIVATION

Red Hat build of OpenJDK 17 removes the **java.rmi.activation** package and its associated **rmid** activation daemon for Java remote method invocation (RMI). Other RMI features are still available in Red Hat build of OpenJDK 17 and later versions.

For more information, see JEP 407: Remove RMI Activation .

3.7. REMOVAL OF THE GRAAL COMPILER

Red Hat build of OpenJDK 17 removes the Graal compiler, which comprises the **jaotc** tool and the **jdk.internal.vm.compiler** and **jdk.internal.vm.compiler.management** modules. From Red Hat build of OpenJDK 17 onward, if you want to use ahead-of-time (AOT) compilation, you can use GraalVM.

For more information, see JEP 410: Remove the Experimental AOT and JIT Compiler .

3.8. ADDITIONAL RESOURCES (OR NEXT STEPS)

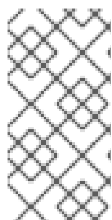
- OpenJDK: JEPs in JDK 17 integrated since JDK 11
- Major differences between Red Hat build of OpenJDK 17 and Red Hat build of OpenJDK 21

CHAPTER 4. MAJOR DIFFERENCES BETWEEN RED HAT BUILD OF OPENJDK 17 AND RED HAT BUILD OF OPENJDK 21

Before migrating your Java applications from Red Hat build of OpenJDK 17 or earlier to Red Hat build of OpenJDK 21, familarize yourself with the changes in version 21. These changes might require that you reconfigure your existing Red Hat build of OpenJDK installation before you migrate to version 21.

Most of your source code should already work in Red Hat build of OpenJDK 21. However, Red Hat build of OpenJDK 21 includes some additional features that can help to make your source code more robust and secure. Users are advised to familarize with these new features and upgrade their applications to use this additional functionality.

Due to version changes in the Common Locale Data Repository (CLDR), OpenJDK 20 and later versions cannot parse some date and time strings that were created in OpenJDK 19 or earlier. Later JDK versions support a “loose match” mechanism that can fix some of these issues. However, if you experience issues parsing date and time strings, consider using the **-Djava.locale.providers=COMPAT** parameter when launching your application and consider migrating these strings to the newer CLDR version.



NOTE

Red Hat does not provide builds of OpenJDK with 32-bit support. In OpenJDK 21, Windows 32-bit x86 support is also now deprecated upstream. This feature will be removed in a future release. For more information, see JEP 449: Deprecate the Windows 32-bit x86 Port for Removal.

4.1. UTF-8 CHARACTER SET USED BY DEFAULT

From Red Hat build of OpenJDK 21 onward, UTF-8 is the default character set when using the APIs for reading and writing files and for processing text. In earlier releases, if no character set was passed as an argument, the character set was based on the runtime environment and depended on the specific method being called. To ensure that your applications are not expecting a different character encoding, review your application source code and environment.

For more information, see JEP 400: UTF-8 by Default .

4.2. Z GARBAGE COLLECTOR IMPROVEMENTS

Red Hat build of OpenJDK 21 extends the Z Garbage Collector (ZGC) to maintain separate allocation regions (that is, generations) for young and old objects. This enhancement allows ZGC to collect young objects more frequently, which helps to improve application performance.

You can enable generational ZGC by specifying the **-XX:+UseZGC** and **-XX:+ZGenerational** JVM options at startup.

For more information, see JEP 439: Generational ZGC.

4.3. WARNINGS ABOUT THE DYNAMIC LOADING OF AGENTS

Red Hat build of OpenJDK 21 issues warnings when agents are loaded dynamically into a running JVM. The dynamic loading of agents will be disallowed by default in a future release. If your applications currently rely on the dynamic loading of agents into the JVM, consider updating your application code to use a different approach.

For more information, see JEP 451: Prepare to Disallow the Dynamic Loading of Agents .

4.4. FINALIZATION DEPRECATED FOR FUTURE REMOVAL

Red Hat build of OpenJDK 21 deprecates the finalization feature, which is used for performing cleanup operations before an object is destroyed. The finalization feature is planned to be removed in a future release.

Red Hat build of OpenJDK 21 still enables finalization by default. To facilitate early testing, you can disable finalization by setting the **--finalization=disabled** command-line option.

If you are maintaining libraries and applications that rely on finalization, consider migrating to other resource management techniques, such as cleaners or the **try**-with-resources statement. Due to this change, maintainers of libraries and applications are advised to test their code as soon as possible.

For more information, see JEP 421: Deprecate Finalization for Removal .

4.5. INTERNET ADDRESS RESOLUTION SPI

Red Hat build of OpenJDK 21 includes a service provider interface (SPI) for host name and address resolution that supports the use of different resolver providers. From Red Hat build of OpenJDK 21 onward, the **InetAddress** API uses a service loader to locate a resolver provider. Similar to previous releases, if no provider is found, the JDK uses the built-in implementation.

For more information, see JEP 418: Internet-Address Resolution SPI .

4.6. SIMPLE WEB SERVER

Red Hat build of OpenJDK 21 includes a **jwebserver** command-line tool that you can use to start a simple web server. The aim of this feature is to support prototyping, ad-hoc coding, and testing, especially for an educational purpose.

The **jwebserver** tool supports the HTTP 1.1 protocol and serves static files only. The simple web server does not support the secure HTTPS protocol and is not suitable for production services.

To run the simple web server, you can use the following command:

```
$ jwebserver
```

By default, files are served from the current directory. If the requested resource is a directory that contains an index file, the index file is served.

In addition to the **jwebserver** tool, this feature also includes an API to support enhanced request handling that can be used to return HTTP headers.

For example:

```
jshell> var h = HttpHandlers.handleOrElse(r -> r.getRequestMethod().equals("PUT"),
   ...> new SomePutHandler(), new SomeHandler());
jshell> var f = Filter.adaptRequest("Add Foo header", r -> r.with("Foo", List.of("Bar")));
jshell> var s = HttpServer.create(new InetSocketAddress(8080),
   ...> 10, "/", h, f);
jshell> s.start();
```

For more information, see JEP 408: Simple Web Server .

4.7. SEQUENCED COLLECTIONS

Red Hat build of OpenJDK 21 introduces a sequenced collections feature that adds new interfaces to represent ordered collections, where each collection has a range of elements with a well-defined encounter order. This feature unifies how each ordered collection accesses its first, last, and other specific elements.

SequencedCollection interface

The **SequencedCollection** interface provides methods for adding, getting, or removing the first and last elements in the collection and for getting a reverse-ordered view of the collection:

```
interface SequencedCollection<E> extends Collection<E> {
    // new method
    SequencedCollection<E> reversed();
    // methods promoted from Deque
    void addFirst(E);
    void addLast(E);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}
```

The **SequencedCollection** interface is implemented by the **SortedSet**, **NavigableSet**, **LInkedHashSet**, **List**, and **Deque** interfaces.

SequencedMap interface

The **SequencedMap** interface provides methods for getting or removing entries at either end of the collection:

```
interface SequencedMap<K,V> extends Map<K,V> {
    // new methods
    SequencedMap<K,V> reversed();
    SequencedSet<K> sequencedKeySet();
    SequencedCollection<V> sequencedValues();
    SequencedSet<Entry<K,V>> sequencedEntrySet();
    V putFirst(K, V);
    V putLast(K, V);
    // methods promoted from NavigableMap
    Entry<K, V> firstEntry();
    Entry<K, V> lastEntry();
    Entry<K, V> pollFirstEntry();
    Entry<K, V> pollLastEntry();
}
```

The **SequencedMap** interface is implemented by the **SortedMap**, **NavigableMap**, and **LInkedHashMap** interfaces.

For more information, see JEP 431: Sequenced Collections .

4.8. PATTERN MATCHING FOR SWITCH STATEMENTS

Red Hat build of OpenJDK 21 enhances the Java programming language with pattern matching for **switch** statements. This enhancement enables you to test **switch** expressions against different patterns that each have a specific action, to ensure that complex data-oriented queries are expressed concisely and safely.

You can implement a **switch** statement that is based on the class of an object.

For example:

```
switch (obj) {
  case Integer i -> String.format("int %d", i);
  case String s  -> String.format("String %s", s);
  default        -> obj.toString();
 };
```

You can also implement more complex **switch** statements by nesting a **when** guard inside the **case** block of the **switch** statement.

For example:

```
switch (response) {
    case null -> { }
    case String s when s.equalsIgnoreCase("YES") -> {
        System.out.println("You got it");
    }
    case String s when s.equalsIgnoreCase("NO") -> {
        System.out.println("Shame");
    }
    case "n", "N" -> {
        System.out.println("Shame");
    }
    case String s -> {
        System.out.println("Sorry?");
    }
}
```

If your applications use **switch** statements, ensure that these statements are up to date.

For more information, see JEP 441: Pattern Matching for switch.

4.9. KEY ENCAPSULATION MECHANISM API

Red Hat build of OpenJDK 21 introduces an API for a key encapsulation mechanism (KEM). KEM is an encryption technique for securing symmetric keys by using public key cryptography. The KEM API facilitates the use of public key cryptography and helps to improve security when handling secrets and messages.

For more information, see JEP 452: Key Encapsulation Mechanism API.

4.10. CODE SNIPPETS IN JAVA API DOCUMENTATION

Red Hat build of OpenJDK 21 includes a **@snippet** tag for the Javadoc tool's standard doclet. The **@snippet** tag helps to simplify the inclusion of example source code in API documentation.

For example:

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet :
 * if (v.isPresent()) {
 *   System.out.println("v: " + v.get()); // @highlight substring="println"
 * }
 * }
 */
```

Markup tags

You can use the **@snippet** tag in combination with markup tags, such as **@highlight**, to modify the styling of the code. For example, the following code snippet uses the **@highlight** tag to highlight the **println** method name in the API documentation:

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet :
 * if (v.isPresent()) {
 *   System.out.println("v: " + v.get()); // @highlight substring="println"
 * }
 * }
 */
```

External files

You can also use the **@snippet** tag in combination with external files that contain the source code.

For example:

```
/**
 * The following code shows how to use {@code Optional.isPresent}:
 * {@snippet file="ShowOptional.java" region="example"}
 */
```

In the preceding example, **ShowOptional.java** is a file that contains the following code:

```
public class ShowOptional {
 void show(Optional<String> v) {
     // @start region="example"
     if (v.isPresent()) {
         System.out.println("v: " + v.get());
     }
     // @end
 }
}
```

You can specify the location of the external code either as a class name by using the **class** attribute or as a relative file path by using the **file** attribute.

For more information, see JEP 413: Code Snippets in Java API Documentation .

4.11. VIRTUAL THREADS

Red Hat build of OpenJDK 21 introduces virtual threads, which are lightweight threads that reduce the effort of writing, maintaining, and observing high-throughput concurrent applications. If your application uses more than a few thousand concurrent threads and your workload is not CPU-bound, the use of virtual threads can optimize your application.

Virtual threads support thread-local variables and interruptions. Virtual threads can also run any code that traditional threads can run. Therefore, you can migrate your source code to use only virtual threads by changing the way the threads are created.

The following example creates a virtual thread that executes a runnable instance:

```
Thread thread = Thread.ofVirtual().name("example").unstarted(runnable);
```

The following example shows how to create virtual threads by using a loop. Because virtual threads do not require pooling, no thread pool is created. If you want to limit concurrency, you can use semaphores instead.

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
 IntStream.range(0, 10000).forEach(i -> {
      executor.submit(() -> {
         Thread.sleep(Duration.ofSeconds(1));
         return i;
    });
   });
}  // executor.close() is called implicitly, and waits
```

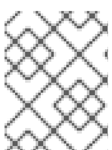
The preceding example creates 10,000 virtual threads to run concurrently. The JDK can run the code on a small number of operating system threads or perhaps only one thread, which reduces the overhead.

For more information, see JEP 444: Virtual Threads .

4.12. VECTOR API

OpenJDK 16 initially introduced the Vector API as an incubating feature. Red Hat build of OpenJDK 21 includes several enhancements to the Vector API based on a sixth round of incubation. The Vector API expresses a wide range of vector computations that can be reliably compiled at runtime to optimal vector instructions on supported CPU architectures. This API achieves better performance levels than equivalent scalar computations.

The latest enhancements ensure the reliability and efficiency of the Vector API in all architectures while also ensuring a graceful degradation of the API in any architecture that does not support all the expected native instructions. Future enhancements to the Vector API might also include alignment with Project Valhalla by defining vector classes as primitive classes.



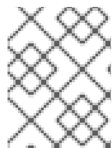
NOTE

Because the Vector API relates to implementation details, your application source code should not not require any updates due to this change.

For more information, see JEP 448: Vector API (Sixth Incubator) .

4.13. REIMPLEMENTATION OF THE CORE REFLECTION FUNCTIONALITY WITH METHOD HANDLES

Red Hat build of OpenJDK 21 includes a reimplementation of the **java.lang.reflect.Method**, **java.lang.reflect.Constructor**, and **java.lang.reflect.Field** classes on top of **java.lang.invoke** method handles. This reimplementation of the core reflection functionality helps to reduce the maintenance and development costs of the **java.lang.reflect** and **java.lang.invoke** APIs.



NOTE

Because this change relates to a reimplementation, your application source code should not require any updates.

For more information, see JEP 416: Reimplement Core Reflection with Method Handles .

4.14. FOREIGN FUNCTION AND MEMORY API (THIRD PREVIEW)

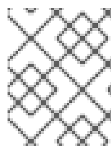
Red Hat build of OpenJDK 21 includes a Foreign Function and Memory (FFM) API, which is a preview feature that enables Java programs to interoperate with code and data that is outside the Java runtime. The FFM API can efficiently invoke foreign functions that are outside the JVM and safely access foreign memory that the JVM does not manage.

The FFM API replaces the existing Java Native Interface (JNI) with a pure Java development model that provides a more efficient, cleaner, and safer way to call native libraries and process native data.

The FFM API defines classes and interfaces that enable client code in libraries and applications to perform the following tasks:

- Control the allocation and deallocation of foreign memory by using the **MemorySegment**, **Arena**, and **SegmentAllocator** interfaces.
- Manipulate and access structured foreign memory by using the **MemoryLayout** and **VarHandle** interfaces.
- Call foreign functions by using the **Linker**, **FunctionDescriptor**, and **SymbolLookup** interfaces.

The FFM API is located in the **java.lang.foreign** package of the **java.base** module.



NOTE

The FFM API is an experimental addition to the Java language that might be subject to improvements in future versions, where no backwards compatibility is guaranteed.

For more information, see JEP 442: Foreign Function & Memory API (Third Preview) .

4.15. RECORD PATTERNS (PREVIEW)

Red Hat build of OpenJDK 21 introduces record patterns, which is a preview feature that enhances the Java programming language to deconstruct record values. This feature allows declaration and assignment variables that use the **instanceof** operator.

Consider the following declaration for a record named **Point** with attributes **x** and **y**:

```
record Point(int x, int y) {}
```

Based on the preceding declaration, the following code can test whether a value is an instance of **Point** and also extract the **x** and **y** components from the value directly:

```
if (obj instanceof Point(int x, int y)) {
    System.out.println(x+y);
}
```

You can also use this feature in records that are nested inside other records. For example, consider the following declarations for a record named **Rectangle** with attributes of a **ColoredPoint** record with attributes of a **Point** record and **Color** enum:

```
record Point(int x, int y) {}
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```

Based on the preceding declarations, the following code can test whether a value is an instance of **Rectangle** and extract the color from the upper-left point:

```
if (r instanceof Rectangle(ColoredPoint ul, ColoredPoint lr)) {
     System.out.println(ul.attribute());
}
```

For more information, see JEP 440: Record Patterns.

4.16. UNNAMED PATTERNS AND VARIABLES (PREVIEW)

Red Hat build of OpenJDK 21 introduces unnamed patterns and variables, which is a preview feature that enhances the Java programming language:

- *Unnamed patterns* match a record component without stating the component's name or type. Unnamed patterns help to improve the readability of record patterns by omitting unnecessary nested patterns.
- *Unnamed variables* can be initialized but not used. Unnamed variables help to improve the maintainability of all code by identifying variables that must be declared (for example, in a **catch** clause) but which are not used.

You can denote unnamed patterns and unnamed variables by using an underscore (_) character. When a variable is not relevant and will not be used, you do not need to name the variable. In this situation, you can use an underscore character to denote that the variable exists but is not for use.

In the following example of a lambda expression, the parameter is irrelevant, which means that the JVM does not need to create a variable for this parameter:

```
...stream.collect(Collectors.toMap(String::toUpperCase, _ -> "NODATA"))
```

In the following example of a **switch** statement, the variables need to match the **case** but these variables will not be used:

```
switch (b) {
 case Box(RedBall _), Box(BlueBall _) -> processBox(b);
 case Box(GreenBall _)             -> stopProcessing();
```

```
 case Box(_)                    -> pickAnotherBox();
}
```

In the preceding example, the first two cases use unnamed pattern variables because their right-hand sides do not use the **Box** record's component. The third case, which is new, uses the unnamed pattern to match a **Box** record with a null component.

For more information, see JEP 443: Unnamed Patterns and Variables (Preview) .

4.17. STRING TEMPLATES (PREVIEW)

Red Hat build of OpenJDK 21 introduces string templates, which is a preview feature that enhances the Java programming language. 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.

The string templates feature includes an **STR** utility, which is a template processor that supports validation and transformation, helping to improve the security of Java programs that compose strings. The **STR** utility also facilitates code readability. You can compose strings by using other variables, attributes, or functions.

For example, consider the following template expression:

```
STR."\{username} access at \{req.date} from \{getLocation()}"
```

The preceding expression produces the following example string:

```
“Guest access at 12/1/2024 from 127.0.0.1”
```

The preview features are experimental additions to the Java language that might be subject to improvements in future versions, where no backwards compatibility is guaranteed.

For more information, see JEP 430: String Templates (Preview) .

4.18. UNNAMED CLASSES AND INSTANCE MAIN() METHODS (PREVIEW)

Red Hat build of OpenJDK 21 introduces unnamed classes and instance **main()** methods, which is a preview feature that enhances the Java programming language for an educational purpose.

Unnamed classes and instance **main()** methods enable students to start writing basic Java programs in a concise manner without needing to understand Java language features for large complex programs. Rather than use a separate dialect of Java, students can write streamlined declarations for single-class programs and seamlessly modify their programs to use more advanced features as their Java knowledge increases.

To simplify the source code and allow students to focus on their own code, Java allows students to launch instance **main()** methods that meet the following criteria:

- Not static
- Not required to be public
- Not required to have a **String[]** parameter

For example:

```
class HelloWorld {
 void main() {
     System.out.println("Hello, World!");
 }
}
```

To allow students to start writing code before learning the object orientation concept, Red Hat build of OpenJDK 21 also introduces unnamed classes to make the class declaration implicit. For example:

```
void main() {
 System.out.println("Hello, World!");
}
```

To test this feature, ensure that you enable preview features. For example:

- If you want to compile and then run the program, enter the following commands:

  ```
  javac --release 21 --enable-preview Main.java
  java --enable-preview Main
  ```

- If you want to use the source code launcher, enter the following command:

  ```
  java --source 21 --enable-preview Main.java
  ```

For more information, see JEP 445: Unnamed Classes and Instance Main Methods (Preview) .

4.19. SCOPED VALUES (PREVIEW)

Red Hat build of OpenJDK 21 introduces scoped values, which is a preview feature that provides an API to support scoped value variables. Typically, a scoped value variable is declared as a final static field that is accessible from many methods.

Scoped values provide a safe and efficient way to share variables across methods without needing to use method parameters. A scoped value is comparable with an implicit method parameter that a method can use without having to declare the parameter within the method itself. Only the methods that have access to the scoped value object can access its data. Scoped values enable applications to pass data securely from a caller to a callee through other intermediate methods that do not declare a parameter for this data and cannot access it. Scoped values are a preferred alternative to thread-local variables, especially when using large numbers of virtual threads.

Consider the following example code where a callee method (**bar**) can use the same scoped value in a nested binding to send a different value to another callee method (**baz**):

```
private static final ScopedValue<String> X = ScopedValue.newInstance();

void main() {
 ScopedValue.where(X, "hello").run(() -> bar());
}

void bar() {
 System.out.println(X.get()); // prints hello
 ScopedValue.where(X, "goodbye").run(() -> baz());
 System.out.println(X.get()); // prints hello
}
```

```
void baz() {
 System.out.println(X.get()); // prints goodbye
}
```

In the preceding example, the **main()** method sets the scoped value **X** to **hello** and sends this value to the **bar()** method. The **bar()** method prints **hello** (the value of **X**) and then uses a nested binding to change the value of **X** and send the changed value to the **baz()** method. Once the **baz()** method returns, the rest of the **bar()** method still has the original **hello** value of **X**.

For more information, see JEP 446: Scoped Values (Preview) .

4.20. STRUCTURED CONCURRENCY (PREVIEW)

Red Hat build of OpenJDK 21 introduces structured concurrency, which is a preview feature that provides an API to treat groups of related tasks running in different threads as a single unit of work. This API simplifies concurrent programming and helps to streamline error handling and cancellation, improve readability, and enhance observability.

In the structured concurrency paradigm, tasks (that is, threads) can be subdivided into subtasks, which can also be nested into further subtasks. This type of task hierarchy enables both cancellation propagation (that is, canceling a task automatically cancels its subtasks) and better control of how tasks interact with each other.

For example:

```
Response handle() throws ExecutionException, InterruptedException {
 try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    Supplier<String>  user  = scope.fork(() -> findUser());
    Supplier<Integer> order = scope.fork(() -> fetchOrder());

    scope.join()          // Join both subtasks
        .throwIfFailed();  // ... and propagate errors

    // Here, both subtasks have succeeded, so compose their results
    return new Response(user.get(), order.get());
 }
}
```

The preceding example first creates a scope and then forks this scope to create the subtasks. At any time, the original thread or any of the subtasks can request a shutdown of the scope, to cancel all unfinished tasks and prevent new subtasks from being created. The **scope.join()** method ensures that the scope must wait for all subtasks to finish running. The **.get()** method returns the results of each subtask.

Each subtask can also create its own **StructuredTaskScope** to divide the workload into further subtasks. In this situation, the additional subtasks are nested under their parent subtask within the overall hierarchy.

For more information, see JEP 453: Structured Concurrency (Preview) .

4.21. ADDITIONAL RESOURCES (OR NEXT STEPS)

- JEPs in JDK 21 integrated since JDK 17

- OpenJDK: JDK 18
- OpenJDK: JDK 19
- OpenJDK: JDK 20
- OpenJDK: JDK 21

CHAPTER 5. PREPARATION FOR MIGRATION

Red Hat build of OpenJDK 21 includes changes that might require you to reconfigure your applications, which were already successfully deployed on Red Hat build of OpenJDK version 8, 11, or 17.

You can ensure an effective migration plan by completing the following tasks, as appropriate:

- If you currently use Red Hat build of OpenJDK 8, review the Major differences between Red Hat build of OpenJDK 8 and Red Hat build of OpenJDK 11 section to familiarize yourself with the changes that are available from version 11 onward.
- If you currently use Red Hat build of OpenJDK 11 or earlier, review the Major differences between Red Hat build of OpenJDK 11 and Red Hat build of OpenJDK 17 section to familiarize yourself with the changes that are available from version 17 onward.
- Review the Major differences between Red Hat build of OpenJDK 17 and Red Hat build of OpenJDK 21 section to familiarize yourself with the changes that are available from version 21 onward.
- Integrate the differences into your migration plan.

Red Hat provides a migration toolkit for applications (MTA) tool that you can use to help with your migration tasks. You can use the MTA tool to migrate Java applications from Red Hat build of OpenJDK version 8, 11, or 17 to Red Hat build of OpenJDK 21.

Additional resources

- For more information about the installation of Red Hat build of OpenJDK 21 on RHEL, see the *Installing and using Red Hat build of OpenJDK 21 on RHEL* guide.
- For more information about the installation of Red Hat build of OpenJDK 21 on Microsoft Windows, see the *Installing and using Red Hat build of OpenJDK 21 for Windows* guide.
- For more information about switching between Red Hat build of OpenJDK versions on RHEL, see Interactively selecting a system-wide Red Hat build of OpenJDK version on RHEL in the *Configuring Red Hat build of OpenJDK 21 on RHEL* guide.
- For more information about the MTA tool, see the *Introduction to the Migration Toolkit for Applications* guide.

CHAPTER 6. TOOLS FOR APPLICATION MIGRATION

Before you migrate your applications from Red Hat build of OpenJDK version 8, 11, or 17 to Red Hat build of OpenJDK 21, you can use tools to test the suitability of your applications to run on Red Hat build of OpenJDK 21.

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) tool to migrate Java applications from Red Hat build of OpenJDK version 8, 11, or 17 to Red Hat build of OpenJDK 21.

Additional resources

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