Java Platform, Standard Edition

Oracle JDK 9 Migration Guide

Release 9

E82965-05

October 2017

Migrating to JDK 9

The purpose of this guide is to help you identify potential issues and give you suggestions on how to proceed as you migrate your existing Java application to JDK 9.

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

To migrate your application, start by following the steps listed in Prepare for Migration.

Then, look at the list of changes that you may encounter as you run your application.

- New Version-String Scheme
- Understanding Runtime Access Warnings
- Changes to the Installed JDK/JRE Image
- Removed or Changed APIs
- Modules Shared with Java EE Not Resolved by Default
- Deployment
- Security Updates
- Changes to Garbage Collection
- Removed Tools and Components
- Removed macOS-Specific Features

Finally, when your application is running successfully on JDK 9, review Looking Forward, which will help you avoid problems with future releases.

This guide focuses on changes required to make your code run on JDK 9. For a comprehensive list of all of the new features of JDK 9, see *Java Platform, Standard Edition What's New in JDK 9* For detailed information about the changes mentioned in this guide, refer to JDK 9 Release Notes.

Prepare for Migration

The following sections will help you successfully migrate your application:

- Download JDK 9
- Run Your Program Before Recompiling
- Update Third-Party Libraries
- Compile Your Application if Needed
- Run jdeps on Your Code

Download JDK 9

Download and install the JDK 9 release.

Run Your Program Before Recompiling

Try running your application on JDK 9. Most code and libraries should work on JDK 9 without any changes, but there may be some libraries that need to be upgraded.

Note:

Migrating is an iterative process. You'll probably find it best to try running your program (this task) first, then complete these three tasks more or less in parallel:

- Update Third-Party Libraries
- Compile Your Application if Needed
- Run jdeps on Your Code.

When you run your application, look for warnings from the JVM about obsolete VM options. If the VM fails to start, then look for removed options, because some VM flags that were deprecated in JDK 8 have been removed in JDK 9. See Removed GC Options.

If your application starts successfully, look carefully at your tests and ensure that the behavior is the same as on JDK 8. For example, a few early adopters have noticed that their dates and currencies are formatted differently. See Use CLDR Locale Data by Default.

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

Update Third-Party Libraries

For every tool and third-party library that you use, you may need to have an updated version that supports JDK 9.

Check the websites for your third-party libraries and your tool vendors for a version of each library or tool that's designed to work on JDK 9. If one exists, then download and install the new version.

If you use Maven or Gradle to build your application, then make sure to upgrade to a more recent version that supports JDK 9.

If you use an IDE to develop your applications, then it can help to migrate existing code. The NetBeans, Eclipse, and IntelliJ IDEs all have versions available that include JDK 9 support.

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

Compile Your Application if Needed

Compiling your code with the JDK 9 compiler will ease migration to future releases since the code may depend on APIs and features which have been identified as problematic. However, it is not strictly necessary.

If you need to compile your code with the JDK 9 compiler then take note of the following:

- If you use the underscore character ("_") as a one-character identifier in source code, then your code won't compile in JDK 9. Its use generates a warning in JDK 8, and an error in JDK 9.

  As an example:

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

  This code generates the following error message from the compiler:

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

- If you use the `-source` and `-target` options with `javac`, then check the values that you use. In JDK 9, `javac` uses a "one plus three back" policy of supporting `-source` and `-target` options.

  The supported `-source/-target` values are 9 (the default), 8, 7, and 6 (6 is deprecated, and a warning is displayed when this value is used).

  In JDK 8, `-source` and `-target` values of 1.5/5 and earlier were deprecated and caused a warning to be generated. In JDK 9, those values cause an error.

  ```
  >javac -source 5 -target 5 Sample.java
  warning: [options] bootstrap class path not set in conjunction with -source 1.5
  error: Source option 1.5 is no longer supported. Use 1.6 or later.
  error: Target option 1.5 is no longer supported. Use 1.6 or later.
  ```

  If possible, use the new `--release` flag instead of the `-source` and `-target` options. The `--release N` flag is conceptually a macro for:

  ```
  -source N -target N -bootclasspath $PATH_TO_rt.jar_FOR_RELEASE_N
  ```

  The valid arguments for the `--release` flag follow the same policy as for `-source` and `-target`, one plus three back.

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

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

- Critical internal JDK APIs such as sun.misc.Unsafe are still accessible in JDK 9, but most of the JDK's internal APIs are not accessible at compile time. You may get compilation errors that indicate that your application or its libraries are dependent on internal APIs.

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

  You may use the --add-exports option as a temporary workaround to compile source code with references to JDK internal classes.

- You may see more deprecation warnings than previously. If you see deprecation with removal warnings, then you should address those to avoid future problems.
- A number of small changes have been made to javac to align it with the Java SE 9 Language Specification.
- You may encounter some source compatibility issues when recompiling.

The JDK 9 Release Notes contains details of changes to the javac compiler and source compatibility issues in JDK 9.

Run jdeps on Your Code

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

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

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

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

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

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

For jdeps syntax, see jdeps in the *Java Platform, Standard Edition Tools Reference*.

Keep in mind that `jdeps` is a static analysis tool, and static analysis of code doesn't tell you everything. If the code uses reflection to call an internal API, then `jdeps` doesn't warn you.

New Version-String Scheme

JDK 9 provides a new simplified version-string format. If your code relies on the version-string format to distinguish major, minor, security, and patch update releases, then you may need to update it.

The format of the new version-string is:

```
$MAJOR.$MINOR.$SECURITY.$PATCH
```

For example, under the old scheme, the Java `9u5` security release would have the version string `1.9.0_5-b20`.

Under the new scheme, the short version of the same release is `9.0.1`, and the long version is `9.0.1+20`.

This change affects `java -version` and related system properties, such as `java.runtime.version`, `java.vm.version`, `java.specification.version`, and `java.vm.specification.version`.

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

See Version String Format in *Java Platform, Standard Edition Installation Guide*, and JEP 223: New Version-String Scheme.

Understanding Runtime Access Warnings

Some tools and libraries use reflection to access parts of the JDK that are meant for internal use only. This illegal reflective access will be disabled in a future release of the JDK. In JDK 9, it is permitted by default and a warning is issued.

For example, here is the warning issued when starting Jython:

```
>java -jar jython-standalone-2.7.0.jar
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/C:/Jython/
jython2.7.0/jython-standalone-2.7.0.jar) to method sun.nio.ch.SelChImpl.getFD()
WARNING: Please consider reporting this to the maintainers of
jnr.posix.JavaLibCHelper
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective
access operations
WARNING: All illegal access operations will be denied in a future release
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
```

If you see a warning like this, contact the maintainers of the tool or library. The second line of the warning names the exact JAR file whose code used reflection to access an internal part of the JDK.

By default, a maximum of one warning about reflective access is issued in the lifetime of the process started by the `java` launcher. The exact timing of the warning depends on the behavior of tools and libraries performing reflective–access operations. The warning may appear early in the lifetime of the process, or a long time after startup.

You can disable the warning message on a library-by-library basis by using the `--add-opens` command line flag. For example, you can start Jython in the following way:

```
>java --add-opens java.base/sun.nio.ch=ALL-UNNAMED --add-opens java.base/java.io=ALL-
UNNAMED -jar jython-standalone-2.7.0.jar
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
```

This time, the warning is not issued because the `java` invocation explicitly acknowledges the reflective access. As you can see, you may need to specify multiple `--add-opens` flags to cover all of the reflective access operations that are attempted by libraries on the class path.

To better understand the behavior of tools and libraries, you can use the `--illegal-access=`**`warn`** command line flag. This flag causes a warning message to be issued for every illegal reflective-access operation. In addition, you can obtain detailed information about illegal reflective-access operations, including stack traces, by setting `--illegal-access=`**`debug`**.

If you have updated libraries, or when you get them, then you can experiment with using the `--illegal-access=`**`deny`** command line flag. It disables all reflective-access operations except for those enabled by other command-line options, such as `--add-opens`. This will be the default mode in a future release.

There are two options that allow you to break encapsulation in specific ways. You could use these in combination with `--illegal-access=`**`deny`**, or, as already mentioned, to suppress warnings.

- If you need to use an internal API that has been made inaccessible, then use the `--add-exports` runtime option. You can also use `--add-exports` at compile time to access internal APIs.
- If you have to allow code on the class path to do *deep reflection* to access nonpublic members, then use the `--add-opens` option.

If you want to suppress all reflective access warnings, then use the `--add-exports` and `--add-opens` options where needed.

--add-exports

If you must use an internal API that has been made inaccessible by default, then you can break encapsulation using the `--add-exports` command-line option.

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

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

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

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

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

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

This example allows code in all unnamed modules (code on the class path) to access the public members of public types in `java.management/sun.management`. If the code on the class path attempts to do *deep reflection* to access nonpublic members, then the code fails.

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

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

You can also break encapsulation with the JAR file manifest:

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

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

See also JEP 261.

--add-opens

If you have to allow code on the class path to do *deep reflection* to access nonpublic members, then use the `--add-opens` runtime option.

Some libraries do deep reflection, meaning `setAccessible(true)`, so they can access all members, including private ones. You can grant this access using the `--add-opens` option on the `java` command line. No warning messages are generated as a result of using this option.

If `--illegal-access=`**`deny`**, and you see `IllegalAccessException` or `InaccessibleObjectException` messages at runtime, you could use the `--add-opens` runtime option, basing the arguments upon the information shown in the exception message.

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

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

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

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

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

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

Note:

If you are using the JNI Invocation API, including, for example, a Java Web Start JNLP file, you must include an equals sign between `--add-opens` and its value.

```
<j2se version="9" java-vm-args="--add-opens=module/package=ALL-UNNAMED"  />
```

That equals sign is optional on the command line.

Changes to the Installed JDK/JRE Image

Significant changes have been made to the JDK and JRE.

Changed JDK and JRE Layout

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

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

In JDK 9, the JDK and JRE are two types of modular runtime images, where each contains the following directories:

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

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

See JEP 220: Modular Run-Time Images and Installed Directory Structure of JDK and JRE in *Java Platform, Standard Edition Installation Guide*.

New Class Loader Implementations

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

- The application class loader is no longer an instance of `URLClassLoader` but, rather, of an internal class. It is the default loader for classes in modules that are neither Java SE nor JDK modules.
- The extension class loader has been renamed; it is now the platform class loader. All classes in the Java SE Platform are guaranteed to be visible through the platform class loader. In addition, the classes in modules that are standardized under the Java Community Process but not part of the Java SE Platform are guaranteed to be visible through the platform class loader.

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

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

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

See JDK 9 Release Notes.

Removed rt.jar and tools.jar

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

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

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

  For example:

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

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

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

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

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

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

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

Removed Extension Mechanism

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

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

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

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

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

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

See JEP 220: Modular Run-Time Images.

Removed Endorsed Standards Override Mechanism

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

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

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

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

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

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

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

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

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

See JEP 220: Modular Run-Time Images.

Windows Registry Key Changes

The JRE installer creates keys in the Windows registry that have a different name than in previous releases. These changes may impact existing applications that use these keys.

If your application searches the registry for the installed version of Java it may fail, as the `Java Runtime Environment` key, which is created in Java SE 8 and previous releases, is abbreviated to `JRE` in Java SE 9.

The Java 9 installer creates these Windows registry keys when installing the JRE:

- `“HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JRE”`
- `“HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JRE\9”`
- `"HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\JRE"`
  `"@CurrentVersion"=9`

If the JDK is installed, the `JDK` key replaces the `JRE` key in the above example.

The Java 8u144 installer creates these Windows registry keys when installing the JRE:

- `“HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment”`
- `“HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment\1.8”`
- `“HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment\1.8.0_144”`
- `"HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment"`
  `"@CurrentVersion"=1.8`

If the JDK is installed, the `Java Development Kit` key replaces the `Java Runtime Environment` key in the above example.

If there are two versions of JDK or JRE installed, one with the new version-string format introduced in JDK 9, and the other with the older version format, then there will be two different Java-related `CurrentVersion` strings in the registry. One indicates the highest version number prior to JDK 9, and the other indicates the highest version number since JDK 9.

Removed or Changed APIs

This section highlights APIs that have been made inaccessible, removed, or altered in their default behavior. You may encounter the issues described in this section when compiling or running your application.

Removed java.* APIs

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

These include:

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

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

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

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

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

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

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

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

- `sun.misc.Unsafe`

The functionality of many of the methods in this class is available by using variable handles, see JEP 193: Variable Handles.

- `sun.reflect.Reflection::getCallerClass(int)`
  Instead, use the stack-walking API, see JEP 259: Stack-Walking API.

See JEP 260: Encapsulate Most Internal APIs.

java.awt.peer Not Accessible

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

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

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

- To see if a peer has been set yet:

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

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

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

- To test if a component is lightweight:

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

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

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

Removed com.sun.image.codec.jpeg Package

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

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

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

Removed Tools Support for Compact Profiles

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

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

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

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

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

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

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

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

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

See JEP 200: The Modular JDK.

Use CLDR Locale Data by Default

In JDK 9, the Unicode Consortium's Common Locale Data Repository (CLDR) data is enabled as the default locale data, so that you can use standard locale data without any further action.

In JDK 8, although CLDR locale data is bundled with the JRE, it isn't enabled by default.

Code that uses locale-sensitive services such as date, time, and number formatting may produce different results with the CLDR locale data. Remember that even `System.out.printf()` is locale-aware.

To enable behavior compatible with JDK 8, set the system property `java.locale.providers` to a value with `COMPAT` ahead of `CLDR`, for example, `java.locale.providers=COMPAT,CLDR`.

See CLDR Locale Data Enabled by Default in the *Java Platform, Standard Edition Internationalization Guide* and JEP 252: Use CLDR Locale Data by Default.

Modules Shared with Java EE Not Resolved by Default

In JDK 9, the modules that contain CORBA or the APIs shared between Java SE and Java EE are not resolved by default when you compile or run code on the class path. These are:

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

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

The code for these APIs was not removed in JDK 9, although the modules are deprecated for removal. The policy of not resolving these modules is a first step toward removing these APIs from Java SE and the JDK in a future release.

The migration options for libraries or applications that use these APIs are:

1. Use the `--add-modules` command-line option to ensure that the module with the API is resolved at startup. For example, specify `--add-module java.xml.bind` to ensure that the `java.xml.bind` module is resolved. This allows existing code that uses the JAXB API and implementation to work as it did in JDK 8.

   This is a temporary workaround because eventually these modules will be removed from the JDK.

   Using `--add-modules java.se.ee` or `--add-modules ALL-SYSTEM` as a workaround is not recommended. These options will resolve all Java EE modules, which is problematic in environments that are also using the standalone versions of APIs.

2. Deploy the standalone version of the API (and implementation if needed) on the class path. Each of the Java EE APIs are standalone technologies with projects published in Maven Central.

3. Deploy the standalone version of these modules on the upgrade module path. The standalone versions are provided by the Java EE project.

Deployment

This section includes issues that appear only when deploying an application.

Removed Launch-Time JRE Version Selection

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

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

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

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

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

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

Removed Support for Serialized Applets

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

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

Instead of serializing applets, use standard deployment strategies.

JNLP Specification Update

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

JNLP has been updated as follows:

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

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

2. Comparing numeric version element types against nonnumeric version element types.

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

   In JDK 9, if the element that can be parsed as an `int` is a shorter string than the other element, it will be padded with leading zeros before being compared lexicographically by ASCII value. This ensures there can be no circularity.

   In the case where both version comparisons and a JNLP servlet are used, you should use only numeric values to represent versions.

3. Component extensions with nested resources in `java` (or `j2se`) elements.
   This is permitted in the specification. It was previously supported, but this support wasn't reflected in the specification.

4. FX XML extension.
   The JNLP specification has been enhanced to add a `type` attribute to `application-desc` element, and add the subelement `param` in `application-desc` (as it already is in `applet-desc`).

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

See the JNLP specification updates at JSR-056.

Security Updates

Some security-related defaults have changed in JDK 9.

JCE Jurisdiction Policy File Default is Unlimited

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

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

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

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

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

Create PKCS12 Keystores

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

See Creating a Keystore to Use with JSSE in *Java Platform, Standard Edition Security Developer's Guide* and keytool in *Java Platform, Standard Edition Tools Reference*.

Changes to Garbage Collection

This section includes changes to garbage collection in JDK 9.

Make G1 the Default Garbage Collector

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

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

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

Removed GC Options

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

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

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

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

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

Removed Permanent Generation

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

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

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

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

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

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

Changes to GC Log Output

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

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

See Enable Logging with the JVM Unified Logging Framework in the *Java Platform, Standard Edition Tools Reference* and JEP 271: Unified GC Logging.

Removed Tools and Components

This list includes tools and components that are no longer bundled with JDK 9.

Removed JavaDB

JavaDB, which was a rebranding of Apache Derby, isn't included in JDK 9.

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

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

Removed the JVM TI hprof Agent

The `hprof` agent library has been removed.

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

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

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

- jmap: `jmap -dump`. See `jmap`.

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

Note:

Java Flight Recorder requires a commercial license for use in production. To learn more about commercial features and how to enable them, visit http://www.oracle.com/technetwork/java/javaseproducts/.

See JEP 240: Remove the JVM TI hprof Agent.

Removed the jhat Tool

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

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

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

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

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

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

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

Removed Support for the IIOP Transport from the JMX RMIConnector

The IIOP transport support from the JMX RMI Connector along with its supporting classes have been removed in JDK 9.

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

Public API changes include:

- The `javax.management.remote.rmi.RMIIIOPServerImpl` class has been deprecated. Upon invocation, all its methods and constructors throw `java.lang.UnsupportedOperationException` with an explanatory message.

- Two classes, `org.omg.stub.javax.management.rmi._RMIConnection_Stub`, and `org.omg.stub.javax.management.rmi._RMIConnection_Tie`, aren't generated.

Dropped Windows 32–bit Client VM

In JDK 9, the Windows 32–bit client VM is not available. Only a server VM is offered.

JDK 8 and earlier releases offered both a client JVM and a server JVM for Windows 32-bit systems. JDK 9 offers only the server JVM. The server JVM is tuned to maximize peak operating speed.

Removed Java VisualVM

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

Java VisualVM isn't bundled with JDK 9. If you would like to use VisualVM with JDK 9, then you can get it from the VisualVM open source project site.

Removed native2ascii Tool

The `native2ascii` tool is removed in JDK 9. Because JDK 9 supports UTF-8 based properties resource bundles, the conversion tool for UTF-8 based properties resource bundles to ISO-8859-1 is no longer needed.

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

Removed macOS-Specific Features

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

Platform-Specific Desktop Features

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

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

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

See JEP 272: Platform-Specific Desktop Features.

Removed AppleScript Engine

The AppleScript engine, a platform-specific `javax.script` implementation, has been removed without any replacement in JDK 9.

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

Looking Forward

After you have your application working on JDK 9, here are some suggestions that can help you get the most from the Java SE Platform:

- Read *Java Platform, Standard Edition What's New in JDK 9* to learn about new features of JDK 9.
- If needed, cross-compile to an older release of the platform using the new `–release` flag in the `javac` tool.
- Take advantage of your IDE's suggestions for updating your code with the latest features.
- Find out if your code is using deprecated APIs by running the static analysis tool `jdeprscan`. As already mentioned in this guide, APIs can be removed from the JDK, but only with advance notice.
- Get familiar with new features like multi-release JAR files (see `jar`) .

Documentation Accessibility

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

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Java Platform, Standard Edition Oracle JDK 9 Migration Guide, Release 9
E82965-05