Java Platform, Standard Edition
Tools Reference

Release 9
E61612-05
October 2017

Java Platform, Standard Edition Tools Reference, Release 9

E61612-05

Contents

Preface

The Java Platform, Standard Edition (Java SE) Command Reference describes the valid options and arguments for Java SE commands. In many cases, examples are included to show correct usage.

Audience

This document is intended for Java SE developers who want to use the tools and commands provided in JDK 9.

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.

Related Documents

For more information, see the documents in the Oracle JDK 9 Documentation

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Tools and Commands Reference

The JDK tools and their commands enable developers to handle development tasks such as compiling and running a program, packaging source files into a Java Archive (JAR) file, applying security policies to a JAR file, and more.

The tools and commands reference topic lists and describes the Java Development Kit (JDK) tools. They're grouped into the following sections based on the related functions that they perform. Details about the tools and the commands that you use to run them are contained in the corresponding sections of this guide.

Main Tools

The following foundation tools and commands let you create and build applications:

- **javac**: You can use the `javac` tool and its options to read Java class and interface definitions and compile them into bytecode and class files.
- **javap**: You use the `javap` command to disassemble one or more class files.
- **javah**: You use the `javah` tool to generate C header and source files from a Java class.
- **javadoc**: You use the `javadoc` tool and its options to generate HTML pages of API documentation from Java source files.
- **java**: You can use the `java` command to launch a Java application.
- **appletviewer**: You use the `appletviewer` command to launch the AppletViewer and run applets outside of a web browser.
- **jar**: You can use the `jar` command to create an archive for classes and resources, and to manipulate or restore individual classes or resources from an archive.
- **jlink**: You can use the `jlink` tool to assemble and optimize a set of modules and their dependencies into a custom runtime image.
- **jmod**: You use the `jmod` tool to create JMOD files and list the content of existing JMOD files.
- **jdeps**: You use the `jdeps` command to launch the Java class dependency analyzer.
- **jdeprscan**: You use the `jdeprscan` tool as a static analysis tool that scans a jar file (or some other aggregation of class files) for uses of deprecated API elements.

Language Shell

The following tool gives you an interactive environment for trying out the Java language:

- **jshell**: You use the `jshell` tool to interactively evaluate declarations, statements, and expressions of the Java programming language in a read-eval-print loop (REPL).

Security Tools

The following security tools set security policies on your system and create applications that can work within the scope of security policies set at remote sites:

- **keytool**: You use the `keytool` command and options to manage a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates.
- **jarsigner**: You use the `jarsigner` tool to sign and verify Java Archive (JAR) files.
- **policytool**: You use `policytool` to read and write a plain text policy file based on user input through the utility GUI.

The following tools obtain, list, and manage Kerberos tickets on Windows:

- **kinit**: You use the `kinit` tool and its options to obtain and cache Kerberos ticket-granting tickets.
- **klist**: You use the `klist` tool to display the entries in the local credentials cache and key table.
- **ktab**: You use the `ktab` tool to manage the principal names and service keys stored in a local key table.

Remote Method Invocation (RMI) Tools

The following tools enable creating applications that interact over the Web or other network:

- **rmic**: You use the `rmic` compiler to generate stub and skeleton class files using the Java Remote Method Protocol (JRMP) and stub and tie class files (IIOP protocol) for remote objects.
- **rmiregistry**: You use the `rmiregistry` command to create and start a remote object registry on the specified port on the current host.
- **rmid**: You use the `rmid` command to start the activation system daemon that enables objects to be registered and activated in a Java Virtual Machine (JVM).
- **serialver**: You use the `serialver` command to return the `serialVersionUID` for one or more classes in a form suitable for copying into an evolving class.

Java IDL and RMI-IIOP Tools

The following tools enable creating applications that use OMG-standard IDL and CORBA/IIOP:

- **tnameserv**: You use the `tnameserv` command as a substitute for Object Request Broker Daemon (ORBD).
- **idlj**: You use the `idlj` command to generate Java bindings for a specified Interface Definition Language (IDL) file.
- **orbd**: You use the `orbd` command for the client to transparently locate and call persistent objects on servers in the CORBA environment.
- **servertool**: You use the `servertool` command-line tool to register, unregister, start up, and shut down a persistent server.

Java Deployment Tools

The following utilities let you deploy Java applications and applets on the web:

- **pack200**: You use the `pack200` command to transform a Java Archive (JAR) file into a compressed pack200 file with the Java gzip compressor.
- **unpack200**: You use the `unpack200` command to transform a packed file into a JAR file for web deployment.
- **javapackager**: You use the `javapackager` command to perform tasks related to packaging Java and JavaFX applications.

Java Web Start

The following utility launches Java Web Start applications:

- **javaws**: You use the `javaws` tool command and its options to start Java Web Start.

Monitoring Tools

The following tools let you monitor performance statistics:

> **Note:**
>
> The following tools that are identified as **experimental** are unsupported and should be used with that understanding. They may not be available in future JDK versions.

- **jconsole**: You use the `jconsole` command to start a graphical console to monitor and manage Java applications.
- **jps**: **Experimental** You use the `jps` command to list the instrumented JVMs on the target system.
- **jstat**: **Experimental** You use the `jstat` command to monitor JVM statistics. This command is experimental and unsupported.
- **jstatd**: **Experimental** You use the `jstatd` command to monitor the creation and termination of instrumented Java HotSpot VMs. This command is experimental and unsupported.
- **jmc**: You use the `jmc` command and its options to launch Java Mission Control. Java Mission Control is a profiling, monitoring, and diagnostics tools suite.

Java Web Services Tools

The following tools let you create applications that provide web services:

- **schemagen**: You can use the `schemagen` tool and commands to generate a schema for every namespace that's referenced in your Java classes.
- **wsgen**: You use the `wsgen` command to generate Java API for XML Web Services (JAX-WS) portable artifacts used in JAX-WS web services.
- **wsimport**: You use the `wsimport` command to generate Java API for XML Web Services (JAX-WS) portable artifacts.
- **xjc**: You use the `xjc` shell script to compile an XML schema file into fully annotated Java classes.

Java Accessibility Utilities

The following utilities let you check the accessibility of Java objects:

- **jaccessinspector**: You use the `jaccessinspector` accessibility evaluation tool for the Java Accessibility Utilities API to examine accessible information about the objects in the Java Virtual Machine.
- **jaccesswalker**: You use the `jaccesswalker` to navigate through the component trees in a particular Java Virtual Machine and presents the hierarchy in a tree view.

Troubleshooting Tools

The following tools let you perform specific troubleshooting tasks:

> **Note:**
>
> The following tools that are identified as **experimental** are unsupported and should be used with that understanding. They may not be available in future JDK versions. Some of these tools aren't currently available on Windows platforms.

- **jcmd**: You use the `jcmd` utility to send diagnostic command requests to a running Java Virtual Machine (JVM).
- **jdb**: You use the `jdb` command and it's options to find and fix bugs in Java platform programs.
- **jhsdb**: You use the `jhsdb` tool to attach to a Java process or to launch a postmortem debugger to analyze the content of a core dump from a crashed Java Virtual Machine (JVM).
- **jinfo**: **Experimental** You use the `jinfo` command to generate Java configuration information for a specified Java process. This command is experimental and unsupported.
- **jmap**: **Experimental** You use the `jmap` command to print details of a specified process. This command is experimental and unsupported.
- **jstack**: **Experimental** You use the `jstack` command to print Java stack traces of Java threads for a specified Java process. This command is experimental and unsupported.

Scripting Tools

The following tools let you run scripts that interact with the Java platform:

> **Note:**
>
> The following tools identified that are **experimental** are unsupported and should be used with that understanding. They may not be available in future JDK versions.

- **jjs**: You use the `jjs` command-line tool to invoke the Nashorn engine.
- **jrunscript**: **Experimental** You use the `jrunscript` command to run a command-line script shell that supports interactive and batch modes.

2

Main Tools to Create and Build Applications

You can use the foundation JDK tools and commands to create and build applications.

The following sections describe the tools and commands that you can use to create and build applications:

- javac: You can use the `javac` tool and its options to read Java class and interface definitions and compile them into bytecode and class files.
- javap: You use the `javap` command to disassemble one or more class files.
- javah: You use the `javah` tool to generate C header and source files from a Java class.
- javadoc: You use the `javadoc` tool and its options to generate HTML pages of API documentation from Java source files.
- java: You can use the `java` command to launch a Java application.
- appletviewer: You use the `appletviewer` command to launch the AppletViewer and run applets outside of a web browser.
- jar: You can use the `jar` command to create an archive for classes and resources, and to manipulate or restore individual classes or resources from an archive.
- jlink: You can use the `jlink` tool to assemble and optimize a set of modules and their dependencies into a custom runtime image.
- jmod: You use the `jmod` tool to create JMOD files and list the content of existing JMOD files.
- jdeps: You use the `jdeps` command to launch the Java class dependency analyzer.
- jdeprscan: You use the `jdeprscan` tool as a static analysis tool that scans a jar file (or some other aggregation of class files) for uses of deprecated API elements.

javac

You can use the `javac` tool and its options to read Java class and interface definitions and compile them into bytecode and class files.

Synopsis

```
javac [ options ] [ sourcefiles ]
```

options
Command-line options. See Overview of javac Options.

sourcefiles
One or more source files to be compiled (such as `MyClass.java`) or processed for annotations (such as `MyPackage.MyClass`).

Description

The `javac` command reads class and interface definitions, written in the Java programming language, and compiles them into bytecode class files. The `javac` command can also process annotations in Java source files and classes.

In JDK 9, a new launcher environment variable, `JDK_JAVAC_OPTIONS`, has been introduced that prepends its content to the command line to `javac` . See Using JDK_JAVAC_OPTIONS Environment Variable.

There are two ways to pass source code file names to `javac`.

- For a small number of source files, you can list the file names on the command line.
- For a large number of source files, you can use the *`@filename`* option on the `javac` command line to include a file that lists the source file names. See Standard Options for javac for a description of the option and javac Command-Line Argument Files for a description of `javac` argument files.

Source code file names must have `.java` suffixes, class file names must have `.class` suffixes, and both source and class files must have root names that identify the class. For example, a class called `MyClass` would be written in a source file called `MyClass.java` and compiled into a bytecode class file called `MyClass.class`.

Inner class definitions produce additional class files. These class files have names that combine the inner and outer class names, such as `MyClass$MyInnerClass.class`.

You should arrange the source files in a directory tree that reflects their package tree. For example:

- **Oracle Solaris, Linux, and OS X:** If all of your source files are in `/workspace`, then put the source code for `com.mysoft.mypack.MyClass` in `/workspace/com/mysoft/mypack/MyClass.java`.
- **Windows:** If all of your source files are in `\workspace`, then put the source code for `com.mysoft.mypack.MyClass` in `\workspace\com\mysoft\mypack\MyClass.java`.

By default, the compiler puts each class file in the same directory as its source file. You can specify a separate destination directory with the `-d` option described in Standard Options for javac.

Programmatic Interface

The `javac` command supports the new Java Compiler API defined by the classes and interfaces in the `javax.tools` package.

Implicitly Loaded Source Files

To compile a set of source files, the compiler might need to implicitly load additional source files. See Searching for Types. Such files are currently not subject to annotation processing. By default, the compiler gives a warning when annotation processing occurs and any implicitly loaded source files are compiled. The `-implicit` option provides a way to suppress the warning.

Using JDK_JAVAC_OPTIONS Environment Variable

The content of the `JDK_JAVAC_OPTIONS` environment variable, separated by white-spaces ( ) or white-space characters (`\n`, `\t`, `\r`, or `\f`) is prepended to the command line arguments passed to `javac` as a list of arguments.

The encoding requirement for the environment variable is the same as the `javac` command line on the system. `JDK_JAVAC_OPTIONS` environment variable content is treated in the same manner as that specified in the command line.

Single (') or double (") quotes can be used to enclose arguments that contain whitespace characters. All content between the open quote and the first matching close quote are preserved by simply removing the pair of quotes. In case a matching quote is not found, the launcher will abort with an error message. *`@files`* are supported as they are specified in the command line. However, as in *`@files`*, use of a wildcard is not supported.

Examples of quoting arguments containing white spaces:

```
export JDK_JAVAC_OPTIONS=‘@"C:\white spaces\argfile”'

export JDK_JAVAC_OPTIONS=‘"@C:\white spaces\argfile”'

export JDK_JAVAC_OPTIONS='@C:\"white spaces"\argfile'
```

Overview of javac Options

The compiler has sets of standard options, and cross-compilation options that are supported on the current development environment. The compiler also has a set of nonstandard options that are specific to the current virtual machine and compiler implementations but are subject to change in the future. The nonstandard options begin with `-X` . The different sets of `javac` options are described in the following sections:

- Standard Options for javac
- Cross-Compilation Options for javac
- Extra Options for javac

Standard Options for javac

`@filename`
Reads options and file names from a file. To shorten or simplify the `javac` command, you can specify one or more files that contain arguments to the `javac` command (except `-J` options). This let’s you to create `javac` commands of any length on any operating system. See javac Command-Line Argument Files.

`-Akey[=`*`value`*`]`
Specifies options to pass to annotation processors. These options aren’t interpreted by `javac` directly, but are made available for use by individual processors. The `key` value should be one or more identifiers separated by a dot (.).

`--add-modules` ***`module , module`***
Specifies root modules to resolve in addition to the initial modules, or all modules on the module path if *`module`* is `ALL-MODULE-PATH`.

`--boot-class-path` *`path`* or `-bootclasspath` *`path`*
Overrides the location of the bootstrap class files.

> **Note:**
>
> This option is not supported when using `--release` *`release`* to compile for JDK 9. See the description of `--release` *`release`* for details about compiling for versions other than JDK 9.

`--class-path` *`path`* , `-classpath` *`path`*, or `-cp` *`path`*
Specifies where to find user class files and annotation processors. This class path overrides the user class path in the `CLASSPATH` environment variable.

- If `--class-path`, `-classpath`, or `-cp` aren't specified, then the user class path is the current directory.
- If the `-sourcepath` option isn't specified, then the user class path is also searched for source files.
- If the `-processorpath` option isn't specified, then the class path is also searched for annotation processors.

`-d` *`directory`*
Sets the destination directory for class files. If a class is part of a package, then `javac` puts the class file in a subdirectory that reflects the package name and creates directories as needed. For example:

- **Oracle Solaris, Linux, and OS X:** If you specify `-d /home/myclasses` and the class is called `com.mypackage.MyClass`, then the class file is `/home/myclasses/com/mypackage/MyClass.class`.
- **Windows:** If you specify `-d C:\myclasses` and the class is called `com.mypackage.MyClass`, then the class file is `C:\myclasses\com\mypackage\MyClass.class`.

If the `-d` option isn't specified, then `javac` puts each class file in the same directory as the source file from which it was generated.

> **Note:**
>
> The directory specified by the `-d` option isn't automatically added to your user class path.

`-deprecation`
Shows a description of each use or override of a deprecated member or class. Without the `-deprecation` option, `javac` shows a summary of the source files that use or override deprecated members or classes. The `-deprecation` option is shorthand for `-Xlint:deprecation`.

`-encoding` *`encoding`*
Specifies character encoding used by source files, such as EUC-JP and UTF-8. If the `-encoding` option isn't specified, then the platform default converter is used.

`-endorseddirs` *`directories`*
Overrides the location of the endorsed standards path.

> **Note:**
>
> This option is not supported when using `--release` *`release`* to compile for JDK 9. See the description of `--release` *`release`* for details about compiling for versions other than JDK 9.

`-extdirs` *`directories`*
Overrides the location of the installed extensions. The *`directories`* variable is a colon-separated list of directories. Each JAR file in the specified directories is searched for class files. All JAR files found become part of the class path.
If you are cross-compiling, then this option specifies the directories that contain the extension classes. See Cross-Compilation Options for javac.

> **Note:**
>
> This option is not supported when using `--release` *`release`* to compile for JDK 9. See the description of `--release` *`release`* for details about compiling for versions other than JDK 9.

`-g`
Generates all debugging information, including local variables. By default, only line number and source file information is generated.

`-g:[lines, vars, source],[lines, vars, source],[lines, vars, source]`
Generates only the kinds of debugging information specified by the comma-separated list of keywords. Valid keywords are:

`lines`
Line number debugging information.

`vars`
Local variable debugging information.

`source`
Source file debugging information.

`-g:none`
Doesn't generate any debugging information.

`-h` *`directory`*
Specifies where to place generated native header files.
When you specify this option, a native header file is generated for each class that contains native methods or that has one or more constants annotated with the `java.lang.annotation.Native` annotation. If the class is part of a package, then the compiler puts the native header file in a subdirectory that reflects the package name and creates directories as needed.

`--help` or `-help`
Prints a synopsis of the standard options.

`--help-extra` or `-X`
Prints the help for extra options.

`-implicit:{[none, class],[none, class]}`
Specifies whether or not to generate class files for implicitly referenced files:

- `-implicit:class` — Automatically generates class files.
- `-implicit:none` — Suppresses class file generation.

If this option isn't specified, then the default automatically generates class files. In this case, the compiler issues a warning if any class files are generated when also doing annotation processing. The warning isn't issued when the `-implicit` option is explicitly set. See Searching for Types.

`-J`*`option`*
Passes *`option`* to the runtime system, where *`option`* is one of the Java options described on javacommand. For example, `-J-Xms48m` sets the startup memory to 48 MB.

> **Note:**
>
> The `CLASSPATH` environment variable, `-classpath` option, `-bootclasspath` option, and `-extdirs` option don't specify the classes used to run `javac`. Trying to customize the compiler implementation with these options and variables is risky and often doesn't accomplish what you want. If you must customize the complier implementation, then use the `-J` option to pass options through to the underlying Java launcher.

`--limit-modules` *`module , module*`*
Limits the universe of observable modules.

`--module` *`module-name`* or `-m` *`module-name`*
Compiles only the specified module and checks time stamps.

`--module-path` *`path`* or `-p` *`path`*
Specifies where to find application modules.

`--module-source-path` *`module-source-path`*
Specifies where to find input source files for multiple modules.

`--module-version` *`version`*
Specifies the version of modules that are being compiled.

`-nowarn`
Disables warning messages. This option operates the same as the `-Xlint:none` option.

`-parameters`
Generates metadata for reflection on method parameters. Stores formal parameter names of constructors and methods in the generated class file so that the method

`java.lang.reflect.Executable.getParameters` from the Reflection API can retrieve them.

`-proc: [none, only] , [none, only]`
Controls whether annotation processing and compilation are done. `-proc:none` means that compilation takes place without annotation processing. `-proc:only` means that only annotation processing is done, without any subsequent compilation.

`-processor` *`class1`* `[,`*`class2`*`,`*`class3`*`...]`
Names of the annotation processors to run. This bypasses the default discovery process.

`--processor-module–path` *`path`* or `-p` *`module-path`*
Specifies the module path used for finding annotation processors.

`--processor–path` *`path`* or `-processorpath` *`path`*
Specifies where to find annotation processors. If this option isn't used, then the class path is searched for processors.

`-profile` *`profile`*
Checks that the API used is available in the specified profile.

> **Note:**
>
> Not supported when using `--release` *`release`* to compile for JDK 9. See the description of `--release` *`release`* for details about compiling for versions other than JDK 9.

`--release` *`release`*
Compiles against the public, supported and documented API for a specific VM version. Supported *`release`* targets are `6`, `7`, `8`, and `9`.

> **Note:**
>
> When using `--release` for a version of the Java Platform that supports modules, you can't use `--add-modules` to access internal JDK modules, nor can you use `--add-exports` to access internal JDK APIs in the modules.

`-s` *`directory`*
Specifies the directory used to place the generated source files. If a class is part of a package, then the compiler puts the source file in a subdirectory that reflects the package name and creates directories as needed. For example:

- **Oracle Solaris, Linux, and OS X:**: If you specify `-s /home/mysrc` and the class is called `com.mypackage.MyClass`, then the source file is put in `/home/mysrc/com/mypackage/MyClass.java`.
- **Windows**: If you specify `-s C:\mysrc` and the class is called `com.mypackage.MyClass`, then the source file is put in `C:\mysrc\com\mypackage\MyClass.java`.

`-source release`
Specifies the version of source code accepted. The following values for *`release`* are allowed:

> **Note:**
>
> As of JDK 9, the `javac` doesn't support `-source` release settings less than or equal to `5`. If settings less than or equal to `5` are used, then the `javac` command behaves as if `-source 6` were specified.

`1.6`
No language changes were introduced in Java SE 6. However, encoding errors in source files are now reported as errors instead of warnings as was done in earlier releases of Java Platform, Standard Edition.

`6`
Synonym for 1.6.

`1.7`
The compiler accepts code with features introduced in Java SE 7.

`7`
Synonym for 1.7.

`1.8`
The compiler accepts code with features introduced in Java SE 8.

`8`
Synonym for 1.8.

`9`
The default value. The compiler accepts code with features introduced in Java SE 9.

`--source-path path` or **`-sourcepath path`**
Specifies where to find input source files. This is the source code path used to search for class or interface definitions. As with the user class path, source path entries are separated by colons (:) on Oracle Solaris and semicolons(;) on Windows. They can be directories, JAR archives, or ZIP archives. If packages are used, then the local path name within the directory or archive must reflect the package name.

> **Note:**
>
> Classes found through the class path might be recompiled when their source files are also found. See Searching for Types.

`--system jdk | none`
Overrides the location of system modules.

`-target release`
Generates class files for a specific VM version.

`--upgrade-module–path` *`path`*
Overrides the location of upgradeable modules.

`-verbose`
Outputs messages about what the compiler is doing. Messages include information about each class loaded and each source file compiled.

`--version` or `-version`
Prints version information.

`-Werror`
Terminates compilation when warnings occur.

Cross-Compilation Options for javac

By default, for releases prior to JDK 9, classes were compiled against the bootstrap classes of the platform that shipped with the`javac` command. But `javac` also supports cross-compiling, in which classes are compiled against bootstrap classes of a different Java platform implementation. It's important to use the `-bootclasspath` and `-extdirs` options when cross-compiling.

> **Note:**
>
> Not supported when using `--release` *`release`* to compile for JDK 9. See the description of `--release` *`release`* for details about compiling for versions other than JDK 9.

Extra Options for javac

`--add-exports` *`module/package=other-module(,other-module)*`*
Specifies a package to be considered as exported from its defining module to additional modules or to all unnamed modules when the value of *`other-module`* is `ALL-UNNAMED`.

`--add-reads` *`module=other-module(,other-module)*`*
Specifies additional modules to be considered as required by a given module.

`-Djava.endorsed.dirs=`*`dirs`*
Overrides the location of the endorsed standards path.

> **Note:**
>
> Not supported when using `--release` *`release`* to compile for JDK 9. See the description of `--release` *`release`* for details about compiling for versions other than JDK 9.

`-Djava.ext.dirs=`*`dirs`*
Overrides the location of installed extensions.

> **Note:**
>
> This option is not supported when using `--release release` to compile for JDK 9. See the description of `--release release` for details about compiling for versions other than JDK 9.

`--doclint-format [html4|html5]`
Specifies the format for documentation comments.

`--patch-module module=file(:file)*`
Overrides or augments a module with classes and resources in JAR files or directories.

`-Xbootclasspath:path`
Overrides the location of the bootstrap class files.

> **Note:**
>
> This option is not supported when using `--release release` to compile for JDK 9. See the description of `--release release` for details about compiling for versions other than JDK 9.

`-Xbootclasspath/a:path`
Adds a suffix to the bootstrap class path.

> **Note:**
>
> This option is not supported when using `--release release` to compile for JDK 9. See the description of `--release release` for details about compiling for versions other than JDK 9.

`-Xbootclasspath/p:path`
Adds a prefix to the bootstrap class path.

> **Note:**
>
> This option is not supported when using `--release release` to compile for JDK 9. See the description of `--release release` for details about compiling for versions other than JDK 9.

`-Xdiags:[compact, verbose]`
Selects a diagnostic mode.

`-Xdoclint`
Enables recommended checks for problems in `javadoc` comments

`-Xdoclint:(all|none|[-]group) [/access]`
Enables or disables specific groups of checks, where *`group`* is one of the following values:

- `accessibility`
- `html`
- `missing`
- `reference`
- `syntax`

For more information about these groups of checks, see the `-Xdoclint` option of the `javadoc` command. The `-Xdoclint` option is disabled by default in the `javac` command. The variable *`access`* specifies the minimum visibility level of classes and members that the `-Xdoclint` option checks. It can have one of the following values (in order of most to least visible:)

- `public`
- `protected`
- `package`
- `private`

The default access level is `private`.
For example, the following option checks classes and members (with all groups of checks) that have the access level of protected and higher (which includes protected and public):

```
-Xdoclint:all/protected
```

The following option enables all groups of checks for all access levels, except it won't check for HTML errors for classes and members that have the access level of package and higher (which includes package, protected and public :)

```
-Xdoclint:all,-html/package
```

`Xdoclint/package:[-]packages(,[-]package)*`
Enables or disables checks in specific packages. Each *`package`* is either the qualified name of a package or a package name prefix followed by `.*`, which expands to all sub-packages of the given package. Each *`package`* can be prefixed with `-` to disable checks for a specified package or packages.

`-Xlint`
Enables all recommended warnings. In this release, enabling all available warnings is recommended.

`-Xlint:key(,key)*`
Supplies warnings to enable or disable, separated by comma. Precede a key by a hypen (-) to disable the specified warning.
Supported values for *`key`* are:

- `all` — Enables all warnings.
- `auxiliaryclass` — Warns about an auxiliary class that's hidden in a source file, and is used from other files.
- `cast` — Warns about the use of unnecessary casts.

- `classfile` — Warns about the issues related to classfile contents.
- `deprecation` — Warns about the use of deprecated items.
- `dep-ann` — Warns about the items marked as deprecated in `javadoc` but without the `@Deprecated` annotation.
- `divzero` — Warns about the division by the constant integer 0.
- `empty` — Warns about an empty statement after `if`.
- `exports` — Warns about the issues regarding module exports.
- `fallthrough` — Warns about the falling through from one case of a switch statement to the next.
- `finally` — Warns about `finally` clauses that don't terminate normally.
- `module` — Warns about the module system-related issues.
- `opens` — Warns about the issues related to module opens.
- `options` — Warns about the issues relating to use of command line options.
- `overloads` — Warns about the issues related to method overloads.
- `overrides` — Warns about the issues related to method overrides.
- `path` — Warns about the invalid path elements on the command l ine.
- `processing` — Warns about the issues related to annotation processing.
- `rawtypes` — Warns about the use of raw types.
- `removal` — Warns about the use of an API that has been marked for removal.
- `serial` — Warns about the serializable classes that don't provide a serial version ID. Also warns about access to non-public members from a serializable element.
- `static` — Warns about the accessing a static member using an instance.
- `try` — Warns about the issues relating to the use of try blocks (that is, try-with-resources).
- `unchecked` — Warns about the unchecked operations.
- `varargs` — Warns about the potentially unsafe `vararg` methods.
- `none` — Disables all warnings.

See Examples of Using -Xlint keys.

`-Xmaxerrs` *`number`*
Sets the maximum number of errors to print.

`-Xmaxwarns` *`number`*
Sets the maximum number of warnings to print.

`-Xpkginfo:[always, legacy, nonempty]`
Specifies when and how the `javac` command generates `package-info.class` files from `package-info.java` files using one of the following options:

`always`
Generates a `package-info.class` file for every `package-info.java` file. This option may be useful if you use a build system such as Ant, which checks that each `.java` file has a corresponding `.class` file.

legacy
Generates a `package-info.class` file only if `package-info.java` contains annotations. This option doesn't generate a `package-info.class` file if `package-info.java` contains only comments.

> **Note:**
>
> A `package-info.class` file might be generated but be empty if all the annotations in the `package-info.java` file have `RetentionPolicy.SOURCE`.

nonempty
Generates a `package-info.class` file only if `package-info.java` contains annotations with `RetentionPolicy.CLASS` or `RetentionPolicy.RUNTIME`.

`-Xplugin:`*`name args`*
Specifies the name and optional arguments for a plug-in to be run.

`-Xprefer:[source` or `newer]`
Specifies which file to read when both a source file and class file are found for an implicitly compiled class using one of the following options. See Searching for Types.

- `-Xprefer:newer` — Reads the newer of the source or class files for a type (default).
- `-Xprefer:source` — Reads the source file. Use `-Xprefer:source` when you want to be sure that any annotation processors can access annotations declared with a retention policy of `SOURCE`.

`-Xprint`
Prints a textual representation of specified types for debugging purposes. This doesn't perform annotation processing or compilation. The format of the output could change.

`-XprintProcessorInfo`
Prints information about which annotations a processor is asked to process.

`-XprintRounds`
Prints information about initial and subsequent annotation processing rounds.

`-Xstdout` *`filename`*
Sends compiler messages to the named file. By default, compiler messages go to `System.err`.

javac Command-Line Argument Files

An argument file can include `javac` options and source file names in any combination. The arguments within a file can be separated by spaces or new line characters. If a file name contains embedded spaces, then put the whole file name in double quotation marks.

File names within an argument file are relative to the current directory, not to the location of the argument file. Wildcards (*) aren't allowed in these lists (such as for specifying `*.java`). Use of the at sign (`@`) to recursively interpret files isn't supported. The `-J` options aren't supported because they're passed to the launcher, which doesn't support argument files.

When executing the `javac` command, pass in the path and name of each argument file with the at sign (@) leading character. When the `javac` command encounters an

argument beginning with the at sign (@), it expands the contents of that file into the argument list.

Examples of Using javac @filename

Single Argument File

You could use a single argument file named `argfile` to hold all `javac` arguments:

```
javac @argfile
```

This argument file could contain the contents of both files shown in Example 2.

Two Argument Files

You can create two argument files: one for the `javac` options and the other for the source file names. Note that the following lists have no line-continuation characters. Create a file named options that contains the following:
Oracle Solaris, Linux, and OS X::

```
-d classes
-g
-sourcepath /java/pubs/ws/1.3/src/share/classes
```

Windows:

```
-d classes
-g
-sourcepath C:\java\pubs\ws\1.3\src\share\classes
```

Create a file named classes that contains the following:

```
MyClass1.java
MyClass2.java
MyClass3.java
```

Then, run the `javac` command as follows:

```
javac @options @classes
```

Argument Files with Paths

The argument files can have paths, but any file names inside the files are relative to the current working directory (not `path1` or `path2`):

```
javac @path1/options @path2/classes
```

Examples of Using -Xlint keys

`cast`
Warns about unnecessary and redundant casts, for example:

```
String s = (String) "Hello!"
```

`classfile`
Warns about issues related to class file contents.

`deprecation`
Warns about the use of deprecated items. For example:

```
java.util.Date myDate = new java.util.Date();
int currentDay = myDate.getDay();
```

The method `java.util.Date.getDay` has been deprecated since JDK 1.1.

`dep-ann`
Warns about items that are documented with the `@deprecated` Javadoc comment, but don't have the `@Deprecated` annotation, for example:

```
/**
  * @deprecated As of Java SE 7, replaced by {@link #newMethod()}
  */
public static void deprecatedMethod() { }
public static void newMethod() { }
```

`divzero`
Warns about division by the constant integer 0, for example:

```
int divideByZero = 42 / 0;
```

`empty`
Warns about empty statements after `if` statements, for example:

```
class E {
    void m() {
         if (true) ;
    }
}
```

`fallthrough`
Checks the switch blocks for fall-through cases and provides a warning message for any that are found. Fall-through cases are cases in a switch block, other than the last case in the block, whose code doesn't include a break statement, allowing code execution to fall through from that case to the next case. For example, the code following the case 1 label in this switch block doesn't end with a break statement:

```
switch (x) {
case 1:
  System.out.println("1");
  // No break statement here.
case 2:
  System.out.println("2");
}
```

If the `-Xlint:fallthrough` option was used when compiling this code, then the compiler emits a warning about possible fall-through into case, with the line number of the case in question.

`finally`
Warns about `finally` clauses that can't be completed normally, for example:

```
public static int m() {
  try {
     throw new NullPointerException();
  }  catch (NullPointerException(); {
     System.err.println("Caught NullPointerException.");
     return 1;
   } finally {
     return 0;
   }
  }
```

The compiler generates a warning for the `finally` block in this example. When the `int` method is called, it returns a value of 0. A `finally` block executes when the `try` block exits. In this example, when control is transferred to the `catch` block, the `int` method exits. However, the `finally` block must execute, so it's executed, even though control was transferred outside the method.

`options`
Warns about issues that related to the use of command-line options. See Cross-Compilation Options for javac.

`overrides`
Warns about issues related to method overrides. For example, consider the following two classes:

```
public class ClassWithVarargsMethod {
  void varargsMethod(String... s) { }
}

public class ClassWithOverridingMethod extends ClassWithVarargsMethod {
   @Override
   void varargsMethod(String[] s) { }
}
```

The compiler generates a warning similar to the following:.

```
warning: [override] varargsMethod(String[]) in ClassWithOverridingMethod
overrides varargsMethod(String...) in ClassWithVarargsMethod; overriding
method is missing '...'
```

When the compiler encounters a `varargs` method, it translates the `varargs` formal parameter into an array. In the method `ClassWithVarargsMethod.varargsMethod`, the compiler translates the `varargs` formal parameter `String... s` to the formal parameter `String[] s`, an array that matches the formal parameter of the method `ClassWithOverridingMethod.varargsMethod`. Consequently, this example compiles.

`path`
Warns about invalid path elements and nonexistent path directories on the command line (with regard to the class path, the source path, and other paths). Such warnings can't be suppressed with the `@SuppressWarnings` annotation. For example:

- **Oracle Solaris, Linux, and OS X:**: `javac -Xlint:path -classpath /nonexistentpath Example.java`
- **Windows**: `javac -Xlint:path -classpath C:\nonexistentpath Example.java`

`processing`
Warns about issues related to annotation processing. The compiler generates this warning when you have a class that has an annotation, and you use an annotation processor that can't handle that type of exception. For example, the following is a simple annotation processor:
Source file AnnocProc.java:

```
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import.javaz.lang.model.element.*;

@SupportedAnnotationTypes("NotAnno")
```

```
public class AnnoProc extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> elems, RoundEnvironment renv){
     return true;
  }

  public SourceVersion getSupportedSourceVersion() {
     return SourceVersion.latest();
   }
}
```

Source file AnnosWithoutProcessors.java:

```
@interface Anno { }

@Anno
class AnnosWithoutProcessors { }
```

The following commands compile the annotation processor `AnnoProc`, then run this annotation processor against the source file `AnnosWithoutProcessors.java`:

```
javac AnnoProc.java
javac -cp . -Xlint:processing -processor AnnoProc -proc:only
AnnosWithoutProcessors.java
```

When the compiler runs the annotation processor against the source file `AnnosWithoutProcessors.java`, it generates the following warning:

```
warning: [processing] No processor claimed any of these annotations: Anno
```

To resolve this issue, you can rename the annotation defined and used in the class `AnnosWithoutProcessors` from `Anno` to `NotAnno`.

`rawtypes`
Warns about unchecked operations on raw types. The following statement generates a `rawtypes` warning:

```
void countElements(List l) { ... }
```

The following example doesn't generate a `rawtypes` warning:

```
void countElements(List<?> l) { ... }
```

`List` is a raw type. However, `List<?>` is an unbounded wildcard parameterized type. Because `List` is a parameterized interface, always specify its type argument. In this example, the `List` formal argument is specified with an unbounded wildcard (`?`) as its formal type parameter, which means that the `countElements` method can accept any instantiation of the `List` interface.

`serial`
Warns about missing `serialVersionUID` definitions on serializable classes, for example:

```
public class PersistentTime implements Serializable
{
  private Date time;

   public PersistentTime() {
     time = Calendar.getInstance().getTime();
```

```
   }

   public Date getTime() {
     return time;
   }
}
```

The compiler generates the following warning:

```
warning: [serial] serializable class PersistentTime has no definition of
serialVersionUID
```

If a serializable class doesn't explicitly declare a field named `serialVersionUID`, then the serialization runtime environment calculates a default `serialVersionUID` value for that class based on various aspects of the class, as described in the Java Object Serialization Specification. However, it's strongly recommended that all serializable classes explicitly declare `serialVersionUID` values because the default process of computing `serialVersionUID` values is highly sensitive to class details that can vary depending on compiler implementations. As a result, might cause an unexpected `InvalidClassExceptions` during deserialization. To guarantee a consistent `serialVersionUID` value across different Java compiler implementations, a serializable class must declare an explicit `serialVersionUID` value.

`static`
Warns about issues relating to the use of statics variables, for example:

```
class XLintStatic {
    static void m1() { }
    void m2() { this.m1(); }
}
```

The compiler generates the following warning:

```
warning: [static] static method should be qualified by type name,
XLintStatic, instead of by an expression
```

To resolve this issue, you can call the `static` method `m1` as follows:

```
XLintStatic.m1();
```

Alternately, you can remove the `static` keyword from the declaration of the method `m1`.

`try`
Warns about issues relating to the use of `try` blocks, including try-with-resources statements. For example, a warning is generated for the following statement because the resource `ac` declared in the `try` block isn't used:

```
try ( AutoCloseable ac = getResource() ) {    // do nothing}
```

`unchecked`
Gives more detail for unchecked conversion warnings that are mandated by the Java Language Specification, for example:

```
List l = new ArrayList<Number>();
List<String> ls = l;       // unchecked warning
```

During type erasure, the types `ArrayList<Number>` and `List<String>` become `ArrayList` and `List`, respectively.

The `ls` command has the parameterized type `List<String>`. When the `List` referenced by `l` is assigned to `ls`, the compiler generates an unchecked warning. At compile time, the compiler and JVM can't determine whether `l` refers to a `List<String>` type. In this case, `l` doesn't refer to a `List<String>` type. As a result, heap pollution occurs.
A heap pollution situation occurs when the `List` object `l`, whose static type is `List<Number>`, is assigned to another `List` object, `ls`, that has a different static type, `List<String>`. However, the compiler still allows this assignment. It must allow this assignment to preserve backward compatibility with releases of Java SE that don't support generics. Because of type erasure, `List<Number>` and `List<String>` both become `List`. Consequently, the compiler allows the assignment of the object `l`, which has a raw type of `List`, to the object `ls`.

`varargs`
Warns about unsafe use of variable arguments (`varargs`) methods, in particular, those that contain non-reifiable arguments, for example:

```
public class ArrayBuilder {
  public static <T> void addToList (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }
}
```

A non-reifiable type is a type whose type information isn't fully available at runtime. The compiler generates the following warning for the definition of the method `ArrayBuilder.addToList`:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the `varargs` formal parameter into an array. However, the Java programming language doesn't permit the creation of arrays of parameterized types. In the method `ArrayBuilder.addToList`, the compiler translates the `varargs` formal parameter `T...` elements to the formal parameter `T[]` elements, an array. However, because of type erasure, the compiler converts the `varargs` formal parameter to `Object[]` elements. Consequently, there's a possibility of heap pollution.

Example of Compiling by Providing Command-Line Arguments

To compile as though providing command-line arguments, use the following syntax:

```
JavaCompiler javac = ToolProvider.getSystemJavaCompiler();
```

The example writes diagnostics to the standard output stream and returns the exit code that `javac` command would give when called from the command line.

You can use other methods in the `javax.tools.JavaCompiler` interface to handle diagnostics, control where files are read from and written to, and more.

Old Interface

> **Note:**
>
> This API is retained for backward compatibility only. All new code should use the Java Compiler API.

The `com.sun.tools.javac.Main` class provides two static methods to call the compiler from a program:

```
public static int compile(String[] args);
public static int compile(String[] args, PrintWriter out);
```

The `args` parameter represents any of the command-line arguments that would typically be passed to the compiler.

The `out` parameter indicates where the compiler diagnostic output is directed.

The `return` value is equivalent to the `exit` value from `javac`.

> **Note:**
>
> All other classes and methods found in a package with names that start with `com.sun.tools.javac` (subpackages of `com.sun.tools.javac`) are strictly internal and subject to change at any time.

Example of Compiling Multiple Source Files

This example compiles the `Aloha.java`, `GutenTag.java`, `Hello.java`, and `Hi.java` source files in the `greetings` package.

Oracle Solaris, Linux, and OS X::

```
% javac greetings/*.java
% ls greetings
Aloha.class         GutenTag.class      Hello.class         Hi.class
Aloha.java          GutenTag.java       Hello.java          Hi.java
```

Windows:

```
C:\>javac greetings\*.java
C:\>dir greetings
Aloha.class         GutenTag.class      Hello.class         Hi.class
Aloha.java          GutenTag.java       Hello.java          Hi.java
```

Example of Specifying a User Class Path

After changing one of the source files in the previous example, recompile it:

Oracle Solaris, Linux, and OS X::

```
pwd
/examples
javac greetings/Hi.java
```

Windows:

```
C:\>cd
\examples
C:\>javac greetings\Hi.java
```

Because `greetings.Hi` refers to other classes in the `greetings` package, the compiler needs to find these other classes. The previous example works because the default user class path is the directory that contains the package directory. If you want to

recompile this file without concern for which directory you are in, then add the examples directory to the user class path by setting `CLASSPATH`. This example uses the `-classpath` option.

Oracle Solaris, Linux, and OS X::

```
javac -classpath /examples /examples/greetings/Hi.java
```

Windows:

```
C:\>javac -classpath \examples \examples\greetings\Hi.java
```

If you change `greetings.Hi` to use a banner utility, then that utility also needs to be accessible through the user class path.

Oracle Solaris, Linux, and OS X::

```
javac -classpath /examples:/lib/Banners.jar \
            /examples/greetings/Hi.java
```

Windows:

```
C:\>javac -classpath \examples;\lib\Banners.jar ^
            \examples\greetings\Hi.java
```

To execute a class in the `greetings` package, the program needs access to the `greetings` package, and to the classes that the `greetings` classes use.

Oracle Solaris, Linux, and OS X::

```
java -classpath /examples:/lib/Banners.jar greetings.Hi
```

Windows:

```
C:\>java -classpath \examples;\lib\Banners.jar greetings.Hi
```

The `-source 1.7` option specifies that release 1.7 (or 7) of the Java programming language must be used to compile OldCode.java. The `-target 1.7` option ensures that the generated class files are compatible with JVM 1.7.

Annotation Processing

The `javac` command provides direct support for annotation processing, superseding the need for the separate annotation processing command, `apt`.

The API for annotation processors is defined in the `javax.annotation.processing` and `javax.lang.model` packages and subpackages.

How Annotation Processing Works

Unless annotation processing is disabled with the `-proc:none` option, the compiler searches for any annotation processors that are available. The search path can be specified with the `-processorpath` option. If no path is specified, then the user class path is used. Processors are located by means of service provider-configuration files named `META-INF/services/javax.annotation.processing`. Processor on the search path. Such files should contain the names of any annotation processors to be used, listed one per line. Alternatively, processors can be specified explicitly, using the `-processor` option.

After scanning the source files and classes on the command line to determine what annotations are present, the compiler queries the processors to determine what annotations they process. When a match is found, the processor is called. A processor can claim the annotations it processes, in which case no further attempt is made to find any processors for those annotations. After all of the annotations are claimed, the compiler does not search for additional processors.

If any processors generate new source files, then another round of annotation processing occurs: Any newly generated source files are scanned, and the annotations processed as before. Any processors called on previous rounds are also called on all subsequent rounds. This continues until no new source files are generated.

After a round occurs where no new source files are generated, the annotation processors are called one last time, to give them a chance to complete any remaining work. Finally, unless the `-proc:only` option is used, the compiler compiles the original and all generated source files.

Searching for Types

To compile a source file, the compiler often needs information about a type, but the type definition is not in the source files specified on the command line.

To compile a source file, the compiler often needs information about a type, but the type definition is not in the source files specified on the command line. The compiler needs type information for every class or interface used, extended, or implemented in the source file. This includes classes and interfaces not explicitly mentioned in the source file, but that provide information through inheritance.

For example, when you create a subclass of `java.awt.Window`, you are also using the ancestor classes of `Window`: `java.awt.Container`, `java.awt.Component`, and `java.lang.Object`.

When the compiler needs type information, it searches for a source file or class file that defines the type. The compiler searches for class files first in the bootstrap and extension classes, then in the user class path (which by default is the current directory). The user class path is defined by setting the `CLASSPATH` environment variable or by using the `-classpath` option.

If you set the `-sourcepath` option, then the compiler searches the indicated path for source files. Otherwise, the compiler searches the user class path for both class files and source files.

You can specify different bootstrap or extension classes with the `-bootclasspath` and the `-extdirs` options. See Cross-Compilation Options for javac.

A successful type search may produce a class file, a source file, or both. If both are found, then you can use the `-Xprefer` option to instruct the compiler which to use. If `newer` is specified, then the compiler uses the newer of the two files. If `source` is specified, the compiler uses the source file. The default is `newer`.

If a type search finds a source file for a required type, either by itself, or as a result of the setting for the `-Xprefer` option, then the compiler reads the source file to get the information it needs. By default the compiler also compiles the source file. You can use the `-implicit` option to specify the behavior. If `none` is specified, then no class files are generated for the source file. If `class` is specified, then class files are generated for the source file.

The compiler might not discover the need for some type information until after annotation processing completes. When the type information is found in a source file and no `-implicit` option is specified, the compiler gives a warning that the file is being compiled without being subject to annotation processing. To disable the warning, either specify the file on the command line (so that it will be subject to annotation processing) or use the `-implicit` option to specify whether or not class files should be generated for such source files.

javap

You use the `javap` command to disassemble one or more class files.

Synopsis

```
javap [options] classes...
```

options
Specifies the command-line options. See Options for javap.

classes
Specifies one or more classes separated by spaces to be processed for annotations. You can specify a class that can be found in the class path by its file name, URL, or by its fully qualified class name.
Examples:

```
path/to/MyClass.class

jar:file:///path/to/MyJar.jar!/mypkg/MyClass.class

 java.lang.Object
```

Description

The `javap` command disassembles one or more class files. The output depends on the options used. When no options are used, the `javap` command prints the protected and public fields, and methods of the classes passed to it.

The `javap` command isn't multirelease JAR aware. Using the class path form of the command results in viewing the base entry in all JAR files, multirelease or not. Using the URL form, you can use the URL form of an argument to specify a specific version of a class to be disassembled.

The `javap` command prints its output to `stdout`.

> **Note:**
>
> In tools that support `--` style options, the GNU-style options can use the equal sign (=) instead of a white space to separate the name of an option from its value.

Options for javap

`-help`, `--help`, or `-?`
Prints a help message for the `javap` command.

`-version`
Prints release information.

`-verbose` or `-v`
Prints additional information about the selected class.

`-l`
Prints line and local variable tables.

`-public`
Shows only public classes and members.

`-protected`
Shows only protected and public classes and members.

`-package`
Shows package/protected/public classes and members (default).

`-private` or `-p`
Shows all classes and members.

`-c`
Prints disassembled code, for example, the instructions that comprise the Java bytecodes, for each of the methods in the class.

`-s`
Prints internal type signatures.

`-sysinfo`
Shows system information (path, size, date, MD5 hash) of the class being processed.

`-constants`
Shows `static final` constants.

`--module` *`module`* or `-m` *`module`*
Specifies the module containing classes to be disassembled.

`--module-path` *`path`*
Specifies where to find application modules.

`--system` *`jdk`*
Specifies where to find system modules.

`--class-path` *`path`*, `-classpath` *`path`*, or `-cp` *`path`*
Specifies the path that the `javap` command uses to find user class files. It overrides the default or the `CLASSPATH` environment variable when it's set.

`-bootclasspath` *`path`*
Overrides the location of bootstrap class files.

-Joption
Passes the specified option to the JVM. For example:

```
javap -J-version

javap -J-Djava.security.manager -J-Djava.security.policy=MyPolicy MyClassName
```

See Overview of Java Options.

javap Example

Compile the following `HelloWorldFrame` class:

```
import java.awt.Graphics;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class HelloWorldFrame extends JFrame {

   String message = "Hello World!";

   public HelloWorldFrame(){
        setContentPane(new JPanel(){
            @Override
            protected void paintComponent(Graphics g) {
                g.drawString(message ,15, 30);
            }
        });
        setSize(100,100);
    }
    public static void main(String[] args) {
        HelloWorldFrame frame = new HelloWorldFrame();
        frame.setVisible(true);

    }

}
```

The output from the `javap HelloWorldFrame.class` command yields the following:

```
Compiled from "HelloWorldFrame.java"
public class HelloWorldFrame extends javax.swing.JFrame {
  java.lang.String message;
  public HelloWorldFrame();
  public static void main(java.lang.String[]);
}
```

The output from the `javap -c HelloWorldFrame.class` command yields the following:

```
Compiled from "HelloWorldFrame.java"
public class HelloWorldFrame extends javax.swing.JFrame {
  java.lang.String message;

  public HelloWorldFrame();
    Code:
       0: aload_0
       1: invokespecial #1        // Method javax/swing/JFrame."<init>":()V
       4: aload_0
       5: ldc           #2        // String Hello World!
       7: putfield      #3        // Field message:Ljava/lang/String;
      10: aload_0
```

```
      11: new           #4        // class HelloWorldFrame$1
      14: dup
      15: aload_0
      16: invokespecial #5        // Method HelloWorldFrame$1."<init>":
(LHelloWorldFrame;)V
      19: invokevirtual #6        // Method setContentPane:(Ljava/awt/Container;)V
      22: aload_0
      23: bipush        100
      25: bipush        100
      27: invokevirtual #7        // Method setSize:(II)V
      30: return

  public static void main(java.lang.String[]);
    Code:
       0: new           #8        // class HelloWorldFrame
       3: dup
       4: invokespecial #9        // Method "<init>":()V
       7: astore_1
       8: aload_1
       9: iconst_1
      10: invokevirtual #10       // Method setVisible:(Z)V
      13: return
}
```

javah

You use the `javah` tool to generate C header and source files from a Java class.

> **Note:**
>
> The `javah` tool is deprecated as of JDK 9 and might be removed in a future JDK release. The tool has been superseded by the `-h` option added to `javac` in JDK 8.

Synopsis

`javah [`*`options`*`] `*`fully-qualified-class-name ...`*

`options`
Specifies the command-line options. See Options for javah.

`fully-qualified-class-name`
Specifies the fully qualified location of the classes to be converted to C header and source files.
Each class must be specified by its fully qualified name, optionally prefixed by a module name followed by the slash (/). For example:
`java.lang.Object`
`java.base/java.io.File`

Description

The `javah` command generates C header and source files that are needed to implement native methods. The generated header and source files are used by C programs to reference an object's instance variables from native source code. The `.h`

file contains a `struct` definition with a layout that parallels the layout of the corresponding class. The fields in the `struct` correspond to instance variables in the class.

The name of the header file and the structure declared within it are derived from the name of the class. When the class passed to the `javah` command is inside a package, the package name is added to the beginning of both the header file name and the structure name. Underscores (_) are used as name delimiters.

By default, the `javah` command creates a header file for each class listed on the command line and puts the files in the current directory. Use the `-stubs` option to create source files. Use the `-o` option to concatenate the results for all listed classes into a single file.

The Java Native Interface (JNI) doesn't require header information or stub files. The `javah` command can still be used to generate native method function prototypes needed for JNI-style native methods. The `javah` command produces JNI-style output and places the result in the `.h` file.

Options for javah

> **Note:**
>
> In tools that support `--` style options, the GNU-style options can use the equal sign (=) instead of a white space to separate the name of an option from its value.

`-o` *`outputfile`*
Concatenates the resulting header or source files for all the classes listed on the command line into an output file. Only one of the options `-o` or `-d` can be used.

`-d` *`directory`*
Sets the directory where the `javah` command saves the header files or the stub files. Only one of the options `-d` or `-o` can be used.

`-v` or `-verbose`
Indicates verbose output and causes the `javah` command to print a message to `stdout` about the status of the generated files.

`-h` , `-help`, or `-?`
Prints a help message for `javah` usage.

`-version`
Prints `javah` command release information.

`-jni`
Causes the `javah` command to create an output file containing JNI-style native method function prototypes. This is the default output; use of `-jni` is optional.

`-force`
Specifies that output files should always be written.

`--module-path` *`path`*
Specifies the path from which to load application modules.

`--system jdk`
Specifies where to find system modules.

`--class-path path`, `-classpath path`, or `-cp path`
Specifies the path that the `javah` command uses to look up classes. Overrides the default or the `CLASSPATH` environment variable when it's set. Directories are separated by colons on Oracle Solaris and semicolons on Windows. The general format for the path is:

- **Oracle Solaris, Linux, and OS X:**

  `:your-path`

  Example: `:/home/avh/classes:/usr/local/java/classes`

- **Windows:**

  ;your-path

  Example: `;C:\users\dac\classes;C:\tools\java\classes`

As a special convenience, a class path element that contains a base name with an asterisk (*) is considered equivalent to specifying a list of all the files in the directory with the extension `.jar` or `.JAR`.
For example, if directory `mydir` contains `a.jar` and `b.JAR`, then the class path element `mydir/*` is expanded to `a.jar:b.JAR`, except that the order of JAR files is unspecified. All JAR files in the specified directory, including hidden ones, are included in the list. A class path entry that consists of an asterisk (*) expands to a list of all the JAR files in the current directory. The `CLASSPATH` environment variable, where defined, is similarly expanded. Any class path wildcard expansion occurs before the Java Virtual Machine (JVM) is started. A Java program never sees unexpanded wild cards except by querying the environment, for example, by calling `System.getenv("CLASSPATH")`.

`-bootclasspath path`
Specifies the path from which to load bootstrap classes.

javadoc

You use the `javadoc` tool and its options to generate HTML pages of API documentation from Java source files.

Synopsis

```
javadoc [options] [packagenames] [sourcefiles] [@files]
```

`options`
Specifies command-line options, separated by spaces. See Options for javadoc, Extended Options, Standard doclet Options, and Nonstandard Options Provided by the Standard doclet.

`packagenames`
Specifies names of packages that you want to document, separated by spaces, for example `java.lang java.lang.reflect java.awt`. If you want to also document the subpackages, then use the `-subpackages` option to specify the packages.
By default, `javadoc` looks for the specified packages in the current directory and subdirectories. Use the `-sourcepath` option to specify the list of directories where to look for packages.

sourcefiles
Specifies names of Java source files that you want to document, separated by spaces, for example `Class.java Object.java Button.java`. By default, `javadoc` looks for the specified classes in the current directory. However, you can specify the full path to the class file and use wildcard characters, for example `/home/src/java/awt/Graphics*.java`. You can also specify the path relative to the current directory.

@files
Specifies names of files that contain a list of `javadoc` command options, package names, and source file names in any order.

Description

The `javadoc` command parses the declarations and documentation comments in a set of Java source files and produces corresponding HTML pages that describe (by default) the public and protected classes, nested classes (but not anonymous inner classes), interfaces, constructors, methods, and fields. You can use the `javadoc` command to generate the API documentation or the implementation documentation for a set of source files.

You can run the `javadoc` command on entire packages, individual source files, or both. When documenting entire packages, you can use the `-subpackages` option either to recursively traverse a directory and its subdirectories, or to pass in an explicit list of package names. When you document individual source files, pass in a list of Java source file names. See javadoc Overview in *Java Platform, Standard Edition Javadoc Guide* for information about using the `javadoc` tool.

Conformance

The standard doclet does not validate the content of documentation comments for conformance, nor does it attempt to correct any errors in documentation comments. Anyone running javadoc is advised to be aware of the problems that may arise when generating non-conformant output or output containing executable content, such as JavaScript. The standard doclet does provide the `doclint` feature to help developers detect common problems in documentation comments; but, it is also recommended to check the generated output with any appropriate conformance and other checking tools.

For more details on the conformance requirements for HTML5 documents, see Conformance requirements in the HTML5 Specification. For more details on security issues related to web pages, see the Open Web Application Security Project (OWASP) page.

Options for javadoc

The following options are the core Javadoc options.

> **Note:**
>
> In tools that support `--` style options, the GNU-style options can use the equal sign (=) instead of a white space to separate the name of an option from its value.

`--add-modules` *`module(,module)*`*
Specifies the root modules to resolve in addition to the initial modules, or all modules on the module path if *`module`* is `ALL-MODULE-PATH`.

`-bootclasspath` *`classpathlist`*
Overrides the location of platform class files used for nonmodular releases. The `bootclasspath` option is part of the search path that the `javadoc` command uses to look up source and class files.
Separate directories in the `classpathlist` parameters with one of the following delimiters:

- **Oracle Solaris, Linux, and OS X:**: colon (:)
- **Windows**: semicolon (;)

`-breakiterator`
Computes the first sentence with `BreakIterator`. The first sentence is copied to the package, class, or member summary and to the alphabetic index. The `BreakIterator` class is used to determine the end of a sentence for all languages except for English.

- English default sentence-break algorithm — Stops at a period followed by a space or an HTML block tag, such as `<P>`.
- Breakiterator sentence-break algorithm — Stops at a period, question mark, or exclamation point followed by a space when the next word starts with a capital letter. This is meant to handle most abbreviations (such as "The serial no. is valid", but will not handle "Mr. Smith"). The `-breakiterator` option doesn't stop at HTML tags or sentences that begin with numbers or symbols. The algorithm stops at the last period in ../filename, even when embedded in an HTML tag.

`--class-path` *`path`* , `-classpath` *`path`* , or `-cp` *`path`*
Specifies the paths where the `javadoc` command searches for referenced classes These are the documented classes plus any classes referenced by those classes.

- **Oracle Solaris, Linux, and OS X:**: Separate multiple paths with a colon (:).
- **Windows**: Separate multiple paths with a semicolon (;).

The `javadoc` command searches all subdirectories of the specified paths. Follow the instructions in the class path documentation for specifying the `classpathlist` value. If you omit `-sourcepath`, then the `javadoc` command uses `-classpath` to find the source files and class files (for backward compatibility). If you want to search for source and class files in separate paths, then use both `-sourcepath` and `-classpath`.

- **Oracle Solaris, Linux, and OS X:**: For example, if you want to document `com.mypackage`, whose source files reside in the directory `/home/user/src/com/mypackage`, and if this package relies on a library in `/home/user/lib`, then you would use the following command:

  ```
  javadoc -sourcepath /home/user/src -classpath /home/user/lib com.mypackage
  ```

- **Windows**: For example, if you want to document `com.mypackage`, whose source files reside in the directory `\user\src\com\mypackage`, and if this package relies on a library in `\user\lib`, then you would use the following command:

  ```
  javadoc -sourcepath \user\lib -classpath \user\src com.mypackage
  ```

Similar to other tools, if you don't specify `-classpath`, then the `javadoc` command uses the `CLASSPATH` environment variable when it is set. If both aren't set, then the `javadoc` command searches for classes from the current directory.

A class path element that contains a base name of * is considered equivalent to specifying a list of all the files in the directory with the extension `.jar` or `.JAR`.
For example, if directory `mydir` contains `a.jar` and `b.JAR`, then the class path element `foo/*` is expanded to a `A.jar:b.JAR`, except that the order of JAR files is unspecified. All JAR files in the specified directory including hidden files are included in the list. A class path entry that consists of * expands to a list of all the jar files in the current directory. The `CLASSPATH` environment variable is similarly expanded. Any class path wildcard expansion occurs before the Java Virtual Machine (JVM) starts. No Java program ever sees unexpanded wild cards except by querying the environment, for example, by calling `System.getenv ("CLASSPATH")`.

`-doclet` *`class`*
Generates output by using an alternate doclet. Use the fully qualified name. This doclet defines the content and formats the output. If the `-doclet` option isn't used, then the `javadoc` command uses the standard doclet for generating the default HTML format. This class must contain the `start(Root)` method. The path to this starting class is defined by the `-docletpath` option.

`-docletpath` *`path`*
Specifies where to find doclet class files (specified with the `-doclet` option) and any JAR files it depends on. If the starting class file is in a JAR file, then this option specifies the path to that JAR file. You can specify an absolute path or a path relative to the current directory. If `classpathlist` contains multiple paths or JAR files, then they should be separated with a colon (:) on Oracle Solaris and a semi-colon (;) on Windows. This option isn't necessary when the `doclet` starting class is already in the search path.

`-encoding` *`name`*
Specifies the encoding name of the source files, such as `EUCJIS/SJIS`. If this option isn't specified, then the platform default converter is used.

`-exclude` *`pkglist`*
Unconditionally, excludes the specified packages and their subpackages from the list formed by `-subpackages`. It excludes those packages even when they would otherwise be included by some earlier or later `-subpackages` option.
The following example would include `java.io`, `java.util`, and `java.math` (among others), but would exclude packages rooted at `java.net` and `java.lang`. Notice that these examples exclude `java.lang.ref`, which is a subpackage of `java.lang`.

- **Oracle Solaris, Linux, and OS X:**:

  ```
  javadoc -sourcepath /home/user/src -subpackages java -exclude java.net:java.lang
  ```

- **Windows**:

  ```
  javadoc -sourcepath \user\src -subpackages java -exclude java.net:java.lang
  ```

`--expand-requires` *`value`*
Instructs the javadoc tool to expand the set of modules to be documented. By default, only the modules given explicitly on the command line are documented. Supports the following values:

- `transitive`: additionally includes all the required transitive dependencies of those modules.
- `all`: includes all dependencies.

`-extdirs` *`dirlist`*
Specifies the directories where extension classes reside. These are any classes that use the Java Extension mechanism. The `extdirs` option is part of the search path the `javadoc` command uses to look up source and class files. See the `-classpath` option for more information. Separate directories in `dirlist` with semicolons (;) for Windows and colons (:) for Oracle Solaris.

`-help` or `--help`
Displays the online help, which lists all of the `javadoc` and `doclet` command-line options.

`-Jflag`
Passes `flag` directly to the Java Runtime Environment (JRE) that runs the `javadoc` command. For example, if you must ensure that the system sets aside 32 MB of memory in which to process the generated documentation, then you would call the `-Xmx` option as follows: `javadoc -J-Xmx32m -J-Xms32m com.mypackage`. Be aware that `-Xms` is optional because it only sets the size of initial memory, which is useful when you know the minimum amount of memory required.
There is no space between the `J` and the `flag`.
Use the `-version` option to find out what version of the `javadoc` command you are using. The version number of the standard doclet appears in its output stream.

```
javadoc -J-version
java version "1.7.0_09"
Java(TM) SE Runtime Environment (build 1.7.0_09-b05)
Java HotSpot(TM) 64-Bit Server VM (build 23.5-b02, mixed mode)
```

`--limit-modules` *`module (,module)*`*
Limits the universe of observable modules.

`-locale` *`name`*
Specifies the locale that the `javadoc` command uses when it generates documentation. The argument is the name of the locale, as described in `java.util.Locale` documentation, such as `en_US` (English, United States) or `en_US_WIN` (Windows variant).

> **Note:**
>
> The `-locale` option must be placed ahead (to the left) of any options provided by the standard doclet or any other doclet. Otherwise, the navigation bars appear in English. This is the only command-line option that depends on order.

Specifying a locale causes the `javadoc` command to choose the resource files of that locale for messages such as strings in the navigation bar, headings for lists and tables, help file contents, comments in the `stylesheet.css` file, and so on. It also specifies the sorting order for lists sorted alphabetically, and the sentence separator to determine the end of the first sentence. The `-locale` option doesn't determine the locale of the documentation comment text specified in the source files of the documented classes.

`--module` *`module(,module)*`*
Documents the specified module.

`--module-path` *`path`* or `-p` *`path`*
Specifies where to find application modules.

`--module-source-path` *`path`*
Specifies where to find input source files for multiple modules.

`-package`
Shows only package, protected, and public classes and members.

`-private`
Shows all classes and members.

`-protected`
Shows only protected and public classes and members. This is the default.

`-public`
Shows only the public classes and members.

`-quiet`
Shuts off messages so that only the warnings and errors appear to make them easier to view. It also suppresses the `version` string.

`--release` *`release`*
Provides source compatibility with specified release.

`--show-members` *`value`*
Specifies which members (fields or methods) are documented, where value can be any of the following:

- `protected`: The default value is protected.
- `public`: Shows only public values.
- `package`: Shows public, protected, and package members.
- `private`: Shows all members.

`--show-module-contents` *`value`*
Specifies the documentation granularity of module declarations. Possible values are `api` or `all`.

`--show-packages` *`value`*
Specifies which modules packages are documented. Possible values are `exported` or `all` packages.

`--show-types` *`value`*
Specifies which types (classes, interfaces, etc.) are documented, where value can be any of the following:

- `protected`: The default value. Shows public and protected types.
- `public`: Shows only public values.
- `package`: Shows public, protected, and package types.
- `private`: Shows all types.

`-source release`
Specifies the release of source code accepted. The following values for the `release` parameter are allowed. Use the value of `release` that corresponds to the value used when you compile code with the `javac` command.

- **Release Value: 9**. The `javadoc` command accepts code containing language features in JDK 9. The compiler defaults to the 9 behavior when the `-source` option isn't used.
- **Release Value: 8**. The `javadoc` command accepts code containing generics and other language features introduced in JDK 8.
- **Release Value: 7**. The `javadoc` command accepts code containing assertions, which were introduced in JDK 7.
- **Release Value: 6**. The `javadoc` command doesn't support assertions, generics, or other language features introduced after JDK 6.

`--source-path path` or `-sourcepath path`
Specifies the search paths for finding source files when passing package names or the `-subpackages` option into the `javadoc` command.

- **Oracle Solaris, Linux, and OS X:**: Separate multiple paths with a colon (`:`).
- **Windows**: Separate multiple paths with a. semicolon (`;`).

The `javadoc` command searches all subdirectories of the specified paths. Note that this option isn't only used to locate the source files being documented, but also to find source files that aren't being documented, but whose comments are inherited by the source files being documented.
You can use the `-sourcepath` option only when passing package names into the `javadoc` command. This will not locate source files passed into the `javadoc` command. To locate source files, change to that directory or include the path ahead of each file. If you omit `-sourcepath`, then the `javadoc` command uses the class path to find the source files (see `-classpath`). The default `-sourcepath` is the value of class path. If `-classpath` is omitted and you pass package names into the `javadoc` command, then the `javadoc` command searches in the current directory and subdirectories for the source files.
Set `sourcepathlist` to the root directory of the source tree for the package you are documenting.

- **Oracle Solaris, Linux, and OS X:**:
  - For example, suppose you want to document a package called `com.mypackage`, whose source files are located at `/home/user/src/com/mypackage/*.java`. Specify `sourcepath` as `/home/user/src`, the directory that contains `com\mypackage`, and then supply the package name, as follows:

    ```
    javadoc -sourcepath /home/user/src/ com.mypackage
    ```

  - Notice that if you concatenate the value of `sourcepath` and the package name together and change the dot to a slash (`/`), then you have the full path to the package:

    ```
    /home/user/src/com/mypackage
    ```

  - To point to two source paths:

```
javadoc -sourcepath /home/user1/src:/home/user2/src com.mypackage
```

- **Windows**:
  - For example, suppose you want to document a package called `com.mypackage`, whose source files are located at `\user\src\com\mypackage\*.java`. Specify `sourcepath` as `\user\src`, the directory that contains `com\mypackage`, and then supply the package name as follows:

    ```
    javadoc -sourcepath C:\user\src com.mypackage
    ```

  - Notice that if you concatenate the value of sourcepath and the package name together and change the dot to a backslash (\), then you have the full path to the package:

    ```
    \user\src\com\mypackage
    ```

  - To point to two source paths:

    ```
    javadoc -sourcepath \user1\src;\user2\src com.mypackage
    ```

`-subpackages` *`subpkglist`*
Generates documentation from source files in the specified packages and recursively in their subpackages. This option is useful when adding new subpackages to the source code because they are automatically included. Each package argument is any top-level subpackage (such as `java`) or fully qualified package (such as `javax.swing`) that doesn't need to contain source files. Arguments are separated by colons on all operating systems. Wild cards aren't allowed. Use `-sourcepath` to specify where to find the packages. This option doesn't process source files that are in the source tree but don't belong to the packages.
For example, the following commands generates documentation for packages named `java` and `javax.swing` and all of their subpackages.

- **Oracle Solaris, Linux, and OS X:**:

  ```
  javadoc -d docs -sourcepath /home/user/src  -subpackages java:javax.swing
  ```

- **Windows**:

  ```
  javadoc -d docs -sourcepath \user\src -subpackages java:javax.swing
  ```

`--system` *`jdk`*
Overrides location of system modules used for modular releases.

`--upgrade-module-path` *`path`*
Overrides location of upgradable options.

`-verbose`
Provides more detailed messages while the `javadoc` command runs. Without the `verbose` option, messages appear for loading the source files, generating the documentation (one message per source file), and sorting. The *`verbose`* option causes the printing of additional messages that specify the number of milliseconds to parse each Java source file.

`-X`
Prints a synopsis of non-standard options and exit.

Extended Options

The following are extended options for `javadoc` and are subject to change without notice.

`--add-exports` *`module/package=other-module(,other-module)*`*
Specifies a package that is to be considered as exported from its defining module from its defining module to additional modules, or to all unnamed modules if *`other-module`* is `ALL-UNNAMED`.

`--add-reads` *`module /package=other-module (,other-module)`*
Specifies additional modules to be considered as required by a given module. If *`other-module`* is `ALL-UNNAMED`, it requires the unamed module.

`--patch-module` *`module=pathlist`*
Replaces the contents of a module such as class files and resources with another version. You can specify a list of JARs or directories containing the new module's contents in the *pathlist*.
Each element in the list is separated by a separator:

- **Oracle Solaris, Linux, and OS X:**: colon (`:`)
- **Windows**: semicolon (`;`)

`-Xmaxerrs` *`number`*
Sets the maximum number of errors to print.

`-Xmaxwarns` *`number`*
Sets the maximum number of warnings to print.

`-Xmodule:`*`module-name`*
Specifies a module to which the classes being compiled belong.

`-Xold`
Invokes the legacy javadoc tool.

Standard doclet Options

The following options are provided by the standard doclet.

`-author`
Includes the `@author` text in the generated docs.

`-bottom` *`html-code`*
Specifies the text to be placed at the bottom of each output file. The text is placed at the bottom of the page, underneath the lower navigation bar. The text can contain HTML tags and white space, but when it does, the text must be enclosed in quotation marks. Use escape characters for any internal quotation marks within text.

`-charset` *`name`*
Specifies the HTML character set for this document. The name should be a preferred MIME name as specified in the IANA Registry, Character Sets.
For example, `javadoc -charset "iso-8859-1" mypackage` inserts the following line in the head of every generated page:

```
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

This `META` tag is described in the HTML standard (4197265 and 4137321), HTML Document Representation.

`-d` *`directory`*
Specifies the destination directory where the `javadoc` command saves the generated HTML files. If you omit the `-d` option, then the files are saved to the current directory.

The `directory` value can be absolute or relative to the current working directory. The destination directory is automatically created when the `javadoc` command runs.

- **Oracle Solaris, Linux, and OS X:**: For example, the following command generates the documentation for the package `com.mypackage` and saves the results in the `/user/doc/` directory:

  ```
  javadoc -d /user/doc/ com.mypackage
  ```

- **Windows**: For example, the following command generates the documentation for the package `com.mypackage` and saves the results in the `\user\doc\` directory:

  ```
  javadoc -d \user\doc\ com.mypackage
  ```

`-docencoding` *`name`*
Specifies the encoding of the generated HTML files. The name should be a preferred MIME name as specified in the IANA Registry, Character Sets.
If you omit the `-docencoding` option but use the `-encoding` option, then the encoding of the generated HTML files is determined by the `-encoding` option, for example: `javadoc -docencoding "iso-8859-1" mypackage`.

`-docfilessubdirs`
Recursively copies doc-file subdirectories

`-doctitle` *`html-code`*
Specifies the title to place near the top of the overview summary file. The text specified in the `title` tag is placed as a centered, level-one heading directly beneath the top navigation bar. The `title` tag can contain HTML tags and white space, but when it does, you must enclose the title in quotation marks. Internal quotation marks within the `title` tag must be escaped. For example, `javadoc -header "<b>My Library</b><br>v1.0" com.mypackage.`

`-excludedocfilessubdir` *`name`*
Excludes any doc files sub directories with the given name. Enables deep copying of doc-files directories. Subdirectories and all contents are recursively copied to the destination. For example, the directory `doc-files/example/images` and all of its contents are copied. There is also an option to exclude subdirectories.

`-footer` *`html-code`*
Specifies the footer text to be placed at the bottom of each output file. The*`html-code`* value is placed to the right of the lower navigation bar. The *`html-code`* value can contain HTML tags and white space, but when it does, the *`html-code`* value must be enclosed in quotation marks. Use escape characters for any internal quotation marks within a footer.

`--frames`
Enables the use of frames in the generated output (default).

`-group` *`namep1:p2`*
Group the specified packages together in the Overview page.

`-header` *`html-code`*
Specifies the header text to be placed at the top of each output file. The header is placed to the right of the upper navigation bar. The `header` can contain HTML tags and white space, but when it does, the `header` must be enclosed in quotation marks. Use escape characters for internal quotation marks within a header. For example, `javadoc -header "<b>My Library</b><br>v1.0" com.mypackage.`

`-helpfile filename`
Includes the file that links to the **HELP** link in the top and bottom navigation bars . Without this option, the `javadoc` command creates a help file help-doc.html that is hard-coded in the `javadoc` command. This option lets you override the default. The *`filename`* can be any name and isn't restricted to `help-doc.html`. The `javadoc` command adjusts the links in the navigation bar accordingly. For example:

- **Oracle Solaris, Linux, and OS X:**:

  ```
  javadoc -helpfile /home/user/myhelp.html java.awt.
  ```

- **Windows**:

  ```
  javadoc -helpfile C:\user\myhelp.html java.awt.
  ```

`-html4`
Generates HTML 4.0.1 output. If the option is not used, `-html4` is the default

`-html5`
Generates HTML 5 output. If the option is not used, `-html4` is the default.

`-keywords`
Adds HTML keyword `<META>` tags to the generated file for each class. These tags can help search engines that look for `<META>` tags find the pages. Most search engines that search the entire Internet don't look at `<META>` tags, because pages can misuse them. Search engines offered by companies that confine their searches to their own website can benefit by looking at `<META>` tags. The `<META>` tags include the fully qualified name of the class and the unqualified names of the fields and methods. Constructors aren't included because they are identical to the class name. For example, the class `String` starts with these keywords:

```
<META NAME="keywords" CONTENT="java.lang.String class">
<META NAME="keywords" CONTENT="CASE_INSENSITIVE_ORDER">
<META NAME="keywords" CONTENT="length()">
<META NAME="keywords" CONTENT="charAt()">
```

`-link url`
Creates links to existing Javadoc-generated documentation of externally referenced classes. The*`url`* argument is the absolute or relative URL of the directory that contains the external Javadoc-generated documentation you want to link. You can specify multiple `-link` options in a specified `javadoc` command run to link to multiple documents.
The package-list file must be found in this directory (otherwise, use the `-linkoffline` option). The `javadoc` command reads the package names from the package-list file and links to those packages at that URL. When the `javadoc` command runs, the `extdocURL` value is copied into the `<A HREF>` links that are created. Therefore, `extdocURL` must be the URL to the directory, and not to a file. You can use an absolute link for *`url`*to enable your documents to link to a document on any web site, or you can use a relative link to link only to a relative location. If you use a relative link, then the value you pass in should be the relative path from the destination directory (specified with the `-d` option) to the directory containing the packages being linked to. When you specify an absolute link, you usually use an HTTP link. However, if you want to link to a file system that has no web server, then you can use a file link. Use a file link only when everyone who wants to access the generated documentation shares the same file system. In all cases, and on all operating systems, use a slash as the separator, whether the URL is absolute or relative, and `http:` or `file:` as specified in the URL Memo: Uniform Resource Locators.

```
-link  http://<host>/<directory>/<directory>/.../<name>
-link file://<host>/<directory>/<directory>/.../<name>
-link <directory>/<directory>/.../<name>
```

`-linkoffline` *`url1 url2`*
This option is a variation of the `-link` option. They both create links to Javadoc-generated documentation for externally referenced classes. Use the `-linkoffline` option when linking to a document on the web when the `javadoc` command can't access the document through a web connection. Use the `-linkoffline` option when package-list file of the external document isn't accessible or doesn't exist at the `url` location, but does exist at a different location that can be specified by `packageListLoc` (typically local). If `url1` is accessible only on the World Wide Web, then the `-linkoffline` option removes the constraint that the `javadoc` command must have a web connection to generate documentation. Another use is as a work-around to update documents: After you have run the `javadoc` command on a full set of packages, you can run the `javadoc` command again on a smaller set of changed packages, so that the updated files can be inserted back into the original set. Examples follow. The `-linkoffline` option takes two arguments. The first is for the string to be embedded in the `<a href>` links, and the second tells the `-linkoffline` option where to find package-list:
The *`url1`* or *`url2`* value is the absolute or relative URL of the directory that contains the external Javadoc-generated documentation you want to link to. When relative, the value should be the relative path from the destination directory (specified with the `-d` option) to the root of the packages being linked to. See *`url`* in the `-link` option. You can specify multiple `-linkoffline` options in a specified `javadoc` command run.

`-linksource`
Creates an HTML version of each source file (with line numbers) and adds links to them from the standard HTML documentation. Links are created for classes, interfaces, constructors, methods, and fields whose declarations are in a source file. Otherwise, links aren't created, such as for default constructors and generated classes.
This option exposes all private implementation details in the included source files, including private classes, private fields, and the bodies of private methods, regardless of the `-public`, `-package`, `-protected`, and `-private` options. Unless you also use the `-private` option, not all private classes or interfaces are accessible through links.
Each link appears on the name of the identifier in its declaration. For example, the link to the source code of the `Button` class would be on the word `Button`:

```
public class Button extends Component implements Accessible
```

The link to the source code of the `getLabel` method in the `Button` class is on the word `getLabel`:

```
public String getLabel()
```

`-nocomment`
Suppresses the entire comment body, including the main description and all tags, and generate only declarations. This option lets you reuse source files that were originally intended for a different purpose so that you can produce skeleton HTML documentation during the early stages of a new project.

`-nodeprecated`
Prevents the generation of any deprecated API in the documentation. This does what the `-nodeprecatedlist` option does, and it doesn't generate any deprecated API

throughout the rest of the documentation. This is useful when writing code when you don't want to be distracted by the deprecated code.

`-nodeprecatedlist`
Prevents the generation of the file that contains the list of deprecated APIs (`deprecated-list.html`) and the link in the navigation bar to that page. The `javadoc` command continues to generate the deprecated API throughout the rest of the document. This is useful when your source code contains no deprecated APIs, and you want to make the navigation bar cleaner.

`--no-frames`
Disables the use of frames in the generated output.

`-nohelp`
Omits the HELP link in the navigation bars at the top and bottom of each page of output.

`-noindex`
Omits the index from the generated documents. The index is produced by default.

`-nonavbar`
Prevents the generation of the navigation bar, header, and footer, that are usually found at the top and bottom of the generated pages. The `-nonavbar` option has no affect on the `-bottom` option. The `-nonavbar` option is useful when you are interested only in the content and have no need for navigation, such as when you are converting the files to PostScript or PDF for printing only.

`-noqualifier` *`name1`*`:` *`name2...`*
Excludes the list of qualifiers from the output. The package name is removed from places where class or interface names appear.
The following example omits all package qualifiers: `-noqualifier all`.
The following example omits `java.lang` and `java.io` package qualifiers: `-noqualifier java.lang:java.io`.
The following example omits package qualifiers starting with `java` and `com.sun` subpackages, but not `javax`: `-noqualifier java.*:com.sun.*`.
Where a package qualifier would appear due to the previous behavior, the name can be suitably shortened. This rule is in effect whether or not the `-noqualifier` option is used.

`-nosince`
Omits from the generated documents the `Since` sections associated with the `@since` tags.

`-notimestamp`
Suppresses the time stamp, which is hidden in an HTML comment in the generated HTML near the top of each page. The `-notimestamp` option is useful when you want to run the `javadoc` command on two source bases and get the differences between `diff` them, because it prevents time stamps from causing a `diff` (which would otherwise be a `diff` on every page). The time stamp includes the `javadoc` command release number.

`-notree`
Omits the class and interface hierarchy pages from the generated documents. These are the pages you reach using the Tree button in the navigation bar. The hierarchy is produced by default.

`-overview` *`filename`*
Specifies that the `javadoc` command should retrieve the text for the overview documentation from the source file specified by*`filename`* and place it on the Overview page (`overview-summary.html`). A relative path specified with the file name is relative to the current working directory.
While you can use any name you want for the `filename` value and place it anywhere you want for the path, it is typical to name it `overview.html` and place it in the source tree at the directory that contains the topmost package directories. In this location, no path is needed when documenting packages, because the `-sourcepath` option points to this file.

- **Oracle Solaris, Linux, and OS X:**: For example, if the source tree for the `java.lang` package is `/src/classes/java/lang/`, then you could place the overview file at `/src/classes/overview.html`.
- **Windows**: For example, if the source tree for the `java.lang` package is `\src \classes\java\lang\`, then you could place the overview file at `\src\classes \overview.html`

The overview page is created only when you pass two or more package names to the `javadoc` command. The title on the overview page is set by `-doctitle`.

`-serialwarn`
Generates compile-time warnings for missing `@serial` tags. By default, Javadoc generates no serial warnings. Use this option to display the serial warnings, which helps to properly document default serializable fields and `writeExternal` methods.

`-sourcetab` *`tablength`*
Specifies the number of spaces each tab uses in the source.

`-splitindex`
Splits the index file into multiple files, alphabetically, one file per letter, plus a file for any index entries that start with non-alphabetical symbols.

`-stylesheetfile` *`path`*
Specifies the path of an alternate HTML stylesheet file. Without this option, the `javadoc` command automatically creates a stylesheet file `stylesheet.css` that is hard-coded in the `javadoc` command. This option lets you override the default. The file name can be any name and isn't restricted to `stylesheet.css`, for example:

- **Oracle Solaris, Linux, and OS X:**:

```
javadoc -stylesheet file /home/user/mystylesheet.css com.mypackage
```

- **Windows**:

```
javadoc -stylesheet file C:\user\mystylesheet.css com.mypackage
```

`-tag` *`name`*`:`*`locations`*`:` *`header`*
Specifies single argument custom tags. For the `javadoc` command to spell-check tag names, it is important to include a `-tag` option for every custom tag that is present in the source code, disabling (with `X`) those that aren't being output in the current run. The colon (:) is always the separator. The `-tag` option outputs the tag heading, *`header`*, in bold, followed on the next line by the text from its single argument. Similar to any block tag, the argument text can contain inline tags, which are also interpreted. The output is similar to standard one-argument tags, such as the `@return` and `@author` tags. Omitting a *`header`* value causes the `tagname` to be the heading.

`-taglet` *`class`*
Specifies the fully qualified name of the taglet used in generating the documentation for that tag. Use the fully qualified name for the `class` value. This taglet also defines the number of text arguments that the custom tag has. The taglet accepts those arguments, processes them, and generates the output.
Taglets are useful for block or inline tags. They can have any number of arguments and implement custom behavior, such as making text bold, formatting bullets, writing out the text to a file, or starting other processes. Taglets can only determine where a tag should appear and in what form. All other decisions are made by the doclet. A taglet can't do things such as remove a class name from the list of included classes. However, it can execute side effects, such as printing the tag's text to a file or triggering another process. Use the `-tagletpath` option to specify the path to the taglet. The following example inserts the To Do taglet after Parameters and ahead of Throws in the generated pages.

```
-taglet com.sun.tools.doclets.ToDoTaglet
-tagletpath /home/taglets
-tag return
-tag param
-tag todo
-tag throws
-tag see
```

Alternately, you can use the `-taglet` option in place of its `-tag` option, but that might be difficult to read.

`-tagletpath` *`tagletpathlist`*
Specifies the search paths for finding taglet class files. The `tagletpathlist` can contain multiple paths by separating them with a colon (`:`). The `javadoc` command searches all subdirectories of the specified paths.

`-top` *`html-code`*
Specifies the text to be placed at the top of each output file.

`-use`
Creates class and package usage pages. Includes one Use page for each documented class and package. The page describes what packages, classes, methods, constructors and fields use any API of the specified class or package. Given class C, things that use class C would include subclasses of C, fields declared as C, methods that return C, and methods and constructors with parameters of type C. For example, you can look at the Use page for the `String` type. Because the `getName` method in the `java.awt.Font` class returns type `String`, the `getName` method uses `String` and so the `getName` method appears on the Use page for `String`. This documents only uses of the API, not the implementation. When a method uses `String` in its implementation, but doesn't take a string as an argument or return a string, that isn't considered a use of `String`.To access the generated Use page, go to the class or package and click the **Use link** in the navigation bar.

`-version`
Includes the version text in the generated docs. This text is omitted by default. To find out what version of the `javadoc` command you are using, use the `-J-version` option.

`-windowtitle` *`title`*
Specifies the title to be placed in the HTML `<title>` tag. The text specified in the `title` tag appears in the window title and in any browser bookmarks (favorite places) that someone creates for this page. This title shouldn't contain any HTML tags because

the browser doesn't interpret them correctly. Use escape characters on any internal quotation marks within the `title` tag. If the `-windowtitle` option is omitted, then the `javadoc` command uses the value of the `-doctitle` option for the `-windowtitle` option. For example, `javadoc -windowtitle "My Library" com.mypackage`.

Nonstandard Options Provided by the Standard doclet

The following are non-standard options provided by the standard doclet and are subject to change without notice.

`-Xdoclint`
Enables recommended checks for problems in Javadoc comments.

`-Xdoclint:(all|none|[-]`*`group`*`)`
Enable or disable specific checks for bad references, lack of accessibility, missing Javadoc comments, and reports errors for invalid Javadoc syntax and missing HTML tags.
This option enables the `javadoc` command to check for all documentation comments included in the generated output. You can select which items to include in the generated output with the standard options `-public`, `-protected`, `-package` and `-private`.
When the `-Xdoclint` is enabled, it reports issues with messages similar to the `javac` command. The `javadoc` command prints a message, a copy of the source line, and a caret pointing at the exact position where the error was detected. Messages may be either warnings or errors, depending on their severity and the likelihood to cause an error if the generated documentation were run through a validator. For example, bad references or missing Javadoc comments don't cause the `javadoc` command to generate invalid HTML, so these issues are reported as warnings. Syntax errors or missing HTML end tags cause the `javadoc` command to generate invalid output, so these issues are reported as errors.
`-Xdoclint` option validates input comments based upon the requested markup.
By default, the `-Xdoclint` option is enabled. Disable it with the option `-Xdoclint:none`.
The following options change what the `-Xdoclint` option reports:

- `-Xdoclint none` : Disables the `-Xdoclint` option
- `-Xdoclint group` : Enables *`group`* checks
- `-Xdoclint all` : Enables all groups of checks
- `-Xdoclint all,-group`: Enables all checks except `group` checks

The *`group`* variable has one of the following values:

- `accessibility`: Checks for the issues to be detected by an accessibility checker (for example, no caption or summary attributes specified in a `<table>` tag).
- `html`: Detects high-level HTML issues, such as putting block elements inside inline elements, or not closing elements that require an end tag. The rules are derived from the HTML 4 Specification or the HTML 5 Specification based on the standard doclet `html` output generation selected. This type of check enables the `javadoc` command to detect HTML issues that some browsers might not interpret as intended.
- `missing` : Checks for missing Javadoc comments or tags (for example, a missing comment or class, or a missing `@return` tag or similar tag on a method).

- `reference` : Checks for issues relating to the references to Java API elements from Javadoc tags (for example, item not found in `@see` , or a bad name after `@param`).
- `syntax` : Checks for low level issues like unescaped angle brackets (`<` and `>`) and ampersands (`&`) and invalid Javadoc tags.

You can specify the `-Xdoclint` option multiple times to enable the option to check errors and warnings in multiple categories. Alternatively, you can specify multiple error and warning categories by using the preceding options. For example, use either of the following commands to check for the HTML, syntax, and accessibility issues in the file `filename`.

```
javadoc -Xdoclint:html -Xdoclint:syntax -Xdoclint:accessibility filename
javadoc -Xdoclint:html,syntax,accessibility filename
```

> **Note:**
>
> The `javadoc` command doesn't guarantee the completeness of these checks. In particular, it isn't a full HTML compliance checker. The goal of the `-Xdoclint` option is to enable the `javadoc` command to report majority of common errors. The `javadoc` command doesn't attempt to fix invalid input, it just reports it.

`-Xdoclint/package:([-]) packages`
Enables or disables checks in specific packages. *`packages`* is a comma separated list of package specifiers. A package specifier is either a qualified name of a package or a package name prefix followed by `*`, which expands to all sub packages of the given package. Prefix the package specifier with — to disable checks for the specified packages.

`-Xdocrootparent url`
Replaces all `@docRoot` items followed by`/..` in Javadoc comments with the *`url`*.

java

You can use the `java` command to launch a Java application.

Synopsis

To execute a class:

```
java [options] mainclass [args...]
```

To execute a JAR file:

```
java [options] -jar jarfile [args...]
```

To execute the main class in a module:

```
java [options] [--module-path modulepath] --module module[/mainclass] [args...]
```

`options`
Specifies command-line options separated by spaces. See Overview of Java Options for a description of available options.

mainclass
Specifies the name of the class to be launched. Command-line entries following *classname* are the arguments for the main method.

jarfile
Specifies the name of the Java Archive (JAR) file to be called. Used only with the `-jar` option.

modulepath
Specifies the path to a semicolon-separated (;) list of directories in which each directory is a directory of modules Used only with the `--module-path` option. See Standard Options for Java.

***module*[*/mainclass*]**
Specifies the name of the initial *module* to resolve and, if it isn't specified by the *module*, then specifies the name of the *mainclass* to execute. Used only with the `-m` or `--module` option. See Standard Options for Java.

args
Specifies the arguments passed to the `main` method separated by spaces.

> **Note:**
>
> Arguments following the main class, `-jar` *jarfile* , `-m` or `--module` *module/mainclass* are passed as the arguments to the main class.

Description

The `java` command starts a Java application. It does this by starting the Java Runtime Environment (JRE), loading the specified class, and calling that class's `main()` method. The method must be declared `public` and `static`, it must not return any value, and it must accept a `String` array as a parameter. The method declaration has the following form:

```
public static void main(String[] args)
```

In JDK 9, a new launcher environment variable, `JDK_JAVA_OPTIONS`, has been introduced that prepends its content to the actual command line of the `java` launcher. See Using the JDK_JAVA_OPTIONS Launcher Environment Variable.

The `java` command can be used to launch a JavaFX application by loading a class that either has a `main()` method or that extends the `javafx.application.Application`. In the latter case, the launcher constructs an instance of the `Application` class, calls its `init()` method, and then calls the `start(javafx.stage.Stage)` method.

By default, the first argument that isn't an option of the `java` command is the fully qualified name of the class to be called. If the `-jar` option is specified, then its argument is the name of the JAR file containing class and resource files for the application. The startup class must be indicated by the `Main-Class` manifest header in its manifest file.

Arguments after the class file name or the JAR file name are passed to the `main()` method.

Windows: The `javaw` command is identical to `java`, except that with `javaw` there's no associated console window. Use `javaw` when you don't want a command prompt

window to appear. The `javaw` launcher will, however, display a dialog box with error information if a launch fails.

Using the JDK_JAVA_OPTIONS Launcher Environment Variable

`JDK_JAVA_OPTIONS` prepends its content to the options parsed from the command line. The content of the `JDK_JAVA_OPTIONS` environment variable is a list of arguments separated by white-space characters (as determined by `isspace()`). These are prepended to the command line arguments passed to `java` launcher. The encoding requirement for the environment variable is the same as the `java` command line on the system. `JDK_JAVA_OPTIONS` environment variable content is treated in the same manner as that specified in the command line.

Single (') or double (") quotes can be used to enclose arguments that contain whitespace characters. All content between the open quote and the first matching close quote are preserved by simply removing the pair of quotes. In case a matching quote is not found, the launcher will abort with an error message. *`@files`* are supported as they are specified in the command line. However, as in *`@files`*, use of a wildcard is not supported. In order to mitigate potential misuse of `JDK_JAVA_OPTIONS` behavior, options that specify the main class (such as `-jar`) or cause the `java` launcher to exit without executing the main class (such as `-h`) are disallowed in the environment variable. If any of these options appear in the environment variable, the launcher will abort with an error message. When `JDK_JAVA_OPTIONS` is set, the launcher prints a message to stderr as a reminder.

Example:

```
export JDK_JAVA_OPTIONS='-g @file1 -Dprop=value @file2 -Dws.prop="white spaces”'
$ java -Xint @file3
```

is equivalent to the command line:

```
java -g @file1 -Dprop=value @file2 -Dws.prop="white spaces" -Xint @file3
```

Overview of Java Options

The `java` command supports a wide range of options in the following categories:

- Standard Options for Java: Options guaranteed to be supported by all implementations of the Java Virtual Machine (JVM). They’re used for common actions, such as checking the version of the JRE, setting the class path, enabling verbose output, and so on.
- Extra Options for Java: General purpose options that are specific to the Java HotSpot Virtual Machine. They aren’t guaranteed to be supported by all JVM implementations, and are subject to change. These options start with `-X`.

The advanced options aren’t recommended for casual use. These are developer options used for tuning specific areas of the Java HotSpot Virtual Machine operation that often have specific system requirements and may require privileged access to system configuration parameters. Several examples of performance tuning are provided in Performance Tuning Examples. These options aren’t guaranteed to be supported by all JVM implementations and are subject to change. Advanced options start with `-XX`.

- Advanced Runtime Options for Java: Control the runtime behavior of the Java HotSpot VM.

- Advanced JIT Compiler Options for java: Control the dynamic just-in-time (JIT) compilation performed by the Java HotSpot VM.
- Advanced Serviceability Options for Java: Enable gathering system information and performing extensive debugging.
- Advanced Garbage Collection Options for Java: Control how garbage collection (GC) is performed by the Java HotSpot

Boolean options are used to either enable a feature that's disabled by default or disable a feature that's enabled by default. Such options don't require a parameter. Boolean `-XX` options are enabled using the plus sign (`-XX:+OptionName`) and disabled using the minus sign (`-XX:-OptionName`).

For options that require an argument, the argument may be separated from the option name by a space, a colon (:), or an equal sign (=), or the argument may directly follow the option (the exact syntax differs for each option). If you're expected to specify the size in bytes, then you can use no suffix, or use the suffix `k` or `K` for kilobytes (KB), `m` or `M` for megabytes (MB), or `g` or `G` for gigabytes (GB). For example, to set the size to 8 GB, you can specify either `8g`, `8192m`, `8388608k`, or `8589934592` as the argument. If you are expected to specify the percentage, then use a number from 0 to 1. For example, specify `0.25` for 25%.

The following sections describe the options that are obsolete, deprecated, and removed in JDK 9:

- Obsolete Java Options: Accepted but ignored. A warning is issued when they're used.
- Deprecated Java Options: Accepted and acted upon. A warning is issued when they're used.
- Removed Java Options: Removed in JDK 9. Using them results in an error.

Standard Options for Java

These are the most commonly used options supported by all implementations of the JVM.

> **Note:**
>
> To specify an argument for a long option, you can use either `--name=value` or `--name value`.

`-agentlib:libname[=options]`
Loads the specified native agent library. After the library name, a comma-separated list of options specific to the library can be used.

- **Oracle Solaris, Linux, and OS X:** If the option `-agentlib:foo` is specified, then the JVM attempts to load the library named `libfoo.so` in the location specified by the `LD_LIBRARY_PATH` system variable (on OS X this variable is `DYLD_LIBRARY_PATH`).
- **Windows:** If the option `-agentlib:foo` is specified, then the JVM attempts to load the library named `foo.dll` in the location specified by the `PATH` system variable.

The following example shows how to load the Java Debug Wire Protocol (JDWP) library and listen for the socket connection on port 8000, suspending the JVM before the main class loads:

```
-agentlib:jdwp=transport=dt_socket,server=y,address=8000
```

`-agentpath:`*`pathname`*`[=`*`options`*`]`
Loads the native agent library specified by the absolute path name. This option is equivalent to `-agentlib` but uses the full path and file name of the library.

`--class-path` *`classpath`*, `-classpath` *`classpath`* , or `-cp` *`classpath`*
A semicolon (;) separated list of directories, JAR archives, and ZIP archives to search for class files.
Specifying *`classpath`* overrides any setting of the `CLASSPATH` environment variable. If the class path option isn't used and *`classpath`* isn't set, then the user class path consists of the current directory (.).
As a special convenience, a class path element that contains a base name of an asterisk (*) is considered equivalent to specifying a list of all the files in the directory with the extension `.jar` or `.JAR` . A Java program can't tell the difference between the two invocations. For example, if the directory `mydir` contains `a.jar` and `b.JAR`, then the class path element `mydir/*` is expanded to `A.jar:b.JAR`, except that the order of JAR files is unspecified. All `.jar` files in the specified directory, even hidden ones, are included in the list. A class path entry consisting of an asterisk (*) expands to a list of all the jar files in the current directory. The `CLASSPATH` environment variable, where defined, is similarly expanded. Any class path wildcard expansion that occurs before the Java VM is started. Java programs never see wildcards that aren't expanded except by querying the environment, such as by calling `System.getenv("CLASSPATH")`.

`--disable-@files`
Can be used anywhere on the command line, including in an argument file, to prevent further `@filename` expansion. This option stops expanding `@argfiles` after the option.

`--module-path` *`modulepath...`* or `-p` *`modulepath`*
Searches for directories from a semicolon-separated (;) list of directories. Each directory is a directory of modules.

`--upgrade-module-path` *`modulepath...`*
Searches for directories from a semicolon-separated (;) list of directories. Each directory is a directory of modules that replace upgradeable modules in the runtime image.

`--add-modules` *`module`*`[,`*`module...`*`]`
Specifies the root modules to resolve in addition to the initial module. *`module`* also can be `ALL-DEFAULT`, `ALL-SYSTEM`, and `ALL-MODULE-PATH`.

`--list-modules`
Lists the observable modules and then exits.

`-d` *`module`* or `--describe-module` *`module`*
Describes a specified module and then exits.

`--dry-run`
Creates the VM but doesn't execute the main method. This `--dry-run` option might be useful for validating the command-line options such as the module system configuration.

`--validate-modules`
Validates all modules and exit. This option is helpful for finding conflicts and other errors with modules on the module path.

`-D`*`property`*`=`*`value`*
Sets a system property value. The *`property`* variable is a string with no spaces that represents the name of the property. The *`value`* variable is a string that represents the value of the property. If *`value`* is a string with spaces, then enclose it in quotation marks (for example `-Dfoo="foo bar"`).

`-disableassertions[:[`*`packagename`*`]...|:`*`classname`*`]` or `-da[:[`*`packagename`*`]...|:`*`classname`*`]`
Disables assertions. By default, assertions are disabled in all packages and classes. With no arguments, `-disableassertions` (`-da`) disables assertions in all packages and classes. With the *`packagename`* argument ending in `...`, the switch disables assertions in the specified package and any subpackages. If the argument is simply `...`, then the switch disables assertions in the unnamed package in the current working directory. With the *`classname`* argument, the switch disables assertions in the specified class. The `-disableassertions` (`-da`) option applies to all class loaders and to system classes (which don't have a class loader). There's one exception to this rule: If the option is provided with no arguments, then it doesn't apply to system classes. This makes it easy to disable assertions in all classes except for system classes. The `-disablesystemassertions` option enables you to disable assertions in all system classes. To explicitly enable assertions in specific packages or classes, use the `-enableassertions` (`-ea`) option. Both options can be used at the same time. For example, to run the `MyClass` application with assertions enabled in the package `com.wombat.fruitbat` (and any subpackages) but disabled in the class `com.wombat.fruitbat.Brickbat`, use the following command:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyClass
```

`-disablesystemassertions` or `-dsa`
Disables assertions in all system classes.

`-enableassertions[:[`*`packagename`*`]...|:`*`classname`*`]` or `-ea[:[`*`packagename`*`]...|:`*`classname`*`]`
Enables assertions. By default, assertions are disabled in all packages and classes. With no arguments, `-enableassertions` (`-ea`) enables assertions in all packages and classes. With the *`packagename`* argument ending in `...`, the switch enables assertions in the specified package and any subpackages. If the argument is simply `...`, then the switch enables assertions in the unnamed package in the current working directory. With the *`classname`* argument, the switch enables assertions in the specified class. The `-enableassertions` (`-ea`) option applies to all class loaders and to system classes (which don't have a class loader). There's one exception to this rule: If the option is provided with no arguments, then it doesn't apply to system classes. This makes it easy to enable assertions in all classes except for system classes. The `-enablesystemassertions` option provides a separate switch to enable assertions in all system classes. To explicitly disable assertions in specific packages or classes, use the `-disableassertions` (`-da`) option. If a single command contains multiple instances of these switches, then they're processed in order, before loading any classes. For example, to run the `MyClass` application with assertions enabled only in the package `com.wombat.fruitbat` (and any subpackages) but disabled in the class `com.wombat.fruitbat.Brickbat`, use the following command:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyClass
```

`-enablesystemassertions` or `-esa`
Enables assertions in all system classes.

`-help` or `-?`
Prints the help message to the error stream.

`--help`
Prints the help message to the output stream.

`-jar` *`filename`*
Executes a program encapsulated in a JAR file. The *`filename`* argument is the name of a JAR file with a manifest that contains a line in the form `Main-Class:`*`classname`* that defines the class with the `public static void main(String[] args)` method that serves as your application's starting point. When you use the `-jar` option, the specified JAR file is the source of all user classes, and other class path settings are ignored. If you're using JAR files, then see: jar

`-javaagent:`*`jarpath`*`[=`*`options`*`]`
Loads the specified Java programming language agent.

`--show-version` or `-showversion`
Displays version information and continues execution of the application. This option is equivalent to the `-version` option except that the latter instructs the JVM to exit after displaying version information.

`--show-module-resolution`
Shows module resolution output during startup.

`-splash:`*`imgname`*
Shows the splash screen with the image specified by *`imgname`*. HiDPI scaled images are automatically supported and used if available. The unscaled image file name, such as `image.ext`, should always be passed as the argument to the `-splash` option. The most appropriate scaled image provided is picked up automatically.
For example, to show the `splash.gif` file from the `images` directory when starting your application, use the following option:

```
-splash:images/splash.gif
```

`-verbose:class`
Displays information about each loaded class.

`-verbose:gc`
Displays information about each garbage collection (GC) event.

`-verbose:jni`
Displays information about the use of native methods and other Java Native Interface (JNI) activity.

`-verbose:module`
Displays information about the modules in use.

`--version` or `-version`
Displays version information and then exits. This option is equivalent to the `-showversion` option except that the latter doesn't instruct the JVM to exit after displaying version information.

`-X`
Prints the help on extra options to the error stream.

`--help-extra`
Prints the help on extra options to the output stream.

`@argument files`
Specifies one or more argument files prefixed by `@` used by the `java` command. It isn't uncommon for the `java` command line to be very long because of the `.jar` files needed in the classpath. The `@argument files` option overcomes command-line length limitations by enabling the launcher to expand the contents of argument files after shell expansion, but before argument processing. Contents in the argument files are expanded because otherwise, they would be specified on the command line until the `-Xdisable-@files` option was encountered.
The argument files can also contain the main class name and all options. If an argument file contains all of the options required by the `java` command, then the command line could simply be:
`java @argument files`
See java Command-Line Argument Files for a description and examples of using `@argument files`.

Extra Options for Java

The following `java` options are general purpose options that are specific to the Java HotSpot Virtual Machine.

`-Xbatch`
Disables background compilation. By default, the JVM compiles the method as a background task, running the method in interpreter mode until the background compilation is finished. The `-Xbatch` flag disables background compilation so that compilation of all methods proceeds as a foreground task until completed. This option is equivalent to `-XX:-BackgroundCompilation`.

`-Xbootclasspath/a:directories|zip|JAR files`
Specifies a list of directories, JAR files, and ZIP archives to append to the end of the default bootstrap class path.
Oracle Solaris, Linux, and OS X: Colons (`:`) separate entities in this list.
Windows: Semicolons (`;`) separate entities in this list.

`-Xcheck:jni`
Performs additional checks for Java Native Interface (JNI) functions. Specifically, it validates the parameters passed to the JNI function and the runtime environment data before processing the JNI request. It also checks for pending exceptions between JNI calls. Any invalid data encountered indicates a problem in the native code, and the JVM terminates with an irrecoverable error in such cases. Expect a performance degradation when this option is used.

`-Xcomp`
Forces compilation of methods on first invocation. By default, the Client VM (`-client`) performs 1,000 interpreted method invocations and the Server VM (`-server`) performs 10,000 interpreted method invocations to gather information for efficient compilation. Specifying the `-Xcomp` option disables interpreted method invocations to increase compilation performance at the expense of efficiency. You can also change the number of interpreted method invocations before compilation using the `-XX:CompileThreshold` option.

`-Xdebug`
Does nothing. Provided for backward compatibility.

`-Xdiag`
Shows additional diagnostic messages.

`-Xfuture`
Enables strict class-file format checks that enforce close conformance to the class-file format specification. Developers should use this flag when developing new code. Stricter checks may become the default in future releases.

`-Xint`
Runs the application in interpreted-only mode. Compilation to native code is disabled, and all bytecode is executed by the interpreter. The performance benefits offered by the just-in-time (JIT) compiler aren't present in this mode.

`-Xinternalversion`
Displays more detailed JVM version information than the `-version` option, and then exits.

`-Xloggc:`*`option`*
Enables the JVM unified logging framework. Logs GC status to a file with time stamps.

`-Xlog:`*`option`*
Configure or enable logging with the Java Virtual Machine (JVM) unified logging framework. See Enable Logging with the JVM Unified Logging Framework.

`-Xmixed`
Executes all bytecode by the interpreter except for hot methods, which are compiled to native code.

`-Xmn` ***`size`***
Sets the initial and maximum size (in bytes) of the heap for the young generation (nursery). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The young generation region of the heap is used for new objects. GC is performed in this region more often than in other regions. If the size for the young generation is too small, then a lot of minor garbage collections are performed. If the size is too large, then only full garbage collections are performed, which can take a long time to complete. Oracle recommends that you keep the size for the young generation greater than 25% and less than 50% of the overall heap size. The following examples show how to set the initial and maximum size of young generation to 256 MB using various units:

```
-Xmn256m
-Xmn262144k
-Xmn268435456
```

Instead of the `-Xmn` option to set both the initial and maximum size of the heap for the young generation, you can use `-XX:NewSize` to set the initial size and `-XX:MaxNewSize` to set the maximum size.

`-Xms` ***`size`***
Sets the initial size (in bytes) of the heap. This value must be a multiple of 1024 and greater than 1 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate

megabytes, `g` or `G` to indicate gigabytes. The following examples show how to set the size of allocated memory to 6 MB using various units:

```
-Xms6291456
-Xms6144k
-Xms6m
```

If you don't set this option, then the initial size is set as the sum of the sizes allocated for the old generation and the young generation. The initial size of the heap for the young generation can be set using the `-Xmn` option or the `-XX:NewSize` option.

`-Xmx` *`size`*
Specifies the maximum size (in bytes) of the memory allocation pool in bytes. This value must be a multiple of 1024 and greater than 2 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default value is chosen at runtime based on system configuration. For server deployments, `-Xms` and `-Xmx` are often set to the same value. The following examples show how to set the maximum allowed size of allocated memory to 80 MB using various units:

```
-Xmx83886080
-Xmx81920k
-Xmx80m
```

The `-Xmx` option is equivalent to `-XX:MaxHeapSize`.

`-Xnoclassgc`
Disables garbage collection (GC) of classes. This can save some GC time, which shortens interruptions during the application run. When you specify `-Xnoclassgc` at startup, the class objects in the application are left untouched during GC and are always be considered live. This can result in more memory being permanently occupied which, if not used carefully, throws an out-of-memory exception.

`-Xprof`
Profiles the running program and sends profiling data to standard output. This option is provided as a utility that's useful in program development and isn't intended to be used in production systems.

`-Xrs`
Reduces the use of operating system signals by the JVM. Shutdown hooks enable the orderly shutdown of a Java application by running user cleanup code (such as closing database connections) at shutdown, even if the JVM terminates abruptly.

- **Oracle Solaris, Linux, and OS X:**
  - The JVM catches signals to implement shutdown hooks for unexpected termination. The JVM uses `SIGHUP`, `SIGINT`, and `SIGTERM` to initiate the running of shutdown hooks.
  - The JVM catches signals to implement shutdown hooks for unexpected termination. The JVM uses `SIGHUP`, `SIGINT`, and `SIGTERM` to initiate the running of shutdown hooks.
  - Applications embedding the JVM frequently need to trap signals such as `SIGINT` or `SIGTERM`, which can lead to interference with the JVM signal

handlers. The `-Xrs` option is available to address this issue. When `-Xrs` is used, the signal masks for `SIGINT`, `SIGTERM`, `SIGHUP`, and `SIGQUIT` aren't changed by the JVM, and signal handlers for these signals aren't installed.

- **Windows:**
  - The JVM watches for console control events to implement shutdown hooks for unexpected termination. Specifically, the JVM registers a console control handler that begins shutdown-hook processing and returns `TRUE` for `CTRL_C_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, and `CTRL_SHUTDOWN_EVENT`.
  - The JVM uses a similar mechanism to implement the feature of dumping thread stacks for debugging purposes. The JVM uses `CTRL_BREAK_EVENT` to perform thread dumps.
  - If the JVM is run as a service (for example, as a servlet engine for a web server), then it can receive `CTRL_LOGOFF_EVENT` but shouldn't initiate shutdown because the operating system doesn't actually terminate the process. To avoid possible interference such as this, the `-Xrs` option can be used. When the `-Xrs` option is used, the JVM doesn't install a console control handler, implying that it doesn't watch for or process `CTRL_C_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, or `CTRL_SHUTDOWN_EVENT`.

There are two consequences of specifying `-Xrs`:

- **Oracle Solaris, Linux, and OS X:** `SIGQUIT` thread dumps aren't available.
- **Windows:** Ctrl + Break thread dumps aren't available.

User code is responsible for causing shutdown hooks to run, for example, by calling the `System.exit()` when the JVM is to be terminated.

`-Xshare:`*`mode`*
Sets the class data sharing (CDS) mode.
Possible *`mode`* arguments for this option include the following:

`auto`
Uses CDS if possible. This is the default value for Java HotSpot 32-Bit Client VM.

`on`
Requires the use of CDS. This option prints an error message and exits if class data sharing can't be used.

`off`
Instructs not to use CDS.

`-XshowSettings`
Shows all settings and then continues.

`-XshowSettings:`*`category`*
Shows settings and continues. Possible *`category`* arguments for this option include the following:

`all`
Shows all categories of settings. This is the default value.

`locale`
Shows settings related to locale.

properties
Shows settings related to system properties.

vm
Shows the settings of the JVM.

-Xss *size*
Sets the thread stack size (in bytes). Append the letter `k` or `K` to indicate KB, `m` or `M` to indicate MB, or `g` or `G` to indicate GB. The default value depends on the platform:

- Linux/x64 (64-bit): 1024 KB
- OS X (64-bit): 1024 KB
- Oracle Solaris/x64 (64-bit): 1024 KB
- Windows: The default value depends on virtual memory

The following examples set the thread stack size to 1024 KB in different units:

```
-Xss1m
-Xss1024k
-Xss1048576
```

This option is similar to `-XX:ThreadStackSize`.

-Xverify:*mode*
Sets the mode of the bytecode verifier. Bytecode verification ensures that class files are properly formed and satisfy the constraints listed in Verification of Class Files in the *The Java Virtual Machine Specification*.
Don't turn off verification because this reduces the protection provided by Java and could cause problems due to ill-formed class files.
Possible *`mode`* arguments for this option include the following:

remote
Verifies those classes that aren't loaded by the bootstrap class loader. This is the default behavior if you don't specify the `-Xverify` option.

all
Enables verification of all bytecodes.

none
Disables verification of all bytecodes. Use of `-Xverify:none` is unsupported.

--add-reads *module=target-module(,target-module)**
Updates *`module`* to read the *`target-module`*, regardless of the module declaration. *`target-module`* can be all unnamed to read all unnamed modules.

--add-exports *module/package=target-module(,target-module)**
Updates *`module`* to export *`package`* to *`target-module`*, regardless of module declaration. The *`target-module`* can be all unnamed to export to all unnamed modules.

--add-opens *module/package=target-module(,target-module)**
Updates *`module`* to open *`package`* to *`target-module`*, regardless of module declaration.

`--illegal-access=`*`parameter`*

> **Note:**
>
> This option is a new option in JDK 9 and may not be available in future JDK versions.

When present at run time, `--illegal-access=` takes a keyword *`parameter`* to specify a mode of operation:

> **Note:**
>
> Illegal-access operations to internal APIs from code on the class path are allowed by default in JDK 9.

- `permit`: This mode opens packages in JDK 9 that existed in JDK 8 to code on the class path. This allows code on class path that relies on the use of setAccessible to break into JDK internals, or to do other illegal access on members of classes in these packages, to work as per previous releases. This enables both static access (such as, by compiled bytecode) and deep reflective access. Deep reflective access is accomplished through the platform's reflection APIs. The first reflective-access operation to any such package causes a warning to be issued. However, no warnings are issued after the first occurrence. This single warning describes how to enable further warnings. This mode is the default for JDK 9 but will change in a future release.
- `warn`: This mode is identical to `permit` except that a warning message is issued for each illegal reflective-access operation.
- `debug`: This mode is identical to `warn` except that both a warning message and a stack trace are issued for each illegal reflective-access operation.
- `deny`: This mode disables all illegal-access operations except for those enabled by other command-line options, such as`--add-opens`. This mode will become the default in a future release.

The default mode, `--illegal-access=permit`, is intended to make you aware of code on the class path that reflectively accesses any JDK-internal APIs at least once. To learn about all such accesses, you can use the `warn` or the `debug` modes. For each library or framework on the class path that requires illegal access, you have two options:

- If the component's maintainers have already released a fixed version that no longer uses JDK-internal APIs then you can consider upgrading to that version.
- If the component still needs to be fixed, then you can contact its maintainers and ask them to replace their use of JDK-internal APIs with the proper exported APIs.

If you must continue to use a component that requires illegal access, then you can eliminate the warning messages by using one or more `--add-opens` options to open only those internal packages to which access is required.
To verify that your application is ready for a future version of the JDK, run it with `--illegal-access=deny` along with any necessary `--add-opens` options. Any remaining illegal-access errors will most likely be due to static references from compiled code to JDK-internal APIs. You can identify those by running the jdeps tool with the `--jdk-`

`internals` option. For performance reasons, JDK 9 does not issue warnings for illegal static-access operations.

`--limit-modules` *`module`*`[,`*`module`*`...]`
Specifies the limit of the universe of observable modules.

`--patch-module` *`module`*`=`*`file`*`(;`*`file`*`)*`
Overrides or augments a module with classes and resources in JAR files or directories.

`--disable-@files`
Can be used anywhere on the command line, including in an argument file, to prevent further `@filename` expansion. This option stops expanding `@argfiles` after the option.

Extra Options for Mac OS X

The following extra options are Mac OS X specific.

`-XstartOnFirstThread`
Runs the `main()` method on the first (AppKit) thread.

`-Xdock:name=`*`application name`*
Overrides the default application name displayed in dock.

`-Xdock:icon=`*`path to icon file`*
Overrides the default icon displayed in dock.

Advanced Runtime Options for Java

These `java` options control the runtime behavior of the Java HotSpot VM.

`-XX:+CheckEndorsedAndExtDirs`
Enables the option to prevent the `java` command from running a Java application if any of these directories exists and isn't empty:

- `lib/endorsed`
- `lib/ext`
- The systemwide platform-specific extension directory

The endorsed standards override mechanism and the extension mechanism are no longer supported.

`-XX:-CompactStrings`
Disables the Compact Strings feature. By default, this option is enabled. When this option is enabled, Java Strings containing only single-byte characters are internally represented and stored as single-byte-per-character Strings using ISO-8859-1 / Latin-1 encoding. This reduces, by 50%, the amount of space required for Strings containing only single-byte characters. For Java Strings containing at least one multibyte character: these are represented and stored as 2 bytes per character using UTF-16 encoding. Disabling the Compact Strings feature forces the use of UTF-16 encoding as the internal representation for all Java Strings.
Cases where it may be beneficial to disable Compact Strings include the following:

- When it’s known that an application overwhelmingly will be allocating multibyte character Strings

- In the unexpected event where a performance regression is observed in migrating from Java SE 8 to Java SE 9 and an analysis shows that Compact Strings introduces the regression

In both of these scenarios, disabling Compact Strings makes sense.

`-XX:CompilerDirectivesFile=file`
Adds directives from a file to the directives stack when a program starts. See Compiler Directives and the Command Line.

`-XX:CompilerDirectivesPrint`
Prints the directives stack when the program starts or when a new directive is added..

`-XX:ConcGCThreads=`*`n`*
Sets the number of parallel marking threads. Sets *`n`* to approximately 1/4 of the number of parallel garbage collection threads (ParallelGCThreads).

`-XX:+DisableAttachMechanism`
Disables the mechanism that lets tools attach to the JVM. By default, this option is disabled, meaning that the attach mechanism is enabled and you can use diagnostics and troubleshooting tools such as `jcmd`, `jstack`, `jmap`, and `jinfo`.

> **Note:**
>
> The tools such as jcmd, jinfo, jmap, and jstack shipped with the JDK aren't supported when using the tools from one JDK version to troubleshoot a different JDK version.

`-XX:ErrorFile=`*`filename`*
Specifies the path and file name to which error data is written when an irrecoverable error occurs. By default, this file is created in the current working directory and named hs_err_pid pid.log where *`pid`* is the identifier of the process that caused the error. The following example shows how to set the default log file (note that the identifier of the process is specified as `%p`):

```
-XX:ErrorFile=./hs_err_pid%p.log
```

- **Oracle Solaris, Linux, and OS X:** The following example shows how to set the error log to `/var/log/java/java_error.log`:

  ```
  -XX:ErrorFile=/var/log/java/java_error.log
  ```

- **Windows:** The following example shows how to set the error log file to `C:/log/java/java_error.log`:

  ```
  -XX:ErrorFile=C:/log/java/java_error.log
  ```

If the file can 't be created in the specified directory (due to insufficient space, permission problem, or another issue), then the file is created in the temporary directory for the operating system:

- **Oracle Solaris, Linux, and OS X:** The temporary directory is `/tmp`.
- **Windows:** The temporary directory is specified by the value of the `TMP` environment variable; if that environment variable isn't defined, then the value of the `TEMP` environment variable is used.

-XX:+FailOverToOldVerifier
Enables automatic failover to the old verifier when the new type checker fails. By default, this option is disabled and it's ignored (that is, treated as disabled) for classes with a recent bytecode version. You can enable it for classes with older versions of the bytecode.

-XX:+FlightRecorder
.Enables the use of the Java Flight Recorder (JFR) during the runtime of the application. This is a commercial feature that requires that you also specify the `-XX:+UnlockCommercialFeatures` option as follows:

```
java -XX:+UnlockCommercialFeatures -XX:+FlightRecorder
```

> **Note:**
>
> The `-XX:+FlightRecorder` option is no longer required to use JFR. This was a change made in JDK 8u40.

-XX:FlightRecorderOptions=*parameter=value*
Sets the parameters that control the behavior of JFR. This is a commercial feature that works in conjunction with the `-XX:+UnlockCommercialFeatures` option. The `-XX:+FlightRecorder` option is no longer required to use JFR. This was a change made in JDK 8u40.
The following list contains all available JFR parameters:

globalbuffersize=*size*
Specifies the total amount of primary memory (in bytes) used for data retention. Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, or `g` or `G` to specify the size in GB. By default, the size is set to 462848 bytes.

loglevel={quiet|error|warning|info|debug|trace}
Specify the amount of data written to the log file by JFR. By default, it's set to `info`.

maxchunksize=*size*
Specifies the maximum size (in bytes) of the data chunks in a recording. Append `k` or `K`, to specify the size in KB, or `m` or `M` to specify the size in MB, or `g` or `G` to specify the size in GB. By default, the maximum size of data chunks is set to 12 MB.

memorysize=*size*
Determines how much buffer memory should be used.

repository=*path*
Specifies the repository (a directory) for temporary disk storage. By default, the system's temporary directory is used.

samplethreads={true|false}
Specifies whether thread sampling is enabled. Thread sampling occurs only if the sampling event is enabled along with this parameter. By default, this parameter is enabled.

`stackdepth=depth`
Stack depth for stack traces by JFR. By default, the depth is set to 64 method calls. The maximum is 2048, and the minimum is 1.

`threadbuffersize=size`
Specifies the per-thread local buffer size (in bytes). Append `k` or `K`, to specify the size in KB, or `m` or `M` to specify the size in MB, `g` or `G` to specify the size in GB. Higher values for this parameter allow more data gathering without contention to flush it to the global storage. It can increase an application footprint in a thread-rich environment. By default, the local buffer size is set to 5 KB.

`transform={true|false}`
Specifies if event classes should be retransformed using JVMTI. If false, instrumentation will be added when event classes are loaded. By default it is true.

You can specify values for multiple parameters by separating them with a comma.

`-XX:InitiatingHeapOccupancyPercent=45`
Sets the Java heap occupancy threshold that triggers a marking cycle. The default occupancy is 45 percent of the entire Java heap.

`-XX:LargePageSizeInBytes=size`
Oracle Solaris: Sets the maximum size (in bytes) for large pages used for the Java heap. The *size* argument must be a power of 2 (2, 4, 8, 16, and so on). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. By default, the size is set to 0, meaning that the JVM chooses the size for large pages automatically. See Large Pages.
The following example describes how to set the large page size to 4 megabytes (MB):

```
-XX:LargePageSizeInBytes=4m
```

`-XX:MaxDirectMemorySize=size`
Sets the maximum total size (in bytes) of the `java.nio` package, direct-buffer allocations. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. By default, the size is set to 0, meaning that the JVM chooses the size for NIO direct-buffer allocations automatically.
The following examples illustrate how to set the NIO size to 1024 KB in different units:

```
-XX:MaxDirectMemorySize=1m
-XX:MaxDirectMemorySize=1024k
-XX:MaxDirectMemorySize=1048576
```

`-XX:-MaxFDLimit`
Disables the attempt to set the soft limit for the number of open file descriptors to the hard limit. By default, this option is enabled on all platforms, but is ignored on Windows. The only time that you may need to disable this is on Mac OS, where its use imposes a maximum of 10240, which is lower than the actual system maximum.

`-XX:MaxGCPauseMillis=200`
Sets a target value for the desired maximum pause time. The default value is 200 milliseconds. The specified value doesn't adapt to your heap size.

`-XX:NativeMemoryTracking=mode`
Specifies the mode for tracking JVM native memory usage. Possible *mode* arguments for this option include the following:

off
Instructs not to track JVM native memory usage. This is the default behavior if you don't specify the `-XX:NativeMemoryTracking` option.

summary
Tracks memory usage only by JVM subsystems, such as Java heap, class, code, and thread.

detail
In addition to tracking memory usage by JVM subsystems, track memory usage by individual `CallSite`, individual virtual memory region and its committed regions.

`-XX:ObjectAlignmentInBytes=`*`alignment`*
Sets the memory alignment of Java objects (in bytes). By default, the value is set to 8 bytes. The specified value should be a power of 2, and must be within the range of 8 and 256 (inclusive). This option makes it possible to use compressed pointers with large Java heap sizes.
The heap size limit in bytes is calculated as:

```
4GB * ObjectAlignmentInBytes
```

> **Note:**
>
> As the alignment value increases, the unused space between objects also increases. As a result, you may not realize any benefits from using compressed pointers with large Java heap sizes.

`-XX:OnError=`*`string`*
Sets a custom command or a series of semicolon-separated commands to run when an irrecoverable error occurs. If the string contains spaces, then it must be enclosed in quotation marks.

- **Oracle Solaris, Linux, and OS X:** The following example shows how the `-XX:OnError` option can be used to run the `gcore` command to create the core image, and the debugger is started to attach to the process in case of an irrecoverable error (the `%p` designates the current process):

  ```
  -XX:OnError="gcore %p;dbx - %p"
  ```

- **Windows:** The following example shows how the `-XX:OnError` option can be used to run the `userdump.exe` utility to obtain a crash dump in case of an irrecoverable error (the `%p` designates the current process). This example assumes that the path to the `userdump.exe` utility is specified in the `PATH` environment variable:

  ```
  -XX:OnError="userdump.exe %p"
  ```

`-XX:OnOutOfMemoryError=`*`string`*
Sets a custom command or a series of semicolon-separated commands to run when an `OutOfMemoryError` exception is first thrown. If the string contains spaces, then it must be enclosed in quotation marks. For an example of a command string, see the description of the `-XX:OnError` option.

`-XX:ParallelGCThreads=`*`n`*
Sets the value of the STW worker threads. Sets the value of n to the number of logical processors. The value of *n* is the same as the number of logical processors up to a

value of 8. If there are more than 8 logical processors, then this option sets the value of `n` to approximately 5/8 of the logical processors. This works in most cases except for larger SPARC systems where the value of `n` can be approximately 5/16 of the logical processors.

`-XX:+PerfDataSaveToFile`
If enabled, saves jstat binary data when the Java application exits. This binary data is saved in a file named `hsperfdata_pid`, where *`pid`* is the process identifier of the Java application that you ran. Use the*`jstat`* command to display the performance data contained in this file as follows:

```
jstat -class file:///path/hsperfdata_pid

jstat -gc file:///path/hsperfdata_pid
```

`-XX:+PrintCommandLineFlags`
Enables printing of ergonomically selected JVM flags that appeared on the command line. It can be useful to know the ergonomic values set by the JVM, such as the heap space size and the selected garbage collector. By default, this option is disabled and flags aren't printed.

`-XX:+PreserveFramePointer`
Selects between using the RBP register as a general purpose register `(-XX:-PreserveFramePointer)` and using the RBP register to hold the frame pointer of the currently executing method `(-XX:+PreserveFramePointer)`. If the frame pointer is available, then external profiling tools (for example, Linux perf) can construct more accurate stack traces.

`-XX:+PrintNMTStatistics`
Enables printing of collected native memory tracking data at JVM exit when native memory tracking is enabled (see `-XX:NativeMemoryTracking`). By default, this option is disabled and native memory tracking data isn't printed.

`-XX:+RelaxAccessControlCheck`
Decreases the amount of access control checks in the verifier. By default, this option is disabled, and it's ignored (that is, treated as disabled) for classes with a recent bytecode version. You can enable it for classes with older versions of the bytecode.

`-XX:+ResourceManagement`
Enables the use of Resource Management during the runtime of the application. This is a commercial feature that requires you to also specify the `-XX:+UnlockCommercialFeatures` option as follows:

```
java -XX:+UnlockCommercialFeatures -XX:+ResourceManagement
```

`-XX:ResourceManagementSampleInterval=`*`value in milliseconds`*
Sets the parameter that controls the sampling interval for Resource Management measurements, in milliseconds.
This option can be used only when Resource Management is enabled (that is, the `-XX:+ResourceManagement` option is specified).

`-XX:SharedArchiveFile=`*`path`*
Specifies the path and name of the class data sharing (CDS) archive file
See Application Class Data Sharing.

`-XX:SharedArchiveConfigFile=`*`shared_config_file`*
Specifies additional shared data added to the archive file.

`-XX:SharedClassListFile=file_name`
Specifies the text file that contains the names of the class files to store in the class data sharing (CDS) archive. This file contains the full name of one class file per line, except slashes (`/`) replace dots (`.`). For example, to specify the classes `java.lang.Object` and `hello.Main`, create a text file that contains the following two lines:

```
java/lang/Object
hello/Main
```

The class files that you specify in this text file should include the classes that are commonly used by the application. They may include any classes from the application, extension, or bootstrap class paths.
See Application Class Data Sharing.

`-XX:+ShowMessageBoxOnError`
Enables the display of a dialog box when the JVM experiences an irrecoverable error. This prevents the JVM from exiting and keeps the process active so that you can attach a debugger to it to investigate the cause of the error. By default, this option is disabled.

`-XX:StartFlightRecording=parameter=value`
Starts a JFR recording for the Java application. This is a commercial feature that works in conjunction with the `-XX:+UnlockCommercialFeatures` option. This option is equivalent to the `JFR.start` diagnostic command that starts a recording during runtime. You can set the following parameters when starting a JFR recording:

`delay=time`
Specifies the delay between the Java application launch time and the start of the recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `10m` means 10 minutes). By default, there's no delay, and this parameter is set to 0.

`duration=time`
Specifies the duration of the recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `5h` means 5 hours). By default, the duration isn't limited, and this parameter is set to 0.

`filename=path`
Specifies the path and name of the JFR recording log file.

`name=identifier`
Takes both the name and the identifier of a recording.

`maxage=time`
Specifies the maximum age of disk data to keep for the default recording. Append `s` to specify the time in seconds, `m` for minutes, `h` for hours, or `d` for days (for example, specifying `30s` means 30 seconds). By default, the maximum age is set to 15 minutes (`15m`).

`maxsize=size`
Specifies the maximum size (in bytes) of disk data to keep for the default recording. Append `k` or `K`, to specify the size in KB, `m` or `M` to specify the size in MB, or `g` or `G` to specify the size in GB. By default, the maximum size of disk data isn't limited, and this parameter is set to 0.

`settings=path`
Specifies the path and name of the event settings file (of type JFC). By default, the `default.jfc` file is used, which is located in `JAVA_HOME/jre/lib/jfr`.

You can specify values for multiple parameters by separating them with a comma.

`-XX:ThreadStackSize=size`
Sets the Java thread stack size (in kilobytes). Use of a scaling suffix, such as `k`, results in the scaling of the kilobytes value so that `-XX:ThreadStackSize=1k` sets the Java thread stack size to 1024*1024 bytes or 1 megabyte. The default value depends on the platform:

- Linux/x64 (64-bit): 1024 KB
- OS X (64-bit): 1024 KB
- Oracle Solaris/x64 (64-bit): 1024 KB
- Windows: The default value depends on the virtual memory.

The following examples show how to set the thread stack size to 1 megabyte in different units:

```
-XX:ThreadStackSize=1k
-XX:ThreadStackSize=1024
```

This option is similar to `-Xss`.

`-XX:+UnlockCommercialFeatures`
Enables the use of commercial features. Commercial features are included with Oracle Java SE Advanced or Oracle Java SE Suite packages, as defined in theOracle Java SE and Oracle Java Embedded Products page.
By default, this option is disabled and the JVM runs without the commercial features. After they're enabled for a JVM process, it isn't possible to disable their use for that process.

`-XX:+UseAppCDS`
Enables application class data sharing (AppCDS). To use AppCDS, you must also specify values for the options `-XX:SharedClassListFile` and `-XX:SharedArchiveFile` during both CDS dump time (see the option `-Xshare:dump`) and application run time. This is a commercial feature that requires you to also specify the `-XX:+UnlockCommercialFeatures` option. This is also an experimental feature; it may change in future releases.
See Application Class Data Sharing.

`-XX:-UseBiasedLocking`
Disables the use of biased locking. Some applications with significant amounts of uncontended synchronization may attain significant speedups with this flag enabled, but applications with certain patterns of locking may see slowdowns. .
By default, this option is enabled.

`-XX:-UseCompressedOops`
Disables the use of compressed pointers. By default, this option is enabled, and compressed pointers are used when Java heap sizes are less than 32 GB. When this option is enabled, object references are represented as 32-bit offsets instead of 64-bit pointers, which typically increases performance when running the application with Java heap sizes of less than 32 GB. This option works only for 64-bit JVMs.

It's also possible to use compressed pointers when Java heap sizes are greater than 32 GB. See the `-XX:ObjectAlignmentInBytes` option.

`-XX:+UseGCLogRotation`
Handles large log files. This option must be used with `-Xloggc:`*`filename`*.

`-XX:NumberOfGClogFiles=`*`number of files`*
Handles large log files. The *`number of files`* must be greater than or equal to `1`. The default is `1`.

`-XX:GCLogFileSize=`*`number`*
Handles large log files. The *`number`* can be in the form of *`number`*`M` or *`number`*`K`. The default is set to `512K`.

`-XX:+UseHugeTLBFS`
Linux only: This option is the equivalent of specifying `-XX:+UseLargePages`. This option is disabled by default. This option pre-allocates all large pages up-front, when memory is reserved; consequently the JVM can't dynamically grow or shrink large pages memory areas; see `-XX:UseTransparentHugePages` if you want this behavior.
See Large Pages.

`-XX:+UseLargePages`
Enables the use of large page memory. By default, this option is disabled and large page memory isn't used.
See Large Pages.

`-XX:+UseMembar`
Enables issuing of membars on thread-state transitions. This option is disabled by default on all platforms except ARM servers, where it's enabled. (It's recommended that you don't disable this option on ARM servers.)

`-XX:+UsePerfData`
Enables the `perfdata` feature. This option is enabled by default to allow JVM monitoring and performance testing. Disabling it suppresses the creation of the `hsperfdata_userid` directories. To disable the `perfdata` feature, specify `-XX:-UsePerfData`.

`-XX:+UseTransparentHugePages`
Linux only: Enables the use of large pages that can dynamically grow or shrink. This option is disabled by default. You may encounter performance problems with transparent huge pages as the OS moves other pages around to create huge pages; this option is made available for experimentation.

`-XX:+AllowUserSignalHandlers`
Enables installation of signal handlers by the application. By default, this option is disabled and the application isn't allowed to install signal handlers.

`-XX:VMOptionsFile=`*`filename`*
Allows user to specify VM options in a file, for example, `java -XX:VMOptionsFile=/var/my_vm_options HelloWorld`.

Advanced JIT Compiler Options for java

These `java` options control the dynamic just-in-time (JIT) compilation performed by the Java HotSpot VM.

`-XX:+AggressiveOpts`
Enables the use of aggressive performance optimization features. By default, this option is disabled and experimental performance features aren't used.

`-XX:AllocateInstancePrefetchLines=`*`lines`*
Sets the number of lines to prefetch ahead of the instance allocation pointer. By default, the number of lines to prefetch is set to 1:

```
-XX:AllocateInstancePrefetchLines=1
```

Only the Java HotSpot Server VM supports this option.

`-XX:AllocatePrefetchDistance=`*`size`*
Sets the size (in bytes) of the prefetch distance for object allocation. Memory about to be written with the value of new objects is prefetched up to this distance starting from the address of the last allocated object. Each Java thread has its own allocation point. Negative values denote that prefetch distance is chosen based on the platform. Positive values are bytes to prefetch. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default value is set to -1. The following example shows how to set the prefetch distance to 1024 bytes:

```
-XX:AllocatePrefetchDistance=1024
```

Only the Java HotSpot Server VM supports this option.

`-XX:AllocatePrefetchInstr=`*`instruction`*
Sets the prefetch instruction to prefetch ahead of the allocation pointer. Only the Java HotSpot Server VM supports this option. Possible values are from 0 to 3. The actual instructions behind the values depend on the platform. By default, the prefetch instruction is set to 0:

```
-XX:AllocatePrefetchInstr=0
```

Only the Java HotSpot Server VM supports this option.

`-XX:AllocatePrefetchLines=`*`lines`*
Sets the number of cache lines to load after the last object allocation by using the prefetch instructions generated in compiled code. The default value is 1 if the last allocated object was an instance, and 3 if it was an array.
The following example shows how to set the number of loaded cache lines to 5:

```
-XX:AllocatePrefetchLines=5
```

Only the Java HotSpot Server VM supports this option.

`-XX:AllocatePrefetchStepSize=`*`size`*
Sets the step size (in bytes) for sequential prefetch instructions. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, `g` or `G` to indicate gigabytes. By default, the step size is set to 16 bytes:

```
-XX:AllocatePrefetchStepSize=16
```

Only the Java HotSpot Server VM supports this option.

`-XX:AllocatePrefetchStyle=`*`style`*
Sets the generated code style for prefetch instructions. The *`style`* argument is an integer from 0 to 3:

0
Don’t generate prefetch instructions.

1
Execute prefetch instructions after each allocation. This is the default parameter.

2
Use the thread-local allocation block (TLAB) watermark pointer to determine when prefetch instructions are executed.

3
Use BIS instruction on SPARC for allocation prefetch.

Only the Java HotSpot Server VM supports this option.

`-XX:+BackgroundCompilation`
Enables background compilation. This option is enabled by default. To disable background compilation, specify `-XX:-BackgroundCompilation` (this is equivalent to specifying `-Xbatch`).

`-XX:CICompilerCount=`*`threads`*
Sets the number of compiler threads to use for compilation. By default, the number of threads is set to 2 for the server JVM, to 1 for the client JVM, and it scales to the number of cores if tiered compilation is used. The following example shows how to set the number of threads to 2:

```
-XX:CICompilerCount=2
```

`-XX:CompileCommand=`*`command`*`,`*`method`*`[,`*`option`*`]`
Specifies a command to perform on a method. For example, to exclude the `indexOf()` method of the `String` class from being compiled, use the following:

```
-XX:CompileCommand=exclude,java/lang/String.indexOf
```

Note that the full class name is specified, including all packages and subpackages separated by a slash (`/`). For easier cut-and-paste operations, it’s also possible to use the method name format produced by the `-XX:+PrintCompilation` and `-XX:+LogCompilation` options:

```
-XX:CompileCommand=exclude,java.lang.String::indexOf
```

If the method is specified without the signature, then the command isapplied to all methods with the specified name. However, you can also specify the signature of the method in the class file format. In this case, you should enclose the arguments in quotation marks, because otherwise the shell treats the semicolon as a command end. For example, if you want to exclude only the `indexOf(String)` method of the `String` class from being compiled, use the following:

```
-XX:CompileCommand="exclude,java/lang/String.indexOf,(Ljava/lang/String;)I"
```

You can also use the asterisk (*) as a wildcard for class and method names. For example, to exclude all `indexOf()` methods in all classes from being compiled, use the following:

```
-XX:CompileCommand=exclude,*.indexOf
```

The commas and periods are aliases for spaces, making it easier to pass compiler commands through a shell. You can pass arguments to `-XX:CompileCommand` using spaces as separators by enclosing the argument in quotation marks:

```
-XX:CompileCommand="exclude java/lang/String indexOf"
```

Note that after parsing the commands passed on the command line using the `-XX:CompileCommand` options, the JIT compiler then reads commands from the `.hotspot_compiler` file. You can add commands to this file or specify a different file using the `-XX:CompileCommandFile` option.
To add several commands, either specify the `-XX:CompileCommand` option multiple times, or separate each argument with the new line separator (`\n`). The following commands are available:

`break`
Sets a breakpoint when debugging the JVM to stop at the beginning of compilation of the specified method.

`compileonly`
Excludes all methods from compilation except for the specified method. As an alternative, you can use the `-XX:CompileOnly` option, which lets you specify several methods.

`dontinline`
Prevents inlining of the specified method.

`exclude`
Excludes the specified method from compilation.

`help`
Prints a help message for the `-XX:CompileCommand` option.

`inline`
Attempts to inline the specified method.

`log`
Excludes compilation logging (with the `-XX:+LogCompilation` option) for all methods except for the specified method. By default, logging is performed for all compiled methods.

`option`
Passes a JIT compilation option to the specified method in place of the last argument (*`option`*). The compilation option is set at the end, after the method name. For example, to enable the `BlockLayoutByFrequency` option for the `append()` method of the `StringBuffer` class, use the following:

```
-XX:CompileCommand=option,java/lang/StringBuffer.append,BlockLayoutByFrequency
```

You can specify multiple compilation options, separated by commas or spaces.

`print`
Prints generated assembler code after compilation of the specified method.

`quiet`
Instructs not to print the compile commands. By default, the commands that you specify with the `-XX:CompileCommand` option are printed; for example, if you exclude

from compilation the `indexOf()` method of the `String` class, then the following is printed to standard output:

```
CompilerOracle: exclude java/lang/String.indexOf
```

You can suppress this by specifying the `-XX:CompileCommand=quiet` option before other `-XX:CompileCommand` options.

`-XX:CompileCommandFile=`*`filename`*
Sets the file from which JIT compiler commands are read. By default, the `.hotspot_compiler` file is used to store commands performed by the JIT compiler. Each line in the command file represents a command, a class name, and a method name for which the command is used. For example, this line prints assembly code for the `toString()` method of the `String` class:

```
print java/lang/String toString
```

If you're using commands for the JIT compiler to perform on methods, then see the `-XX:CompileCommand` option.

`-XX:CompileOnly=`*`methods`*
Sets the list of methods (separated by commas) to which compilation should be restricted. Only the specified methods are compiled. Specify each method with the full class name (including the packages and subpackages). For example, to compile only the `length()` method of the `String` class and the `size()` method of the `List` class, use the following:

```
-XX:CompileOnly=java/lang/String.length,java/util/List.size
```

Note that the full class name is specified, including all packages and subpackages separated by a slash (`/`). For easier cut and paste operations, it's also possible to use the method name format produced by the `-XX:+PrintCompilation` and `-XX:+LogCompilation` options:

```
-XX:CompileOnly=java.lang.String::length,java.util.List::size
```

Although wildcards aren't supported, you can specify only the class or package name to compile all methods in that class or package, as well as specify just the method to compile methods with this name in any class:

```
-XX:CompileOnly=java/lang/String
-XX:CompileOnly=java/lang
-XX:CompileOnly=.length
```

`-XX:CompileThreshold=`*`invocations`*
Sets the number of interpreted method invocations before compilation. By default, in the server JVM, the JIT compiler performs 10,000 interpreted method invocations to gather information for efficient compilation. For the client JVM, the default setting is 1,500 invocations. This option is ignored when tiered compilation is enabled; see the option `-XX:-TieredCompilation`. The following example shows how to set the number of interpreted method invocations to 5,000:

```
-XX:CompileThreshold=5000
```

You can completely disable interpretation of Java methods before compilation by specifying the `-Xcomp` option.

`-XX:CompileThresholdScaling=scale`
Provides unified control of first compilation. This option controls when methods are first compiled for both the tiered and the nontiered modes of operation. The `CompileThresholdScaling` option has an integer value between 0 and +Inf and scales the thresholds corresponding to the current mode of operation (both tiered and nontiered). Setting `CompileThresholdScaling` to a value less than 1.0 results in earlier compilation while values greater than 1.0 delay compilation. Setting `CompileThresholdScaling` to 0 is equivalent to disabling compilation.

`-XX:+DoEscapeAnalysis`
Enables the use of escape analysis. This option is enabled by default. To disable the use of escape analysis, specify `-XX:-DoEscapeAnalysis`. Only the Java HotSpot Server VM supports this option.

`-XX:InitialCodeCacheSize=size`
Sets the initial code cache size (in bytes). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default value is set to 500 KB. The initial code cache size shouldn't be less than the system's minimal memory page size. The following example shows how to set the initial code cache size to 32 KB:

```
-XX:InitialCodeCacheSize=32k
```

`-XX:+Inline`
Enables method inlining. This option is enabled by default to increase performance. To disable method inlining, specify `-XX:-Inline`.

`-XX:InlineSmallCode=size`
Sets the maximum code size (in bytes) for compiled methods that should be inlined. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. Only compiled methods with the size smaller than the specified size is inlined. By default, the maximum code size is set to 1000 bytes:

```
-XX:InlineSmallCode=1000
```

`-XX:+LogCompilation`
Enables logging of compilation activity to a file named `hotspot.log` in the current working directory. You can specify a different log file path and name using the `-XX:LogFile` option.
By default, this option is disabled and compilation activity isn't logged. The `-XX:+LogCompilation` option has to be used together with the `-XX:UnlockDiagnosticVMOptions` option that unlocks diagnostic JVM options.
You can enable verbose diagnostic output with a message printed to the console every time a method is compiled by using the `-XX:+PrintCompilation` option.

`-XX:MaxInlineSize=size`
Sets the maximum bytecode size (in bytes) of a method to be inlined. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. By default, the maximum bytecode size is set to 35 bytes:

```
-XX:MaxInlineSize=35
```

`-XX:MaxNodeLimit=nodes`
Sets the maximum number of nodes to be used during single method compilation. By default, the maximum number of nodes is set to 65,000:

```
-XX:MaxNodeLimit=65000
```

`-XX:NonNMethodCodeHeapSize=size`
Sets the size in bytes of the code segment containing nonmethod code.
A nonmethod code segment containing nonmethod code, such as compiler buffers and the bytecode interpreter. This code type stays in the code cache forever. This flag is used only if `–XX:SegmentedCodeCache` is enabled.

`–XX:NonProfiledCodeHeapSize=size`
Sets the size in bytes of the code segment containing nonprofiled methods. This flag is used only if `–XX:SegmentedCodeCache` is enabled.

`-XX:MaxTrivialSize=size`
Sets the maximum bytecode size (in bytes) of a trivial method to be inlined. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. By default, the maximum bytecode size of a trivial method is set to 6 bytes:

```
-XX:MaxTrivialSize=6
```

`-XX:+OptimizeStringConcat`
Enables the optimization of `String` concatenation operations. This option is enabled by default. To disable the optimization of `String` concatenation operations, specify `-XX:-OptimizeStringConcat`. Only the Java HotSpot Server VM supports this option.

`-XX:+PrintAssembly`
Enables printing of assembly code for bytecoded and native methods by using the external `hsdis-<arch>.so` or `.dll` library. For 64-bit VM on Windows, it's `hsdis-amd64.dll`. This let's you to see the generated code, which may help you to diagnose performance issues.
By default, this option is disabled and assembly code isn't printed. The `-XX:+PrintAssembly` option has to be used together with the `-XX:UnlockDiagnosticVMOptions` option that unlocks diagnostic JVM options.

`-XX:ProfiledCodeHeapSize=size`
Sets the size in bytes of the code segment containing profiled methods. This flag is used only if `–XX:SegmentedCodeCache` is enabled.

`-XX:+PrintCompilation`
Enables verbose diagnostic output from the JVM by printing a message to the console every time a method is compiled. This let's you to see which methods actually get compiled. By default, this option is disabled and diagnostic output isn't printed.
You can also log compilation activity to a file by using the `-XX:+LogCompilation` option.

`-XX:+PrintInlining`
Enables printing of inlining decisions. This let's you to see which methods are getting inlined.
By default, this option is disabled and inlining information isn't printed. The `-XX:+PrintInlining` option has to be used together with the `-XX:+UnlockDiagnosticVMOptions` option that unlocks diagnostic JVM options.

`-XX:ReservedCodeCacheSize=size`
Sets the maximum code cache size (in bytes) for JIT-compiled code. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default maximum code cache size is 240 MB; if you disable tiered compilation with the option `-XX:-TieredCompilation`, then the default size is 48 MB. This option has a limit of 2 GB; otherwise, an error is generated. The maximum code cache size shouldn't be less than the initial code cache size; see the option `-XX:InitialCodeCacheSize`. This option is equivalent to `-Xmaxjitcodesize`.

-XX:RTMAbortRatio=*abort_ratio*
Specifies the RTM abort ratio is specified as a percentage (%) of all executed RTM transactions. If a number of aborted transactions becomes greater than this ratio, then the compiled code is deoptimized. This ratio is used when the `-XX:+UseRTMDeopt` option is enabled. The default value of this option is 50. This means that the compiled code is deoptimized if 50% of all transactions are aborted.

-XX:+SegmentedCodeCache
Enables segmentation of the code cache. Without the `–XX:+SegmentedCodeCache`, the code cache consists of one large segment. With `–XX:+SegmentedCodeCache`, we have separate segments for nonmethod, profiled method, and nonprofiled method code. These segments aren't resized at runtime. The feature is enabled by default if tiered compilation is enabled (`-XX:+TieredCompilation`) and `-XX:ReservedCodeCacheSize` >= 240 MB. The advantages are better control of the memory footprint, reduced code fragmentation, and better iTLB/iCache behavior due to improved locality. iTLB/iCache is a CPU-specific term meaning Instruction Translation Lookaside Buffer (ITLB). ICache is an instruction cache in theCPU. The implementation of the code cache can be found in the file: `/share/vm/code/codeCache.cpp`.

-XX:StartAggressiveSweepingAt=*percent*
Forces stack scanning of active methods to aggressively remove unused code when only the given percentage of the code cache is free. The default value is 10%.

-XX:RTMRetryCount=*number_of_retries*
Specifies the number of times that the RTM locking code is retried, when it is aborted or busy, before falling back to the normal locking mechanism. The default value for this option is 5. The `-XX:UseRTMLocking` option must be enabled.

-XX:-TieredCompilation
Disables the use of tiered compilation. By default, this option is enabled. Only the Java HotSpot Server VM supports this option.

-XX:+UseAES
Enables hardware-based AES intrinsics for Intel, AMD, and SPARC hardware. Intel Westmere (2010 and newer), AMD Bulldozer (2011 and newer), and SPARC (T4 and newer) are the supported hardware. The `-XX:+UseAES` is used in conjunction with UseAESIntrinsics. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.

-XX:+UseAESIntrinsics
Enables `-XX:+UseAES` and `-XX:+UseAESIntrinsics` flags by default and are supported only for the Java HotSpot Server VM. To disable hardware-based AES intrinsics, specify `-XX:-UseAES -XX:-UseAESIntrinsics`. For example, to enable hardware AES, use the following flags:

```
-XX:+UseAES -XX:+UseAESIntrinsics
```

Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`. To support UseAES and UseAESIntrinsics flags, use the `-server` option to select the Java HotSpot Server VM. These flags aren't supported on Client VM.

-XX:+UseCMoveUnconditionally
Generates CMove (scalar and vector) instructions regardless of profitability analysis.

-XX:+UseCodeCacheFlushing
Enables flushing of the code cache before shutting down the compiler. This option is enabled by default. To disable flushing of the code cache before shutting down the compiler, specify `-XX:-UseCodeCacheFlushing`.

-XX:+UseCondCardMark
Enables checking if the card is already marked before updating the card table. This option is disabled by default. It should be used only on machines with multiple sockets, where it increases the performance of Java applications that rely on concurrent operations. Only the Java HotSpot Server VM supports this option.

-XX:+UseCountedLoopSafepoints
Keeps safepoints in counted loops. Its default value is false.

-XX:+UseFMA
Enables hardware-based FMA intrinsics for hardware where FMA instructions are available (such as, Intel, SPARC, and ARM64). FMA intrinsics are generated for the `java.lang.Math.fma(a, b, c)` methods that calculate the value of (`a * b + c`) expressions.

-XX:+UseRTMDeopt
Autotunes RTM locking depending on the abort ratio. This ratio is specified by the `-XX:RTMAbortRatio` option. If the number of aborted transactions exceeds the abort ratio, then the method containing the lock is deoptimized and recompiled with all locks as normal locks. This option is disabled by default. The `-XX:+UseRTMLocking` option must be enabled.

-XX:+UseRTMLocking
Generates Restricted Transactional Memory (RTM) locking code for all inflated locks, with the normal locking mechanism as the fallback handler. This option is disabled by default. Options related to RTM are available only for the Java HotSpot Server VM on x86 CPUs that support Transactional Synchronization Extensions (TSX).
RTM is part of Intel's TSX, which is an x86 instruction set extension and facilitates the creation of multithreaded applications. RTM introduces the new instructions `XBEGIN`, `XABORT`, `XEND`, and `XTEST`. The `XBEGIN` and `XEND` instructions enclose a set of instructions to run as a transaction. If no conflict is found when running the transaction, then the memory and register modifications are committed together at the `XEND` instruction. The `XABORT` instruction can be used to explicitly abort a transaction and the `XEND` instruction checks if a set of instructions is being run in a transaction.
A lock on a transaction is inflated when another thread tries to access the same transaction, thereby blocking the thread that didn't originally request access to the transaction. RTM requires that a fallback set of operations be specified in case a transaction aborts or fails. An RTM lock is a lock that has been delegated to the TSX's system.
RTM improves performance for highly contended locks with low conflict in a critical region (which is code that must not be accessed by more than one thread concurrently). RTM also improves the performance of coarse-grain locking, which typically doesn't perform well in multithreaded applications. (Coarse-grain locking is the strategy of holding locks for long periods to minimize the overhead of taking and releasing locks, while fine-grained locking is the strategy of trying to achieve maximum parallelism by locking only when necessary and unlocking as soon as possible.) Also, for lightly contended locks that are used by different threads, RTM can reduce false cache line sharing, also known as cache line ping-pong. This occurs when multiple threads from different processors are accessing different resources, but the resources share the same cache line. As a result, the processors repeatedly

invalidate the cache lines of other processors, which forces them to read from main memory instead of their cache.

-XX:+UseSHA
Enables hardware-based intrinsics for SHA crypto hash functions for SPARC hardware. The `UseSHA` option is used in conjunction with the `UseSHA1Intrinsics`, `UseSHA256Intrinsics`, and `UseSHA512Intrinsics` options.
The `UseSHA` and `UseSHA*Intrinsics` flags are enabled by default, and are supported only for Java HotSpot Server VM 64-bit on SPARC T4 and newer.
This feature is applicable only when using the `sun.security.provider.Sun` provider for SHA operations. Flags that control intrinsics now require the option `-XX:+UnlockDiagnosticVMOptions`.
To disable all hardware-based SHA intrinsics, specify the `-XX:-UseSHA`. To disable only a particular SHA intrinsic, use the appropriate corresponding option. For example: `-XX:-UseSHA256Intrinsics`.

-XX:+UseSHA1Intrinsics
Enables intrinsics for SHA-1 crypto hash function. Flags that control intrinsics now require the option`-XX:+UnlockDiagnosticVMOptions`.

-XX:+UseSHA256Intrinsics
Enables intrinsics for SHA-224 and SHA-256 crypto hash functions. Flags that control intrinsics now require the option`-XX:+UnlockDiagnosticVMOptions`.

-XX:+UseSHA512Intrinsics
Enables intrinsics for SHA-384 and SHA-512 crypto hash functions. Flags that control intrinsics now require the option`-XX:+UnlockDiagnosticVMOptions`.

-XX:+UseSuperWord
Enables the transformation of scalar operations into superword operations. Superword is a vectorization optimization. This option is enabled by default. To disable the transformation of scalar operations into superword operations, specify `-XX:-UseSuperWord`. Only the Java HotSpot Server VM supports this option.

Advanced Serviceability Options for Java

These `java` options provide the ability to gather system information and perform extensive debugging.

-XX:+ExtendedDTraceProbes
Oracle Solaris, Linux, and OS X: Enables additional `dtrace` tool probes that affect the performance. By default, this option is disabled and `dtrace` performs only standard probes.

-XX:+HeapDumpOnOutOfMemoryError
Enables the dumping of the Java heap to a file in the current directory by using the heap profiler (HPROF) when a `java.lang.OutOfMemoryError` exception is thrown. You can explicitly set the heap dump file path and name using the `-XX:HeapDumpPath` option. By default, this option is disabled and the heap isn't dumped when an `OutOfMemoryError` exception is thrown.

-XX:HeapDumpPath=*path*
Sets the path and file name for writing the heap dump provided by the heap profiler (HPROF) when the `-XX:+HeapDumpOnOutOfMemoryError` option is set. By default, the file is created in the current working directory, and it's named `java_pidpid.hprof` where

`pid` is the identifier of the process that caused the error. The following example shows how to set the default file explicitly (`%p` represents the current process identifier):

```
-XX:HeapDumpPath=./java_pid%p.hprof
```

- **Oracle Solaris, Linux, and OS X:** The following example shows how to set the heap dump file to `/var/log/java/java_heapdump.hprof`:

  ```
  -XX:HeapDumpPath=/var/log/java/java_heapdump.hprof
  ```

- **Windows:** The following example shows how to set the heap dump file to `C:/log/java/java_heapdump.log`:

  ```
  -XX:HeapDumpPath=C:/log/java/java_heapdump.log
  ```

`-XX:LogFile=`*`path`*
Sets the path and file name to where log data is written. By default, the file is created in the current working directory, and it's named `hotspot.log`.

- **Oracle Solaris, Linux, and OS X:** The following example shows how to set the log file to `/var/log/java/hotspot.log`:

  ```
  -XX:LogFile=/var/log/java/hotspot.log
  ```

- **Windows:** The following example shows how to set the log file to `C:/log/java/hotspot.log`:

  ```
  -XX:LogFile=C:/log/java/hotspot.log
  ```

`-XX:+PrintClassHistogram`
Enables printing of a class instance histogram after one of the following events:

- **Oracle Solaris, Linux, and OS X:** `Control+Break`
- **Windows:** `Control+C` (`SIGTERM`)

By default, this option is disabled.
Setting this option is equivalent to running the `jmap -histo` command, or the `jcmd` *`pid`* `GC.class_histogram` command, where *`pid`* is the current Java process identifier.

`-XX:+PrintConcurrentLocks`
Enables printing of `java.util.concurrent` locks after one of the following events:

- **Oracle Solaris, Linux, and OS X:** `Control+Break`
- **Windows:** `Control+C` (`SIGTERM`)

By default, this option is disabled.
Setting this option is equivalent to running the `jstack -l` command or the`jcmd` *`pid`* `Thread.print -l` command, where *`pid`* is the current Java process identifier.

`-XX:+PrintFlagsRanges`
Prints the range specified and allows automatic testing of the values. See Validate Java Virtual Machine Flag Arguments.

`-XX:+UnlockDiagnosticVMOptions`
Unlocks the options intended for diagnosing the JVM. By default, this option is disabled and diagnostic options aren't available.

Advanced Garbage Collection Options for Java

These `java` options control how garbage collection (GC) is performed by the Java HotSpot VM.

-XX:+AggressiveHeap
Enables Java heap optimization. This sets various parameters to be optimal for long-running jobs with intensive memory allocation, based on the configuration of the computer (RAM and CPU). By default, the option is disabled and the heap isn't optimized.

-XX:+AlwaysPreTouch
Enables touching of every page on the Java heap during JVM initialization. This gets all pages into memory before entering the `main()` method. The option can be used in testing to simulate a long-running system with all virtual memory mapped to physical memory. By default, this option is disabled and all pages are committed as JVM heap space fills.

-XX:+CMSClassUnloadingEnabled
Enables class unloading when using the concurrent mark-sweep (CMS) garbage collector. This option is enabled by default. To disable class unloading for the CMS garbage collector, specify `-XX:-CMSClassUnloadingEnabled`.

-XX:CMSExpAvgFactor=*percent*
Sets the percentage of time (0 to 100) used to weight the current sample when computing exponential averages for the concurrent collection statistics. By default, the exponential averages factor is set to 25%. The following example shows how to set the factor to 15%:

```
-XX:CMSExpAvgFactor=15
```

-XX:CMSIncrementalDutyCycle=*percent*
Sets the percentage (0 to 100) of time between minor collections that the CMS collector is allowed to run. If `CMSIncrementalPacing` is enabled, then this is just the initial value. The default value is 10.

-XX:CMSIncrementalDutyCycleMin=*percent*
Sets the percentage (0 to 100) that's the lower bound on the duty cycle when `CMSIncrementalPacing` is enabled. The default value is 0.

-XX:CMSIncrementalDutySafetyFactor=*percent*
Sets the percentage (0 to 100) used to add conservatism when computing the duty cycle. The default value is 10.

-XX:CMSIncrementalOffset=*percent*
Sets the percentage (0 to 100) by which the incremental mode duty cycle is shifted to the right within the period between minor collections. The default value is 0.

-XX:+CMSIncrementalPacing
Enables automatic pacing. The incremental mode duty cycle is automatically adjusted based on statistics collected while the JVM is running. By default, this option is disabled.

-XX:+CMSScavengeBeforeRemark
Enables scavenging attempts before the CMS remark step. By default, this option is disabled.

-XX:CMSTriggerRatio=*percent*
Sets the percentage (0 to 100) of the value specified by the option `-XX:MinHeapFreeRatio` that's allocated before a CMS collection cycle commences. The default value is set to 80%.

The following example shows how to set the occupancy fraction to 75%:

```
-XX:CMSTriggerRatio=75
```

-XX:ConcGCThreads=*threads*
Sets the number of threads used for concurrent GC. Sets *threads* to approximately 1/4 of the number of parallel garbage collection threads. The default value depends on the number of CPUs available to the JVM.
For example, to set the number of threads for concurrent GC to 2, specify the following option:

```
-XX:ConcGCThreads=2
```

-XX:+DisableExplicitGC
Enables the option that disables processing of calls to the `System.gc()` method. This option is disabled by default, meaning that calls to `System.gc()` are processed. If processing of calls to `System.gc()` is disabled, then the JVM still performs GC when necessary.

-XX:+ExplicitGCInvokesConcurrent
Enables invoking of concurrent GC by using the `System.gc()` request. This option is disabled by default and can be enabled only together with the `-XX:+UseConcMarkSweepGC` and `-XX:+UseG1GC` options.

-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses
Enables invoking of concurrent GC by using the `System.gc()` request and unloading of classes during the concurrent GC cycle. This option is disabled by default and can be enabled only together with the `-XX:+UseConcMarkSweepGC` option.

-XX:G1HeapRegionSize=*size*
Sets the size of the regions into which the Java heap is subdivided when using the garbage-first (G1) collector. The value is a power of 2 and can range from 1 MB to 32 MB. The goal is to have around 2048 regions based on the minimum Java heap size. The default region size is determined ergonomically based on the heap size.
The following example sets the size of the subdivisions to 16 MB:

```
-XX:G1HeapRegionSize=16m
```

-XX:G1HeapWastePercent=*percent*
Sets the percentage of heap that you're willing to waste. The Java HotSpot VM doesn't initiate the mixed garbage collection cycle when the reclaimable percentage is less than the heap waste percentage. The default is 5 percent.

-XX:G1MaxNewSizePercent=*percent*
Sets the percentage of the heap size to use as the maximum for the young generation size. The default value is 60 percent of your Java heap.
This is an experimental flag. This setting replaces the `-XX:DefaultMaxNewGenPercent` setting.
This setting isn't available in Java HotSpot VM build 23 or earlier.

-XX:G1MixedGCCountTarget=*number*
Sets the target number of mixed garbage collections after a marking cycle to collect old regions with at most `G1MixedGCLIveThresholdPercent` live data. The default is 8 mixed garbage collections. The goal for mixed collections is to be within this target number.
This setting isn't available in Java HotSpot VM build 23 or earlier.

`-XX:G1MixedGCLiveThresholdPercent=percent`
Sets the occupancy threshold for an old region to be included in a mixed garbage collection cycle. The default occupancy is 85 percent.
This is an experimental flag. This setting replaces the `-XX:G1OldCSetRegionLiveThresholdPercent` setting.
This setting isn't available in Java HotSpot VM build 23 or earlier.

`-XX:G1NewSizePercent=percent`
Sets the percentage of the heap to use as the minimum for the young generation size. The default value is 5 percent of your Java heap.
This is an experimental flag. This setting replaces the `-XX:DefaultMinNewGenPercent` setting.
This setting isn't available in Java HotSpot VM build 23 or earlier.

`-XX:G1OldCSetRegionThresholdPercent=percent`
Sets an upper limit on the number of old regions to be collected during a mixed garbage collection cycle. The default is 10 percent of the Java heap.
This setting isn't available in Java HotSpot VM build 23 or earlier.

`-XX:G1ReservePercent=percent`
Sets the percentage of the heap (0 to 50) that's reserved as a false ceiling to reduce the possibility of promotion failure for the G1 collector. When you increase or decrease the percentage, ensure that you adjust the total Java heap by the same amount. By default, this option is set to 10%.
The following example sets the reserved heap to 20%:

```
-XX:G1ReservePercent=20
```

`-XX:InitialHeapOccupancyPercent=percent`
Sets the Java heap occupancy threshold that triggers a marking cycle. The default occupancy is 45 percent of the entire Java heap.

`-XX:InitialHeapSize=size`
Sets the initial size (in bytes) of the memory allocation pool. This value must be either 0, or a multiple of 1024 and greater than 1 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default value is selected at run time based on the system configuration.
The following examples show how to set the size of allocated memory to 6 MB using various units:

```
-XX:InitialHeapSize=6291456
-XX:InitialHeapSize=6144k
-XX:InitialHeapSize=6m
```

If you set this option to 0, then the initial size is set as the sum of the sizes allocated for the old generation and the young generation. The size of the heap for the young generation can be set using the `-XX:NewSize` option.

`-XX:InitialSurvivorRatio=ratio`
Sets the initial survivor space ratio used by the throughput garbage collector (which is enabled by the `-XX:+UseParallelGC` and/or `-XX:+UseParallelOldGC` options). Adaptive sizing is enabled by default with the throughput garbage collector by using the `-XX:+UseParallelGC` and `-XX:+UseParallelOldGC` options, and the survivor space is resized according to the application behavior, starting with the initial value. If adaptive sizing is disabled (using the `-XX:-UseAdaptiveSizePolicy` option), then the `-XX:SurvivorRatio`

option should be used to set the size of the survivor space for the entire execution of the application.
The following formula can be used to calculate the initial size of survivor space (S) based on the size of the young generation (Y), and the initial survivor space ratio (R):

```
S=Y/(R+2)
```

The 2 in the equation denotes two survivor spaces. The larger the value specified as the initial survivor space ratio, the smaller the initial survivor space size.
By default, the initial survivor space ratio is set to 8. If the default value for the young generation space size is used (2 MB), then the initial size of the survivor space is 0.2 MB.
The following example shows how to set the initial survivor space ratio to 4:

```
-XX:InitialSurvivorRatio=4
```

`-XX:InitiatingHeapOccupancyPercent=`*`percent`*
Sets the percentage of the heap occupancy (0 to 100) at which to start a concurrent GC cycle. It's used by garbage collectors that trigger a concurrent GC cycle based on the occupancy of the entire heap, not just one of the generations (for example, the G1 garbage collector).
By default, the initiating value is set to 45%. A value of 0 implies nonstop GC cycles.
The following example shows how to set the initiating heap occupancy to 75%:

```
-XX:InitiatingHeapOccupancyPercent=75
```

`-XX:MaxGCPauseMillis=`*`time`*
Sets a target for the maximum GC pause time (in milliseconds). This is a soft goal, and the JVM will make its best effort to achieve it. The specified value doesn't adapt to your heap size. By default, there's no maximum pause time value.
The following example shows how to set the maximum target pause time to 500 ms:

```
-XX:MaxGCPauseMillis=500
```

`-XX:MaxHeapSize=`*`size`*
Sets the maximum size (in byes) of the memory allocation pool. This value must be a multiple of 1024 and greater than 2 MB. Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default value is selected at run time based on the system configuration. For server deployments, the options `-XX:InitialHeapSize` and `-XX:MaxHeapSize` are often set to the same value.
The following examples show how to set the maximum allowed size of allocated memory to 80 MB using various units:

```
-XX:MaxHeapSize=83886080
-XX:MaxHeapSize=81920k
-XX:MaxHeapSize=80m
```

On Oracle Solaris 7 and Oracle Solaris 8 SPARC platforms, the upper limit for this value is approximately 4,000 MB minus overhead amounts. On Oracle Solaris 2.6 and x86 platforms, the upper limit is approximately 2,000 MB minus overhead amounts. On Linux platforms, the upper limit is approximately 2,000 MB minus overhead amounts.
The `-XX:MaxHeapSize` option is equivalent to `-Xmx`.

`-XX:MaxHeapFreeRatio=`*`percent`*
Sets the maximum allowed percentage of free heap space (0 to 100) after a GC event. If free heap space expands above this value, then the heap is shrunk. By default, this value is set to 70%.

Minimize the Java heap size by lowering the values of the parameters `MaxHeapFreeRatio` (default value is 70%) and `MinHeapFreeRatio` (default value is 40%) with the command-line options `-XX:MaxHeapFreeRatio` and `-XX:MinHeapFreeRatio`. Lowering `MaxHeapFreeRatio` to as low as 10% and `MinHeapFreeRatio` to 5% has successfully reduced the heap size without too much performance regression; however, results may vary greatly depending on your application. Try different values for these parameters until they're as low as possible yet still retain acceptable performance.

```
-XX:MaxHeapFreeRatio=10 -XX:MinHeapFreeRatio=5
```

Customers trying to keep the heap small should also add the option `-XX:-ShrinkHeapInSteps`. See Performance Tuning Examples for a description of using this option to keep the Java heap small by reducing the dynamic footprint for embedded applications.

`-XX:MaxMetaspaceSize=`*`size`*
Sets the maximum amount of native memory that can be allocated for class metadata. By default, the size isn't limited. The amount of metadata for an application depends on the application itself, other running applications, and the amount of memory available on the system.
The following example shows how to set the maximum class metadata size to 256 MB:

```
-XX:MaxMetaspaceSize=256m
```

`-XX:MaxNewSize=`*`size`*
Sets the maximum size (in bytes) of the heap for the young generation (nursery). The default value is set ergonomically.

`-XX:MaxTenuringThreshold=`*`threshold`*
Sets the maximum tenuring threshold for use in adaptive GC sizing. The largest value is 15. The default value is 15 for the parallel (throughput) collector, and 6 for the CMS collector.
The following example shows how to set the maximum tenuring threshold to 10:

```
-XX:MaxTenuringThreshold=10
```

`-XX:MetaspaceSize=`*`size`*
Sets the size of the allocated class metadata space that triggers a garbage collection the first time it's exceeded. This threshold for a garbage collection is increased or decreased depending on the amount of metadata used. The default size depends on the platform.

`-XX:MinHeapFreeRatio=`*`percent`*
Sets the minimum allowed percentage of free heap space (0 to 100) after a GC event. If free heap space falls below this value, then the heap is expanded. By default, this value is set to 40%.
Minimize Java heap size by lowering the values of the parameters `MaxHeapFreeRatio` (default value is 70%) and `MinHeapFreeRatio` (default value is 40%) with the command-line options `-XX:MaxHeapFreeRatio` and `-XX:MinHeapFreeRatio`. Lowering `MaxHeapFreeRatio` to as low as 10% and `MinHeapFreeRatio` to 5% has successfully reduced the heap size without too much performance regression; however, results may vary greatly depending on your application. Try different values for these parameters until they're as low as possible, yet still retain acceptable performance.

```
-XX:MaxHeapFreeRatio=10 -XX:MinHeapFreeRatio=5
```

Customers trying to keep the heap small should also add the option `-XX:-ShrinkHeapInSteps`. See Performance Tuning Examples for a description of using this option to keep the Java heap small by reducing the dynamic footprint for embedded applications.

`-XX:NewRatio=`*`ratio`*
Sets the ratio between young and old generation sizes. By default, this option is set to 2. The following example shows how to set the young-to-old ratio to 1:

```
-XX:NewRatio=1
```

`-XX:NewSize=`*`size`*
Sets the initial size (in bytes) of the heap for the young generation (nursery). Append the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes.
The young generation region of the heap is used for new objects. GC is performed in this region more often than in other regions. If the size for the young generation is too low, then a large number of minor GCs are performed. If the size is too high, then only full GCs are performed, which can take a long time to complete. Oracle recommends that you keep the size for the young generation greater than 25% and less than 50% of the overall heap size.
The following examples show how to set the initial size of the young generation to 256 MB using various units:

```
-XX:NewSize=256m
-XX:NewSize=262144k
-XX:NewSize=268435456
```

The `-XX:NewSize` option is equivalent to `-Xmn`.

`-XX:ParallelGCThreads=`*`threads`*
Sets the value of the stop-the-world (STW) worker threads. This option sets the value of *`threads`* to the number of logical processors. The value of *`threads`* is the same as the number of logical processors up to a value of 8.
If there are more than 8 logical processors, then this option sets the value of *`threads`* to approximately 5/8 of the logical processors. This works in most cases except for larger SPARC systems where the value of *`threads`* can be approximately 5/16 of the logical processors.
The default value depends on the number of CPUs available to the JVM.
For example, to set the number of threads for parallel GC to 2, specify the following option:

```
-XX:ParallelGCThreads=2
```

`-XX:+ParallelRefProcEnabled`
Enables parallel reference processing. By default, this option is disabled.

`-XX:+PrintAdaptiveSizePolicy`
Enables printing of information about adaptive-generation sizing. By default, this option is disabled.

`-XX:+ScavengeBeforeFullGC`
Enables GC of the young generation before each full GC. This option is enabled by default. Oracle recommends that you *don't* disable it, because scavenging the young generation before a full GC can reduce the number of objects reachable from the old generation space into the young generation space. To disable GC of the young generation before each full GC, specify the option `-XX:-ScavengeBeforeFullGC`.

-XX:-ShrinkHeapInSteps
Incrementally reduces the Java heap to the target size, specified by the option –XX:MaxHeapFreeRatio. This option is enabled by default. If disabled, then it immediately reduces the Java heap to the target size instead of requiring multiple garbage collection cycles. Disable this option if you want to minimize the Java heap size. You will likely encounter performance degradation when this option is disabled.
See Performance Tuning Examples for a description of using the MaxHeapFreeRatio option to keep the Java heap small by reducing the dynamic footprint for embedded applications.

-XX:StringDeduplicationAgeThreshold=*threshold*
Identifies String objects reaching the specified age that are considered candidates for deduplication. An object's age is a measure of how many times it has survived garbage collection. This is sometimes referred to as tenuring. See the -XX:+PrintTenuringDistribution option.

> **Note:**
>
> String objects that are promoted to an old heap region before this age has been reached are always considered candidates for deduplication. The default value for this option is 3. See the -XX:+UseStringDeduplication option.

-XX:SurvivorRatio=*ratio*
Sets the ratio between eden space size and survivor space size. By default, this option is set to 8. The following example shows how to set the eden/survivor space ratio to 4:

```
-XX:SurvivorRatio=4
```

-XX:TargetSurvivorRatio=*percent*
Sets the desired percentage of survivor space (0 to 100) used after young garbage collection. By default, this option is set to 50%.
The following example shows how to set the target survivor space ratio to 30%:

```
-XX:TargetSurvivorRatio=30
```

-XX:TLABSize=*size*
Sets the initial size (in bytes) of a thread-local allocation buffer (TLAB). Append the letter k or K to indicate kilobytes, m or M to indicate megabytes, or g or G to indicate gigabytes. If this option is set to 0, then the JVM selects the initial size automatically.
The following example shows how to set the initial TLAB size to 512 KB:

```
-XX:TLABSize=512k
```

-XX:+UseAdaptiveSizePolicy
Enables the use of adaptive generation sizing. This option is enabled by default. To disable adaptive generation sizing, specify -XX:-UseAdaptiveSizePolicy and set the size of the memory allocation pool explicitly. See the -XX:SurvivorRatio option.

-XX:+UseCMSInitiatingOccupancyOnly
Enables the use of the occupancy value as the only criterion for initiating the CMS collector. By default, this option is disabled and other criteria may be used.

-XX:+UseG1GC
Enables the use of the garbage-first (G1) garbage collector. It's a server-style garbage collector, targeted for multiprocessor machines with a large amount of RAM. This option meets GC pause time goals with high probability, while maintaining good throughput. The G1 collector is recommended for applications requiring large heaps (sizes of around 6 GB or larger) with limited GC latency requirements (a stable and predictable pause time below 0.5 seconds). By default, this option is enabled and G1 is used as the default garbage collector.

-XX:+UseGCOverheadLimit
Enables the use of a policy that limits the proportion of time spent by the JVM on GC before an `OutOfMemoryError` exception is thrown. This option is enabled, by default, and the parallel GC will throw an `OutOfMemoryError` if more than 98% of the total time is spent on garbage collection and less than 2% of the heap is recovered. When the heap is small, this feature can be used to prevent applications from running for long periods of time with little or no progress. To disable this option, specify the option `-XX:-UseGCOverheadLimit`.

-XX:+UseNUMA
Enables performance optimization of an application on a machine with nonuniform memory architecture (NUMA) by increasing the application's use of lower latency memory. By default, this option is disabled and no optimization for NUMA is made. The option is available only when the parallel garbage collector is used (`-XX:+UseParallelGC`).

-XX:+UseParallelGC
Enables the use of the parallel scavenge garbage collector (also known as the throughput collector) to improve the performance of your application by leveraging multiple processors.
By default, this option is disabled and the collector is chosen automatically based on the configuration of the machine and type of the JVM. If it's enabled, then the `-XX:+UseParallelOldGC` option is automatically enabled, unless you explicitly disable it.

-XX:+UseParallelOldGC
Enables the use of the parallel garbage collector for full GCs. By default, this option is disabled. Enabling it automatically enables the `-XX:+UseParallelGC` option.

-XX:+UseSerialGC
Enables the use of the serial garbage collector. This is generally the best choice for small and simple applications that don't require any special functionality from garbage collection. By default, this option is disabled and the collector is selected automatically based on the configuration of the machine and type of the JVM.

-XX:+UseSHM
Linux only: Enables the JVM to use shared memory to set up large pages.
See Large Pages for setting up large pages.

-XX:+UseStringDeduplication
Enables string deduplication. By default, this option is disabled. To use this option, you must enable the garbage-first (G1) garbage collector.
String deduplication reduces the memory footprint of `String` objects on the Java heap by taking advantage of the fact that many `String` objects are identical. Instead of each `String` object pointing to its own character array, identical `String` objects can point to and share the same character array.

-XX:+UseTLAB
Enables the use of thread-local allocation blocks (TLABs) in the young generation space. This option is enabled by default. To disable the use of TLABs, specify the option `-XX:-UseTLAB`.

Obsolete Java Options

These `java` options are still accepted but ignored, and a warning is issued when they're used.

-Xusealtsigs / -XX:+UseAltSigs
Oracle Solaris only: Use alternative signals instead of SIGUSR1 and SIGUSR2 for JVM internal signals. Since Solaris 10, two dedicated signals have been made available to the VM and so, since JDK 6, these flags have been documented as having no effect. The flags have now been made obsolete, and their use generates a warning. In a future release these flags will be removed completely.

Deprecated Java Options

These `java` options are deprecated and might be removed in a future JDK release. They're still accepted and acted upon, but a warning is issued when they're used.

-d32
This option is deprecated and will be removed in a future release.

-d64
This option is deprecated and will be removed in a future release.
Oracle Solaris, Linux, and OS X: Runs the application in a 64-bit environment. If a 64-bit environment isn't installed or isn't supported, then an error is reported.
Only the Java HotSpot Server VM supports 64-bit operation and the `-server` option is implicit with the use of `-d64`. The `-client` option is ignored with the use of `-d64`.

-Xloggc:garbage-collection.log
Sets the file to which verbose GC events information should be redirected for logging. The information written to this file is similar to the output of `-verbose:gc` with the time elapsed since the first GC event preceding each logged event. The `-Xloggc` option overrides `-verbose:gc` if both are given with the same java command.
Example:

```
-Xlog:gc:garbage-collection.log
```

-XX:CMSInitiatingOccupancyFraction=*percent*
Sets the percentage of the old generation occupancy (0 to 100) at which to start a CMS collection cycle. The default value is set to -1. Any negative value (including the default) implies that the option `-XX:CMSTriggerRatio` is used to define the value of the initiating occupancy fraction.
The following example shows how to set the occupancy fraction to 20%:

```
-XX:CMSInitiatingOccupancyFraction=20
```

-XX:CMSInitiatingPermOccupancyFraction=*percent*
Sets the percentage of the permanent generation occupancy (0 to 100) at which to start a GC. This option was deprecated in JDK 8 with no replacement.

-XX:+G1PrintHeapRegions
Enables the printing of information about which regions are allocated and which are reclaimed by the G1 collector. By default, this option is disabled. See Enable Logging with the JVM Unified Logging Framework.

-XX:MaxPermSize=*size*
Sets the maximum permanent generation space size (in bytes). This option was deprecated in JDK 8 and superseded by the `-XX:MaxMetaspaceSize` option.

-XX:PermSize=*size*
Sets the space (in bytes) allocated to the permanent generation that triggers a garbage collection if it's exceeded. This option was deprecated in JDK 8 and superseded by the `-XX:MetaspaceSize` option.

-XX:+PrintGC
Enables printing of messages at every GC. By default, this option is disabled. If you're using this flag, then see Enable Logging with the JVM Unified Logging Framework. In JDK 9, this option is deprecated.

-XX:+PrintGCApplicationConcurrentTime
Enables printing of how much time elapsed since the last pause (for example, a GC pause). By default, this option is deprecated.

-XX:+PrintGCApplicationStoppedTime
Enables printing of how much time the pause (for example, a GC pause) lasted. By default, this option is deprecated

-XX:+PrintGCDateStamps
Enables printing of a date stamp at every GC. By default, this option is deprecated.

-XX:+PrintGCDetails
Enables printing of detailed messages at every GC. By default, this option is disabled. See Enable Logging with the JVM Unified Logging Framework.

-XX:+PrintGCTaskTimeStamps
Enables printing of time stamps for every individual GC worker thread task. By default, this option is disabled. See Enable Logging with the JVM Unified Logging Framework.

-XX:+PrintGCTimeStamps
Enables printing of time stamps at every GC. By default, this option is disabled. See Enable Logging with the JVM Unified Logging Framework.

-XX:+PrintStringDeduplicationStatistics
Prints detailed deduplication statistics. By default, this option is disabled. See the `-XX:+UseStringDeduplication` option.

-XX:+PrintTenuringDistribution
Enables printing of tenuring age information. The following is an example of the output:

```
Desired survivor size 48286924 bytes, new threshold 10 (max 10)
- age 1: 28992024 bytes, 28992024 total
- age 2: 1366864 bytes, 30358888 total
- age 3: 1425912 bytes, 31784800 total
...
```

Age 1 objects are the youngest survivors (they were created after the previous scavenge, survived the latest scavenge, and moved from eden to survivor space). Age 2 objects have survived two scavenges (during the second scavenge they were copied from one survivor space to the next). This pattern is repeated for all objects in the output.
In the preceding example, 28,992,024bytes survived one scavenge and were copied from eden to survivor space, 1,366,864 bytes are occupied by age 2 objects, and so on. The third value in each row is the cumulative size of objects of age *n* or less.
By default, this option is disabled.

`-XX:SoftRefLRUPolicyMSPerMB=`*`time`*
Sets the amount of time (in milliseconds) a softly reachable object is kept active on the heap after the last time it was referenced. The default value is one second of lifetime per free megabyte in the heap. The `-XX:SoftRefLRUPolicyMSPerMB` option accepts integer values representing milliseconds per one megabyte of the current heap size (for Java HotSpot Client VM) or the maximum possible heap size (for Java HotSpot Server VM). This difference means that the Client VM tends to flush soft references rather than grow the heap, whereas the Server VM tends to grow the heap rather than flush soft references. In the latter case, the value of the `-Xmx` option has a significant effect on how quickly soft references are garbage collected.
The following example shows how to set the value to 2.5 seconds:

```
-XX:SoftRefLRUPolicyMSPerMB=2500
```

`-XX:+TraceClassLoading`
Enables tracing of classes as they are loaded. By default, this option is disabled and classes aren't traced.
The replacement Unified Logging syntax is `-Xlog:class+load=`*`level`*. See Enable Logging with the JVM Unified Logging Framework
Use *`level`*`=info` for regular information, or *`level`*`=debug` for additional information. In Unified Logging syntax, `-verbose:class` equals `-Xlog:class+load=info,class +unload=info`..

`-XX:+TraceClassLoadingPreorder`
Enables tracing of all loaded classes in the order in which they're referenced. By default, this option is disabled and classes aren't traced.
The replacement Unified Logging syntax is `-Xlog:class+preorder=debug`. See Enable Logging with the JVM Unified Logging Framework.

`-XX:+TraceClassResolution`
Enables tracing of constant pool resolutions. By default, this option is disabled and constant pool resolutions aren't traced.
The replacement Unified Logging syntax is `-Xlog:class+resolve=debug`. See Enable Logging with the JVM Unified Logging Framework.

`-XX:+TraceClassUnloading`
Enables tracing of classes as they're unloaded. By default, this option is disabled and classes aren't traced.
The replacement Unified Logging syntax is `-Xlog:class+unload=`*`level`*. See Enable Logging with the JVM Unified Logging Framework.
Use *`level`*`=info` for regular information, and *`level`*`=trace` for additional information. In Unified Logging syntax, `-verbose:class` equals `-Xlog:class+unload=info,class +unload=info` .

`-XX:+TraceLoaderConstraints`
Enables tracing of the loader constraints recording. By default, this option is disabled and loader constraints recording isn't traced.
The replacement Unified Logging syntax is `-Xlog:class+loader+constraints=info`. See Enable Logging with the JVM Unified Logging Framework.

`-XX:+UseConcMarkSweepGC`
Enables the use of the CMS garbage collector for the old generation. CMS is an alternative to the default garbage collector (G1), which also focuses on meeting application latency requirements. By default, this option is disabled and the collector is selected automatically based on the configuration of the machine and type of the JVM. In JDK 9, the CMS garbage collector is deprecated.

`-XX:+UseParNewGC`
Enables the use of parallel threads for collection in the young generation. By default, this option is disabled. It's automatically enabled when you set the `-XX:+UseConcMarkSweepGC` option. Using the `-XX:+UseParNewGC` option without the `-XX:+UseConcMarkSweepGC` option was deprecated in JDK 8. Starting with JDK 9, all uses of the `-XX:+UseParNewGC` option are deprecated. Using the option without `-XX:+UseConcMarkSweepGC` isn't possible.

`-XX:+UseSplitVerifier`
Enables splitting the verification process. By default, this option was enabled in the previous releases, and verification was split into two phases: type referencing (performed by the compiler) and type checking (performed by the JVM runtime). Verification is now split by default without a way to disable it.

Removed Java Options

These `java` options were removed in JDK 9 and using them results in an error of:

```
Unrecognized VM option option-name
```

`-d32`
Oracle Solaris, Linux, and OS X: Ran the application in a 32-bit environment. 32-bit JDKs/JREs are no longer supported.

> **Note:**
>
> The `-d32` and `-d64` options were added to allow multiple architectures (data model) JDKs and JREs to coexist on the same system. The user could invoke the other data model by using these launcher options. Oracle Solaris was the only platform supporting these options, and the 32-bit JDKs/JREs are no longer supported.

`-Xincgc`
Enabled incremental garbage collection. This option and the GC mode are removed in JDK 9.

`-Xmaxjitcodesize=`*`size`*
Specified the maximum code cache size (in bytes) for JIT-compiled code. Appended the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. The default maximum code cache size is 240 MB; if you disable tiered compilation with the option `-XX:-TieredCompilation`, then the default size is 48 MB:

```
-Xmaxjitcodesize=240m
```

This option is equivalent to `-XX:ReservedCodeCacheSize`.

`-Xrun`*`libname`*
Loaded the specified debugging/profiling library. This option was superseded by the `-agentlib` option.

`-XX:CMSIncrementalDutyCycle=`*`percent`*
Set the percentage of time (0 to 100) between minor collections that the concurrent collector was allowed to run.

`-XX:CMSIncrementalDutyCycleMin=`*`percent`*
Sets the percentage of time (0 to 100) between minor collections that was the lower bound for the duty cycle when `-XX:+CMSIncrementalPacing` option was enabled. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option. The option was removed in JDK 9, because the entire incremental mode was removed.

`-XX:+CMSIncrementalMode`
Enabled incremental mode. Note that the CMS collector must also be enabled (with `-XX:+UseConcMarkSweepGC)` for this option to work. The option was removed in JDK 9, because the entire incremental mode was removed.

`-XX:CMSIncrementalOffset=`*`percent`*
Set the percentage of time (0 to 100) by which the incremental mode duty cycle was shifted to the right within the period between minor collections.

`-XX:+CMSIncrementalPacing`
Enabled automatic adjustment of the incremental mode duty cycle based on statistics collected while the JVM was running. This option was deprecated with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option. The option was removed, because the entire incremental mode was removed.

`-XX:CMSIncrementalSafetyFactor=`*`percent`*
Set the percentage of time (0 to 100) used to add conservatism when computing the duty cycle. This option was deprecated in JDK 8 with no replacement, following the deprecation of the `-XX:+CMSIncrementalMode` option. The option was removed, because the entire incremental mode was removed.

`-XX:CodeCacheMinimumFreeSpace=`*`size`*
Set the minimum free space (in bytes) required for compilation. Appended the letter `k` or `K` to indicate kilobytes, `m` or `M` to indicate megabytes, or `g` or `G` to indicate gigabytes. When less than the minimum free space remained, compiling stopped. By default, this option was set to 500 KB.

java Command-Line Argument Files

You can shorten or simplify the `java` command by using *`@argument files`* to specify a text file that contains arguments, such as options and class names, passed to the `java` command. This let's you to create `java` commands of any length on any operating system.

In the command line, use the at sign (`@`) prefix to identify an argument file that contains `java` options and class names. When the `java` command encounters a file beginning

with the at sign (@) , it expands the contents of that file into an argument list just as they would be specified on the command line.

The `java` launcher expands the argument file contents until it encounters the `-Xdisable-@files` option. You can use the `-Xdisable-@files` option anywhere on the command line, including in an argument file, to stop `@{argument files}` expansion.

The following items describe the syntax of `java` argument files:

- The argument file must contain only ASCII characters or characters in system default encoding that's ASCII friendly, such as UTF-8.
- The argument file size must not exceed MAXINT (2,147,483,647) bytes.
- The launcher doesn't expand wildcards that are present within an argument file.
- Use white space or new line characters to separate arguments included in the file.
- White space includes a white space character, `\t`, `\n`, `\r`, and `\f`.

  For example, it is possible to have a path with a space, such as `c:\Program Files` that can be specified as either `"c:\\Program Files"` or, to avoid an escape, `c:\Program" "Files`.
- Any option that contains spaces, such as a path component, must be within quotation marks using quotation ("") characters in its entirety.
- A string within quotation marks may contain the characters `\n`, `\r`, `\t`, and `\f`. They are converted to their respective ASCII codes.
- If a file name contains embedded spaces, then put the whole file name in double quotation marks.
- File names in an argument file are relative to the current directory, not to the location of the argument file.
- Use the number sign `#` in the argument file to identify comments. All characters following the`#` are ignored until the end of line.
- Additional at sign `@` prefixes to `@` prefixed options act as an escape, (the first `@` is removed and the rest of the arguments are presented to the launcher literally).
- Lines may be continued using the continuation character (\) at the end-of-line. The two lines are concatenated with the leading white spaces trimmed. To prevent trimming the leading white spaces, a continuation character (\) may be placed at the first column.
- Because backslash (\) is an escape character, a backslash character must be escaped with another backslash character.
- Partial quote is allowed and is closed by an end-of-file.
- An open quote stops at end-of-line unless `\` is the last character, which then joins the next line by removing all leading white space characters.
- Wildcards (*) aren't allowed in these lists (such as specifying `*.java`).
- Use of the at sign (@) to recursively interpret files isn't supported.

Example of Open or Partial Quotes in an Argument File

In the argument file,

```
-cp "lib/
cool/
```

```
app/
jars
```

this is interpreted as:

```
-cp lib/cool/app/jars
```

Example of a Backslash Character Escaped with Another Backslash Character in an Argument File

To output the following:

```
-cp c:\Program Files (x86)\Java\jre\lib\ext;c:\Program Files\Java\jre9\lib\ext
```

The backslash character must be specified in the argument file as:

```
-cp  "c:\\Program Files (x86)\\Java\\jre\\lib\\ext;c:\\Program Files\\Java\\jre9\
\lib\\ext"
```

Example of an EOL Escape Used to Force Concatenation of Lines in an Argument File

In the argument file,

```
-cp "/lib/cool app/jars:\
    /lib/another app/jars"
```

This is interpreted as:

```
-cp /lib/cool app/jars:/lib/another app/jars
```

Example of Line Continuation with Leading Spaces in an Argument File

In the argument file,

```
-cp "/lib/cool\
\app/jars”
```

This is interpreted as:

```
-cp /lib/cool app/jars
```

Examples of Using Single Argument File

You can use a single argument file, such as `myargumentfile` in the following example, to hold all required `java` arguments:

```
java @myargumentfile
```

Examples of Using Argument Files with Paths

You can include relative paths in argument files; however, they’re relative to the current working directory and not to the paths of the argument files themselves. In the following example, `path1/options` and `path2/options` represent argument files with different paths. Any relative paths that they contain are relative to the current working directory and not to the argument files:

```
java @path1/options @path2/classes
```

Enable Logging with the JVM Unified Logging Framework

You use the `-Xlog` option to configure or enable logging with the Java Virtual Machine (JVM) unified logging framework.

Synopsis

```
-Xlog[:[what][:[output][:[decorators][:output-options [,...]]]]]
```

what
Specifies a combination of tags and levels of the form *tag1*[+*tag2*...][*][=*level*][,...]. Unless the wildcard (*) is specified, only log messages tagged with exactly the tags specified are matched. See -Xlog Tags and Levels.

output
Sets the type of output. Omitting the *output* type defaults to `stdout`. See -Xlog Output.

decorators
Configures the output to use a custom set of decorators. Omitting *decorators* defaults to `uptime`, `level`, and `tags`. See Decorations.

output-options
Sets the `-Xlog` logging output options.

Description

The Java Virtual Machine (JVM) unified logging framework provides a common logging system for all components of the JVM. GC logging for the JVM has been changed to use the new logging framework. The mapping of old GC flags to the corresponding new Xlog configuration is described in Convert GC Logging Flags to Xlog. In addition, runtime logging has also been changed to use the JVM unified logging framework. The mapping of legacy runtime logging flags to the corresponding new Xlog configuration is described in Convert Runtime Logging Flags to Xlog.

The following provides quick reference to the `-Xlog` command and syntax for options:

`-Xlog`
Enables JVM logging on an `info` level.

`-Xlog:help`
Prints `-Xlog` usage syntax and available tags, levels, and decorators along with example command lines with explanations.

`-Xlog:disable`
Turns off all logging and clears all configuration of the logging framework including the default configuration for warnings and errors.

`-Xlog[:`*`option`*`]`
Applies multiple arguments in the order that they appear on the command line. Multiple `-Xlog` arguments for the same output override each other in their given order. The *option* is set as:

```
[tag selection][:[output][:[decorators][:output-options]]]
```

Omitting the *tag selection* defaults to a tag-set of `all` and a level of `info`.

tag[+...] all

The `all` tag is a meta tag consisting of all tag-sets available. The asterisk `*` in a tag set definition denotes a wildcard tag match. Matching with a wildcard selects all tag sets that contain *at least* the specified tags. Without the wildcard, only exact matches of the specified tag sets are selected.
output_options is

filecount=*file count* filesize=*file size with optional K, M or G suffix*

Default Configuration

When the `-Xlog`option and nothing else is specified on the command line, the default configuration is used. The default configuration logs all messages with a level that matches either the warning or error regardless of what tags the message is associated with. The default configuration is equivalent to entering the following on the command line:

```
-Xlog:all=warning:stdout:uptime,level,tags
```

Controlling Logging at Runtime

Logging can also be controlled at run time through Diagnostic Commands (with the `jcmd` utility). Everything that can be specified on the command line can also be specified dynamically with the `VM.log` command. As the diagnostic commands are automatically exposed as MBeans, you can use JMX to change logging configuration at run time.

-Xlog Tags and Levels

Each log message has a level and a tag set associated with it. The level of the message corresponds to its details, and the tag set corresponds to what the message contains or which JVM component it involves (such as, GC, compiler, or threads). Mapping GC flags to the Xlog configuration is described in Convert GC Logging Flags to Xlog. Mapping legacy runtime logging flags to the corresponding Xlog configuration is described in Convert Runtime Logging Flags to Xlog.

Available log levels:

- `off`
- `trace`
- `debug`
- `info`
- `warning`
- `error`

Available log tags:
The following are the available log tags. Specifying `all` instead of a tag combination matches all tag combinations.

- `add`
- `age`
- `alloc`

- annotation
- aot
- arguments
- attach
- barrier
- biasedlocking
- blocks
- bot
- breakpoint
- bytecode
- census
- class
- classhisto
- cleanup
- compaction
- comparator
- constraints
- constantpool
- coops
- cpu
- cset
- data
- defaultmethods
- dump
- ergo
- event
- exceptions
- exit
- fingerprint
- freelist
- gc
- hashtables
- heap
- humongous
- ihop

- iklass
- init
- itables
- jfr
- jni
- jvmti
- liveness
- load
- loader
- logging
- mark
- marking
- metadata
- metaspace
- method
- mmu
- modules
- monitorinflation
- monitormismatch
- nmethod
- normalize
- objecttagging
- obsolete
- oopmap
- os
- pagesize
- parser
- patch
- path
- phases
- plab
- preorder
- promotion
- protectiondomain
- purge

- redefine
- ref
- refine
- region
- remset
- resolve
- safepoint
- scavenge
- scrub
- setting
- stackmap
- stacktrace
- stackwalk
- start
- startuptime
- state
- stats
- stringdedup
- stringtable
- subclass
- survivor
- sweep
- system
- task
- thread
- time
- timer
- tlab
- unload
- update
- verification
- verify
- vmoperation
- vtables
- workgang

-Xlog Output

The `-Xlog` option supports the following types of outputs:

- `stdout` — Sends output to stdout
- `stderr` — Sends output to stderr
- `file=`*`filename`* — Sends output to text file(s).

When using `file=`*`filename`*, specifying `%p` and/or `%t` in the file name expands to the JVM's PID and startup timestamp, respectively. You can also configure text files to handle file rotation based on file size and a number of files to rotate. For example, to rotate the log file every 10 MB and keep 5 files in rotation, specify the options `filesize=10M, filecount=5`. The target size of the files isn't guaranteed to be exact, it's just an approximate value. Files are rotated by default with up to 5 rotated files of target size 20 MB, unless configured otherwise. Specifying `filecount=0` means that the log file shouldn't be rotated. There's a possibility of the pre-existing log file getting overwritten.

Decorations

Logging messages are decorated with information about the message. You can configure each output to use a custom set of decorators. The order of the output is always the same as listed in the table. You can configure the decorations to be used at run time. Decorations are prepended to the log message. For example:

```
[6.567s][info][gc,old] Old collection complete
```

Omitting *`decorators`* defaults to `uptime`, `level`, and `tags`. The `none` decorator is special and is used to turn off all decorations.

`time` (`t`), `utctime` (`utc`), `uptime` (`u`), `timemillis` (`tm`), `uptimemillis` (`um`), `timenanos` (`tn`), `uptimenanos` (`un`), `hostname` (`hn`), `pid` (`p`), `tid` (`ti`), `level` (`l`), `tags` (`tg`) decorators can also be specified as `none` for no decoration.

Decorations	Description
`time` or `t`	Current time and date in ISO-8601 format.
`utctime` or `utc`	Universal Time Coordinated or Coordinated Universal Time.
`uptime` or `u`	Time since the start of the JVM in seconds and milliseconds. For example, 6.567s.
`timemillis` or `tm`	The same value as generated by `System.currentTimeMillis()`.
`uptimemillis` or `um`	Milliseconds since the JVM started.
`timenanos` or `tn`	The same value generated by `System.nanoTime()`.
`uptimenanos` or `un`	Nanoseconds since the JVM started.
`hostname` or `hn`	The host name.
`pid` or `p`	The process identifier.
`tid` or `ti`	The thread identifier.
`level` or `l`	The level associated with the log message.
`tags` or `tg`	The tag-set associated with the log message.

Convert GC Logging Flags to Xlog

Table 2-1 Mapping Legacy Garbage Collection Logging Flags to the Xlog Configuration

Legacy Garbage Collection (GC) Flag	Xlog Configuration	Comment
`G1PrintHeapRegions`	`-Xlog:gc+region=trace`	Not Applicable
`GCLogFileSize`	No configuration available	Log rotation is handled by the framework.
`NumberOfGCLogFiles`	Not Applicable	Log rotation is handled by the framework.
`PrintAdaptiveSizePolicy`	`-Xlog:ergo*=`*`level`	Use a *`level`* of `debug` for most of the information, or a *`level`* of `trace` for all of what was logged for `PrintAdaptiveSizePolicy`.
`PrintGC`	`-Xlog:gc`	Not Applicable
`PrintGCApplicationConcurrentTime`	`-Xlog:safepoint`	Note that `PrintGCApplicationConcurrentTime` and `PrintGCApplicationStoppedTime` are logged on the same tag and aren't separated in the new logging.
`PrintGCApplicationStoppedTime`	`-Xlog:safepoint`	Note that `PrintGCApplicationConcurrentTime` and `PrintGCApplicationStoppedTime` are logged on the same tag and not separated in the new logging.
`PrintGCCause`	Not Applicable	GC cause is now always logged.
`PrintGCDateStamps`	Not Applicable	Date stamps are logged by the framework.
`PrintGCDetails`	`-Xlog:gc*`	Not Applicable
`PrintGCID`	Not Applicable	GC ID is now always logged.
`PrintGCTaskTimeStamps`	`-Xlog:task*=debug`	Not Applicable
`PrintGCTimeStamps`	Not Applicable	Time stamps are logged by the framework.
`PrintHeapAtGC`	`-Xlog:gc+heap=trace`	Not Applicable
`PrintReferenceGC`	`-Xlog:ref*=debug`	Note that in the old logging, `PrintReferenceGC` had an effect only if `PrintGCDetails` was also enabled.
`PrintStringDeduplicationStatistics`	`-Xlog:stringdedup*=debug`	Not Applicable
`PrintTenuringDistribution`	`-Xlog:age*=`*`level`	Use a *`level`* of `debug` for the most relevant information, or a *`level`* of `trace` for all of what was logged for `PrintTenuringDistribution`.
`UseGCLogFileRotation`	Not Applicable	What was logged for `PrintTenuringDistribution`.

Convert Runtime Logging Flags to Xlog

Table 2-2 Mapping Runtime Logging Flags to the Xlog Configuration

Legacy Runtime Flag	Xlog Configuration	Comment
`TraceExceptions`	`-Xlog:exceptions=info`	Not Applicable
`TraceClassLoading`	`-Xlog:class+load=`*`level`*	Use *`level`*`=info` for regular information, or *`level`*`=debug` for additional information. In Unified Logging syntax, `-verbose:class` equals `-Xlog:class +load=info,class+unload=info`.
`TraceClassLoadingPreorder`	`-Xlog:class+preorder=debug`	Not Applicable
`TraceClassUnloading`	`-Xlog:class+unload=`*`level`*	Use *`level`*`=info` for regular information, or *`level`*`=trace` for additional information. In Unified Logging syntax, `-verbose:class` equals `-Xlog:class +load=info,class+unload=info`.
`VerboseVerification`	`-Xlog:verification=info`	Not Applicable
`TraceClassPaths`	`-Xlog:class+path=info`	Not Applicable
`TraceClassResolution`	`-Xlog:class+resolve=debug`	Not Applicable
`TraceClassInitialization`	`-Xlog:class+init=info`	Not Applicable
`TraceLoaderConstraints`	`-Xlog:class+loader +constraints=info`	Not Applicable
`TraceClassLoaderData`	`-Xlog:class+loader+data=`*`level`*	Use `level=debug` for regular information or `level=trace` for additional information.
`TraceSafepointCleanupTime`	`-Xlog:safepoint+cleanup=info`	Not Applicable
`TraceSafepoint`	`-Xlog:safepoint=debug`	Not Applicable
`TraceMonitorInflation`	`-Xlog:monitorinflation=debug`	Not Applicable
`TraceBiasedLocking`	`-Xlog:biasedlocking=`*`level`*	Use `level=info` for regular information, or `level=trace` for additional information.
`TraceRedefineClasses`	`-Xlog:redefine+class*=`*`level`*	`level=info`, `=debug`, and `=trace` provide increasing amounts of information.

-Xlog Usage Examples

The following are `-Xlog` examples.

`-Xlog`
Logs all messages by using the `info`level to `stdout` with `uptime`, `levels`, and `tags` decorations. This is equivalent to using:

```
-Xlog:all=info:stdout:uptime,levels,tags
```

-Xlog:gc
Logs messages tagged with the `gc` tag using `info` level to `stdout`. The default configuration for all other messages at level `warning` is in effect.

-Xlog:gc,safepoint
Logs messages tagged either with the `gc` or `safepoint` tags, both using the `info` level, to `stdout`, with default decorations. Messages tagged with both `gc` and `safepoint`won't be logged.

-Xlog:gc+ref=debug
Logs messages tagged with both `gc` and `ref` tags, using the `debug` level to `stdout`, with default decorations. Messages tagged only with one of the two tags won't be logged.

-Xlog:gc=debug:file=gc.txt:none
Logs messages tagged with the `gc` tag using the `debug` level to a file called `gc.txt` with no decorations. The default configuration for all other messages at level `warning` is still in effect.

-Xlog:gc=trace:file=gctrace.txt:uptimemillis,pids:filecount=5,filesize=1024
Logs messages tagged with the `gc` tag using the `trace` level to a rotating file set with 5 files with size 1 MB with the base name `gctrace.txt` and uses decorations `uptimemillis` and `pid`.
The default configuration for all other messages at level`warning` is still in effect.

-Xlog:gc::uptime,tid
Logs messages tagged with the `gc` tag using the default 'info' level to default the output`stdout` and uses decorations `uptime` and `tid`. The default configuration for all other messages at level`warning` is still in effect.

-Xlog:gc*=info,safepoint*=off
Logs messages tagged with at least `gc` using the `info` level, but turns off logging of messages tagged with `safepoint`. Messages tagged with both `gc` and `safepoint` won't be logged.

-Xlog:disable -Xlog:safepoint=trace:safepointtrace.txt
Turns off all logging, including warnings and errors, and then enables messages tagged with `safepoint`using `trace`level to the file `safepointtrace.txt`. The default configuration doesn't apply, because the command line started with `-Xlog:disable`.

Complex -Xlog Usage Examples

The following describes a few complex examples of using the `-Xlog` option.

-Xlog:gc+class*=debug
Logs messages tagged with at least `gc` and `class` tags using the `debug` level to `stdout`. The default configuration for all other messages at the level `warning` is still in effect

-Xlog:gc+meta*=trace,class*=off:file=gcmetatrace.txt
Logs messages tagged with at least the `gc` and `meta` tags using the`trace` level to the file `metatrace.txt` but turns off all messages tagged with `class`. Messages tagged with `gc`, `meta`, and`class` aren't be logged as`class*` is set to off. The default configuration for all other messages at level `warning` is in effect except for those that include `class`.

`-Xlog:gc+meta=trace`
Logs messages tagged with exactly the `gc` and `meta` tags using the `trace` level to `stdout`. The default configuration for all other messages at level `warning` is still be in effect.

`-Xlog:gc+class+heap*=debug,meta*=warning,threads*=off`
Logs messages tagged with at least `gc`, `class`, and `heap` tags using the `trace` level to `stdout` but only log messages tagged with `meta` with level. The default configuration for all other messages at the level `warning` is in effect except for those that include `threads`.

Validate Java Virtual Machine Flag Arguments

You use values provided to all Java Virtual Machine (JVM) command-line flags for validation and, if the input value is invalid or out-of-range, then an appropriate error message is displayed.

Whether they're set ergonomically, in a command line, by an input tool, or through the APIs (for example, classes contained in the package `java.lang.management`) the values provided to all Java Virtual Machine (JVM) command-line flags are validated. Ergonomics are described in *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide*.
Range and constraints are validated either when all flags have their values set during JVM initialization or a flag's value is changed during runtime (for example using the `jcmd` tool). The JVM is terminated if a value violates either the range or constraint check and an appropriate error message is printed on the error stream.

For example, if a flag violates a range or a constraint check, then the JVM exits with an error:

```
java -XX:AllocatePrefetchStyle=5 -version
intx AllocatePrefetchStyle=5 is outside the allowed range [ 0 ... 3 ]
Improperly specified VM option 'AllocatePrefetchStyle=5'
Error: Could not create the Java Virtual Machine.
Error: A fatal exception has occurred. Program will exit.
```

The flag `-XX:+PrintFlagsRanges` prints the range of all the flags. This flag allows automatic testing of the flags by the values provided by the ranges. For the flags that have the ranges specified, the type, name, and the actual range is printed in the output.

For example,

```
intx   ThreadStackSize [ 0 ... 9007199254740987 ] {pd product}
```

For the flags that don't have the range specified, the values aren't displayed in the print out. For example,:

```
size_t NewSize         [   ...                  ] {product}
```

This helps to identify the flags that need to be implemented. The automatic testing framework can skip those flags that don't have values and aren't implemented.

Large Pages

You use large pages, also known as huge pages, as memory pages that are significantly larger than the standard memory page size (which varies depending on

the processor and operating system). Large pages optimize processor Translation-Lookaside Buffers.

A Translation-Lookaside Buffer (TLB) is a page translation cache that holds the most-recently used virtual-to-physical address translations. A TLB is a scarce system resource. A TLB miss can be costly because the processor must then read from the hierarchical page table, which may require multiple memory accesses. By using a larger memory page size, a single TLB entry can represent a larger memory range. This results in less pressure on a TLB, and memory-intensive applications may have better performance.

However, large pages page memory can negatively affect system performance. For example, when a large mount of memory is pinned by an application, it may create a shortage of regular memory and cause excessive paging in other applications and slow down the entire system. Also, a system that has been up for a long time could produce excessive fragmentation, which could make it impossible to reserve enough large page memory. When this happens, either the OS or JVM reverts to using regular pages.

Large Pages Support

Oracle Solaris, Linux, and Windows Server 2003 support large pages.

Large Pages Support for Oracle Solaris

Oracle Solaris 9 and later include Multiple Page Size Support (MPSS). No additional configuration is necessary. See Features and Benefits - Scalability.

Large Pages Support for Linux

The 2.6 kernel supports large pages. Some vendors have backported the code to their 2.4-based releases. To check if your system can support large page memory, try the following:

```
# cat /proc/meminfo | grep Huge
HugePages_Total: 0
HugePages_Free: 0
Hugepagesize: 2048 kB
```

If the output shows the three "Huge" variables, then your system can support large page memory but it needs to be configured. If the command prints nothing, then your system doesn't support large pages. To configure the system to use large page memory, login as `root`, and then follow these steps:

1. If you're using the option `-XX:+UseSHM` (instead of `-XX:+UseHugeTLBFS`), then increase the `SHMMAX` value. It must be larger than the Java heap size. On a system with 4 GB of physical RAM (or less), the following makes all the memory sharable:

   ```
   # echo 4294967295 > /proc/sys/kernel/shmmax
   ```

2. If you're using the option `-XX:+UseSHM` or `-XX:+UseHugeTLBFS`, then specify the number of large pages. In the following example, 3 GB of a 4 GB system are reserved for large pages (assuming a large page size of 2048kB, then 3 GB = 3 * 1024 MB = 3072 MB = 3072 * 1024 kB = 3145728 kB and 3145728 kB / 2048 kB = 1536):

   ```
   # echo 1536 > /proc/sys/vm/nr_hugepages
   ```

> **Note:**
>
> - Note that the values contained in `/proc` resets after you reboot your system, so may want to set them in an initialization script (for example, `rc.local` or `sysctl.conf`).
> - If you configure (or resize) the OS kernel parameters `/proc/sys/kernel/shmmax` or `/proc/sys/vm/nr_hugepages`, Java processes may allocate large pages for areas in addition to the Java heap. These steps can allocate large pages for the following areas:
>   - Java heap
>   - Code cache
>   - The marking bitmap data structure for the parallel GC
>
>   Consequently, if you configure the `nr_hugepages` parameter to the size of the Java heap, then the JVM can fail in allocating the code cache areas on large pages because these areas are quite large in size.

Large Pages Support for Windows Server 2003

Only Windows Server 2003 supports large pages. To use this feature, the administrator must first assign additional privileges to the user who's running the application:

1. Select **Control Panel**, **Administrative Tools**, and then **Local Security Policy**.
2. Select **Local Policies** and then **User Rights Assignment**.
3. Double-click **Lock pages in memory**, then add users and/or groups.
4. Reboot your system.

Note that these steps are required even if it's the administrator who's running the application, because administrators by default don't have the privilege to lock pages in memory.

Application Class Data Sharing

Application Class Data Sharing (AppCDS) extends class data sharing to enable application classes to be placed in the shared archive.

In addition to the core library classes, AppCDS supports Class Data Sharing from the following locations:

- Platform classes from the module image
- Application classes from the module image
- Application classes from `-cp` path

> **Note:**
>
> In JDK 9, application classes from module path is not supported by AppCDS.

When running multiple JVM processes, AppCDS reduces the runtime footprint with memory sharing for read-only metadata.

This is a commercial feature that requires you to specify `-XX:+UnlockCommercialFeatures`.

Creating a Shared Archive File and Using It to Run an Application

The following steps create a shared archive file that contains all the classes used by the `test.Hello` application. The last step runs the application with the shared archive file.

1. Create a list of all classes used by the `test.Hello` application. The following command creates a file named `hello.classlist` that contains a list of all classes used by this application:

   ```
   java -Xshare:off -XX:+UnlockCommercialFeatures -
   XX:DumpLoadedClassList=hello.classlist -XX:+UseAppCDS -cp hello.jar test.Hello
   ```

   Note that the `-cp` parameter must contain only JAR files; the `-XX:+UseAppCDS` option doesn't support class paths that contain directory names.

2. Create a shared archive, named `hello.jsa`, that contains all the classes in `hello.classlist`:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:dump -XX:+UseAppCDS -
   XX:SharedArchiveFile=hello.jsa -XX:SharedClassListFile=hello.classlist -cp
   hello.jar
   ```

   Note that the `-cp` parameter used at archive creation time must be the same as (or a prefix of) the `-cp` used at run time.

3. Run the application `test.Hello` with the shared archive `hello.jsa`:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:+UseAppCDS -
   XX:SharedArchiveFile=hello.jsa -cp hello.jar test.Hello
   ```

   Ensure that you have specified the option `-Xshare:on` or `-Xshare:auto`. If the option is not specified,`-Xshare:auto` is the default .

4. **Optional**: Verify that the `test.Hello` application is using the class contained in the `hello.jsa` shared archive:

   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:+UseAppCDS -
   XX:SharedArchiveFile=hello.jsa -cp hello.jar -verbose:class test.Hello
   ```

   The output of this command should contain the following text:

   ```
   Loaded test.Hello from shared objects file by sun/misc/Launcher$AppClassLoader
   ```

Sharing a Shared Archive Across Multiple Application Processes

You can share the same archive file across multiple applications processes. This reduces memory usage because the archive is memory-mapped into the address space of the processes. The operating system automatically shares the read-only pages across these processes.

The following steps demonstrate how to create a common archive that can be shared by different applications. Only the classes from `common.jar` are archived in the `common.jsa` (step 3). Classes from `hello.jar` and `hi.jar` are not archived in this

particular example because they are not in the `-cp` path during the archiving step (step 3).

To include classes from `hello.jar` and `hi.jar`, the `.jar` files must be added to the `-cp` path.

1. Create a list of all classes used by the `Hello` application and another list for the `Hi` application:

   ```
   java -XX:+UnlockCommercialFeatures -XX:DumpLoadedClassList=hello.classlist -XX:
   +UseAppCDS -cp common.jar:hello.jar Hello

   java -XX:+UnlockCommercialFeatures -XX:DumpLoadedClassList=hi.classlist -XX:
   +UseAppCDS -cp common.jar:hi.jar Hi
   ```

2. Create a single list of classes used by all the applications that will share the shared archive file.

   Oracle Solaris, Linux, and OS X:: The following commands combine the files `hello.classlist` and `hi.classlist` into one file, `common.classlist`:

   ```
   cat hello.classlist hi.classlist > common.classlist
   ```

   Windows: The following commands combine the files `hello.classlist` and `hi.classlist` into one file, `common.classlist`:

   ```
   type hello.classlist hi.classlist > common.classlist
   ```

3. Create a shared archive, named `common.jsa`, that contains all the classes in `common.classlist`:

   ```
    java -XX:+UnlockCommercialFeatures -Xshare:dump -
   XX:SharedArchiveFile=common.jsa -XX:+UseAppCDS -
   XX:SharedClassListFile=common.classlist -cp common.jar:hello.jar:hi.jar
   ```

   The value of the `-cp` parameter is the common class path prefix shared by the `Hello` and `Hi` applications.

4. Run the `Hello` and `Hi` applications with the same shared archive:

   ```
    java -XX:+UnlockCommercialFeatures -Xshare:on -XX:SharedArchiveFile=common.jsa -
   XX:+UseAppCDS -cp common.jar:hello.jar:hi.jar Hello

   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:SharedArchiveFile=common.jsa -
   XX:+UseAppCDS -cp common.jar:hello.jar:hi.jar Hi
   ```

Specifying Additional Shared Data Added to an Archive File

```
-XX:SharedArchiveConfigFile=shared_config_file
```

The option is used to specify additional shared data added to the archive file. In JDK 9, it supports strings and symbols. The string data and symbol data should be generated by the `jcmd` tool attaching to a running JVM process. See jcmd.

The following is an example of the string and symbol dumping command in `jcmd`:

```
jcmd pid VM.stringtable -verbose
jcmd pid VM.symboltable -verbose
```

The following is an example of a configuration file:

```
VERSION: 1.0
@SECTION: String
```

```
7: test123
1: *
8: segments
@SECTION: Symbol
10 -1: linkMethod
```

In the configuration file example:

- The string entries under `@SECTION: String` use the following format:

  length: string

- The `@SECTION: Symbol` entry uses the following format:

  ```
  length refcount: symbol
  ```

  The *refcount* for a shared symbol is always `-1`.

`@SECTION` specifies the type of the section that follows it. All data within the section must be the same type that's specified by `@SECTION`. Different types of data can't be mixed. Multiple separated data sections for the same type specified by different `@SECTION` are allowed within one *shared_config_file* .

Performance Tuning Examples

You can use the Java advanced runtime options to optimize the performance of your applications.

Tuning for Higher Throughput

Use the following commands and advanced runtime options to achieve higher throughput performance for your application:

```
java -d64 -server -XX:+UseParallelGC -XX:+AggressiveOpts -XX:+UseLargePages -Xmn10g
-Xms26g -Xmx26g
```

Tuning for Lower Response Time

Use the following commands and advanced runtime options to achieve lower response times for your application:

```
java -d64 -XX:+UseG1GC -Xms26g Xmx26g -XX:MaxGCPauseMillis=500 -XX:+PrintGCTimeStamp
```

Keeping the Java Heap Small and Reducing the Dynamic Footprint of Embedded Applications

Use the following advanced runtime options to keep the Java heap small and reduce the dynamic footprint of embedded applications:

```
-XX:MaxHeapFreeRatio=10 -XX:MinHeapFreeRatio=5
```

> **Note:**
>
> The defaults for these two options are 70% and 40% respectively. Because performance sacrifices can occur when using these small settings, you should optimize for a small footprint by reducing these settings as much as possible without introducing unacceptable performance degradation.

Exit Status

The following exit values are typically returned by the launcher when the launcher is called with the wrong arguments, serious errors, or exceptions thrown by the JVM. However, a Java application may choose to return any value by using the API call `System.exit(exitValue)`. The values are:

- `0`: Successful completion
- `>0`: An error occurred

appletviewer

Note: You use the `appletviewer` command to launch the AppletViewer and run applets outside of a web browser. Although available and supported in JDK 9, the Applet API is marked as deprecated in preparation for removal in a future release. Instead of applets, consider alternatives such as Java Web Start or self-contained applications.

Synopsis

`appletviewer` [*`options`*] *`url...`*

`options`
Specifies the command-line options separated by spaces. See **Options for appletviewer**.

`url`
Specifies the location of the documents or resources to be displayed. You can specify multiple URLs separated by spaces.

Description

The `appletviewer` command connects to the documents or resources designated by *`url`* and displays each applet referenced by the documents in its own AppletViewer window. If the documents referred to by *`url`* don't reference any applets with the `OBJECT`, `EMBED`, or `APPLET` tag, then the `appletviewer` command does nothing. The `OBJECT`, `EMBED`, and `APPLET`tags are described in AppletViewer Tags.

The `appletviewer` command requires encoded URLs according to the escaping mechanism defined in RFC2396. Only encoded URLs are supported. However, file names must be unencoded, as specified in RFC2396.

> **Note:**
>
> The `appletviewer` command is intended for development purposes only.

Options for appletviewer

`-encoding` ***`encoding-name`***
Specifies the input HTML file encoding name.

-Jjavaoption
Passes the string *javaoption* as a single argument to the Java interpreter, which runs the AppletViewer. The argument shouldn't contain spaces. Multiple argument words must all begin with the prefix `-J`. This is useful for adjusting the compiler's execution environment or memory usage. Seejava command documentation for more information about JVM options.

AppletViewer Tags

The AppletViewer makes it possible to run a Java applet without using a browser.

The AppletViewer ignores any HTML that isn't immediately relevant to launching an applet. However, it recognizes a wide variety of applet-launching syntax. The HTML code that the AppletViewer recognizes is described in this section. All other HTML code is ignored.

object

The `object` tag is the HTML 4.0 tag for embedding applets and multmedia objects into an HTML page. It's also an Internet Explorer 4.n extension to HTML 3.2 which enables IE to run a Java applet using the latest Java plug-in.

```
<object
  width="pixelWidth"
  height="pixelHeight"
>
  <param name="code" value="yourClass.class">
  <param name="object" value="serializedObjectOrJavaBean">
  <param name="codebase" value="classFileDirectory">
   ...alternate-text
</object>
```

Note:

- The AppletViewer ignores the `classID` attribute, on the assumption that it'spointing to the Java plug-in, with the value:

  ```
  classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  ```

- The AppletViewer also ignores the `codebase` attribute that's usually included as part of the `object` tag, assuming that it points to a Java plug-in in a network cab file with a value like:

  ```
  codebase="http://java.sun.com/products/plugin/1.1/jinstall-11-
  win32.cab#Version=1,1,0,0"
  ```

- The optional `codebase` parameter tag supplies a relative URL that specifies the location of the applet class.
- Either code or object is specified, not both.
- The `type` parameter tag isn't used by AppletViewer, but should be present so that browsers load the plug-in properly. For an applet, the value should be similar to:

  ```
  <param name="type" value="application/x-java-applet;version=1.1">
  ```

  or

  ```
  <param name="type__1" value="application/x-java-applet">
  ```

  For a serialized object or JavaBean, the `type` parameter value should be similar to:

  ```
  <param name="type__2" value="application/x-java-bean;version=1.1">
  ```

  or

  ```
  <param name="type__3" value="application/x-java-bean">
  ```

- Other parameter tags are argument values supplied to the applet.
- The `object` tag recognized by IE4.n and the embed tag recognized by Netscape 4.n can be combined so that an applet can use the latest Java plug-in, regardless of the browser that downloads the applet.
- The AppletViewer doesn't recognize the `java_code`, `java_codebase`, `java_object`, or `java_type param` tags. These tags are needed only when the applet defines parameters with the names `code`, `codebase`, `object`, or `type`, respectively. In that situation, the plug-in recognizes and uses the `java_ version`option in preference to the version is be used by the applet. If the applet requires a parameter with one of these four names, then it might not run in the AppletViewer.

embed

The `embed` tag is the Netscape extension to HTML 3.2 that allows embedding an applet or a multimedia object in an HTML page. It allows a Netscape 4.n browser (which supports HTML 3.2) to run a Java applet using the Java plug-in.

```
<embed
  code="yourClass.class"
```

```
  object="serializedObjectOrJavaBean"
  codebase="classFileDirectory"
  width="pixelWidth"
  height="pixelHeight"
>
   ...
</embed>
```

Note:

- The `object` and `embed` tags can be combined so that an applet can use the latest Java plug-in, regardless of the browser that downloads the applet.
- Unlike the `object` tag, all values specified in an `embed` tag are attributes (part of the tag) rather than parameters (between the start tag and end tag), specified with a `param` tag.
- To supply argument values for applet parameters, you add additional attributes to the `embed` tag.
- The AppletViewer ignores the `src` attribute that's usually part of an `embed` tag.
- Either code or object is specified, not both.
- The optional `codebase` attribute supplies a relative URL that specifies the location of the applet class.
- The `type` attribute isn't used by the AppletViewer, but should be present so that browsers load the plug-in properly.

  For an applet, the value should be similar to:

  ```
  <type="application/x-java-applet;version=1.1">...
  ```

  or

  ```
  <type="application/x-java-applet">...
  ```

  For a serialized object or JavaBean, the `type` parameter value should be similar to:

  ```
  <type="application/x-java-bean;version=1.1">...
  ```

  or

  ```
  <type="application/x-java-bean">...
  ```

- The `pluginspage` attribute isn't used by the AppletViewer, but should be present so that browsers load the plug-in properly. It should point to a Java plug-in in a network cab file with a value like:

  ```
  pluginspage="http://java.sun.com/products/plugin/1.1/jinstall-11-
  win32.cab#Version=1,1,0,0"
  ```

applet

The `applet` tag is the original HTML 3.2 tag for embedding an applet in an HTML page. Applets loaded using the applet tag are run by the browser, which may not be using the latest version of the Java platform. To ensure that the applet runs with the latest

version, use the `object` tag to load the Java plug-in into the browser. The plug-in then runs the applet.

```
<applet
  code="yourClass.class"
  object="serializedObjectOrJavaBean"
  codebase="classFileDirectory"
  width="pixelWidth"
  height="pixelHeight"
>
  <param name="..." value="...">
   ...alternate-text
</applet>
```

> **Note:**
>
> - Either `code` or `object` is specified, not both.
> - The optional `codebase` attribute supplies a relative URL that specifies the location of the applet class.
> - The `param` tags supply argument values for applet parameters.

app

The `app` tag was a short-lived abbreviation for applet that's no longer supported. The AppletViewer translates the tag and prints an equivalent tag that's supported.

```
<app
  class="classFileName" (without a .class suffix)
  src="classFileDirectory"
  width="pixelWidth"
  height="pixelHeight"
>
<param name="..." value="...">
 ...
</app>
```

jar

You can use the `jar` command to create an archive for classes and resources, and to manipulate or restore individual classes or resources from an archive.

Synopsis

```
jar [OPTION...] [ [--release VERSION] [-C dir] files] ...
```

Description

The `jar` command is a general-purpose archiving and compression tool, based on the ZIP and ZLIB compression formats. Initially, the `jar` command was designed to package Java applets or applications; however, beginning with JDK 9, users can use the `jar` command to create modular JARs. For transportation and deployment, it's usually more convenient to package modules as modular JARs.

The syntax for the `jar` command resembles the syntax for the `tar` command. It has several main operation modes, defined by one of the mandatory operation arguments. Other arguments are either options that modify the behavior of the operation or are required to perform the operation.

> **Note:**
>
> Although available and supported in JDK 9, the Applet API is marked as deprecated in preparation for removal in a future release. Instead of applets, consider alternatives such as Java Web Start or self-contained applications.

When modules or the components of an applet or application (files, images and sounds) are combined into a single archive, they can be downloaded by a Java agent (such as a browser) in a single HTTP transaction, rather than requiring a new connection for each piece. This dramatically improves download times. The `jar` command also compresses files, which further improves download time. The `jar` command also enables individual entries in a file to be signed so that their origin can be authenticated. A JAR file can be used as a class path entry, whether or not it's compressed.

An archive becomes a modular JAR when you include a module descriptor, `module-info.class`, in the root of the given directories or in the root of the `.jar` archive. The following operations described in Operation Modifiers Valid Only in Create and Update Modes are valid only when creating or updating a modular jar or updating an existing non-modular jar:

- `--module-version`
- `--hash-modules`
- `--module-path`

> **Note:**
>
> All mandatory or optional arguments for long options are also mandatory or optional for any corresponding short options.

Main Operation Modes

When using the `jar` command, you must specify the operation for it to perform. You specify the operation mode for the `jar` command by including the appropriate operation arguments described in this section. You can mix an operation argument with other one-letter options. Generally the operation argument is the first argument specified on the command line.

`–c` or `--create`
Creates the archive.

`–i=`*`FILE`* or `--generate-index=`*`FILE`*
Generates index information for the specified JAR file.

–t or --list
Lists the table of contents for the archive.

–u or --update
Updates an existing JAR file.

–x or --extract
Extracts the named (or all) files from the archive.

-d or --print-module-descriptor
Prints the module descriptor.

Operation Modifiers Valid in Any Mode

You can use the following options to customize the actions of any operation mode included in the `jar` command.

-C *DIR*
Changes the specified directory and includes the *files* specified at the end of the command line.

```
jar [OPTION...] [ [--release VERSION] [-C dir] files]
```

-f=*FILE* or --file=*FILE*
Specifies the archive file name.

--release *VERSION*
Creates a multirelease JAR file. Places all files specified after the option into a versioned directory of the JAR file named `META-INF/versions/`*VERSION*`/`, where *VERSION* must be must be a positive integer whose value is 9 or greater.
At run time, where more than one version of a class exists in the JAR, the JDK will use the first one it finds, searching initially in the directory tree whose VERSION number matches the JDK's major version number. It will then look in directories with successively lower VERSION numbers, and finally look in the root of the JAR.

-v or --verbose
Sends or prints verbose output to standard output.

Operation Modifiers Valid Only in Create and Update Modes

You can use the following options to customize the actions of the create and the update main operation modes:

–e=*CLASSNAME* or --main-class=*CLASSNAME*
Specifies the application entry point for standalone applications bundled into a modular or executable modular JAR file.

-m=*FILE* or --manifest=*FILE*
Includes the manifest information from the given manifest file.

-M or --no-manifest
Doesn’t create a manifest file for the entries.

--module-version=*VERSION*
Specifies the module version, when creating or updating a modular JAR file, or updating a non-modular JAR file.

`--hash-modules=PATTERN`
Computes and records the hashes of modules matched by the given pattern and that depend upon directly or indirectly on a modular JAR file being created or a non-modular JAR file being updated.

`-p` or `--module-path`
Specifies the location of module dependence for generating the hash.

`@files`
Reads `jar` options and file names from a text file.

Operation Modifiers Valid Only in Create, Update, and Generate-index Modes

You can use the following options to customize the actions of the create (`–c` or `--create`) the update (`–u` or `--update`) and the generate-index (`–i` or `--generate-index=FILE`) main operation modes:

`-0` or `--no-compress`
Stores without using ZIP compression.

Other Options

The following options are recognized by the `jar` command and not used with operation modes:

`-h` or `--help[:compat]`
Displays the command-line help for the `jar` command or optionally the compatibility help.

`--help-extra`
Displays help on extra options.

`--version`
Prints the program version.

Examples of jar Command Syntax

Creates an archive, `classes.jar`, that contains two class files,`Foo.class` and `Bar.class`.

```
jar --create --file classes.jar Foo.class Bar.class
```

Creates an archive, `classes.jar`, by using an existing manifest,`mymanifest`, that contains all of the files in the directory `foo/`.

```
jar --create --file classes.jar --manifest mymanifest -C foo/
```

Creates a modular JAR archive,`foo.jar` , where the module descriptor is located in `classes/module-info.class`.

```
jar --create --file foo.jar --main-class com.foo.Main --module-version 1.0 -C foo/
classes resources
```

Updates an existing non-modular JAR, `foo.jar`, to a modular JAR file.

```
jar --update --file foo.jar --main-class com.foo.Main --module-version 1.0 -C foo/
module-info.class
```

Creates a versioned or multi-release JAR, `foo.jar`, that places the files in the `classes` directory at the root of the JAR, and the files in the `classes–9` directory in the `META-INF/versions/9` directory of the JAR.
In this example, the `classes/com/foo` directory contains two classes, `com.foo.Hello` (the entry point class) and `com.foo.NameProvider`, both compiled for JDK 8. The `classes-9/com/foo` directory contains a different version of the `com.foo.NameProvider` class, this one containing JDK 9 specific code and compiled for JDK 9.
Given this setup, create a multirelease JAR file `foo.jar` by running the following command from the directory containing the directories `classes` and `classes-9` .

```
 jar --create --file foo.jar --main-class com.foo.Hello -C classes . --release 9 -C
classes-9 .
```

The JAR file `foo.jar` now contains:

```
% jar -tf foo.jar

META-INF/
META-INF/MANIFEST.MF
com/
com/foo/
com/foo/Hello.class
com/foo/NameProvider.class
META-INF/versions/9/com/
META-INF/versions/9/com/foo/
META-INF/versions/9/com/foo/NameProvider.class
```

As well as other information, the file `META-INF/MANIFEST.MF`, will contain the following lines to indicate that this is a multirelease JAR file with an entry point of `com.foo.Hello`.

```
...
Main-Class: com.foo.Hello
Multi-Release: true
```

Assuming that the `com.foo.Hello` class calls a method on the `com.foo.NameProvider` class, running the program using JDK 9 will ensure that the `com.foo.NameProvider` class is the one in `META-INF/versions/9/com/foo/`. Running the program using JDK 8 will ensure that the `com.foo.NameProvider` class is the one at the root of the JAR, in `com/foo`.

Creates an archive, `my.jar`, by reading options and lists of class files from the file `classes.list`.

> **Note:**
>
> To shorten or simplify the `jar` command, you can specify arguments in a separate text file and pass it to the `jar` command with the at sign (`@`) as a prefix.

```
jar --create --file my.jar @classes.list
```

jlink

You can use the `jlink` tool to assemble and optimize a set of modules and their dependencies into a custom runtime image.

Synopsis

```
jlink [options] --module-path modulepath --add-modules module [,module...]
```

options
Command-line options separated by spaces. See jlink Options.

modulepath
The path where the `jlink` tool discovers observable modules. These modules can be modular JAR files, JMOD files, or exploded modules.

module
The names of the modules to add to the runtime image. The `jlink` tool adds these modules and their transitive dependencies.

Description

The `jlink` tool links a set of modules, along with their transitive dependences, to create a custom runtime image.

> **Note:**
>
> Developers are responsible for updating their custom runtime images.
>
> Unlike custom runtime images, web-deployed Java applications automatically download application updates from the web as soon as they're available. The Java Auto Update mechanism takes care of updating the JRE to the latest secure version several times every year. Custom runtime images don't have built-in support for automatic updates.

jlink Options

`--add-modules` *`mod`* `[,`*`mod`*`...]`
Adds the named modules, *`mod`*, to the default set of root modules. The default set of root modules is empty.

`--bind-services`
Link service provider modules and their dependencies.

`-c ={0|1|2}` or `--compress={0|1|2}`
Enable compression of resources:

- `0`: No compression
- `1`: Constant string sharing
- `2`: ZIP

`--disable-plugin` *`pluginname`*
Disables the specified plug-in. See jlink Plug-ins for the list of supported plug-ins.

`--endian {little|big}`
Specifies the byte order of the generated image. The default value is the format of your system's architecture.

`-h` or `--help`
Prints the help message.

`--ignore-signing-information`
Suppresses a fatal error when signed modular JARs are linked in the runtime image. The signature-related files of the signed modular JARs aren't copied to the runtime image.

`--launcher` *`command=module`* or `--launcher` *`command=module/main`*
Specifies the launcher command name for the module or the command name for the module and main class (the module and the main class names are separated by a slash (`/`)).

`--limit-modules` *`mod`* `[,`*`mod`*`...]`
Limits the universe of observable modules to those in the transitive closure of the named modules, *`mod`*, plus the main module, if any, plus any further modules specified in the `--add-modules` option.

`--list-plugins`
Lists available plug-ins, which you can access through command-line options; see jlink Plug-ins.

`-p` or `--module-path` *`modulepath`*
Specifies the module path.

`--no-header-files`
Excludes header files.

`--no-man-pages`
Excludes man pages.

`--output` *`path`*
Specifies the location of the generated runtime image.

`--save-opts` *`filename`*
Saves `jlink` options in the specified file.

`--suggest-providers [`*`name`*`, ...]`
Suggest providers that implement the given service types from the module path.

`--version`
Prints version information.

`@filename`
Reads options from the specified file.
An options file is a text file that contains the options and values that you would typically enter in a command prompt. Options may appear on one line or on several lines. You may not specify environment variables for path names. You may comment out lines by prefixing a hash symbol (`#`) to the beginning of the line.

The following is an example of an options file for the `jlink` command:

```
#Wed Dec 07 00:40:19 EST 2016
--module-path C:/Java/jdk9/jmods;mlib
--add-modules com.greetings
--output greetingsapp
```

jlink Plug-ins

> **Note:**
>
> Plug-ins not listed in this section aren't supported and are subject to change.

For plug-in options that require a *`pattern-list`*, the value is a comma-separated list of elements, with each element using one the following forms:

- *`glob-pattern`*
- `glob:`*`glob-pattern`*
- `regex:`*`regex-pattern`*
- *`@filename`*
  - *`filename`* is the name of a file that contains patterns to be used, one pattern per line.

For a complete list of all available plug-ins, run the command `jlink --list-plugins`.

Table 2-3 List of Available jlink plugins

Plugin Name	Option	Description
`class-for-name`	`--class-for-name`	Class optimization, converts `Class.forName` calls to constant loads.
`compress`	`--compress={0|1|2}` `[:filter=`*`pattern-list`*`]`	Compresses all resources in the output image. • Level 0: No compression • Level 1: Constant string sharing • Level 2: ZIP An optional *`pattern-list`* filter can be specified to list the pattern of files to include.
`dedup-legal-notices`	`--dedup-legal-notices=[error-if-not-same-content]`	De-duplicates all legal notices. If `error-if-not-same-content` is specified then it will be an error if two files of the same filename are different.
`exclude-files`	`--exclude-files=`*`pattern-list`*	Specifies files to exclude. such as: `--exclude-files=**.java,glob:/java.base/lib/client/**`

Table 2-3 (Cont.) List of Available jlink plugins

Plugin Name	Option	Description
`exclude-jmod-section`	`--exclude-jmod-section=`*`section-name`*	Specifies a JMOD section to exclude where *`section-name`* is `man` or `headers`.
`exclude-resources`	`--exclude-resources=`*`pattern-list`*	Specify resources to exclude. such as: `--exclude-resources=**.jcov,glob:**/META-INF/**`
`generate-jli-classes`	`--generate-jli-classes=@`*`filename`*`[:ignore-version=<true|false>]`	Specify a file listing the `java.lang.invoke` classes to pre-generate. By default, this plugin may use a built-in list of classes to pre-generate. If this plugin runs on a different runtime version than the image being created, then code generation will be disabled by default to guarantee correctness. Add `ignore-version=true` to override this behavior.
`include-locales`	`--include-locales=`*`langtag`*`[,`*`langtag`*`]*`	Includes the list of locales where *`langtag`* is a BCP 47 language tag. This option supports locale matching as defined in RFC 4647. Ensure that you add the module `jdk.localedata` when using this option. Example: `--add-modules jdk.localedata --include-locales=en,ja,*-IN`
`order-resources`	`--order-resources=`*`pattern-list`*	Orders the specified paths in priority order. If *`@filename`* is specified, then each line in *`pattern-list`* must be an exact match for the paths to be ordered. Example: `--order-resources=**/module-info.class,@classlist,/java.base/java/lang/**`

Table 2-3 (Cont.) List of Available jlink plugins

Plugin Name	Option	Description
`release-info`	`--release-info={`*`file`*`|add:`*`key1`*`=`*`value1`*`:`*`key2`*`=`*`value2`*`:...|del:`*`key-list`*`}`	Loads, adds, or deletes release properties where: • *`file`*: Loads release properties from the specified file. • `add`: Adds specified properties to the release file. You can specify any number of *`key`*`=`*`value`* pairs. • `del`: Deletes the list of keys in the release file *`key-list`*.
`strip-debug`	`--strip-debug`	Strips debug information from the output image
`strip-native-commands`	`--strip-native-commands`	Excludes native commands (such as `java/java.exe`) from the image
`system-modules`	`--system-modules=retainModuleTarget`	Fast loads module descriptors (always enabled)
`vm`	`--vm={client|server|minimal|all}`	Selects the HotSpot VM in the output image. Default is `all`.

jlink Examples

The following command creates a runtime image in the directory `greetingsapp`. This command links the module `com.greetings`, whose module definition is contained in the directory `mlib`. The directory `$JAVA_HOME/jmods` contains `java.base.jmod` and the other standard and JDK modules.

```
jlink --module-path $JAVA_HOME/jmods:mlib --add-modules com.greetings --output
greetingsapp
```

The following command lists the modules in the runtime image `greetingsapp`:

```
greetingsapp/bin/java --list-modules
com.greetings
java.base@9
java.logging@9
org.astro@1.0
```

The following command creates a runtime image in the directory `compressedrt` that's stripped of debug symbols, uses compression to reduce space, and includes French language locale information:

```
jlink --module-path $JAVA_HOME/jmods --add-modules jdk.localedata --strip-debug --
compress=2 --include-locales=fr --output compressedrt
```

The following example compares the size of the runtime image `compressedrt` with `fr_rt`, which isn't stripped of debug symbols and doesn't use compression:

```
jlink --module-path $JAVA_HOME/jmods --add-modules jdk.localedata --include-
locales=fr --output fr_rt
```

```
du -sh ./compressedrt ./fr_rt
23M     ./compressedrt
36M     ./fr_rt
```

The following example lists the providers that implement `java.security.Provider`:

```
jlink --module-path $JAVA-HOME/jmods --suggest-providers java.security.Provider

Suggested providers:
   java.naming provides java.security.Provider used by java.base
   java.security.jgss provides java.security.Provider used by java.base
   java.security.sasl provides java.security.Provider used by java.base
   java.smartcardio provides java.security.Provider used by java.base
   java.xml.crypto provides java.security.Provider used by java.base
   jdk.crypto.cryptoki provides java.security.Provider used by java.base
   jdk.crypto.ec provides java.security.Provider used by java.base
   jdk.crypto.mscapi provides java.security.Provider used by java.base
   jdk.deploy provides java.security.Provider used by java.base
   jdk.security.jgss provides java.security.Provider used by java.base
```

The following example creates a custom runtime image named `mybuild` that includes only `java.naming` and `jdk.crypto.cryptoki` and their dependencies but no other providers. Note that these dependencies must exist in the module path:

```
jlink --module-path $JAVA_HOME/jmods --add-modules java.naming,jdk.crypto.cryptoki --
output mybuild
```

The following command is similar to the one that creates a runtime image named `greetingsapp`, except that it will link the modules resolved from root modules with service binding; see the `Configuration.resolveAndBind` method.

```
jlink --module-path $JAVA_HOME/jmods:mlib --add-modules com.greetings --output
greetingsapp --bind-services
```

The following command lists the modules in the runtime image `greetingsapp` created by this command:

```
greetingsapp/bin/java --list-modules
com.greetings
java.base@9
java.compiler@9
java.datatransfer@9
java.desktop@9
java.logging@9
java.management@9
java.management.rmi@9
java.naming@9
java.prefs@9
java.rmi@9
java.scripting@9
java.security.jgss@9
java.security.sasl@9
java.smartcardio@9
java.xml@9
java.xml.crypto@9
jdk.accessibility@9
jdk.charsets@9
jdk.compiler@9
jdk.crypto.cryptoki@9
jdk.crypto.ec@9
```

```
jdk.crypto.mscapi@9
jdk.deploy@9
jdk.dynalink@9
jdk.internal.opt@9
jdk.jartool@9
jdk.javadoc@9
jdk.jdeps@9
jdk.jlink@9
jdk.localedata@9
jdk.management@9
jdk.naming.dns@9
jdk.naming.rmi@9
jdk.scripting.nashorn@9
jdk.security.auth@9
jdk.security.jgss@9
jdk.unsupported@9
jdk.zipfs@9
org.astro@1.0
```

jmod

You use the `jmod` tool to create JMOD files and list the content of existing JMOD files.

Synopsis

`jmod (create|extract|list|describe|hash) [`*`options`*`] `*`jmod-file`*

Includes the following:

Main operation modes

`create`
Creates a new JMOD archive file.

`extract`
Extracts all the files from the JMOD archive file.

`list`
Prints the names of all the entries.

`describe`
Prints the module details.

`hash`
Determines leaf modules and records the hashes of the dependencies that directly and indirectly require them.

Options

`options`
See Options for jmod.

Required

`jmod-file`
Specifies the name of the JMOD file to create or from which to retrieve information.

Description

> **Note:**
>
> For most development tasks, including deploying modules on the module path or publishing them to a Maven repository, continue to package modules in modular JAR files. The `jmod` tool is intended for modules that have native libraries or other configuration files or for modules that you intend to link, with the `jlink` tool, to a runtime image.

The JMOD file format let's you aggregate files other than `.class` files, metadata, and resources. This format is transportable but not executable, which means that you can use it during compile time or link time but not at run time.

Many `jmod` options involve specifying a path whose contents are copied into the resulting JMOD files. These options copy all the contents of the specified path, including subdirectories and their contents, but exclude files whose names match the pattern specified by the `--exclude` option.

With the `--hash-modules` option or the `jmod hash` command, you can, in each module's descriptor, record hashes of the content of the modules that are allowed to depend upon it, thus "tying" together these modules. This let's you to allow a package to be exported to one or more specifically-named modules and to no others through qualified exports. The runtime verifies if the recorded hash of a module matches the one resolved at run time; if not, the runtime returns an error.

Options for jmod

`--class-path` *`path`*
Specifies the location of application JAR files or a directory containing classes to copy into the resulting JMOD file.

`--cmds` *`path`*
Specifies the location of native commands to copy into the resulting JMOD file.

`--config` *`path`*
Specifies the location of user-editable configuration files to copy into the resulting JMOD file.

`--dir` *`path`*
Specifies the location where `jmod` puts extracted files from the specified JMOD archive.

`--dry-run`
Performs a dry run of hash mode. It identifies leaf modules and their required modules without recording any hash values.

`--exclude` *`pattern-list`*
Excludes files matching the supplied comma-separated pattern list, each element using one the following forms:

- *`glob-pattern`*

- `glob:glob-pattern`
- `regex:regex-pattern`

See the `FileSystem.getPathMatcher` method for the syntax of *`glob-pattern`*. See the `Pattern` class for the syntax of *`regex-pattern`*, which represents a regular expression.

`--hash-modules` *`regex-pattern`*
Determines the leaf modules and records the hashes of the dependencies directly and indirectly requiring them, based on the module graph of the modules matching the given *`regex-pattern`*. The hashes are recorded in the JMOD archive file being created, or a JMOD archive or modular JAR on the module path specified by the `jmod hash` command.

`--header-files` *`path`*
Specifies the location of header files to copy into the resulting JMOD file.

`--help` or `-h`
Prints a usage message.

`--help-extra`
Prints help for extra options.

`--legal-notices` *`path`*
Specifies the location of legal notices to copy into the resulting JMOD file.

`--libs` *`path`*
Specifies location of native libraries to copy into the resulting JMOD file.

`--main-class` *`class-name`*
Specifies main class to record in the `module-info.class` file.

`--man-pages` *`path`*
Specifies the location of man pages to copy into the resulting JMOD file.

`--module-version` *`module-version`*
Specifies the module version to record in the `module-info.class` file.

`--module-path` *`path`* or `-p` *`path`*
Specifies the module path. This option is required if you also specify `--hash-modules`.

`--target-platform` *`platform`*
Specifies the target platform.

`--version`
Prints version information of the `jmod` tool.

`@filename`
Reads options from the specified file.
An options file is a text file that contains the options and values that you would ordinarily enter in a command prompt. Options may appear on one line or on several lines. You may not specify environment variables for path names. You may comment out lines by prefixinga hash symbol (#) to the beginning of the line.
The following is an example of an options file for the `jmod` command:

```
#Wed Dec 07 00:40:19 EST 2016
create --class-path mods/com.greetings --module-path mlib
  --cmds commands --config configfiles --header-files src/h
  --libs lib --main-class com.greetings.Main
  --man-pages man --module-version 1.0
  --os-arch "x86_x64" --os-name "Mac OS X"
  --os-version "10.10.5" greetingsmod
```

Extra Options for jmod

In addition to the options described in Options for jmod , the following are extra options that can be used with the command.

`--do-not-resolve-by-default`
Exclude from the default root set of modules

`--warn-if-resolved`
Hint for a tool to issue a warning if the module is resolved. One of deprecated, deprecated-for-removal, or incubating.

jmod Create Example

The following is an example of creating a JMOD file:

```
jmod create --class-path mods/com.greetings --cmds commands
  --config configfiles --header-files src/h --libs lib
  --main-class com.greetings.Main --man-pages man --module-version 1.0
  --os-arch "x86_x64" --os-name "Mac OS X"
  --os-version "10.10.5" greetingsmod
```

jmod Hash Example

The following example demonstrates what happens when you try to link a leaf module (in this example, `ma`) with a required module (`mb`), and the hash value recorded in the required module doesn't match that of the leaf module.

1. Create and compile the following `.java` files:
   - `jmodhashex/src/ma/module-info.java`

     ```
     module ma {
       requires mb;
     }
     ```

   - `jmodhashex/src/mb/module-info.java`

     ```
     module mb {
     }
     ```

   - `jmodhashex2/src/ma/module-info.java`

     ```
     module ma {
       requires mb;
     }
     ```

   - `jmodhashex2/src/mb/module-info.java`

     ```
     module mb {
     }
     ```

2. Create a JMOD archive for each module. Create the directories `jmodhashex/jmods` and `jmodhashex2/jmods`, and then run the following commands from the `jmodhashex` directory, then from the `jmodhashex2` directory:

- `jmod create --class-path mods/ma jmods/ma.jmod`
- `jmod create --class-path mods/mb jmods/mb.jmod`

3. Optionally preview the `jmod hash` command. Run the following command from the `jmodhashex` directory:

   ```
   jmod hash --dry-run -module-path jmods --hash-modules .*
   ```

   The command prints the following:

   ```
   Dry run:
   mb
     hashes ma SHA-256
   07667d5032004b37b42ec2bb81b46df380cf29e66962a16481ace2e71e74073a
   ```

   This indicates that the `jmod hash` command (without the `--dry-run` option) will record the hash value of the leaf module `ma` in the module `mb`.

4. Record hash values in the JMOD archive files contained in the `jmodhashex` directory. Run the following command from the `jmodhashex` directory:

   ```
   jmod hash --module-path jmods --hash-modules .*
   ```

   The command prints the following:

   ```
   Hashes are recorded in module mb
   ```

5. Print information about each JMOD archive contained in the `jmodhashex`directory. Run the highlighted commands from the `jmodhashex` directory:

   ```
   jmod describe jmods/ma.jmod

   ma
     requires mandated java.base
     requires mb

   jmod describe jmods/mb.jmod

   mb
     requires mandated java.base
     hashes ma SHA-256
   07667d5032004b37b42ec2bb81b46df380cf29e66962a16481ace2e71e74073a
   ```

6. Attempt to create a runtime image that contains the module `ma` from the directory `jmodhashex2` but the module `mb` from the directory `jmodhashex`. Run the following command from the `jmodhashex2` directory:

   - **Oracle Solaris, Linux, and OS X:** `jlink --module-path $JAVA_HOME/jmods:jmods/ma.jmod:../jmodhashex/jmods/mb.jmod --add-modules ma --output ma-app`
   - **Windows:** `jlink --module-path %JAVA_HOME%/jmods;jmods/ma.jmod;../jmodhashex/jmods/mb.jmod --add-modules ma --output ma-app`

   The command prints an error message similar to the following:

   ```
   Error: Hash of ma
   (a2d77889b0cb067df02a3abc39b01ac1151966157a68dc4241562c60499150d2) differs to
   expected hash (07667d5032004b37b42ec2bb81b46df380cf29e66962a16481ace2e71e74073a)
   recorded in mb
   ```

jdeps

You use the `jdeps` command to launch the Java class dependency analyzer.

Synopsis

```
jdeps [options] path ...
```

options
Command-line options. For detailed descriptions of the options that can be used, see

- Possible Options
- Module Dependence Analysis Options
- Options to Filter Dependences
- Options to Filter Classes to be Analyzed

path
A pathname to the `.class` file, directory, or JAR file to analyze.

Description

The `jdeps` command shows the package-level or class-level dependencies of Java class files. The input class can be a path name to a `.class` file, a directory, a JAR file, or it can be a fully qualified class name to analyze all class files. The options determine the output. By default, the `jdeps` command writes the dependencies to the system output. The command can generate the dependencies in DOT language (see the `-dotoutput` option).

Possible Options

`-dotoutput` *`dir`* or `--dot-output` *`dir`*
Specifies the destination directory for DOT file output. If this option is specified, then the `jdeps` command generates one `.dot` file for each analyzed archive named `archive-file-name.dot` that lists the dependencies, and also a summary file named `summary.dot` that lists the dependencies among the archive files.

`-s` or `-summary`
Prints a dependency summary only.

`-v` or `-verbose`
Prints all class-level dependencies. This is equivalent to

```
-verbose:class -filter:none
```

`-verbose:package`
Prints package-level dependencies excluding, by default, dependences within the same package.

`-verbose:class`
Prints class-level dependencies excluding, by default, dependencies within the same archive.

`-apionly` or `--api-only`
Restricts the analysis to APIs, for example, dependences from the signature of `public` and `protected` members of public classes including field type, method parameter types, returned type, and checked exception types.

`-jdkinternals` or `--jdk-internals`
Finds class-level dependences in the JDK internal APIs. By default, this option analyzes all classes specified in the `--classpath` option and input files unless you specified the `-include` option. You can't use this option with the `-p`, `-e`, and `-s` options.
Warning: The JDK internal APIs are inaccessible.

`-cp` *`path`*, `-classpath` *`path`* , or `--classpath` *`path`*
Specifies where to find class files.

`--module-path` *`module-path`*
Specifies the module path.

`--upgrade-module-path` *`module-path`*
Specifies the upgrade module path.

`--system` *`java-home`*
Specifies an alternate system module path.

`--add-modules` *`module-name`* `[,` *`module-name`*`...]`
Adds modules to the root set for analysis.

`--multi-release` *`version`*
Specifies the version when processing multi-release JAR files *`version`* should be an integer >=9 or base.

`–q` or `-quite`
Doesn't show missing dependencies from `–generate-module-info` output.

`-version` or `--version`
Prints version information.

Module Dependence Analysis Options

`–m` *`module-name`* or `--module` *`module-name`*
Specifies the root module for analysis.

`--generate-module-info` *`dir`*
Generates `module-info.java` under the specified directory. The specified JAR files will be analyzed. This option cannot be used with `--dot-output` or `--class-path` options. Use the `--generate-open-module` option for open modules.

`--generate-open-module` *`dir`*
Generates `module-info.java` for the specified JAR files under the specified directory as open modules. This option cannot be used with the `--dot-output` or `--class-path` options.

`--check` *`module-name`* `[,` *`module-name`*`...]`
Analyzes the dependence of the specified modules. It prints the module descriptor, the resulting module dependences after analysis and the graph after transition reduction. It also identifies any unused qualified exports.

`--list-deps`
Lists the module dependences and also the package names of JDK internal APIs (if referenced).

`--list–reduced-deps`
Same as `--list-deps` without listing the implied reads edges from the module graph. If module M1 reads M2, and M2 requires transitive on M3, then M1 reading M3 is implied and is not shown in the graph.

Options to Filter Dependences

`-p` *`pkg name`*, `-package` *`pkg name`*, or `--package` *`pkg name`*
Finds dependences matching the specified package name. You can specify this option multiple times for different packages. The `-p` and `-e` options are mutually exclusive.

`-e` *`regex`* , `-regex` *`regex`* , or `--regex` *`regex`*
Finds dependences matching the specified pattern. The `-p` and `-e` options are mutually exclusive.

`--require` *`module-name`*
Finds dependences matching the given module name (may be given multiple times). The `--package`, `--regex`, and `--require` options are mutually exclusive.

`-f` *`regex`* or `-filter`*`regex`*
Filters dependences matching the given pattern. If give multiple times, the last one will be selected.

`-filter:package`
Filters dependences within the same package. This is the default.

`-filter:archive`
Filters dependences within the same archive.

`-filter:module`
Filters dependences within the same module.

`-filter:none`
No `-filter:package` and `-filter:archive` filtering. Filtering specified via the `-filter` option still applies.

Options to Filter Classes to be Analyzed

`-include` *`regex`*
Restricts analysis to the classes matching pattern. This option filters the list of classes to be analyzed. It can be used together with `-p` and `-e`, which apply the pattern to the dependencies.

`-P` or `-profile`
Shows the profile containing a package.

`-R` or `-recursive`
Recursively traverses all run-time dependences. The `-R` option implies `-filter:none`. If `-p`, `-e`, or `-f` options are specified, only the matching dependences are analyzed.

`-I` or `-inverse`
Analyzes the dependences per other given options and then finds all artifacts that directly and indirectly depend on the matching nodes. This is equivalent to the inverse of the compile-time view analysis and the print dependency summary. This option must be used with the `--require`, `--package`, or `--regex` options.

`--compile-time`
Analyzes the compile-time view of transitive dependencies, such as the compile-time view of the `-R` option. Analyzes the dependences per other specified options. If a dependency is found from a directory, a JAR file or a module, all classes in that containing archive are analyzed.

Example of Analyzing Dependencies

The following example demonstrates analyzing the dependencies of the `Notepad.jar` file.

Oracle Solaris, Linux, and OS X:

```
$ jdeps demo/jfc/Notepad/Notepad.jar
Notepad.jar -> java.base
Notepad.jar -> java.desktop
Notepad.jar -> java.logging
   <unnamed> (Notepad.jar)
      -> java.awt
      -> java.awt.event
      -> java.beans
      -> java.io
      -> java.lang
      -> java.net
      -> java.util
      -> java.util.logging
      -> javax.swing
      -> javax.swing.border
      -> javax.swing.event
      -> javax.swing.text
      -> javax.swing.tree
      -> javax.swing.undo
```

Windows:

```
C:\Java\jdk1.9.0>jdeps demo\jfc\Notepad\Notepad.jar
Notepad.jar -> java.base
Notepad.jar -> java.desktop
Notepad.jar -> java.logging
   <unnamed> (Notepad.jar)
      -> java.awt
      -> java.awt.event
      -> java.beans
      -> java.io
      -> java.lang
      -> java.net
      -> java.util
      -> java.util.logging
      -> javax.swing
      -> javax.swing.border
      -> javax.swing.event
      -> javax.swing.text
      -> javax.swing.tree
      -> javax.swing.undo
```

Example Using the --inverse Option

```
 $ jdeps --inverse --require java.xml.bind
Inverse transitive dependences on [java.xml.bind]
java.xml.bind <- java.se.ee
java.xml.bind <- jdk.xml.ws
java.xml.bind <- java.xml.ws <- java.se.ee
java.xml.bind <- java.xml.ws <- jdk.xml.ws
java.xml.bind <- jdk.xml.bind <- jdk.xml.ws
```

jdeprscan

You use the `jdeprscan` tool as a static analysis tool that scans a jar file (or some other aggregation of class files) for uses of deprecated API elements.

Synopsis

`jdeprscan [ options ]{dir|jar|class}`

`options`
See Options for the jdeprscan Command

`dir|jar|class`
`jdeprscan` command scans each argument for usages of deprecated APIs. The arguments can be a:

- *`dir`*: Directory
- *`jar`*: JAR file
- *`class`*: Class name or class file

The class name should use a dot (`.`) as a separator. For example:
`java.lang.Thread`
For nested classes, the dollar sign `$` separator character should be used. For example:
`java.lang.Thread$State`
A class file can also be named. For example:
`build/classes/java/lang/Thread$State.class`

Description

The `jdeprscan` tool is a static analysis tool provided by the JDK that scans a JAR file or some other aggregation of class files for uses of deprecated API elements. The deprecated APIs identified by the `jdeprscan` tool are only those that are defined by Java SE. Deprecated APIs defined by third-party libraries aren't reported.

To scan a JAR file or a set of class files, you must first ensure that all of the classes that the scanned classes depend upon are present in the class path. Set the class path using the `--class-path` option described in Options for the jdeprscan Command.

Typically, you would use the same class path as the one that you use when invoking your application.

If the `jdeprscan` can't find all the dependent classes, it will generate an error message for each class that's missing. These error messages are typically of the form:

```
error: cannot find class ...
```

If these errors occur, then you must adjust the class path so that it includes all dependent classes.

Options for the jdeprscan Command

The following options are available:

`--class-path` *`PATH`*
Provides a search path for resolution of dependent classes.
`PATH` can be a search path that consists of one or more directories separated by the system-specific path separator. For example:

- **Oracle Solaris, Linux, and OS X:**

  ```
  --class-path /some/directory:/another/different/dir
  ```

> **Note:**
>
> On Windows, use a semicolon (`;`) as the separator instead of a colon (`:`).

- **Windows:**

  ```
  --class-path \some\directory;\another\different\dir
  ```

`--for-removal`
Limits scanning or listing to APIs that are deprecated for removal. Can't be used with a release value of 6, 7, or 8.

`--full-version`
Prints out the full version string of the tool.

`--help` or `–h`
Prints out a full help message.

`--list` or `–l`
Prints the set of deprecated APIs. No scanning is done, so no directory, jar, or class arguments should be provided.

`--release` *`6|7|8|9`*
Specifies the Java SE release that provides the set of deprecated APIs for scanning.

`--verbose` or `–v`
Enables additional message output during processing.

`--version`
Prints out the abbreviated version string of the tool.

Example of jdeprscan Output

The JAR file for this library will be named something similar to `commons-math3-3.6.1.jar`. To scan this JAR file for the use of deprecated APIs, run the following command:

```
jdeprscan commons-math3-3.6.1.jar
```

This command produces several lines of output. For example, one line of output might be:

```
class org/apache/commons/math3/util/MathUtils uses deprecated method java/lang/
Double::<init>(D)V
```

> **Note:**
>
> The class name is specified using the slash-separated binary name as described in JVMS 4.2.1. This is the form used internally in class files.

The deprecated API it uses is a method on the `java.lang.Double` class.

The name of the deprecated method is `<init>`, which is a special name that means that the method is actually a constructor. Another special name is `<clinit>`, which indicates a class static initializer.

Other methods are listed just by their method name. Following the method name is the argument list and return type:

```
(D)V
```

This indicates that it takes just one double value (a primitive) and returns void. The argument and return types can become cryptic. For example, another line of output might be:

```
class org/apache/commons/math3/util/Precision uses deprecated method java/math/
BigDecimal::setScale(II)Ljava/math/BigDecimal;
```

In this line of output, the deprecated method is on class `java.math.BigDecimal`, and the method is `setScale()`. In this case, the `(II)` means that it takes two `int` arguments . The `Ljava/math/BigDecimal;` after the parentheses means that it returns a reference to `java.math.BigDecimal`.

jdeprscan Analysis Can Be Version-Specific

You can use `jdeprscan` relative to the previous three JDK releases. For example, if you are running JDK 9, then you can check against JDK 8, 7, and 6.

As an example, look at this code snippet:

```
public class Deprecations {
   SecurityManager sm = new RMISecurityManager();    // deprecated in 8
   Boolean b2 = new Boolean(true);          // deprecated in 9
}
```

The complete class compiles without warnings in JDK 7.

If you run `jdeprscan` on a system with JDK 9, then you see:

```
$ jdeprscan --class-path classes --release 7 example.Deprecations
(no output)
```

Run `jdeprscan` with a release value of 8:

```
$ jdeprscan --class-path classes --release 8 example.Deprecations
class example/Deprecations uses type java/rmi/RMISecurityManager deprecated
class example/Deprecations uses method in type java/rmi/RMISecurityManager deprecated
```

Run `jdeprscan` on JDK 9:

```
$ jdeprscan --class-path classes example.Deprecations
class example/Deprecations uses type java/rmi/RMISecurityManager deprecated
class example/Deprecations uses method in type java/rmi/RMISecurityManager deprecated
class example/Deprecations uses method java/lang/Boolean <init> (Z)V deprecated
```

3
Language Shell

You use the language shell to learn the Java language, explore new features and APIs, and prototype new code.

The following topic describes the Java language shell:

- jshell: Interactively evaluates declarations, statements, and expressions of the Java programming language in a read-eval-print loop (REPL).

jshell

You use the `jshell` tool to interactively evaluate declarations, statements, and expressions of the Java programming language in a read-eval-print loop (REPL).

Synopsis

`jshell [options] [load-files]`

options
Command-line options, separated by spaces. See Options for jshell.

load-files
One or more scripts to run when the tool is started. Scripts can contain any valid code snippets or JShell commands.
The script can be a local file or one of following predefined scripts:

DEFAULT
Loads the default entries, which are commonly used as imports.

JAVASE
Imports all Java SE packages.

PRINTING
Defines `print`, `println`, and `printf` as `jshell` methods for use within the tool.

For more than one script, use a space to separate the names. Scripts are run in the order in which they're entered on the command line. Command-line scripts are run after startup scripts. To run a script after JShell is started, use the `/open` command.

Description

JShell provides a way to interactively evaluate declarations, statements, and expressions of the Java programming language, making it easier to learn the language, explore unfamiliar code and APIs, and prototype complex code. Java statements, variable definitions, method definitions, class definitions, import statements, and expressions are accepted. The bits of code entered are called snippets.

As snippets are entered, they're evaluated, and feedback is provided. Feedback varies from the results and explanations of actions to nothing, depending on the snippet

entered and the feedback mode chosen. Errors are described regardless of the feedback mode. Start with the verbose mode to get the most feedback while learning the tool.

Command-line options are available for configuring the initial environment when JShell is started. Within JShell, commands are available for modifying the environment as needed.

Existing snippets can be loaded from a file to initialize a JShell session, or at any time within a session. Snippets can be modified within the session to try out different variations and make corrections. To keep snippets for later use, save them to a file.

Options for jshell

`--add-modules` *`module`*`[,`*`module`*`...]`
Specifies the root modules to resolve in addition to the initial module. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`-C`*`flag`*
Provides a flag to pass to the compiler. To pass more than one flag, provide an instance of this option for each flag or flag argument needed.

`--class-path` *`path`*
Specifies the directories and archives that are searched to locate class files. This option overrides the path in the `CLASSPATH` environment variable. If the environment variable isn't set and this option isn't used, then the current directory is searched. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`--feedback` *`mode`*
Sets the initial level of feedback provided in response to what's entered. The initial level can be overridden within a session by using the `/set feedback` *`mode`* command. The default is `normal`.
The following values are valid for *`mode`*:

`verbose`
Provides detailed feedback for entries. Additional information about the action performed is displayed after the result of the action. The next prompt is separated from the feedback by a blank line.

`normal`
Provides an average amount of feedback. The next prompt is separated from the feedback by a blank line.

`concise`
Provides minimal feedback. The next prompt immediately follows the code snippet or feedback.

`silent`
Provides no feedback. The next prompt immediately follows the code snippet.

`custom`
Provides custom feedback based on how the mode is defined. Custom feedback modes are created within JShell by using the `/set mode` command.

`--help`
Prints a summary of standard options and exits the tool.

`--help-extra` or **`-X`**
Prints a summary of nonstandard options and exits the tool. Nonstandard options are subject to change without notice.

`-J`*`flag`*
Provides a flag to pass to the runtime system. To pass more than one flag, provide an instance of this option for each flag or flag argument needed.

`--module-path` ***`modulepath`***
Specifies where to find application modules. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`--no-startup`
Prevents startup scripts from running when JShell starts. Use this option to run only the scripts entered on the command line when JShell is started, or to start JShell without any preloaded information if no scripts are entered. This option can't be used if the `--startup` option is used.

`-q`
Sets the feedback mode to `concise`, which is the same as entering `--feedback concise`.

`-R`*`flag`*
Provides a flag to pass to the remote runtime system. To pass more than one flag, provide an instance of this option for each flag or flag argument to pass.

`-s`
Sets the feedback mode to `silent`, which is the same as entering `--feedback silent`.

`--show-version`
Prints version information and enters the tool.

`--startup` ***`file`***
Overrides the default startup script for this session. The script can contain any valid code snippets or commands.
The script can be a local file or one of the following predefined scripts:

`DEFAULT`
Loads the default entries, which are commonly used as imports.

`JAVASE`
Imports all Java SE packages.

`PRINTING`
Defines `print`, `println`, and `printf` as `jshell` methods for use within the tool.

For more than one script, provide a separate instance of this option for each script. Startup scripts are run when JShell is first started and when the session is restarted with the `/reset`, `/reload`, or `/env` command. Startup scripts are run in the order in which they're entered on the command line.
This option can't be used if the `--no-startup` option is used.

`-v`
Sets the feedback mode to `verbose`, which is the same as entering `--feedback verbose`.

`--version`
Prints version information and exits the tool.

jshell Commands

Within the `jshell` tool, commands are used to modify the environment and manage code snippets.

`/drop [`*`name`*`[ `*`name`*`...]|`*`id`*`[ `*`id`*`...]]`
Drops a snippet, making it inactive. Provide either the name or the ID of an import, class, method, or variable. For more than one snippet, separate the names and IDs with a space. Use the `/list` command to see the IDs of code snippets.

`/edit [`*`option`*`]`
Opens an editor. If no option is entered, then the editor opens with the active snippets.
The following options are valid:

> ***`name`*`[ `*`name`*`...]|`*`id`*`[ `*`id`*`...]`**
> Opens the editor with the snippets identified by *`name`* or *`id`*. For more than one snippet, separate the names and IDs with a space.
>
> **`-all`**
> Opens the editor with all snippets, including startup snippets and snippets that failed, were overwritten, or were dropped.
>
> **`-start`**
> Opens the editor with startup snippets that were evaluated when JShell was started.

To exit edit mode, close the editor window, or respond to the prompt provided if the `-wait` option was used when the editor was set.
Use the `/set editor` command to specify the editor to use. If no editor is set, then the following environment variables are checked in order: `JSHELLEDITOR`, `VISUAL`, and `EDITOR`. If no editor is set in JShell and none of the editor environment variables is set, then a simple default editor is used.

`/env [`*`options`*`]`
Displays the environment settings, or updates the environment settings and restarts the session. If no option is entered, then the current environment settings are displayed. If one or more options are entered, then the session is restarted as follows:

- Updates the environment settings with the provided options.
- Resets the execution state.
- Runs the startup scripts.
- Silently replays the history in the order entered. The history includes all valid snippets or `/drop` commands entered at the `jshell` prompt, in scripts entered on the command line, or scripts entered with the `/open` command.

Environment settings entered on the command line or provided with a previous `/reset`, `/env`, or `/reload` command are maintained unless an *`option`* is entered that overwrites the setting.
The following options are valid:

--add-modules *module*[,*module*...]
Specifies the root modules to resolve in addition to the initial module. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

--add-exports *source-module/package=target-module*[,*target-module*]*
Adds an export of *package* from *source-module* to *target-module*.

--class-path *path*
Specifies the directories and archives that are searched to locate class files. This option overrides the path in the CLASSPATH environment variable. If the environment variable isn't set and this option isn't used, then the current directory is searched. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

--module-path *modulepath*
Specifies where to find application modules. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

/exit
Exits the tool.

/history
Displays what was entered in this session.

/help [*command*|*subject*]
Displays information about commands and subjects. If no options are entered, then a summary of information for all commands and a list of available subjects are displayed. If a valid command is provided, then expanded information for that command is displayed. If a valid subject is entered, then information about that subject is displayed.
The following values for *subject* are valid:

context
Describes the options that are available for configuring the environment.

intro
Provides an introduction to the tool.

shortcuts
Describes keystrokes for completing commands and snippets.

/imports
Displays the current active imports, including those from the startup scripts and scripts that were entered on the command line when JShell was started.

/list [*option*]
Displays a list of snippets and their ID. If no option is entered, then all active snippets are displayed, but startup snippets aren't.
The following options are valid:

***name*[*name*...]|*id*[*id*...]**
Displays the snippets identified by *name* or *id*. For more than one snippet, separate the names and IDs with a space.

-all
Displays all snippets, including startup snippets and snippets that failed, were overwritten, or were dropped. IDs that begin with `s` are startup snippets. IDs that begin with `e` are snippets that failed.

-start
Displays startup snippets that were evaluated when JShell was started.

/methods [*option*]
Displays information about the methods that were entered. If no option is entered, then the name, parameter types, and return type of all active methods are displayed. The following options are valid:

***name*[*name*...]|*id*[*id*...]**
Displays information for methods identified by *name* or *id*. For more than one method, separate the names and IDs with a space. Use the `/list` command to see the IDs of code snippets.

-all
Displays information for all methods, including those added when JShell was started, and methods that failed, were overwritten, or were dropped.

-start
Displays information for startup methods that were added when JShell was started.

/open *file*
Opens the script specified and reads the snippets into the tool. The script can be a local file or one of the following predefined scripts:

DEFAULT
Loads the default entries, which are commonly used as imports.

JAVASE
Imports all Java SE packages.

PRINTING
Defines `print`, `println`, and `printf` as `jshell` methods for use within the tool.

/reload [*options*]
Restarts the session as follows:

- Updates the environment settings with the provided options, if any.
- Resets the execution state.
- Runs the startup scripts.
- Replays the history in the order entered. The history includes all valid snippets or `/drop` commands entered at the `jshell` prompt, in scripts entered on the command line, or scripts entered with the `/open` command.

Environment settings entered on the command line or provided with a previous `/reset`, `/env`, or `/reload` command are maintained unless an *option* is entered that overwrites the setting.
The following options are valid:

`--add-modules` *`module`*`[,`*`module`*`...]`
Specifies the root modules to resolve in addition to the initial module. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`--add-exports` *`module/package=target-module`*`[,`*`target-module`*`]*`
Adds an export of *`package`* from *`source-module`* to *`target-module`*.

`--class-path` *`path`*
Specifies the directories and archives that are searched to locate class files. This option overrides the path in the `CLASSPATH` environment variable. If the environment variable isn't set and this option isn't used, then the current directory is searched. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`--module-path` *`modulepath`*
Specifies where to find application modules. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`-quiet`
Replays the valid history without displaying it. Errors are displayed.

`-restore`
Resets the environment to the state at the start of the previous run of the tool or to the last time a `/reset` , `/reload`, or `/env` command was executed in the previous run. The valid history since that point is replayed. Use this option to restore a previous JShell session.

`/reset [`*`options`*`]`
Discards all entered snippets and restarts the session as follows:

- Updates the environment settings with the provided options, if any.
- Resets the execution state.
- Runs the startup scripts.

History is not replayed. All code that was entered is lost.
Environment settings entered on the command line or provided with a previous `/reset`, `/env`, or `/reload` command are maintained unless an *`option`* is entered that overwrites the setting.
The following options are valid:

`--add-modules` *`module`*`[,`*`module`*`...]`
Specifies the root modules to resolve in addition to the initial module. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`--add-exports` *`module/package=target-module`*`[,`*`target-module`*`]*`
Adds an export of *`package`* from *`source-module`* to *`target-module`*.

`--class-path` *`path`*
Specifies the directories and archives that are searched to locate class files. This option overrides the path in the `CLASSPATH` environment variable. If the environment variable isn't set and this option isn't used, then the current directory is searched. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`--module-path` *`modulepath`*
Specifies where to find application modules. For Linux and macOS, use a colon (:) to separate items in the list. For Windows, use a semicolon (;) to separate items.

`/save [`*`options`*`]` *`file`*
Saves snippets and commands to the file specified. If no options are entered, then active snippets are saved.
The following options are valid:

`-all`
Saves all snippets, including startup snippets and snippets that were overwritten or failed.

`-history`
Saves the sequential history of all commands and snippets entered in the current session.

`-start`
Saves the current startup settings. If no startup scripts were provided, then an empty file is saved.

`/set [`*`setting`*`]`
Sets configuration information, including the external editor, startup settings, and feedback mode. This command is also used to create a custom feedback mode with customized prompt, format, and truncation values. If no setting is entered, then the current setting for the editor, startup settings, and feedback mode are displayed.
The following values are valid for *`setting`*:

`editor [`*`options`*`] [`*`command`*`]`
Sets the command used to start an external editor when the `/edit` command is entered. The command can include command arguments separated by spaces. If no command or options are entered, then the current setting is displayed.
The following options are valid:

`-default`
Sets the editor to the default editor provided with JShell. Cannot be used if a command for starting an editor is entered.

`-delete`
Sets the editor to the one in effect when the session started. If used with the `-retain` option, then the retained editor setting is deleted and the editor is set to the first of the following environment variables found: `JSHELLEDITOR`, `VISUAL`, or `EDITOR`. If none of the editor environment variables are set, then this option sets the editor to the default editor.
This option can't be used if a command for starting an editor is entered.

`-retain`
Saves the editor setting across sessions. If no other option or a command is entered, then the current setting is saved.

`-wait`
Prompts the user to indicate when editing is complete. Otherwise control returns to JShell when the editor exits. Use this option if the editor being used

exits immediately, for example, when an edit window already exists. This option is valid only when a command for starting an editor is entered.

feedback [*mode*]
Sets the feedback mode used to respond to input. If no mode is entered, then the current mode is displayed.
The following modes are valid: `concise`, `normal`, `silent`, `verbose`, and any custom mode created with the `/set mode` command.

format *mode field* "*format-string*" *selector*
Sets the format of the feedback provided in response to input. If no mode is entered, then the current formats for all fields for all feedback modes are displayed. If only a mode is entered, then the current formats for that mode are displayed. If only a mode and field are entered, then the current formats for that field are displayed.
To define a format, the following arguments are required:

mode
Specifies a feedback mode to which the response format is applied. Only custom modes created with the `/set mode` command can be modified.

field
Specifies a context-specific field to which the response format is applied. The fields are described in the online help, which is accessed from JShell using the `/help /set format` command.

"***format-string***"
Specifies the string to use as the response format for the specified field and selector. The structure of the format string is described in the online help, which is accessed from JShell using the `/help /set format` command.

selector
Specifies the context in which the response format is applied. The selectors are described in the online help, which is accessed from JShell using the `/help /set format` command.

mode [*mode-name*] [*existing-mode*] [*options*]
Creates a custom feedback mode with the mode name provided. If no mode name is entered, then the settings for all modes are displayed, which includes the mode, prompt, format, and truncation settings. If the name of an existing mode is provided, then the settings from the existing mode are copied to the mode being created.
The following options are valid:

-command|-quiet
Specifies the level of feedback displayed for commands when using the mode. This option is required when creating a feedback mode. Use `-command` to show information and verification feedback for commands. Use `-quiet` to show only essential feedback for commands, such as error messages.

-delete
Deletes the named feedback mode for this session. The name of the mode to delete is required. To permanently delete a retained mode, use the `-retain` option with this option. Predefined modes can't be deleted.

`-retain`
Saves the named feedback mode across sessions. The name of the mode to retain is required.

Configure the new feedback mode using the `/set prompt`, `/set format`, and `/set truncation` commands.
To start using the new mode, use the `/set feedback` command.

`prompt` ***`mode`*** **`"`*****`prompt-string`***`"`** **`"`*****`continuation-prompt-string`***`"`**
Sets the prompts for input within JShell. If no mode is entered, then the current prompts for all feedback modes are displayed. If only a mode is entered, then the current prompts for that mode are displayed.
To define a prompt, the following arguments are required:

`mode`
Specifies the feedback mode to which the prompts are applied. Only custom modes created with the `/set mode` command can be modified.

`"`***`prompt-string`***`"`**
Specifies the string to use as the prompt for the first line of input.

`"`***`continuation–prompt-string`***`"`**
Specifies the string to use as the prompt for the additional input lines needed to complete a snippet.

`start [-retain] [`***`file`***`**`**`[ `*****`file`***`**`**`...]|`*****`option`***`**`**`]`**
Sets the names of the startup scripts used when the next `/reset`, `/reload`, or `/env` command is entered. If more than one script is entered, then the scripts are run in the order entered. If no scripts or options are entered, then the current startup settings are displayed.
The scripts can be local files or one of the following predefined scripts:

`DEFAULT`
Loads the default entries, which are commonly used as imports.

`JAVASE`
Imports all Java SE packages.

`PRINTING`
Defines `print`, `println`, and `printf` as `jshell` methods for use within the tool.

The following options are valid:

`-default`
Sets the startup settings to the default settings.

`-none`
Specifies that no startup settings are used.

Use the `-retain` option to save the start setting across sessions.

`truncation` ***`mode length selector`***
Sets the maximum length of a displayed value. If no mode is entered, then the current truncation values for all feedback modes are displayed. If only a mode is entered, then the current truncation values for that mode are displayed.

To define truncation values, the following arguments are required:

mode
Specifies the feedback mode to which the truncation value is applied. Only custom modes created with the `/set mode` command can be modified.

length
Specifies the unsigned integer to use as the maximum length for the specified selector.

selector
Specifies the context in which the truncation value is applied. The selectors are described in the online help, which is accessed from JShell using the `/help /set truncation` command.

`/types [`*`option`*`]`
Displays classes, interfaces, and enums that were entered. If no option is entered, then all current active classes, interfaces, and enums are displayed.
The following options are valid:

***`name`*`[ `*`name...`*`]|`*`id`*`[ `*`id...`*`]`**
Displays information for classes, interfaces, and enums identified by *`name`* or *`id`*. For more than one variable, separate the names and IDs with a space. Use the `/list` command to see the IDs of the code snippets.

`-all`
Displays information for all classes, interfaces, and enums, including those added when JShell was started, and classes, interfaces, and enums that failed, were overwritten, or were dropped.

`-start`
Displays information for startup classes, interfaces, and enums that were added when JShell was started.

`/vars [`*`option`*`]`
Displays the name, type, and value of variables that were entered. If no option is entered, then all current active variables are displayed.
The following options are valid:

***`name`*`[ `*`name...`*`]|`*`id`*`[ `*`id...`*`]`**
Displays information for variables identified by *`name`* or *`id`*. For more than one variable, separate the names and IDs with a space. Use the `/list` command to see the IDs of the code snippets.

`-all`
Displays information for all variables, including those added when JShell was started, and variables that failed, were overwritten, or were dropped.

`-start`
Displays information for startup variables that were added when JShell was started.

`/?`
Same as the `/help` command.

`/!`
Reruns the last snippet.

`/id`
Reruns the snippet with the ID specified. Use the `/list` command to see the IDs of the code snippets.

`/-n`
Reruns the *-n*th previous snippet. For example, if 15 code snippets were entered, then `/-4` runs the 11th snippet. Commands aren't included in the count.

Input Shortcuts

The following table describes shortcuts that are available for entering commands and snippets in JShell.

Shortcut	Usage
Tab completion	When entering snippets, commands, subcommands, command arguments, or command options, use the Tab key to automatically complete the item. If the item can't be determined from what was entered, then possible options are provided. When entering a method call, use the Tab key after the method call's opening parenthesis to see the parameters for the method. If the method has more than one signature, then all signatures are displayed. Pressing the Tab key a second time displays the description of the method and the parameters for the first signature. Continue pressing the Tab key for a description of any additional signatures.
Command abbreviations	An abbreviations of a command is accepted if the abbreviation uniquely identifies a command. For example, `/l` is recognized as the `/list` command. However, `/s` isn't a valid abbreviation because it can't be determined if the `/set` or `/save` command is meant. Use `/se` for the `/set` command or `/sa` for the `/save` command. Abbreviations are also accepted for subcommands, command arguments, and command options. For example, use `/m -a` to display all methods.
History navigation	A history of what was entered is maintained across sessions. Use the up and down arrows to scroll through commands and snippets from the current and past sessions. Use the Ctrl key with the up and down arrows to skip all but the first line of multiline snippets.
History search	Use the Ctrl+R key combination to search the history for the string entered. The prompt changes to show the string and the match. Ctrl+R searches backwards from the current location in the history through earlier entries. Ctrl+S searches forward from the current location in the history though later entries.

Input Editing

The editing capabilities of JShell are similar to that of other common shells. Keyboard keys and key combinations provide line editing shortcuts. The Ctrl key and Meta key are used in key combinations. If your keyboard doesn't have a Meta key, then the Alt key is often mapped to provide Meta key functionality.

Key or Key Combination	Action
Return	Enter the current line.
Left arrow	Move the cursor to the left one character.
Right arrow	Move the cursor to the right one character.
Ctrl+A	Move the cursor to the beginning of the line.
Ctrl+E	Move the cursor to the end of the line.
Meta+B	Move the cursor to the left one word.
Meta+F	Move the cursor to the right one word.
Delete	Delete the character under the cursor.
Backspace	Delete the character before the cursor.
Ctrl+K	Delete the text from the cursor to the end of the line.
Meta+D	Delete the text from the cursor to the end of the word.
Ctrl+W	Delete the text from the cursor to the previous white space.
Ctrl+Y	Paste the most recently deleted text into the line.
Meta+Y	After Ctrl+Y, press to cycle through the previously deleted text.

Example of Starting and Stopping a JShell Session

JShell is provided with the JDK. To start a session, enter `jshell` on the command line. A welcome message is printed, and a prompt for entering commands and snippets is provided.

```
% jshell
|  Welcome to JShell -- Version 9
|  For an introduction type: /help intro

jshell>
```

To see which snippets were automatically loaded when JShell started, use the `/list -start` command. The default startup snippets are import statements for common packages. The ID for each snippet begins with the letter *s*, which indicates it's a startup snippet.

```
jshell> /list -start

  s1 : import java.io.*;
  s2 : import java.math.*;
  s3 : import java.net.*;
  s4 : import java.nio.file.*;
  s5 : import java.util.*;
  s6 : import java.util.concurrent.*;
  s7 : import java.util.function.*;
  s8 : import java.util.prefs.*;
  s9 : import java.util.regex.*;
 s10 : import java.util.stream.*;

jshell>
```

To end the session, use the `/exit` command.

```
jshell> /exit
|  Goodbye

%
```

Example of Entering Snippets

Snippets are Java statements, variable definitions, method definitions, class definitions, import statements, and expressions. Terminating semicolons are automatically added to the end of a completed snippet if they're missing.

The following example shows two variables and a method being defined, and the method being run. Note that a scratch variable is automatically created to hold the result because no variable was provided.

```
jshell> int a=4
a ==> 4

jshell> int b=8
b ==> 8

jshell> int square(int i1) {
   ...> return i1 * i1;
   ...> }
|  created method square(int)

jshell> square(b)
$5 ==> 64
```

Example of Changing Snippets

Change the definition of a variable, method, or class by entering it again.

The following examples shows a method being defined and the method run:

```
jshell> String grade(int testScore) {
   ...>     if (testScore >= 90) {
   ...>         return "Pass";
   ...>     }
   ...>     return "Fail";
   ...> }
|  created method grade(int)

jshell> grade(88)
$3 ==> "Fail"
```

To change the method `grade` to allow more students to pass, enter the method definition again and change the pass score to `80`. Use the up arrow key to retrieve the previous entries to avoid having to reenter them and make the change in the `if` statement. The following example shows the new definition and reruns the method to show the new result:

```
jshell> String grade(int testScore) {
   ...>     if (testScore >= 80) {
   ...>         return "Pass";
   ...>     }
   ...>     return "Fail";
   ...> }
|  modified method grade(int)
```

```
jshell> grade(88)
$5 ==> "Pass"
```

For snippets that are more than a few lines long, or to make more than a few changes, use the `/edit` command to open the snippet in an editor. After the changes are complete, close the edit window to return control to the JShell session. The following example shows the command and the feedback provided when the edit window is closed. The `/list` command is used to show that the pass score was changed to `85`.

```
jshell> /edit grade
|  modified method grade(int)
jshell> /list grade

   6 : String grade(int testScore) {
           if (testScore >= 85) {
               return "Pass";
           }
           return "Fail";
       }
```

Example of Creating a Custom Feedback Mode

The feedback mode determines the prompt that's displayed, the feedback messages that are provided as snippets are entered, and the maximum length of a displayed value. Predefined feedback modes are provided. Commands for creating custom feedback modes are also provided.

Use the `/set mode` command to create a new feedback mode. In the following example, the new mode `mymode`, is based on the predefined feedback mode, `normal`, and verifying command feedback is displayed:

```
jshell> /set mode mymode normal -command
|  Created new feedback mode: mymode
```

Because the new mode is based on the `normal` mode, the prompts are the same. The following example shows how to see what prompts are used and then changes the prompts to custom strings. The first string represents the standard JShell prompt. The second string represents the prompt for additional lines in multiline snippets.

```
jshell> /set prompt mymode
|  /set prompt mymode "\njshell> " "   ...> "

jshell> /set prompt mymode "\nprompt$ " "   continue$ "
```

The maximum length of a displayed value is controlled by the truncation setting. Different types of values can have different lengths. The following example sets an overall truncation value of 72, and a truncation value of 500 for variable value expressions:

```
jshell> /set truncation mymode 72

jshell> /set truncation mymode 500 varvalue
```

The feedback displayed after snippets are entered is controlled by the format setting and is based on the type of snippet entered and the action taken for that snippet. In the predefined mode `normal`, the string `created` is displayed when a method is created. The following example shows how to change that string to `defined`:

```
jshell> /set format mymode action "defined" added-primary
```

Use the `/set feedback` command to start using the feedback mode that was just created. The following example shows the custom mode in use:

```
jshell> /set feedback mymode
|  Feedback mode: mymode

prompt$ int square (int num1){
   continue$ return num1*num1;
   continue$ }
|  defined method square(int)

prompt$
```

4
Security Tools and Commands

You use specific JDK security tools and commands to set security policies on your local system and create applications that can work within the scope of the security policies set at remote sites.

The following sections describe the security tools and commands used to set security policies and to create applications:

- keytool: You use the `keytool` command and options to manage a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates.
- jarsigner: You use the `jarsigner` tool to sign and verify Java Archive (JAR) files.
- policytool: You use `policytool` to read and write a plain text policy file based on user input through the utility GUI.

> **Note:**
>
> The `policytool` tool has been deprecated in JDK 9 and might be removed in the next major JDK release.

The following sections describe the Kerberos security tools and commands for Windows systems:

- kinit: You use the `kinit` tool and its options to obtain and cache Kerberos ticket-granting tickets.
- klist: You use the `klist` tool to display the entries in the local credentials cache and key table.
- ktab: You use the `ktab` tool to manage the principal names and service keys stored in a local key table.

keytool

You use the `keytool` command and options to manage a keystore (database) of cryptographic keys, X.509 certificate chains, and trusted certificates.

Synopsis

`keytool [commands]`

commands

See Commands. These commands are categorized by task as follows:

- Create or Add Data to the Keystore: `-gencert`, `-genkeypair`, `-genseckey`, `-importcert`, `-importpass`
- Import Contents From Another Keystore: `-importkeystore`

- Generate Certificate Request: `-certreq`
- Export Data: `-exportcert`
- Display Data: `-list`, `-printcert`, `-printcertreq`, `-printcrl`
- Manage the Keystore: `-storepasswd`, `-keypasswd`, `-delete`, `-changealias`
- Get Help: `-help`

Description

The `keytool` command is a key and certificate management utility. It enables users to administer their own public/private key pairs and associated certificates for use in self-authentication (where the user authenticates himself or herself to other users and services) or data integrity and authentication services, using digital signatures. The `keytool` command also enables users to cache the public keys (in the form of certificates) of their communicating peers.

A certificate is a digitally signed statement from one entity (person, company, and so on.), that says that the public key (and some other information) of some other entity has a particular value. When data is digitally signed, the signature can be verified to check the data integrity and authenticity. Integrity means that the data hasn't been modified or tampered with, and authenticity means the data comes from whoever claims to have created and signed it.

The `keytool` command also enables users to administer secret keys and passphrases used in symmetric encryption and decryption (DES).

The `keytool` command stores the keys and certificates in a keystore.

Command and Option Notes

See Commands for a listing and description of the various commands.

- All command and option names are preceded by a hyphen sign (-).
- The options for each command can be provided in any order.
- All items not italicized or in braces or brackets are required to appear as is.
- Braces surrounding an option signify that a default value will be used when the option isn't specified on the command line. Braces are also used around the `-v`, `-rfc`, and `-J` options, which only have meaning when they appear on the command line. They don't have any default values other than not existing.
- Brackets surrounding an option signify that the user is prompted for the values when the option isn't specified on the command line. For the `-keypass` option, if you don't specify the option on the command line, then the `keytool` command first attempts to use the keystore password to recover the private/secret key. If this attempt fails, then the `keytool` command prompts you for the private/secret key password.
- Items in italics (option values) represent the actual values that must be supplied. For example, here is the format of the `-printcert` command:

  ```
  keytool -printcert {-file cert_file} {-v}
  ```

  When you specify a `-printcert` command, replace *`cert_file`* with the actual file name, as follows: `keytool -printcert -file VScert.cer`
- Option values must be put in quotation marks when they contain a blank (space).

- The `-help` option is the default. The `keytool` command is the same as `keytool -help`.

Commands

`-gencert`

`{-rfc} {-infile` *`infile`*`} {-outfile` *`outfile`*`} {-alias` *`alias`*`} {-sigalg` *`sigalg`*`}`

`{-dname` *`dname`*`} {-startdate` *`startdate`* `{-ext` *`ext`*`}* {-validity` *`valDays`*`}`

`[-keypass` *`keypass`*`] {-keystore` *`keystore`*`} [-storepass` *`storepass`*`]`

`{-storetype` *`storetype`*`} {-providername` *`provider_name`*`}`

`{-providerClass` *`provider_class_name`* `{-providerArg` *`provider_arg`*`}}`

`{-v} {-protected} {-Jjavaoption}`

Generates a certificate as a response to a certificate request file (which can be created by the `keytool -certreq` command). The command reads the request from*`infile`* (if omitted, from the standard input), signs it using alias's private key, and outputs the X.509 certificate into *outfile* (if omitted, to the standard output). When`-rfc` is specified, the output format is Base64-encoded PEM; otherwise, a binary DER is created.
The *`sigalg`* value specifies the algorithm that should be used to sign the certificate.
The `startdate` argument is the start time and date that the certificate is valid. The `valDays` argument tells the number of days for which the certificate should be considered valid.
When *`dname`* is provided, it is used as the subject of the generated certificate. Otherwise, the one from the certificate request is used.
The *`ext`* value shows what X.509 extensions will be embedded in the certificate. Read Common Options for the grammar of `-ext`.
The *`-gencert`* option enables you to create certificate chains. The following example creates a certificate, `e1`, that contains three certificates in its certificate chain.
The following commands creates four key pairs named `ca`, `ca1`, `ca2`, and `e1`:

```
keytool -alias ca -dname CN=CA -genkeypair
keytool -alias ca1 -dname CN=CA -genkeypair
keytool -alias ca2 -dname CN=CA -genkeypair
keytool -alias e1 -dname CN=E1 -genkeypair
```

The following two commands create a chain of signed certificates; `ca` signs `ca1` and `ca1` signs `ca2`, all of which are self-issued:

```
keytool -alias ca1 -certreq |
    keytool -alias ca -gencert -ext san=dns:ca1 |
    keytool -alias ca1 -importcert

keytool -alias ca2 -certreq |
    keytool -alias ca1 -gencert -ext san=dns:ca2 |
    keytool -alias ca2 -importcert
```

The following command creates the certificate `e1` and stores it in the file `e1.cert`, which is signed by `ca2`. As a result, `e1` should contain `ca`, `ca1`, and `ca2` in its certificate chain:

```
keytool -alias e1 -certreq | keytool -alias ca2 -gencert > e1.cert
```

-genkeypair

```
{-alias alias} {-keyalg keyalg} {-keysize keysize} {-sigalg sigalg}

[-dname dname] [-keypass keypass] {-startdate value} {-ext ext}*

{-validity valDays} {-storetype storetype} {-keystore keystore}

[-storepass storepass]

{-providerClass provider_class_name {-providerArg provider_arg}}

{-v} {-protected} {-Jjavaoption}
```

Generates a key pair (a public key and associated private key). Wraps the public key into an X.509 v3 self-signed certificate, which is stored as a single-element certificate chain. This certificate chain and the private key are stored in a new keystore entry identified by alias.
The `keyalg` value specifies the algorithm to be used to generate the key pair, and the `keysize` value specifies the size of each key to be generated. The `sigalg` value specifies the algorithm that should be used to sign the self-signed certificate. This algorithm must be compatible with the `keyalg` value.
The `dname` value specifies the X.500 Distinguished Name to be associated with the value of `alias`, and is used as the issuer and subject fields in the self-signed certificate. If no distinguished name is provided at the command line, then the user is prompted for one.
The value of `keypass` is a password used to protect the private key of the generated key pair. If no password is provided, then the user is prompted for it. If you press the **Return** key at the prompt, then the key password is set to the same password as the keystore password. The `keypass` value must be at least 6 characters.
The value of `startdate` specifies the issue time of the certificate, also known as the "Not Before" value of the X.509 certificate's Validity field.
The option value can be set in one of these two forms:
`([+-]nnn[ymdHMS])+`
`[yyyy/mm/dd] [HH:MM:SS]`
With the first form, the issue time is shifted by the specified value from the current time. The value is a concatenation of a sequence of subvalues. Inside each subvalue, the plus sign (+) means shift forward, and the minus sign (-) means shift backward. The time to be shifted is `nnn` units of years, months, days, hours, minutes, or seconds (denoted by a single character of `y`, `m`, `d`, `H`, `M`, or `S` respectively). The exact value of the issue time is calculated using the `java.util.GregorianCalendar.add(int field, int amount)` method on each subvalue, from left to right. For example, the issue time can be specified by:

```
Calendar c = new GregorianCalendar();
c.add(Calendar.YEAR, -1);
c.add(Calendar.MONTH, 1);
c.add(Calendar.DATE, -1);
return c.getTime()
```

With the second form, the user sets the exact issue time in two parts, year/month/day and hour:minute:second (using the local time zone). The user can provide only one part, which means the other part is the same as the current date (or time). The user must provide the exact number of digits as shown in the format definition (padding with 0 when shorter). When both the date and time are provided, there is one (and

only one) space character between the two parts. The hour should always be provided in 24 hour format.
When the option isn't provided, the start date is the current time. The option can be provided at most once.
The value of `valDays` specifies the number of days (starting at the date specified by `-startdate`, or the current date when `-startdate` isn't specified) for which the certificate should be considered valid.

`-genseckey`

`{-alias alias} {-keyalg keyalg} {-keysize keysize} [-keypass keypass]`

`{-storetype storetype} {-keystore keystore} [-storepass storepass]`

`{-providerClass provider_class_name {-providerArg provider_arg}} {-v}`

`{-protected} {-Jjavaoption}`

Generates a secret key and stores it in a new `KeyStore.SecretKeyEntry` identified by `alias`.
The value of `keyalg` specifies the algorithm to be used to generate the secret key, and the value of `keysize` specifies the size of the key to be generated. The `keypass` value is a password that protects the secret key. If no password is provided, then the user is prompted for it. If you press the **Return** key at the prompt, then the key password is set to the same password that is used for the `keystore`. The `keypass` value must be at least 6 characters.

`-importcert`

`{-alias alias} {-file cert_file} [-keypass keypass] {-noprompt} {-trustcacerts}`

`{-storetype storetype} {-keystore keystore} {-cacerts cacerts}[-storepass storepass]`

`{-providerName provider_name}`

`{-providerClass provider_class_name {-providerArg provider_arg}}`

`{-v} {-protected} {-Jjavaoption}`

Reads the certificate or certificate chain (where the latter is supplied in a PKCS#7 formatted reply or a sequence of X.509 certificates) from the file `cert_file`, and stores it in the `keystore` entry identified by `alias`. If no file is specified, then the certificate or certificate chain is read from `stdin`.
The `keytool` command can import X.509 v1, v2, and v3 certificates, and PKCS#7 formatted certificate chains consisting of certificates of that type. The data to be imported must be provided either in binary encoding format or in printable encoding format (also known as Base64 encoding) as defined by the Internet RFC 1421 standard. In the latter case, the encoding must be bounded at the beginning by a string that starts with `-----BEGIN`, and bounded at the end by a string that starts with `-----END`.
You import a certificate for two reasons: To add it to the list of trusted certificates, and to import a certificate reply received from a certificate authority (CA) as the result of submitting a Certificate Signing Request to that CA (see -certreq option in Commands).
Which type of import is intended is indicated by the value of the `-alias` option. If the alias doesn't point to a key entry, then the `keytool` command assumes you are adding a trusted certificate entry. In this case, the alias shouldn't already exist in the keystore.

If the alias does already exist, then the `keytool` command outputs an error because there is already a trusted certificate for that alias, and doesn't import the certificate. If the alias points to a key entry, then the `keytool` command assumes you are importing a certificate reply.

`-importpass`

`{-alias *alias*} [-keypass *keypass*] {-storetype *storetype*} {-keystore *keystore*}`

`[-storepass *storepass*]`

`{-providerClass *provider_class_name* {-providerArg *provider_arg*}}`

`{-v} {-protected} {-Jjavaoption}`

Imports a passphrase and stores it in a new `KeyStore.SecretKeyEntry` identified by `alias`. The passphrase may be supplied via the standard input stream; otherwise the user is prompted for it. `keypass` is a password used to protect the imported passphrase. If no password is provided, the user is prompted for it. If you press the **Return** key at the prompt, the key password is set to the same password as that used for the `keystore`. `keypass` must be at least 6 characters long.

`-importkeystore`

`-srckeystore *srckeystore* {-destkeystore *destkeystore*}`

> **Note:**
>
> This is the first line of all options.

`{-srcstoretype *srcstoretype*} {-deststoretype *deststoretype*}`

`[-srcstorepass *srcstorepass*] [-deststorepass *deststorepass*] {-srcprotected}`

`{-destprotected}`

`{-srcalias *srcalias* {-destalias *destalias*} [-srckeypass *srckeypass*]}`

`[-destkeypass *destkeypass*] {-noprompt}`

`{-srcProviderName *src_provider_name*} {-destProviderName *dest_provider_name*}`

`{-providerClass *provider_class_name* {-providerArg *provider_arg*}} {-v}`

`{-protected} {-Jjavaoption}`

Imports a single entry or all entries from a source keystore to a destination keystore.

> **Note:**
>
> When using the keytool `-importkeystore` command, if `-destkeystore` is not specified, the default keystore used is `$HOME/.keystore`.

When the `-srcalias` option is provided, the command imports the single entry identified by the alias to the destination keystore. If a destination alias isn't provided with `destalias`, then `srcalias` is used as the destination alias. If the source entry is protected by a password, then `srckeypass` is used to recover the entry. If *srckeypass* isn't provided, then the `keytool` command attempts to use `srcstorepass` to recover the entry. If `srcstorepass` is either not provided or is incorrect, then the user is prompted for a password. The destination entry is protected with `destkeypass`. If `destkeypass` isn't provided, then the destination entry is protected with the source entry password. For example, most third-party tools require `storepass` and `keypass` in a PKCS #12 keystore to be the same. In order to create a PKCS#12 keystore for these tools, always specify a `-destkeypass` to be the same as `-deststorepass`.
If the `-srcalias` option isn't provided, then all entries in the source keystore are imported into the destination keystore. Each destination entry is stored under the alias from the source entry. If the source entry is protected by a password, then `srcstorepass` is used to recover the entry. If `srcstorepass` is either not provided or is incorrect, then the user is prompted for a password. If a source keystore entry type isn't supported in the destination keystore, or if an error occurs while storing an entry into the destination keystore, then the user is prompted whether to skip the entry and continue or to quit. The destination entry is protected with the source entry password. If the destination alias already exists in the destination keystore, then the user is prompted to either overwrite the entry or to create a new entry under a different alias name.
If the `-noprompt` option is provided, then the user isn't prompted for a new destination alias. Existing entries are overwritten with the destination alias name. Entries that can't be imported are skipped and a warning is displayed.

`-printcertreq`

`{-file file}`

Prints the content of a PKCS #10 format certificate request, which can be generated by the `keytool -certreq` command. The command reads the request from file. If there is no file, then the request is read from the standard input.

`-certreq`

`{-alias alias} {-dname dname} {-sigalg sigalg} {-ext  ext}* {-file certreq_file}`

`[-keypass keypass] {-storetype storetype} {-keystore keystore}`

`[-storepass storepass] {-providerName provider_name}`

`{-providerClass provider_class_name {-providerArg provider_arg}}`

`{-v} {-protected} {-Jjavaoption}`

Generates a Certificate Signing Request (CSR) using the PKCS #10 format.
A CSR is intended to be sent to a certificate authority (CA). The CA authenticates the certificate requestor (usually off-line) and will return a certificate or certificate chain, used to replace the existing certificate chain (which initially consists of a self-signed certificate) in the keystore.
The private key associated with alias is used to create the PKCS #10 certificate request. To access the private key, the correct password must be provided. If `keypass` isn't provided at the command line and is different from the password used to protect the integrity of the keystore, then the user is prompted for it. If `dname` is provided, then

it is used as the subject in the CSR. Otherwise, the X.500 Distinguished Name associated with alias is used.
The `sigalg` value specifies the algorithm that should be used to sign the CSR.
The CSR is stored in the file certreq_file. If no file is specified, then the CSR is output to `stdout`.
Use the `importcert` command to import the response from the CA.

`-exportcert`

```
{-alias alias} {-file cert_file} {-storetype storetype} {-keystore keystore}{-
cacerts cacerts}

[-storepass storepass] {-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}}

{-rfc} {-v} {-protected} {-Jjavaoption}
```

Reads from the keystore the certificate associated with *alias* and stores it in the cert_file file. When no file is specified, the certificate is output to `stdout`.
The certificate is by default output in binary encoding. If the `-rfc` option is specified, then the output in the printable encoding format defined by the *Internet RFC 1421 Certificate Encoding Standard*.
If `alias` refers to a trusted certificate, then that certificate is output. Otherwise, `alias` refers to a key entry with an associated certificate chain. In that case, the first certificate in the chain is returned. This certificate authenticates the public key of the entity addressed by `alias`.

`-list`

```
{-alias alias} {-storetype storetype} {-keystore keystore} {-cacerts cacerts}[-
storepass storepass]

{-providerName provider_name}

{-providerClass provider_class_name {-providerArg provider_arg}}

{-v | -rfc} {-protected} {-Jjavaoption}
```

Prints to `stdout` the contents of the keystore entry identified by `alias`. If no `alias` is specified, then the contents of the entire keystore are printed.
This command by default prints the SHA-256 fingerprint of a certificate. If the `-v` option is specified, then the certificate is printed in human-readable format, with additional information such as the owner, issuer, serial number, and any extensions. If the `-rfc` option is specified, then the certificate contents are printed using the printable encoding format, as defined by the *Internet RFC 1421 Certificate Encoding Standard*. You can't specify both `-v` and `-rfc`.

`-printcert`

```
{-file cert_file | -sslserver host[:port]} {-jarfile JAR_file {-rfc} {-v}

{-Jjavaoption}
```

Reads the certificate from the file cert_file, the SSL server located at host:port, or the signed JAR file `JAR_file` (with the `-jarfile` option and prints its contents in a human-readable format. When no port is specified, the standard HTTPS port 443 is assumed. Note that `-sslserver` and -file options can't be provided at the same time. Otherwise,

an error is reported. If neither option is specified, then the certificate is read from `stdin`.
When `-rfc` is specified, the `keytool` command prints the certificate in PEM mode as defined by the *Internet RFC 1421 Certificate Encoding* standard.
If the certificate is read from a file or `stdin`, then it might be either binary encoded or in printable encoding format, as defined by the RFC 1421 Certificate Encoding standard.
If the SSL server is behind a firewall, then the `-J-Dhttps.proxyHost=proxyhost` and `-J-Dhttps.proxyPort=proxyport` options can be specified on the command line for proxy tunneling.
Note: This option can be used independently of a keystore.

`-printcrl`

`-file crl_ {-v}`

Reads the Certificate Revocation List (CRL) from the file *`crl_`*. A CRL is a list of digital certificates that were revoked by the CA that issued them. The CA generates the *`crl_`* file.
Note: This option can be used independently of a keystore.

`-storepasswd`

`[-new new_storepass] {-storetype storetype} {-keystore keystore}{-cacerts cacerts}`

`[-storepass storepass] {-providerName provider_name}`

`{-providerClass provider_class_name {-providerArg provider_arg}}`

`{-v} {-Jjavaoption}`

Changes the password used to protect the integrity of the keystore contents. The new password is `new_storepass`, which must be at least 6 characters.

`-keypasswd`

`{-alias alias} [-keypass old_keypass] [-new new_keypass] {-storetype storetype}`

`{-keystore keystore}  [-storepass storepass] {-providerName provider_name}`

`{-providerClass provider_class_name {-providerArg provider_arg}} {-v}`

`{-Jjavaoption}`

Changes the password under which the private/secret keys identified by `alias` is protected, from `old_keypass` to `new_keypass`, which must be at least 6 characters.
If the `-keypass` option isn't provided at the command line, and the key password is different from the keystore password, then the user is prompted for it.
If the `-new` option isn't provided at the command line, then the user is prompted for it

`-delete`

`[-alias alias] {-storetype storetype} {-keystore keystore} {-cacerts cacerts}[-storepass storepass]`

`{-providerName provider_name}`

`{-providerClass provider_class_name {-providerArg provider_arg}}`

`{-v} {-protected} {-Jjavaoption}`

Deletes from the keystore the entry identified by `alias`. The user is prompted for the alias, when no alias is provided at the command line.

`-changealias`

`{-alias` *`alias`*`} [-destalias` *`destalias`*`] [-keypass` *`keypass`*`] {-storetype` *`storetype`*`}`

`{-keystore` *`keystore`*`}{-cacerts` *`cacerts`*`} [-storepass` *`storepass`*`] {-providerName` *`provider_name`*`}`

`{-providerClass` *`provider_class_name`* `{-providerArg` *`provider_arg`*`}} {-v}`

`{-protected} {-Jjavaoption}`

Move an existing keystore entry from the specified `alias` to a new alias, `destalias`. If no destination alias is provided, then the command prompts for one. If the original entry is protected with an entry password, then the password can be supplied with the `-keypass` option. If no key password is provided, then the `storepass` (if provided) is attempted first. If the attempt fails, then the user is prompted for a password.

`–help`
Lists the basic commands and their options.
For more information about a specific command, enter the following, where *`command_name`* is the name of the command: `keytool -`*`command_name`*` -help`.

Common Options

The `-v` option can appear for all commands except `-help`. When the `-v` option appears, it signifies verbose mode, which means that more information is provided in the output.

There is also a `-Jjavaoption` argument that can appear for any command. When the `-Jjavaoption` appears, the specified `javaoption` string is passed directly to the Java interpreter. This option doesn't contain any spaces. It is useful for adjusting the execution environment or memory usage. For a list of possible interpreter options, type `java -h` or `java -X` at the command line.

These options can appear for all commands operating on a keystore:

`-storetype` ***`storetype`***
This qualifier specifies the type of keystore to be instantiated.

`-keystore` ***`keystore`***
The keystore location.
If the JKS `storetype` is used and a keystore file doesn't yet exist, then certain `keytool` commands can result in a new keystore file being created. For example, if `keytool -genkeypair` is called and the `-keystore` option isn't specified, the default keystore file named `.keystore` in the user's home directory is created when it doesn't already exist. Similarly, if the `-keystore ks_file` option is specified but ks_file doesn't exist, then it is created. For more information on the JKS `storetype`, see KeyStore Implementation section in KeyStore aliases.
Note that the input stream from the `-keystore` option is passed to the `KeyStore.load` method. If `NONE` is specified as the URL, then a null stream is passed to the `KeyStore.load` method. `NONE` should be specified if the keystore isn't file-based. For example, when it resides on a hardware token device.

`-cacerts` *`cacerts`*
Operates on the `cacerts` keystore . This option is equivalent to "`-keystore path_to_cacerts -storetype type_of_cacerts`". An error will be reported if the`-keystore` or `-storetype` option is used with the `-cacerts` option.

`-storepass [:env | :file ] argument`
The password that is used to protect the integrity of the keystore.
If the modifier `env` or `file` isn't specified, then the password has the `value` argument, which must be at least 6 characters long. Otherwise, the password is retrieved as follows:

- `env`: Retrieve the password from the environment variable named `argument`.
- `file`: Retrieve the password from the file named argument.

Note: All other options that require passwords, such as `-keypass`, `-srckeypass`, `-destkeypass`, `-srcstorepass`, and `-deststorepass`, accept the *`env`* and *`file`* modifiers. Remember to separate the password option and the modifier with a colon (:).
The password must be provided to all commands that access the keystore contents. For such commands, when the `-storepass` option isn't provided at the command line, the user is prompted for it.
When retrieving information from the keystore, the password is optional. If no password is specified, then the integrity of the retrieved information can't be verified and a warning is displayed.

`-providerName` *`provider_name`*
Used to identify a cryptographic service provider's name when listed in the security properties file.

`-providerClass` *`provider_class_name`*
Used to specify the name of a cryptographic service provider's master class file when the service provider isn't listed in the security properties file.

`-providerArg` *`provider_arg`*
Used with the `-providerClass` option to represent an optional string input argument for the constructor of *`provider_class_name`*.

`-protected=true|false`
This value should be specified as `true` when a password must be specified by way of a protected authentication path such as a dedicated PIN reader.Because there are two keystores involved in the `-importkeystore` command, the following two options `-srcprotected` and `-destprotected` are provided for the source keystore and the destination keystore respectively.

`-ext` *`{name{:critical} {=value}}`*
Denotes an X.509 certificate extension. The option can be used in `-genkeypair` and `-gencert` to embed extensions into the certificate generated, or in `-certreq` to show what extensions are requested in the certificate request. The option can appear multiple times. The `name` argument can be a supported extension name (see **Named Extensions** below) or an arbitrary OID number. The `value` argument, when provided, denotes the argument for the extension. When *`value`* is omitted, that means that the default value of the extension or the extension requires no argument. The `:critical` modifier, when provided, means the extension's `isCritical` attribute is `true`; otherwise, it is `false`. You can use `:c` in place of `:critical`.

Examples of Option Values

The following examples show the defaults for various option values.

```
-alias "mykey"

-keyalg
    "DSA" (when using -genkeypair)
    "DES" (when using -genseckey)

-keysize
    2048 (when using -genkeypair and -keyalg is "RSA")
    2048 (when using -genkeypair and -keyalg is "DSA")
    256 (when using -genkeypair and -keyalg is "EC")
    56 (when using -genseckey and -keyalg is "DES")
    168 (when using -genseckey and -keyalg is "DESede")

-validity 90

-keystore <the file named .keystore in the user's home directory>

-destkeystore <the file named .keystore in the user's home directory>

 -storetype <the value of the "keystore.type" property in the
    security properties file, which is returned by the static
    getDefaultType method in java.security.KeyStore>

-file
    stdin (if reading)
    stdout (if writing)

-protected false
```

In generating a certificate or a certificate request, the default signature algorithm (`-sigalg` option) is derived from the algorithm of the underlying private key to provide an appropriate level of security strength as shown:

keyalg	keysize	default sigalg
DSA	any size	SHA256withDSA
RSA	<= 3072	SHA256withRSA
	<= 7680	SHA384withRSA
	> 7680	SHA512withRSA
EC	<384	SHA256withECDSA
	<512	SHA384withECDSA
	= 512	SHA512withECDSA

> **Note:**
>
> In order to improve out of the box security, default key size and signature algorithm names are periodically updated to stronger values with each release of the JDK. If interoperability with older releases of the JDK is important, please make sure the defaults are supported by those releases, or alternatively use the `-keysize` or `-sigalg` options to override the default values at your own risk.

Supported Named Extensions

The `keytool` command supports these named extensions. The names aren't case-sensitive.

`BC` or `BasicContraints`
Values: The full form is: `ca:{true|false}[,pathlen:`*`len`*`]` or *len*, which is short for `ca:true,pathlen:`*`len`*. When *len*is omitted, you have `ca:true`.

`KU` or `KeyUsage`
Values: *`usage`*(,`usage`)*, where *usage* can be one of `digitalSignature`, `nonRepudiation` (`contentCommitment`), `keyEncipherment`, `dataEncipherment`, `keyAgreement`, `keyCertSign`, `cRLSign`, `encipherOnly`, `decipherOnly`. The *`usage`* argument can be abbreviated with the first few letters (`dig` for `digitalSignature`) or in camel-case style (`dS` for `digitalSignature` or `cRLS` for `cRLSign`), as long as no ambiguity is found. The `usage` values are case-sensitive.

`EKU` or `ExtendedKeyUsage`
Values: `usage`(,`usage`)*, where *usage* can be one of `anyExtendedKeyUsage`, `serverAuth`, `clientAuth`, `codeSigning`, `emailProtection`, `timeStamping`, `OCSPSigning`, or any OID string. The`usage` argument can be abbreviated with the first few letters or in camel-case style, as long as no ambiguity is found. The `usage` values are case-sensitive.

`SAN` or `SubjectAlternativeName`
Values: `type:value`(,`type:value`)*, where *`type`* can be `EMAIL`, `URI`, `DNS`, `IP`, or `OID`. The *`value`* argument is the string format value for the *`type`*.

`IAN` or `IssuerAlternativeName`
Values: Same as `SubjectAlternativeName`.

`SIA` or `SubjectInfoAccess`
Values: `method:location-type:location-value` (,`method:location-type:location-value`)*, where `method` can be `timeStamping`, `caRepository` or any OID. The `location-type` and `location-value` arguments can be any `type:value` supported by the `SubjectAlternativeName` extension.

`AIA` or `AuthorityInfoAccess`
Values: Same as `SubjectInfoAccess`. The `method` argument can be `ocsp`,`caIssuers`, or any OID.

When `name` is OID, the value is the hexadecimal dumped DER encoding of the `extnValue` for the extension excluding the OCTET STRING type and length bytes. Any extra character other than standard hexadecimal numbers (0-9, a-f, A-F) are ignored in the HEX string. Therefore, both 01:02:03:04 and 01020304 are accepted as identical values. When there is no value, the extension has an empty value field.

A special name `honored`, used in `-gencert` only, denotes how the extensions included in the certificate request should be honored. The value for this name is a comma separated list of `all` (all requested extensions are honored), `name{:[critical|non-critical]}` (the named extension is honored, but using a different `isCritical` attribute) and `-name` (used with `all`, denotes an exception). Requested extensions aren't honored by default.

If, besides the `-ext honored` option, another named or OID `-ext` option is provided, this extension is added to those already honored. However, if this name (or OID) also appears in the honored value, then its value and criticality overrides the one in the

request. If extension of the same type is provided multiple times through either a name or an OID, only the last one will be used.

The `subjectKeyIdentifier` extension is always created. For non-self-signed certificates, the `authorityKeyIdentifier` is created.

Note: Users should be aware that some combinations of extensions (and other certificate fields) may not conform to the Internet standard. See **Certificate Conformance Warning**..

Examples of Tasks Performed in Creating a keystore

The following examples describe the sequence actions in creating a keystore for managing public/private key pair and certificates from trusted entities.

- Example of Generating the Key Pair
- Example of Requesting a Signed Certificate from a CA
- Example of Importing a Certificate for the CA
- Example of Importing the Certificate Reply from the CA
- Example of Exporting a Certificate That Authenticates the Public Key
- Example of Importing Keystore
- Example of Generating Certificates for an SSL Server

Example of Generating the Key Pair

First, create a keystore and generate the key pair. You can use a command such as the following typed as a single line:

```
keytool -genkeypair -dname "cn=myname, ou=mygroup, o=mycompany, c=mycountry"
    -alias business -keypass password
    -keystore /working/mykeystore
    -storepass password -validity 180
```

The command creates the keystore named `mykeystore` in the working directory (assuming it doesn't already exist), and assigns it the password specified by `<new password for keystore>`. It generates a public/private key pair for the entity whose distinguished name has a common name of Mark Jones, organizational unit of Java, organization of Oracle and two-letter country code of US. It uses the default DSA key generation algorithm to create the keys; both are 2048 bits

The command uses the default SHA256withDSA signature algorithm to create a self-signed certificate that includes the public key and the distinguished name information. The certificate is valid for 180 days, and is associated with the private key in a keystore entry referred to by the alias `business`. The private key is assigned the password specified by *`new password for private key`*.

The command is significantly shorter when the option defaults are accepted. In this case, no options are required, and the defaults are used for unspecified options that have default values. You are prompted for any required values. You could have the following:

```
keytool -genkeypair
```

In this case, a keystore entry with the alias `mykey` is created, with a newly generated key pair and a certificate that is valid for 90 days. This entry is placed in the keystore named `.keystore` in your home directory. The keystore is created when it doesn't

already exist. You are prompted for the distinguished name information, the keystore password, and the private key password.

The rest of the examples assume you executed the `-genkeypair` command without options specified, and that you responded to the prompts with values equal to those specified in the first `-genkeypair` command. For example, a distinguished name of `cn=myname`, `ou=mygroup`, `o=mycompany`, `c=mycountry`).

Example of Requesting a Signed Certificate from a CA

Generating the key pair created a self-signed certificate. A certificate is more likely to be trusted by others when it is signed by a Certification Authority (CA). To get a CA signature, first generate a Certificate Signing Request (CSR), as follows:

```
keytool -certreq -file myname.csr
```

This creates a CSR for the entity identified by the default alias `mykey` and puts the request in the file named `myname.csr`. Submit this file to a CA, such as VeriSign. The CA authenticates you, the requestor (usually off-line), and returns a certificate, signed by them, authenticating your public key. In some cases, the CA returns a chain of certificates, each one authenticating the public key of the signer of the previous certificate in the chain.

Example of Importing a Certificate for the CA

You now need to replace the self-signed certificate with a certificate chain, where each certificate in the chain authenticates the public key of the signer of the previous certificate in the chain, up to a root CA.

Before you import the certificate reply from a CA, you need one or more trusted certificates in your keystore or in the `cacerts` keystore file. See -importcert in **Commands**.

- If the certificate reply is a certificate chain, then you need the top certificate of the chain. The root CA certificate that authenticates the public key of the CA.
- If the certificate reply is a single certificate, then you need a certificate for the issuing CA (the one that signed it). If that certificate isn't self-signed, then you need a certificate for its signer, and so on, up to a self-signed root CA certificate.

The `cacerts` keystore file ships with several VeriSign root CA certificates, so you probably will not need to import a VeriSign certificate as a trusted certificate in your keystore. But if you request a signed certificate from a different CA, and a certificate authenticating that CA's public key wasn't added to `cacerts`, then you must import a certificate from the CA as a trusted certificate.

A certificate from a CA is usually either self-signed or signed by another CA, in which case you need a certificate that authenticates that CA's public key. Suppose company ABC, Inc., is a CA, and you obtain a file named `ABCCA.cer` that is supposed to be a self-signed certificate from ABC, that authenticates that CA's public key. Be careful to ensure the certificate is valid before you import it as a trusted certificate. View it first with the `keytool -printcert` command or the `keytool -importcert` command without the `-noprompt` option, and make sure that the displayed certificate fingerprints match the expected ones. You can call the person who sent the certificate, and compare the fingerprints that you see with the ones that they show or that a secure public key repository shows. Only when the fingerprints are equal is it guaranteed that the certificate wasn't replaced in transit with somebody else's (for example, an attacker's)

certificate. If such an attack takes place, and you didn't check the certificate before you imported it, then you would be trusting anything the attacker has signed.

If you trust that the certificate is valid, then you can add it to your keystore with the following command:

```
keytool -importcert -alias abc -file ABCCA.cer
```

This command creates a trusted certificate entry in the keystore, with the data from the file ABCCA.cer, and assigns the alias `abc` to the entry.

Example of Importing the Certificate Reply from the CA

After you import a certificate that authenticates the public key of the CA you submitted your certificate signing request to (or there is already such a certificate in the cacerts file), you can import the certificate reply and replace your self-signed certificate with a certificate chain. This chain is the one returned by the CA in response to your request (when the CA reply is a chain), or one constructed (when the CA reply is a single certificate) using the certificate reply and trusted certificates that are already available in the keystore where you import the reply or in the `cacerts` keystore file.

For example, if you sent your certificate signing request to VeriSign, then you can import the reply with the following, which assumes the returned certificate is named `VS`*`myname`*`.cer`:

```
keytool -importcert -trustcacerts -file VSmyname.cer
```

Example of Exporting a Certificate That Authenticates the Public Key

If you used the `jarsigner` command to sign a Java Archive (JAR) file, then clients that want to use the file will want to authenticate your signature. One way the clients can authenticate you is by first importing your public key certificate into their keystore as a trusted entry.

You can export the certificate and supply it to your clients. As an example, you can copy your certificate to a file named *`myname`*`.cer` with the following command that assumes the entry has an alias of `mykey`:

```
keytool -exportcert -alias mykey -file myname.cer
```

With the certificate and the signed JAR file, a client can use the `jarsigner` command to authenticate your signature.

Example of Importing Keystore

The command `importkeystore` is used to import an entire keystore into another keystore, which means all entries from the source keystore, including keys and certificates, are all imported to the destination keystore within a single command. You can use this command to import entries from a different type of keystore. During the import, all new entries in the destination keystore will have the same alias names and protection passwords (for secret keys and private keys). If the `keytool` command can't recover the private keys or secret keys from the source keystore, then it prompts you for a password. If it detects alias duplication, then it asks you for a new alias, and you can specify a new alias or simply allow the `keytool` command to overwrite the existing one.

For example, to import entries from a typical JKS type keystore `key.jks` into a PKCS #11 type hardware-based keystore, use the command:

```
keytool -importkeystore
    -srckeystore key.jks -destkeystore NONE
    -srcstoretype JKS -deststoretype PKCS11
    -srcstorepass password
    -deststorepass password
```

The `importkeystore` command can also be used to import a single entry from a source keystore to a destination keystore. In this case, besides the options you see in the previous example, you need to specify the alias you want to import. With the `-srcalias` option specified, you can also specify the destination alias name in the command line, as well as protection password for a secret or private key and the destination protection password you want. The following command demonstrates this:

```
keytool -importkeystore
    -srckeystore key.jks -destkeystore NONE
    -srcstoretype JKS -deststoretype PKCS11
    -srcstorepass password
    -deststorepass password
    -srcalias myprivatekey -destalias myoldprivatekey
    -srckeypass password
    -destkeypass password
    -noprompt
```

Example of Generating Certificates for an SSL Server

The following are `keytool` commands to generate key pairs and certificates for three entities: Root CA (`root`), Intermediate CA (`ca`), and SSL server (`server`). Ensure that you store all the certificates in the same keystore. In these examples, RSA is the recommended the key algorithm.

```
keytool -genkeypair -keystore root.jks -alias root -ext bc:c
keytool -genkeypair -keystore ca.jks -alias ca -ext bc:c
keytool -genkeypair -keystore server.jks -alias server

keytool -keystore root.jks -alias root -exportcert -rfc > root.pem

keytool -storepass password -keystore ca.jks -certreq -alias ca |
    keytool -storepass password -keystore root.jks
    -gencert -alias root -ext BC=0 -rfc > ca.pem
keytool -keystore ca.jks -importcert -alias ca -file ca.pem

keytool -storepass password -keystore server.jks -certreq -alias server |
    keytool -storepass password -keystore ca.jks -gencert -alias ca
    -ext ku:c=dig,kE -rfc > server.pem
cat root.pem ca.pem server.pem |
    keytool -keystore server.jks -importcert -alias server
```

Terms

Keystore
A keystore is a storage facility for cryptographic keys and certificates.

Keystore entries
Keystores can have different types of entries. The two most applicable entry types for the `keytool` command include the following:
Key entries: Each entry holds very sensitive cryptographic key information, which is stored in a protected format to prevent unauthorized access. Typically, a key stored in this type of entry is a secret key, or a private key accompanied by the certificate chain for the corresponding public key. See Certificate Chains. The `keytool` command can

handle both types of entries, while the `jarsigner` tool only handles the latter type of entry, that is private keys and their associated certificate chains.
Trusted certificate entries: Each entry contains a single public key certificate that belongs to another party. The entry is called a trusted certificate because the keystore owner trusts that the public key in the certificate belongs to the identity identified by the subject (owner) of the certificate. The issuer of the certificate vouches for this, by signing the certificate.

Keystore aliases
All keystore entries (key and trusted certificate entries) are accessed by way of unique aliases.
An alias is specified when you add an entity to the keystore with the `-genseckey` command to generate a secret key, the `-genkeypair` command to generate a key pair (public and private key), or the `-importcert` command to add a certificate or certificate chain to the list of trusted certificates. Subsequent `keytool` commands must use this same alias to refer to the entity.
For example, you can use the alias `duke` to generate a new public/private key pair and wrap the public key into a self-signed certificate with the following command. See **Certificate Chains**.

```
keytool -genkeypair -alias duke -keypass dukekeypasswd
```

This example specifies an initial password of `dukekeypasswd` required by subsequent commands to access the private key associated with the alias `duke`. If you later want to change Duke's private key password, use a command such as the following:

```
keytool -keypasswd -alias duke -keypass dukekeypasswd -new newpass
```

This changes the password from `dukekeypasswd` to `newpass`. A password shouldn't be specified on a command line or in a script unless it is for testing purposes, or you are on a secure system. If you don't specify a required password option on a command line, then you are prompted for it.

Keystore implementation
The `KeyStore` class provided in the `java.security` package supplies well-defined interfaces to access and modify the information in a keystore. It is possible for there to be multiple different concrete implementations, where each implementation is that for a particular type of keystore.
Currently, two command-line tools (`keytool` and `jarsigner`) and a GUI-based tool named Policy Tool make use of keystore implementations. Because the `KeyStore` class is `public`, users can write additional security applications that use it.
As of JDK 9, the default keystore implementation is `PKCS12`. This is a cross platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This standard is primarily meant for storing or transporting a user's private keys, certificates, and miscellaneous secrets. There is another built-in implementation, provided by Oracle. It implements the keystore as a file with a proprietary keystore type (format) named `JKS`. It protects each private key with its individual password, and also protects the integrity of the entire keystore with a (possibly different) password.
Keystore implementations are provider-based. More specifically, the application interfaces supplied by `KeyStore` are implemented in terms of a Service Provider Interface (SPI). That is, there is a corresponding abstract `KeystoreSpi` class, also in the `java.security package`, which defines the Service Provider Interface methods that providers must implement. The term *provider* refers to a package or a set of packages that supply a concrete implementation of a subset of services that can be accessed

by the Java Security API. To provide a keystore implementation, clients must implement a provider and supply a `KeystoreSpi` subclass implementation, as described in Steps to Implement and Integrate a Provider.
Applications can choose different types of keystore implementations from different providers, using the `getInstance` factory method supplied in the `KeyStore` class. A keystore type defines the storage and data format of the keystore information, and the algorithms used to protect private/secret keys in the keystore and the integrity of the keystore. Keystore implementations of different types aren't compatible.
The `keytool` command works on any file-based keystore implementation. It treats the keystore location that is passed to it at the command line as a file name and converts it to a `FileInputStream`, from which it loads the keystore information.)The `jarsigner` and `policytool` commands can read a keystore from any location that can be specified with a URL.
For `keytool` and `jarsigner`, you can specify a keystore type at the command line, with the `-storetype` option. For Policy Tool, you can specify a keystore type with the **Keystore** menu.
If you don't explicitly specify a keystore type, then the tools choose a keystore implementation based on the value of the `keystore.type` property specified in the security properties file. The security properties file is called `java.security`, and resides in the security properties directory:

- **Oracle Solaris, Linux, and OS X:**: *`java.home`*`/lib/security`
- **Windows**: `java.home\lib\security`

Each tool gets the `keystore.type` value and then examines all the currently installed providers until it finds one that implements a keystores of that type. It then uses the keystore implementation from that provider.The `KeyStore` class defines a static method named `getDefaultType` that lets applications and applets retrieve the value of the `keystore.type` property. The following line of code creates an instance of the default keystore type as specified in the `keystore.type` property:

```
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
```

The default keystore type is `pkcs12`, which is a cross-platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This is specified by the following line in the security properties file:

```
keystore.type=pkcs12
```

To have the tools utilize a keystore implementation other than the default, you can change that line to specify a different keystore type. For example, if you want to use the Oracle's `jks` keystore implementation, then change the line to the following:

```
keystore.type=jks
```

Note: Case doesn't matter in keystore type designations. For example, `JKS` would be considered the same as jks.

Certificate
A certificate (or public-key certificate) is a digitally signed statement from one entity (the issuer), saying that the public key and some other information of another entity (the subject) has some specific value. The following terms are related to certificates:
Public Keys: These are numbers associated with a particular entity, and are intended to be known to everyone who needs to have trusted interactions with that entity. Public keys are used to verify signatures.

Digitally Signed: If some data is digitally signed, then it is stored with the identity of an entity and a signature that proves that entity knows about the data. The data is rendered unforgeable by signing with the entity's private key.
Identity: A known way of addressing an entity. In some systems, the identity is the public key, and in others it can be anything from an Oracle Solaris UID to an email address to an X.509 distinguished name.
Signature: A signature is computed over some data using the private key of an entity. The signer, which in the case of a certificate is also known as the issuer.
Private Keys: These are numbers, each of which is supposed to be known only to the particular entity whose private key it is (that is, it is supposed to be kept secret). Private and public keys exist in pairs in all public key cryptography systems (also referred to as public key crypto systems). In a typical public key crypto system, such as DSA, a private key corresponds to exactly one public key. Private keys are used to compute signatures.
Entity: An entity is a person, organization, program, computer, business, bank, or something else you are trusting to some degree.
Public key cryptography requires access to users' public keys. In a large-scale networked environment, it is impossible to guarantee that prior relationships between communicating entities were established or that a trusted repository exists with all used public keys. Certificates were invented as a solution to this public key distribution problem. Now a Certification Authority (CA) can act as a trusted third party. CAs are entities such as businesses that are trusted to sign (issue) certificates for other entities. It is assumed that CAs only create valid and reliable certificates because they are bound by legal agreements. There are many public Certification Authorities, such as VeriSign, Thawte, Entrust, and so on.
You can also run your own Certification Authority using products such as Microsoft Certificate Server or the Entrust CA product for your organization. With the `keytool` command, it is possible to display, import, and export certificates. It is also possible to generate self-signed certificates.
The `keytool` command currently handles X.509 certificates.

X.509 Certificates
The X.509 standard defines what information can go into a certificate and describes how to write it down (the data format). All the data in a certificate is encoded with two related standards called ASN.1/DER. Abstract Syntax Notation 1 describes data. The Definite Encoding Rules describe a single way to store and transfer that data.
All X.509 certificates have the following data, in addition to the signature:
Version: This identifies which version of the X.509 standard applies to this certificate, which affects what information can be specified in it. Thus far, three versions are defined. The `keytool` command can import and export v1, v2, and v3 certificates. It generates v3 certificates.
X.509 Version 1 has been available since 1988, is widely deployed, and is the most generic.
X.509 Version 2 introduced the concept of subject and issuer unique identifiers to handle the possibility of reuse of subject or issuer names over time. Most certificate profile documents strongly recommend that names not be reused and that certificates shouldn’t make use of unique identifiers. Version 2 certificates aren’t widely used.
X.509 Version 3 is the most recent (1996) and supports the notion of extensions where anyone can define an extension and include it in the certificate. Some common extensions are: KeyUsage (limits the use of the keys to particular purposes such as `signing-only`) and AlternativeNames (allows other identities to also be associated with this public key, for example. DNS names, email addresses, IP addresses). Extensions can be marked critical to indicate that the extension should be checked and enforced or used. For example, if a certificate has the KeyUsage extension marked critical and

set to `keyCertSign`, then when this certificate is presented during SSL communication, it should be rejected because the certificate extension indicates that the associated private key should only be used for signing certificates and not for SSL use.
Serial number: The entity that created the certificate is responsible for assigning it a serial number to distinguish it from other certificates it issues. This information is used in numerous ways. For example, when a certificate is revoked its serial number is placed in a Certificate Revocation List (CRL).
Signature algorithm identifier: This identifies the algorithm used by the CA to sign the certificate.
Issuer name: The X.500 Distinguished Name of the entity that signed the certificate. See X.500 Distinguished Names. This is typically a CA. Using this certificate implies trusting the entity that signed this certificate. In some cases, such as root or top-level CA certificates, the issuer signs its own certificate.
Validity period: Each certificate is valid only for a limited amount of time. This period is described by a start date and time and an end date and time, and can be as short as a few seconds or almost as long as a century. The validity period chosen depends on a number of factors, such as the strength of the private key used to sign the certificate, or the amount one is willing to pay for a certificate. This is the expected period that entities can rely on the public value, when the associated private key has not been compromised.
Subject name: The name of the entity whose public key the certificate identifies. This name uses the X.500 standard, so it is intended to be unique across the Internet. This is the X.500 Distinguished Name (DN) of the entity. For example,

```
CN=Java Duke, OU=Java Software Division, O=Oracle Corporation, C=US
```

These refer to the subject's common name (CN), organizational unit (OU), organization (O), and country (C).
Subject public key information: This is the public key of the entity being named with an algorithm identifier that specifies which public key crypto system this key belongs to and any associated key parameters.

Certificate Chains
The `keytool` command can create and manage keystore key entries that each contain a private key and an associated certificate chain. The first certificate in the chain contains the public key that corresponds to the private key.
When keys are first generated, the chain starts off containing a single element, a self-signed certificate. See -genkeypair in **Commands**. A self-signed certificate is one for which the issuer (signer) is the same as the subject. The subject is the entity whose public key is being authenticated by the certificate. Whenever the `-genkeypair` command is called to generate a new public/private key pair, it also wraps the public key into a self-signed certificate.
Later, after a Certificate Signing Request (CSR) was generated with the `-certreq` command and sent to a Certification Authority (CA), the response from the CA is imported with `-importcert`, and the self-signed certificate is replaced by a chain of certificates. See -certreq and -importcert options in **Commands**. At the bottom of the chain is the certificate (reply) issued by the CA authenticating the subject's public key. The next certificate in the chain is one that authenticates the CA's public key.
In many cases, this is a self-signed certificate, which is a certificate from the CA authenticating its own public key, and the last certificate in the chain. In other cases, the CA might return a chain of certificates. In this case, the bottom certificate in the chain is the same (a certificate signed by the CA, authenticating the public key of the key entry), but the second certificate in the chain is a certificate signed by a different CA that authenticates the public key of the CA you sent the CSR to. The next certificate in the chain is a certificate that authenticates the second CA's key, and so

on, until a self-signed root certificate is reached. Each certificate in the chain (after the first) authenticates the public key of the signer of the previous certificate in the chain.
Many CAs only return the issued certificate, with no supporting chain, especially when there is a flat hierarchy (no intermediates CAs). In this case, the certificate chain must be established from trusted certificate information already stored in the keystore.
A different reply format (defined by the PKCS #7 standard) includes the supporting certificate chain in addition to the issued certificate. Both reply formats can be handled by the `keytool` command.
The top-level (root) CA certificate is self-signed. However, the trust into the root's public key doesn't come from the root certificate itself, but from other sources such as a newspaper. This is because anybody could generate a self-signed certificate with the distinguished name of, for example, the VeriSign root CA. The root CA public key is widely known. The only reason it is stored in a certificate is because this is the format understood by most tools, so the certificate in this case is only used as a vehicle to transport the root CA's public key. Before you add the root CA certificate to your keystore, you should view it with the `-printcert` option and compare the displayed fingerprint with the well-known fingerprint obtained from a newspaper, the root CA's Web page, and so on.

The cacerts Certificates File
A certificates file named `cacerts` resides in the security properties directory:

- **Oracle Solaris, Linux, and OS X:**: *`JAVA_HOME`* `/lib/security`
- **Windows**: `java.home\lib\security`

`java.home` is the runtime environment directory, which is the `jre` directory in the JDK or the top-level directory of the Java Runtime Environment (JRE).
The `cacerts` file represents a system-wide keystore with CA certificates. System administrators can configure and manage that file with the `keytool` command by specifying `jks` as the keystore type. The `cacerts` keystore file ships with a default set of root CA certificates. For Oracle Solaris, Linux, OS X, and Windows, you can list the default certificates with the following command:

```
keytool -list -cacerts
```

The initial password of the `cacerts` keystore file is `changeit`. System administrators should change that password and the default access permission of that file upon installing the SDK.
Note: It is important to verify your `cacerts` file. Because you trust the CAs in the `cacerts` file as entities for signing and issuing certificates to other entities, you must manage the `cacerts` file carefully. The `cacerts` file should contain only certificates of the CAs you trust. It is your responsibility to verify the trusted root CA certificates bundled in the `cacerts` file and make your own trust decisions.
To remove an untrusted CA certificate from the `cacerts` file, use the `delete` option of the `keytool` command. You can find the `cacerts` file in the JRE installation directory. Contact your system administrator if you don't have permission to edit this file

Internet RFC 1421 Certificate Encoding Standard
Certificates are often stored using the printable encoding format defined by the Internet RFC 1421 standard, instead of their binary encoding. This certificate format, also known as Base64 encoding, makes it easy to export certificates to other applications by email or through some other mechanism.
Certificates read by the `-importcert` and `-printcert` commands can be in either this format or binary encoded. The `-exportcert` command by default outputs a certificate in

binary encoding, but will instead output a certificate in the printable encoding format, when the `-rfc` option is specified.
The `-list` command by default prints the SHA-256 fingerprint of a certificate. If the `-v` option is specified, then the certificate is printed in human-readable format. If the `-rfc` option is specified, then the certificate is output in the printable encoding format.
In its printable encoding format, the encoded certificate is bounded at the beginning and end by the following text:

```
-----BEGIN CERTIFICATE-----

encoded certificate goes here.

-----END CERTIFICATE-----
```

X.500 Distinguished Names
X.500 Distinguished Names are used to identify entities, such as those that are named by the `subject` and `issuer` (signer) fields of X.509 certificates. The `keytool` command supports the following subparts:
commonName: The common name of a person such as Susan Jones.
organizationUnit: The small organization (such as department or division) name. For example, Purchasing.
localityName: The locality (city) name, for example, Palo Alto.
stateName: State or province name, for example, California.
country: Two-letter country code, for example, CH.
When you supply a distinguished name string as the value of a `-dname` option, such as for the `-genkeypair` command, the string must be in the following format:

```
CN=cName, OU=orgUnit, O=org, L=city, S=state, C=countryCode
```

All the following items represent actual values and the previous keywords are abbreviations for the following:

```
CN=commonName
OU=organizationUnit
O=organizationName
L=localityName
S=stateName
C=country
```

A sample distinguished name string is:

```
CN=Mark Smith, OU=Java, O=Oracle, L=Cupertino, S=California, C=US
```

A sample command using such a string is:

```
keytool -genkeypair -dname "CN=Mark Smith, OU=Java, O=Oracle, L=Cupertino,
S=California, C=US" -alias mark
```

Case doesn't matter for the keyword abbreviations. For example, CN, cn, and Cn are all treated the same.
Order matters; each subcomponent must appear in the designated order. However, it isn't necessary to have all the subcomponents. You can use a subset, for example:

```
CN=Smith, OU=Java, O=Oracle, C=US
```

If a distinguished name string value contains a comma, then the comma must be escaped by a backslash (\) character when you specify the string on a command line, as in:

```
cn=Jack, ou=Java\, Product Development, o=Oracle, c=US
```

It is never necessary to specify a distinguished name string on a command line. When the distinguished name is needed for a command, but not supplied on the command line, the user is prompted for each of the subcomponents. In this case, a comma doesn't need to be escaped by a backslash (\).

Warnings

Importing Trusted Certificates Warning

Important: Be sure to check a certificate very carefully before importing it as a trusted certificate.

Windows Example:

View the certificate first with the `-printcert` command or the `-importcert` command without the `-noprompt` option. Ensure that the displayed certificate fingerprints match the expected ones. For example, suppose someone sends or emails you a certificate that you put it in a file named `\tmp\cert`. Before you consider adding the certificate to your list of trusted certificates, you can execute a `-printcert` command to view its fingerprints, as follows:

```
  keytool -printcert -file \tmp\cert
    Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
    Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
    Serial Number: 59092b34
    Valid from: Thu Jun 24 18:01:13 PDT 2016 until: Wed Jun 23 17:01:13 PST 2016
    Certificate Fingerprints:

                  SHA-1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:
5E:FE
                SHA-256: 90:7B:70:0A:EA:DC:16:79:92:99:41:FF:8A:FE:EB:90:
                         17:75:E0:90:B2:24:4D:3A:2A:16:A6:E4:11:0F:67:A4
```

Oracle Solaris Example:

View the certificate first with the `-printcert` command or the `-importcert` command without the `-noprompt` option. Ensure that the displayed certificate fingerprints match the expected ones. For example, suppose someone sends or emails you a certificate that you put it in a file named `/tmp/cert`. Before you consider adding the certificate to your list of trusted certificates, you can execute a `-printcert` command to view its fingerprints, as follows:

```
  keytool -printcert -file /tmp/cert
    Owner: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
    Issuer: CN=ll, OU=ll, O=ll, L=ll, S=ll, C=ll
    Serial Number: 59092b34
    Valid from: Thu Jun 24 18:01:13 PDT 2016 until: Wed Jun 23 17:01:13 PST 2016
    Certificate Fingerprints:

                  SHA-1: 20:B6:17:FA:EF:E5:55:8A:D0:71:1F:E8:D6:9D:C0:37:13:0E:
5E:FE
                  SHA-256: 90:7B:70:0A:EA:DC:16:79:92:99:41:FF:8A:FE:EB:90:
                           17:75:E0:90:B2:24:4D:3A:2A:16:A6:E4:11:0F:67:A4
```

Then call or otherwise contact the person who sent the certificate and compare the fingerprints that you see with the ones that they show. Only when the fingerprints are equal is it guaranteed that the certificate wasn't replaced in transit with somebody else's certificate such as an attacker's certificate. If such an attack took place, and you

didn't check the certificate before you imported it, then you would be trusting anything the attacker signed, for example, a JAR file with malicious class files inside.

Note: It isn't required that you execute a `-printcert` command before importing a certificate. This is because before you add a certificate to the list of trusted certificates in the keystore, the `-importcert` command prints out the certificate information and prompts you to verify it. You can then stop the import operation. However, you can do this only when you call the `-importcert` command without the `-noprompt` option. If the `-noprompt` option is specified, then there is no interaction with the user.

Passwords Warning

Most commands that operate on a keystore require the store password. Some commands require a private/secret key password. Passwords can be specified on the command line in the `-storepass` and `-keypass` options. However, a password shouldn't be specified on a command line or in a script unless it is for testing, or you are on a secure system. When you don't specify a required password option on a command line, you are prompted for it.

Certificate Conformance Warning

Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile defined a profile on conforming X.509 certificates, which includes what values and value combinations are valid for certificate fields and extensions.

The `keytool` command doesn't enforce all of these rules so it can generate certificates that don't conform to the standard, such as self-signed certificates that would be used for internal testing purposes. Certificates that don't conform to the standard might be rejected by JRE or other applications. Users should ensure that they provide the correct options for `-dname`, `-ext`, and so on.

Import a New Trusted Certificate

Before you add the certificate to the keystore, the `keytool` command verifies it by attempting to construct a chain of trust from that certificate to a self-signed certificate (belonging to a root CA), using trusted certificates that are already available in the keystore.

If the `-trustcacerts` option was specified, then additional certificates are considered for the chain of trust, namely the certificates in a file named `cacerts`.

If the `keytool` command fails to establish a trust path from the certificate to be imported up to a self-signed certificate (either from the keystore or the `cacerts` file), then the certificate information is printed, and the user is prompted to verify it by comparing the displayed certificate fingerprints with the fingerprints obtained from some other (trusted) source of information, which might be the certificate owner. Be very careful to ensure the certificate is valid before importing it as a trusted certificate. The user then has the option of stopping the import operation. If the `-noprompt` option is specified, then there is no interaction with the user.

Import a Certificate Reply

When you import a certificate reply, the certificate reply is validated with trusted certificates from the keystore, and optionally, the certificates configured in the `cacerts` keystore file when the `-trustcacerts` option is specified. See The cacerts Certificates File.

The methods of determining whether the certificate reply is trusted are as follows:

- If the reply is a single X.509 certificate, then the `keytool` command attempts to establish a trust chain, starting at the certificate reply and ending at a self-signed certificate (belonging to a root CA). The certificate reply and the hierarchy of certificates is used to authenticate the certificate reply from the new certificate chain of aliases. If a trust chain can't be established, then the certificate reply isn't imported. In this case, the `keytool` command doesn't print the certificate and prompt the user to verify it, because it is very difficult for a user to determine the authenticity of the certificate reply.
- If the reply is a PKCS #7 formatted certificate chain or a sequence of X.509 certificates, then the chain is ordered with the user certificate first followed by zero or more CA certificates. If the chain ends with a self-signed root CA certificate and the `-trustcacerts` option was specified, the `keytool` command attempts to match it with any of the trusted certificates in the keystore or the `cacerts` keystore file. If the chain doesn't end with a self-signed root CA certificate and the `-trustcacerts` option was specified, the `keytool` command tries to find one from the trusted certificates in the keystore or the `cacerts` keystore file and add it to the end of the chain. If the certificate isn't found and the `-noprompt` option isn't specified, the information of the last certificate in the chain is printed, and the user is prompted to verify it.

If the public key in the certificate reply matches the user's public key already stored with `alias`, then the old certificate chain is replaced with the new certificate chain in the reply. The old chain can only be replaced with a valid `keypass`, and so the password used to protect the private key of the entry is supplied. If no password is provided, and the private key password is different from the keystore password, the user is prompted for it.

This command was named `-import` in earlier releases. This old name is still supported in this release. The new name, `-importcert`, is preferred going forward.

jarsigner

You use the `jarsigner` tool to sign and verify Java Archive (JAR) files.

Synopsis

```
jarsigner [ options ] jar-file alias jarsigner -verify [ options ] jar-file
[alias ...]

jarsigner -verify [ options ] jar-file [alias ...]
```

options
The command-line options. See Options for jarsigner.

`-verify`
The `-verify` option can take zero or more keystore alias names after the JAR file name. When the `-verify` option is specified, the `jarsigner` command checks that the certificate used to verify each signed entry in the JAR file matches one of the keystore aliases. The aliases are defined in the keystore specified by `-keystore` or the default keystore.
If you also specify the `-strict` option, and the `jarsigner` command detects severe warnings, the message, "jar verified, with signer errors" is displayed.

jar-file
The JAR file to be signed.

If you also specified the `-strict` option, and the `jarsigner` command detected severe warnings, the message, "jar signed, with signer errors" is displayed.

alias
The aliases are defined in the keystore specified by `-keystore` or the default keystore.

Description

The `jarsigner` tool has two purposes:

- To sign Java Archive (JAR) files.
- To verify the signatures and integrity of signed JAR files.

The JAR feature enables the packaging of class files, images, sounds, and other digital data in a single file for faster and easier distribution. A tool named `jar` enables developers to produce JAR files. (Technically, any ZIP file can also be considered a JAR file, although when created by the `jar` command or processed by the `jarsigner` command, JAR files also contain a `META-INF/MANIFEST.MF` file.)

A digital signature is a string of bits that is computed from some data (the data being signed) and the private key of an entity (a person, company, and so on). Similar to a handwritten signature, a digital signature has many useful characteristics:

- Its authenticity can be verified by a computation that uses the public key corresponding to the private key used to generate the signature.
- It can't be forged, assuming the private key is kept secret.
- It is a function of the data signed and thus can't be claimed to be the signature for other data as well.
- The signed data can't be changed. If the data is changed, then the signature can't be verified as authentic.

To generate an entity's signature for a file, the entity must first have a public/private key pair associated with it and one or more certificates that authenticate its public key. A certificate is a digitally signed statement from one entity that says that the public key of another entity has a particular value.

The `jarsigner` command uses key and certificate information from a keystore to generate digital signatures for JAR files. A keystore is a database of private keys and their associated X.509 certificate chains that authenticate the corresponding public keys. The `keytool` command is used to create and administer keystores.

The `jarsigner` command uses an entity's private key to generate a signature. The signed JAR file contains, among other things, a copy of the certificate from the keystore for the public key corresponding to the private key used to sign the file. The `jarsigner` command can verify the digital signature of the signed JAR file using the certificate inside it (in its signature block file).

The `jarsigner` command can generate signatures that include a time stamp that enables a systems or deployer (including Java Plug-in) to check whether the JAR file was signed while the signing certificate was still valid.

> **Note:**
>
> Although available and supported in JDK 9, the Java Plug-in has been marked as deprecated in preparation for removal in a future release. Alternatives for applets and embedded JavaFX applications, which require the plug-in, include Java Web Start and self-contained applications.

In addition, APIs allow applications to obtain the timestamp information.

At this time, the `jarsigner` command can only sign JAR files created by the `jar` command or zip files. JAR files are the same as zip files, except they also have a `META-INF/MANIFEST.MF` file. A `META-INF/MANIFEST.MF` file is created when the `jarsigner` command signs a zip file.

The default `jarsigner` command behavior is to sign a JAR or zip file. Use the `-verify` option to verify a signed JAR file.

The `jarsigner` command also attempts to validate the signer's certificate after signing or verifying. If there is a validation error or any other problem, the command generates warning messages. If you specify the `-strict` option, then the command treats severe warnings as errors. See Errors and Warnings.

Keystore Aliases

All keystore entities are accessed with unique aliases.

When you use the `jarsigner` command to sign a JAR file, you must specify the alias for the keystore entry that contains the private key needed to generate the signature. If no output file is specified, it overwrites the original JAR file with the signed JAR file.

Keystores are protected with a password, so the store password must be specified. You are prompted for it when you don't specify it on the command line. Similarly, private keys are protected in a keystore with a password, so the private key's password must be specified, and you are prompted for the password when you don't specify it on the command line and it isn't the same as the store password.

Keystore Location

The `jarsigner` command has a `-keystore` option for specifying the URL of the keystore to be used. The keystore is by default stored in a file named `.keystore` in the user's home directory, as determined by the `user.home` system property.

Oracle Solaris, Linux, and OS X:: `user.home` defaults to the user's home directory.

The input stream from the `-keystore` option is passed to the `KeyStore.load` method. If `NONE` is specified as the URL, then a null stream is passed to the `KeyStore.load` method. `NONE` should be specified when the `KeyStore` class isn't file based, for example, when it resides on a hardware token device.

Keystore Implementation

The `KeyStore` class provided in the `java.security` package supplies a number of well-defined interfaces to access and modify the information in a keystore. You can have multiple different concrete implementations, where each implementation is for a particular type of keystore.

Currently, there are two command-line tools that use keystore implementations (`keytool` and `jarsigner`). You can also use a GUI-based tool named policytool but it is deprecated and might be removed in a future JDK release.. Because the `KeyStore` class is publicly available, JDK users can write additional security applications that use it.

The default keystore implementation is `PKCS12`. This is a cross platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This standard is primarily meant for storing or transporting a user's private keys, certificates, and miscellaneous secrets. There is another built-in implementation, provided by Oracle. It implements the keystore as a file with a proprietary keystore type (format) named `JKS`. It protects each private key with its individual password, and also protects the integrity of the entire keystore with a (possibly different) password.

Keystore implementations are provider-based, which means the application interfaces supplied by the `KeyStore` class are implemented in terms of a Service Provider Interface (SPI). There is a corresponding abstract `KeystoreSpi` class, also in the `java.security package`, that defines the Service Provider Interface methods that providers must implement. The term provider refers to a package or a set of packages that supply a concrete implementation of a subset of services that can be accessed by the Java Security API. To provide a keystore implementation, clients must implement a provider and supply a `KeystoreSpi` subclass implementation, as described in How to Implement a Provider in the Java Cryptography Architecture.

Applications can choose different types of keystore implementations from different providers, with the `getInstance` factory method in the `KeyStore` class. A keystore type defines the storage and data format of the keystore information and the algorithms used to protect private keys in the keystore and the integrity of the keystore itself. Keystore implementations of different types aren’t compatible.

The `jarsigner` and `policytool` commands can read file-based keystores from any location that can be specified using a URL. In addition, these commands can read non-file-based keystores such as those provided by MSCAPI on Windows and PKCS11 on all platforms.

For the `jarsigner` and `keytool` commands, you can specify a keystore type at the command line with the `-storetype` option. For Policy Tool, you can specify a keystore type with the **Edit** command in the **KeyStore** menu.

If you don’t explicitly specify a keystore type, then the tools choose a keystore implementation based on the value of the `keystore.type` property specified in the security properties file. The security properties file is called `java.security`, and it resides in the JDK security properties directory, `java.home/conf/security`.

Each tool gets the `keystore.type` value and then examines all the installed providers until it finds one that implements keystores of that type. It then uses the keystore implementation from that provider.

The `KeyStore` class defines a static method named `getDefaultType` that lets applications and applets retrieve the value of the `keystore.type` property. The following line of code creates an instance of the default keystore type as specified in the `keystore.type property`:

```
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
```

The default keystore type is `pkcs12`, which is a cross platform keystore based on the RSA PKCS12 Personal Information Exchange Syntax Standard. This is specified by the following line in the security properties file:

```
keystore.type=pkcs12
```

Case doesn't matter in keystore type designations. For example, `JKS` is the same as `jks`.

To have the tools utilize a keystore implementation other than the default, you can change that line to specify a different keystore type. For example, if you want to use the Oracle's `jks` keystore implementation, then change the line to the following:

```
keystore.type=jks
```

Supported Algorithms

By default, the `jarsigner` command signs a JAR file using one of the following algorithms files depending on the type and size of the private key:

keyalg	keysize	default sigalg
DSA	any size	SHA256withDSA
RSA	<= 3072	SHA256withRSA
	<= 7680	SHA384withRSA
	> 7680	SHA512withRSA
EC	<384	SHA256withECDSA
	<512	SHA384withECDSA
	= 512	SHA512withECDSA

These default signature algorithms can be overridden by using the `-sigalg` option.

Signed JAR file algorithms are checked against the `jdk.jar.disabledAlgorithms` security property during verification (`-verify`). If the JAR file was signed with any algorithms that are disabled, it will be treated as an unsigned JAR file. For detailed verification output, include `-J-Djava.security.debug=jar`. The default value for the `jdk.jar.disabledAlgorithms` security property is defined in the `java.security` file (located in the JRE's `$JAVA_HOME/conf/security` directory).

> **Note:**
>
> In order to improve out of the box security, default key size and signature algorithm names are periodically updated to stronger values with each release of the JDK. If interoperability with older releases of the JDK is important, please make sure the defaults are supported by those releases, or alternatively use the `-sigalg` option to override the default values at your own risk.

The Signed JAR File

When the `jarsigner` command is used to sign a JAR file, the output signed JAR file is exactly the same as the input JAR file, except that it has two additional files placed in the META-INF directory:

- A signature file with an `.SF` extension
- A signature block file with a `.DSA`, `.RSA`, or `.EC` extension

The base file names for these two files come from the value of the `-sigFile` option. For example, when the option is `-sigFile MKSIGN`, the files are named `MKSIGN.SF` and `MKSIGN.DSA`

If no `-sigfile` option appears on the command line, then the base file name for the `.SF` and `.DSA` files is the first 8 characters of the alias name specified on the command line, all converted to uppercase. If the alias name has fewer than 8 characters, then the full alias name is used. If the alias name contains any characters that aren't allowed in a signature file name, then each such character is converted to an underscore (_) character in forming the file name. Valid characters include letters, digits, underscores, and hyphens.

Signature File

A signature file (`.SF` file) looks similar to the manifest file that is always included in a JAR file when the `jarsigner` command is used to sign the file. For each source file included in the JAR file, the `.SF` file has three lines, such as in the manifest file, that list the following:

- File name
- Name of the digest algorithm (SHA)
- SHA digest value

In the manifest file, the SHA digest value for each source file is the digest (hash) of the binary data in the source file. In the `.SF` file, the digest value for a specified source file is the hash of the three lines in the manifest file for the source file.

The signature file, by default, includes a header with a hash of the whole manifest file. The header also contains a hash of the manifest header. The presence of the header enables verification optimization. See JAR File Verification.

Signature Block File

The `.SF` file is signed and the signature is placed in the signature block file. This file also contains, encoded inside it, the certificate or certificate chain from the keystore that authenticates the public key corresponding to the private key used for signing. The file has the extension `.DSA`, `.RSA`, or `.EC`, depending on the digest algorithm used.

Signature Time Stamp

The `jarsigner` command used with the following options generates and stores a signature time stamp when signing a JAR file:

- `-tsa` *`url`*
- `-tsacert` *`alias`*
- `-tsapolicyid` *`policyid`*
- `-tsadigestalg` *`algorithm`*

See Options for jarsigner.

JAR File Verification

A successful JAR file verification occurs when the signatures are valid, and none of the files that were in the JAR file when the signatures were generated have changed since then. JAR file verification involves the following steps:

1. Verify the signature of the `.SF` file.

The verification ensures that the signature stored in each signature block (`.DSA`) file was generated using the private key corresponding to the public key whose certificate (or certificate chain) also appears in the `.DSA` file. It also ensures that the signature is a valid signature of the corresponding signature (`.SF`) file, and thus the `.SF` file wasn't tampered with.

2. Verify the digest listed in each entry in the `.SF` file with each corresponding section in the manifest.

   The `.SF` file by default includes a header that contains a hash of the entire manifest file. When the header is present, the verification can check to see whether or not the hash in the header matches the hash of the manifest file. If there is a match, then verification proceeds to the next step.

   If there is no match, then a less optimized verification is required to ensure that the hash in each source file information section in the `.SF` file equals the hash of its corresponding section in the manifest file. See Signature File.

   One reason the hash of the manifest file that is stored in the `.SF` file header might not equal the hash of the current manifest file is that one or more files were added to the JAR file (with the `jar` tool) after the signature and `.SF` file were generated. When the `jar` tool is used to add files, the manifest file is changed by adding sections to it for the new files, but the `.SF` file isn't changed. A verification is still considered successful when none of the files that were in the JAR file when the signature was generated have been changed since then. This happens when the hashes in the non-header sections of the `.SF` file equal the hashes of the corresponding sections in the manifest file.

3. Read each file in the JAR file that has an entry in the `.SF` file. While reading, compute the file's digest and compare the result with the digest for this file in the manifest section. The digests should be the same or verification fails.

   If any serious verification failures occur during the verification process, then the process is stopped and a security exception is thrown. The `jarsigner` command catches and displays the exception.

4. Check for disabled algorithm usage. See Supported Algorithms.

> **Note:**
>
> You should read any addition warnings (or errors if you specified the `-strict` option), as well as the content of the certificate (by specifying the `-verbose` and `-certs` options) to determine if the signature can be trusted.

Multiple Signatures for a JAR File

A JAR file can be signed by multiple people by running the `jarsigner` command on the file multiple times and specifying the alias for a different person each time, as follows:

```
jarsigner myBundle.jar susan
jarsigner myBundle.jar kevin
```

When a JAR file is signed multiple times, there are multiple `.SF` and `.DSA` files in the resulting JAR file, one pair for each signature. In the previous example, the output JAR file includes files with the following names:

```
SUSAN.SF
SUSAN.DSA
KEVIN.SF
KEVIN.DSA
```

Options for jarsigner

The following sections describe the options for the `jarsigner`. Be aware of the following standards:

- All option names are preceded by a hyphen sign (-).
- The options can be provided in any order.
- Items that are in italics or underlined (option values) represent the actual values that must be supplied.
- The `-storepass`, `-keypass`, `-sigfile`, `-sigalg`, `-digestalg`, `-signedjar`, and TSA-related options are only relevant when signing a JAR file; they aren't relevant when verifying a signed JAR file. The `-keystore` option is relevant for signing and verifying a JAR file. In addition, aliases are specified when signing and verifying a JAR file.

`-keystore` *`url`*
Specifies the URL that tells the keystore location. This defaults to the file `.keystore` in the user's home directory, as determined by the `user.home` system property.
A keystore is required when signing. You must explicitly specify a keystore when the default keystore doesn't exist or if you want to use one other than the default.
A keystore isn't required when verifying, but if one is specified or the default exists and the `-verbose` option was also specified, then additional information is output regarding whether or not any of the certificates used to verify the JAR file are contained in that keystore.
The `-keystore` argument can be a file name and path specification rather than a URL, in which case it is treated the same as a file: URL, for example, the following are equivalent:

```
-keystore filePathAndName
-keystore file:filePathAndName
```

If the Sun PKCS #11 provider was configured in the `java.security` security properties file (located in the JRE's `$JAVA_HOME/conf/security` directory), then the `keytool` and `jarsigner` tools can operate on the PKCS #11 token by specifying these options:

```
-keystore NONE
-storetype PKCS11
```

For example, the following command lists the contents of the configured PKCS#11 token:

```
keytool -keystore NONE -storetype PKCS11 -list
```

`-storepass[:env | :file]` *`argument`*
Specifies the password that is required to access the keystore. This is only needed when signing (not verifying) a JAR file. In that case, if a `-storepass` option isn't provided at the command line, then the user is prompted for the password.
If the modifier `env` or `file` isn't specified, then the password has the value `argument`. Otherwise, the password is retrieved as follows:

- `env`: Retrieve the password from the environment variable named *argument*.
- `file`: Retrieve the password from the file named *argument*.

> **Note:**
>
> The password shouldn't be specified on the command line or in a script unless it is for testing purposes, or you are on a secure system.

`-storetype storetype`
Specifies the type of keystore to be instantiated. The default keystore type is the one that is specified as the value of the `keystore.type` property in the security properties file, which is returned by the static `getDefaultType` method in `java.security.KeyStore`. The PIN for a PCKS #11 token can also be specified with the `-storepass` option. If none is specified, then the `keytool` and `jarsigner` commands prompt for the token PIN. If the token has a protected authentication path (such as a dedicated PIN-pad or a biometric reader), then the `-protected` option must be specified and no password options can be specified.

`-keypass [:env | :file] argument -certchain file`
Specifies the password used to protect the private key of the keystore entry addressed by the alias specified on the command line. The password is required when using `jarsigner` to sign a JAR file. If no password is provided on the command line, and the required password is different from the store password, then the user is prompted for it.
If the modifier `env` or `file` isn't specified, then the password has the value `argument`. Otherwise, the password is retrieved as follows:

- `env`: Retrieve the password from the environment variable named `argument`.
- `file`: Retrieve the password from the file named `argument`.

> **Note:**
>
> The password shouldn't be specified on the command line or in a script unless it is for testing purposes, or you are on a secure system.

`-certchain file`
Specifies the certificate chain to be used when the certificate chain associated with the private key of the keystore entry that is addressed by the alias specified on the command line isn't complete. This can happen when the keystore is located on a hardware token where there isn't enough capacity to hold a complete certificate chain. The file can be a sequence of concatenated X.509 certificates, or a single PKCS#7 formatted data block, either in binary encoding format or in printable encoding format (also known as Base64 encoding) as defined by InternetRFC 1421 Certificate Encoding Standard.

`-sigfile file`
Specifies the base file name to be used for the generated `.SF` and `.DSA` files. For example, if file is `DUKESIGN`, then the generated `.SF` and `.DSA` files are named `DUKESIGN.SF` and `DUKESIGN.DSA`, and placed in the `META-INF` directory of the signed JAR file.

The characters in the file must come from the set `a-zA-Z0-9_-`. Only letters, numbers, underscore, and hyphen characters are allowed. All lowercase characters are converted to uppercase for the `.SF` and `.DSA` file names.
If no `-sigfile` option appears on the command line, then the base file name for the `.SF` and `.DSA` files is the first 8 characters of the alias name specified on the command line, all converted to upper case. If the alias name has fewer than 8 characters, then the full alias name is used. If the alias name contains any characters that aren't valid in a signature file name, then each such character is converted to an underscore (_) character to form the file name.

`-signedjar` *`file`*
Specifies the name of signed JAR file.

`-digestalg` *`algorithm`*
Specifies the name of the message digest algorithm to use when digesting the entries of a JAR file.
For a list of standard message digest algorithm names, see "Appendix A: Standard Names" in the *Java Cryptography Architecture (JCA) Reference Guide*.
If this option isn't specified, then `SHA256` is used. There must either be a statically installed provider supplying an implementation of the specified algorithm or the user must specify one with the `-providerClass` option; otherwise, the command will not succeed.

`-sigalg` *`algorithm`*
Specifies the name of the signature algorithm to use to sign the JAR file.
This algorithm must be compatible with the private key used to sign the JAR file. If this option isn't specified, then use a default algorithm matching the private key as described in the Supported Algorithms section. There must either be a statically installed provider supplying an implementation of the specified algorithm or you must specify one with the `-providerClass` option; otherwise, the command doesn't succeed.
For a list of standard signature algorithm names, see "Appendix A: Standard Names" in the *Java Cryptography Architecture (JCA) Reference Guide*.

`-verify`
Verifies a signed JAR file.

`-verbose [:`*`suboptions`*`]`
When the `-verbose` option appears on the command line, it indicates that the `jarsigner` use the verbose mode when signing or verifying with the suboptions determining how much information is shown.. This causes the , which causes `jarsigner` to output extra information about the progress of the JAR signing or verification. The *`suboptions`* can be `all`, `grouped`, or `summary`.
If the `-certs` option is also specified, then the default mode (or suboption `all`) displays each entry as it is being processed, and after that, the certificate information for each signer of the JAR file.
If the `-certs` and the `-verbose:grouped` suboptions are specified, then entries with the same signer info are grouped and displayed together with their certificate information.
If `-certs` and the `-verbose:summary` suboptions are specified, then entries with the same signer information are grouped and displayed together with their certificate information.
Details about each entry are summarized and displayed as *one entry (and more)*. See Example of Verifying a Signed JAR File and Example of Verification with Certificate Information.

`-certs`
If the `-certs` option appears on the command line with the `-verify` and `-verbose` options, then the output includes certificate information for each signer of the JAR file. This information includes the name of the type of certificate (stored in the `.DSA` file) that certifies the signer's public key, and if the certificate is an X.509 certificate (an instance of the `java.security.cert.X509Certificate`), then the distinguished name of the signer.
The keystore is also examined. If no keystore value is specified on the command line, then the default keystore file (if any) is checked. If the public key certificate for a signer matches an entry in the keystore, then the alias name for the keystore entry for that signer is displayed in parentheses.

`-tsa` *`url`*
If `-tsa http://example.tsa.url` appears on the command line when signing a JAR file then a time stamp is generated for the signature. The URL, `http://example.tsa.url`, identifies the location of the Time Stamping Authority (TSA) and overrides any URL found with the `-tsacert` option. The `-tsa` option doesn't require the TSA public key certificate to be present in the keystore.
To generate the time stamp, `jarsigner` communicates with the TSA with the Time-Stamp Protocol (TSP) defined in RFC 3161. When successful, the time stamp token returned by the TSA is stored with the signature in the signature block file.

`-tsacert` *`alias`*
When `-tsacert alias` appears on the command line when signing a JAR file, a time stamp is generated for the signature. The alias identifies the TSA public key certificate in the keystore that is in effect. The entry's certificate is examined for a Subject Information Access extension that contains a URL identifying the location of the TSA. The TSA public key certificate must be present in the keystore when using the `-tsacert` option.

`-tsapolicyid` *`policyid`*
Specifies the object identifier (OID) that identifies the policy ID to be sent to the TSA server. If this option isn't specified, no policy ID is sent and the TSA server will choose a default policy ID.
Object identifiers are defined by X.696, which is an ITU Telecommunication Standardization Sector (ITU-T) standard. These identifiers are typically period-separated sets of non-negative digits like `1.2.3.4`, for example.

`–tsadigestalg` *`algorithm`*
Specifies the message digest algorithm that is used to generate the message imprint to be sent to the TSA server. If this option isn't specified, SHA-256 will be used.
See Supported Algorithms. For a list of standard message digest algorithm names, see "Appendix A: Standard Names" in *Java Cryptography Architecture (JCA) Reference Guide*.

`-internalsf`
In the past, the `.DSA` (signature block) file generated when a JAR file was signed included a complete encoded copy of the `.SF` file (signature file) also generated. This behavior has been changed. To reduce the overall size of the output JAR file, the `.DSA` file by default doesn't contain a copy of the `.SF` file anymore. If `-internalsf` appears on the command line, then the old behavior is utilized. This option is useful for testing. In practice, don't use the `-internalsf` option because it incurs higher overhead.

`-sectionsonly`
If the `-sectionsonly` option appears on the command line, then the `.SF` file (signature file) generated when a JAR file is signed doesn't include a header that contains a hash of the whole manifest file. It contains only the information and hashes related to each individual source file included in the JAR file. See Signature File.
By default, this header is added, as an optimization. When the header is present, whenever the JAR file is verified, the verification can first check to see whether the hash in the header matches the hash of the whole manifest file. When there is a match, verification proceeds to the next step. When there is no match, it is necessary to do a less optimized verification that the hash in each source file information section in the `.SF` file equals the hash of its corresponding section in the manifest file. See JAR File Verification.
The `-sectionsonly` option is primarily used for testing. It shouldn't be used other than for testing because using it incurs higher overhead.

`-protected`
Values can be either `true` or `false`. Specify `true` when a password must be specified through a protected authentication path such as a dedicated PIN reader.

`-providerName` *`providerName`*
If more than one provider was configured in the `java.security` security properties file, then you can use the `-providerName` option to target a specific provider instance. The argument to this option is the name of the provider.
For the Oracle PKCS #11 provider, *`providerName`* is of the form `SunPKCS11-TokenName`, where `TokenName` is the name suffix that the provider instance has been configured with, as detailed in the configuration attributes table. For example, the following command lists the contents of the `PKCS #11` keystore provider instance with name suffix `SmartCard`:

```
jarsigner -keystore NONE -storetype PKCS11
        -providerName SunPKCS11-SmartCard
        -list
```

`-addprovider` *`name`*`[-providerArg` *`arg`*`]`
Adds a security provider by name (such as SunPKCS11) and an optional configure argument for `-addprovider`.

`-providerClass` *`provider-class-name`*`[-providerArg` *`arg`*`]`
Used to specify the name of cryptographic service provider's master class file when the service provider isn't listed in the `java.security` security properties file. Adds a security provider by fully-qualified class name and an optional configure argument for the `-providerClass`.
Used with the `-providerArg ConfigFilePath` option, the `keytool` and `jarsigner` tools install the provider dynamically and use `ConfigFilePath` for the path to the token configuration file. The following example shows a command to list a `PKCS #11` keystore when the Oracle PKCS #11 provider wasn't configured in the security properties file.

```
jarsigner -keystore NONE -storetype PKCS11
          -providerClass sun.security.pkcs11.SunPKCS11
          -providerArg /mydir1/mydir2/token.config
          -list
```

`-Jjavaoption`
Passes through the specified `javaoption` string directly to the Java interpreter. The `jarsigner` command is a wrapper around the interpreter. This option shouldn't contain

any spaces. It is useful for adjusting the execution environment or memory usage. For a list of possible interpreter options, type `java -h` or `java -X` at the command line.

`-strict`
During the signing or verifying process, the command may issue warning messages. If you specify this option, the exit code of the tool reflects the severe warning messages that this command found. See Errors and Warnings.

`-conf` *`url`*
Specifies a pre-configured options file.

Deprecated Options

The following `jarsigner` options are deprecated as of JDK 9 and might be removed in a future JDK release.

`-altsigner` *`class`*
This option specifies an alternative signing mechanism. The fully qualified class name identifies a class file that extends the `com.sun.jarsigner.ContentSigner` abstract class. The path to this class file is defined by the `-altsignerpath` option. If the `-altsigner` option is used, then the `jarsigner` command uses the signing mechanism provided by the specified class. Otherwise, the `jarsigner` command uses its default signing mechanism.
For example, to use the signing mechanism provided by a class named `com.sun.sun.jarsigner.AuthSigner`, use the jarsigner option `-altsigner com.sun.jarsigner.AuthSigner`.

`-altsignerpath` *`classpathlist`*
Specifies the path to the class file and any JAR file it depends on. The class file name is specified with the `-altsigner` option. If the class file is in a JAR file, then this option specifies the path to that JAR file.
An absolute path or a path relative to the current directory can be specified. If *`classpathlist`* contains multiple paths or JAR files, then they should be separated with a:

- Colon (:) on Oracle Solaris, Linux, and macOS
- Semicolon (;) on Windows

This option isn't necessary when the class is already in the search path.
The following example shows how to specify the path to a JAR file that contains the class file. The JAR file name is included.

```
-altsignerpath /home/user/lib/authsigner.jar
```

The following example shows how to specify the path to the JAR file that contains the class file. The JAR file name is omitted.

```
-altsignerpath /home/user/classes/com/sun/tools/jarsigner/
```

Errors and Warnings

During the signing or verifying process, the `jarsigner` command may issue various errors or warnings.

If there is a failure, the `jarsigner` command exits with code 1. If there is no failure, but there are one or more severe warnings, the `jarsigner` command exits with code 0 when the `-strict` option is **not** specified, or exits with the OR-value of the warning

codes when the `-strict` is specified. If there is only informational warnings or no warning at all, the command always exits with code 0.

For example, if a certificate used to sign an entry is expired and has a KeyUsage extension that doesn't allow it to sign a file, the `jarsigner` command exits with code 12 (=4+8) when the `-strict` option is specified.

Note: Exit codes are reused because only the values from 0 to 255 are legal on Oracle Solaris, Linux, and OS X.

The following sections describes the names, codes, and descriptions of the errors and warnings that the `jarsigner` command can issue.

Failure

Reasons why the `jarsigner` command fails include (but aren't limited to) a command line parsing error, the inability to find a keypair to sign the JAR file, or the verification of a signed JAR fails.

failure
Code 1. The signing or verifying fails.

Severe Warnings

> **Note:**
>
> Severe warnings are reported as errors if you specify the `-strict` option.

Reasons why the `jarsigner` command issues a severe warning include the certificate used to sign the JAR file has an error or the signed JAR file has other problems.

hasExpiredCert
Code 4. This JAR contains entries whose signer certificate has expired.

notYetValidCert
Code 4. This JAR contains entries whose signer certificate isn't yet valid.

chainNotValidated
Code 4. This JAR contains entries whose certificate chain isn't validated.

signerSelfSigned
Code 4. This JAR contains entries whose signer certificate is self signed.

weakAlg
Code 4. An algorithm specified on the command line is considered a security risk.

badKeyUsage
Code 8. This JAR contains entries whose signer certificate's KeyUsage extension doesn't allow code signing.

badExtendedKeyUsage
Code 8. This JAR contains entries whose signer certificate's ExtendedKeyUsage extension doesn't allow code signing.

badNetscapeCertType
Code 8. This JAR contains entries whose signer certificate's NetscapeCertType extension doesn't allow code signing.

hasUnsignedEntry
Code 16. This JAR contains unsigned entries which haven't been integrity-checked.

notSignedByAlias
Code 32. This JAR contains signed entries which aren't signed by the specified alias(es).

aliasNotInStore
Code 32. This JAR contains signed entries that aren't signed by alias in this keystore.

Informational Warnings

Informational warnings include those that aren't errors but regarded as bad practice. They don't have a code.

hasExpiringCert
This JAR contains entries whose signer certificate expires within six months.

noTimestamp
This JAR contains signatures that doesn't include a timestamp. Without a timestamp, users may not be able to validate this JAR file after the signer certificate's expiration date (`YYYY-MM-DD`) or after any future revocation date.

Example of Signing a JAR File

Use the following command to sign `bundle.jar` with the private key of a user whose keystore alias is `jane` in a keystore named `mystore` in the `working` directory and name the signed JAR file `sbundle.jar`:

```
jarsigner -keystore /working/mystore
    -storepass <keystore password>
    -keypass <private key password>
    -signedjar sbundle.jar bundle.jar jane
```

There is no `-sigfile` specified in the previous command so the generated `.SF` and `.DSA` files to be placed in the signed JAR file have default names based on the alias name. They are named `JANE.SF` and `JANE.DSA`.

If you want to be prompted for the store password and the private key password, then you could shorten the previous command to the following:

```
jarsigner -keystore /working/mystore
    -signedjar sbundle.jar bundle.jar jane
```

If the `keystore` is the default `keystore` (`.keystore` in your home directory), then you don't need to specify a `keystore`, as follows:

```
jarsigner -signedjar sbundle.jar bundle.jar jane
```

If you want the signed JAR file to overwrite the input JAR file (`bundle.jar`), then you don't need to specify a `-signedjar` option, as follows:

```
jarsigner bundle.jar jane
```

Example of Verifying a Signed JAR File

To verify a signed JAR file to ensure that the signature is valid and the JAR file wasn't been tampered with, use a command such as the following:

```
jarsigner -verify sbundle.jar
```

When the verification is successful, `jar verified` is displayed. Otherwise, an error message is displayed. You can get more information when you use the `-verbose` option. A sample use of `jarsigner` with the `-verbose` option follows:

```
jarsigner -verify -verbose sbundle.jar

          198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
          199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
         1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
  smk    2752 Fri Sep 26 16:12:30 PDT 1997 AclEx.class
  smk     849 Fri Sep 26 16:12:46 PDT 1997 test.class

    s = signature was verified
    m = entry is listed in manifest
    k = at least one certificate was found in keystore

  jar verified.
```

Example of Verification with Certificate Information

If you specify the `-certs` option with the `-verify` and `-verbose` options, then the output includes certificate information for each signer of the JAR file. The information includes the certificate type, the signer distinguished name information (when it is an X.509 certificate), and in parentheses, the keystore alias for the signer when the public key certificate in the JAR file matches the one in a keystore entry, for example:

```
jarsigner -keystore /working/mystore -verify -verbose -certs myTest.jar

          198 Fri Sep 26 16:14:06 PDT 1997 META-INF/MANIFEST.MF
          199 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.SF
         1013 Fri Sep 26 16:22:10 PDT 1997 META-INF/JANE.DSA
          208 Fri Sep 26 16:23:30 PDT 1997 META-INF/JAVATEST.SF
         1087 Fri Sep 26 16:23:30 PDT 1997 META-INF/JAVATEST.DSA
  smk    2752 Fri Sep 26 16:12:30 PDT 1997 Tst.class

    X.509, CN=Test Group, OU=Java Software, O=Oracle, L=CUP, S=CA, C=US (javatest)
    X.509, CN=Jane Smith, OU=Java Software, O=Oracle, L=cup, S=ca, C=us (jane)

    s = signature was verified
    m = entry is listed in manifest
    k = at least one certificate was found in keystore

  jar verified.
```

If the certificate for a signer isn't an X.509 certificate, then there is no distinguished name information. In that case, just the certificate type and the alias are shown. For example, if the certificate is a PGP certificate, and the alias is `bob`, then you would get: `PGP, (bob)`.

policytool

You use `policytool` to read and write a plain text policy file based on user input through the utility GUI.

> **Note:**
>
> The `policytool` tool has been deprecated in JDK 9 and might be removed in the next major JDK release.

Synopsis

`policytool [ -file ] [ filename ]`

`-file`
Directs the `policytool` command to load a policy file.

`filename`
The name of the file to be loaded.

Description

The `policytool` command calls an administrator's GUI that enables system administrators to manage the contents of local policy files. A policy file is a plain-text file with a `.policy` extension, that maps remote requestors by domain, to permission objects. For details, see Default Policy Implementation and Policy File Syntax.

Examples

To run the policy tool administrator utility, use the following command:

```
policytool
```

To run the `policytool` command and load the specified file, use the following command line:

```
policytool -file mypolicyfile
```

kinit

You use the `kinit` tool and its options to obtain and cache Kerberos ticket-granting tickets.

This tool is similar in functionality to the `kinit` tool that is commonly found in other Kerberos implementations, such as SEAM and MIT Reference implementations. The user must be registered as a principal with the Key Distribution Center (KDC) prior to running `kinit`.

Synopsis

Initial ticket request:

```
kinit [-A] [-f] [-p] [-c cache_name] [-l lifetime] [-r renewable_time] [[-k [-t
keytab_file_name]] [principal] [password]
```

Renew a ticket:

```
kinit -R [-c cachename] [principal]
```

Description

By default, on Windows, a cache file named *USER_HOME*\krb5cc_*USER_NAME* is generated.

The identifier *USER_HOME* is obtained from the `java.lang.System` property `user.home`. *USER_NAME* is obtained from the `java.lang.System` property `user.name`. If *USER_HOME* is null, the cache file is stored in the current directory from which the program is running. *USER_NAME* is the operating system's login user name. This user name could be different than the user's principal name. For example, on Windows, the cache file could be `C:\Windows\Users\duke\krb5cc_duke`, in which `duke` is the *USER_NAME* and `C:\Windows\Users\duke` is the *USER_HOME*.

By default, the keytab name is retrieved from the Kerberos configuration file. If the keytab name isn't specified in the Kerberos configuration file, the kinit tool assumes that the name is *USER_HOME*`\krb5.keytab`

If you don't specify the password using the `password` option on the command line, the `kinit` tool prompts you for the password.

> **Note:**
>
> The `password` option is provided only for testing purposes. Don't specify your password in a script or provide your password on the command line. Doing so will compromise your password.

Commands

You can specify one of the following commands. After the command, specify the options for it.

`-A`
Doesn't include addresses.

`-f`
Issues a forwardable ticket.

`-p`
Issues a proxiable ticket.

`-c` *`cache_name`*
The cache name (for example, `FILE:D:\temp\mykrb5cc`).

`-l` *`lifetime`*
Sets the lifetime of a ticket.

`-r` *`renewable_time`*
Sets the total lifetime that a ticket can be renewed.

`-R`
Renews a ticket.

`-k`
Uses keytab

`-t keytab_filename`
The keytab name (for example, `D:\winnt\profiles\duke\krb5.keytab`).

`principal`
The principal name (for example, `duke@example.com`).

`password`
The*`principal`*'s Kerberos password. **Don't specify this on the command line or in a script.**

`-help`
Displays instructions.

Examples

Requests credentials valid for authentication from the current client host, for the default services, storing the credentials cache in the default location (`C:\Windows\Users\duke\krb5cc_duke`):

```
kinit duke@example.com
```

Requests proxiable credentials for a different principal and store these credentials in a specified file cache:

```
kinit -p -c FILE:C:\Windows\Users\duke\credentials\krb5cc_cafebeef
    cafebeef@example.com
```

Requests proxiable and forwardable credentials for a different principal and stores these credentials in a specified file cache:

```
kinit -f -p -c FILE:C:\Windows\Users\duke\credentials\krb5cc_cafebeef
    cafebeef@example.com
```

Displays the help menu for the `kinit` tool:

```
kinit -help
```

klist

You use the `klist` tool to display the entries in the local credentials cache and key table. The `ktab` tool enables you to view entries in the local credentials cache and key table.

Synopsis

```
klist [ [-c] [-f] [-e] [-a [-n]] ] [-k [-t] [-K] ] [name] [-help]]
```

Description

The `klist` tool displays the entries in the local credentials cache and key table. After you modify the credentials cache with the `kinit` tool or modify the keytab with the `ktab`

tool, the only way to verify the changes is to view the contents of the credentials cache or keytab using the `klist` tool. The `klist` tool doesn't change the Kerberos database.

Commands

`-c`
Specifies that the credential cache is to be listed.
The following are the options for credential cache entries:

`-f`
Show credential flags.

`-e`
Show the encryption type.

`-a`
Show addresses.

`-n`
If the `-a` option is specified, don't reverse resolve addresses.

`-k`
Specifies that key tab is to be listed.
List the keytab entries. The following are the options for keytab entries:

`-t`
Show keytab entry timestamps.

`-K`
Show keytab entry DES keys.

`-e`
Shows keytab entry key type.

`name`
Specifies the credential cache name or the keytab name. File-based cache or keytab's prefix is `FILE:`. If the name isn't specified, the `klist` tool uses default values for the cache name and keytab. The `kinit` documentation lists these default values.

`-help`
Displays instructions.

Examples

List entries in the keytable specified including keytab entry timestamps and DES keys:

```
klist -k -t -K FILE:\temp\mykrb5cc
```

List entries in the credentials cache specified including credentials flag and address list:

```
klist -c -f FILE:\temp\mykrb5cc
```

ktab

You use the `ktab` tool to manage the principal names and service keys stored in a local key table.

Synopsis

`ktab` *commands options*

commands options
Lists the keytab name and entries, adds new key entries to the keytab, deletes existing key entries, and displays instructions. See Commands and Options.

Description

The `ktab` enables the user to manage the principal names and service keys stored in a local key table. Principal and key pairs listed in the keytab enable services running on a host to authenticate themselves to the Key Distribution Center (KDC).

Before configuring a server to use Kerberos, you must set up a keytab on the host running the server. Note that any updates made to the keytab using the `ktab` tool don't affect the Kerberos database.

A *keytab* is a host's copy of its own keylist, which is analogous to a user's password. An application server that needs to authenticate itself to the Key Distribution Center (KDC) must have a keytab which contains its own principal and key. If you change the keys in the keytab, you must also make the corresponding changes to the Kerberos database. The `ktab` tool enables you to list, add, update or delete principal names and key pairs in the key table. None of these operations affect the Kerberos database.

Security Alert

Don't specify your password on the command line. Doing so can be a security risk. For example, an attacker could discover your password while running the UNIX `ps` command.

Just as it is important for users to protect their passwords, it is equally important for hosts to protect their keytabs. You should always store keytab files on the local disk and make them readable only by root. You should never send a keytab file over a network in the clear.

Commands and Options

`-l [-e] [-t]`
Lists the keytab name and entries. When `-e` is specified, the encryption type for each entry is displayed. When `-t` is specified, the timestamp for each entry is displayed.

`-a` *`principal_name`* `[`*`password`*`] [-n` *`kvno`*`] [-append]`
Adds new key entries to the keytab for the given principal name with an optional *`password`*. If a *`kvno`* is specified, new keys' Key Version Numbers equal to the value, otherwise, automatically incrementing the Key Version Numbers. If `-append` is specified, new keys are appended to the keytab, otherwise, old keys for the same principal are removed.

No changes are made to the Kerberos database. **Don't specify the password on the command line or in a script.** This tool will prompt for a password if it isn't specified.

`-d principal_name [–f] [-e etype] [kvno | all| old]`
Deletes key entries from the keytab for the specified principal. No changes are made to the Kerberos database.

- If *kvno* is specified, the tool deletes keys whose Key Version Numbers match kvno. If `all` is specified, delete all keys.
- If `old` is specified, the tool deletes all keys except those with the highest *kvno*. The default action is `all`.
- If *etype* is specified, the tool only deletes keys of this encryption type. *etype* should be specified as the numberic value *etype* defined in RFC 3961, section 8. A prompt to confirm the deletion is displayed unless `-f` is specified.

When *etype* is provided, only the entry matching this encryption type is deleted. Otherwise, all entries are deleted.

`-help`
Displays instructions.

Common Options

This option can be used with the `-l`, `-a` or `-d` commands.

`-k keytab name`
Specifies the keytab name and path with the `FILE:` prefix.

Examples

Lists all the entries in the default keytable

```
ktab -l
```

Adds a new principal to the key table (note that you will be prompted for your password)

```
ktab -a duke@example.com
```

Deletes a principal from the key table

```
ktab -d duke@example.com
```

5

Remote Method Invocation (RMI) Tools and Commands

You use the RMI tools and commands to create applications that interact over the web or with another network.

The following sections describe the RMI tools and commands:

- rmic: You use the `rmic` compiler to generate stub and skeleton class files using the Java Remote Method Protocol (JRMP) and stub and tie class files (IIOP protocol) for remote objects.
- rmiregistry: You use the `rmiregistry` command to create and start a remote object registry on the specified port on the current host.
- rmid: You use the `rmid` command to start the activation system daemon that enables objects to be registered and activated in a Java Virtual Machine (JVM).
- serialver: You use the `serialver` command to return the `serialVersionUID` for one or more classes in a form suitable for copying into an evolving class.

rmic

You use the `rmic` compiler to generate stub and skeleton class files using the Java Remote Method Protocol (JRMP) and stub and tie class files (IIOP protocol) for remote objects. The `rmic` compiler generates Object Management Group (OMG) Interface Definition Language (IDL).

Synopsis

```
rmic [ options ] package-qualified-class-names
```

options
This represent the command-line *`options`* for the`rmic` compiler. See Options for the rmic Compiler.

package-qualified-class-names
Class names that include their packages, for example, `java.awt.Color`.

Description

Deprecation Note: Support for static generation of Java Remote Method Protocol (JRMP) stubs and skeletons has been deprecated. Oracle recommends that you use dynamically generated JRMP stubs instead, eliminating the need to use this tool for JRMP-based applications.

The `rmic` compiler generates stub and skeleton class files using the JRMP and stub and tie class files (IIOP protocol) for remote objects. These class files are generated from compiled Java programming language classes that are remote object implementation classes. A remote implementation class is a class that implements the interface `java.rmi.Remote`. The class names in the `rmic` command must be for classes

that were compiled successfully with the `javac` command and must be fully package qualified. For example, running the `rmic` command on the class file name `HelloImpl` as shown here creates the `HelloImpl_Stub.class` file in the `hello` subdirectory (named for the class's package):

```
rmic hello.HelloImpl
```

A skeleton for a remote object is a JRMP protocol server-side entity that has a method that dispatches calls to the remote object implementation.

A tie for a remote object is a server-side entity similar to a skeleton, but communicates with the client with the IIOP protocol.

A stub is a client-side proxy for a remote object that's responsible for communicating method invocations on remote objects to the server where the actual remote object implementation resides. A client's reference to a remote object, therefore, is actually a reference to a local stub.

By default, the `rmic` command generates stub classes that use the 1.2 JRMP stub protocol version only, as though the `-v1.2` option were specified. See Options for the rmic Compiler.

A stub implements only the remote interfaces, and not local interfaces that the remote object also implements. Because a JRMP stub implements the same set of remote interfaces as the remote object, a client can use the Java programming language built-in operators for casting and type checking. For IIOP, the `PortableRemoteObject.narrow` method must be used.

Options for the rmic Compiler

`-bootclasspath` *`path`*
Overrides the location of bootstrap class files.

`-classpath` *`path`*
Specifies the path the `rmic` command uses to look up classes. This option overrides the default or the `CLASSPATH` environment variable when it is set. Directories are separated by colons or semicolons, depending on your operating system. The following is the general format for *`path`*:

- **Oracle Solaris, Linux, and OS X**: `.:`*`your_path`*, for example: `.:/usr/local/java/classes`
- **Windows**: `.;`*`your_path`*, for example: `.;/usr/local/java/classes`

`-d` *`directory`*
Specifies the root destination directory for the generated class hierarchy. You can use this option to specify a destination directory for the stub, skeleton, and tie files.

- **Oracle Solaris, Linux, and OS X**: For example, the following command places the stub and skeleton classes derived from `MyClass` into the directory `/java/classes/exampleclass`:

  ```
  rmic -d /java/classes exampleclass.MyClass
  ```

- **Windows**: For example, the following command places the stub and skeleton classes derived from `MyClass` into the directory `C:\java\classes\exampleclass`:

  ```
  rmic -d C:\java\classes exampleclass.MyClass
  ```

If the `-d` option isn't specified, then the default behavior is as though `-d` was specified. The package hierarchy of the target class is created in the current directory, and stub/tie/skeleton files are placed within it.

`-g`
Enables the generation of all debugging information, including local variables. By default, only line number information is generated.

`-idl`
Causes the `rmic` command to generate OMG IDL for the classes specified and any classes referenced. IDL provides a purely declarative, programming language-independent way to specify an API for an object. The IDL is used as a specification for methods and data that can be written in and called from any language that provides CORBA bindings. This includes Java and C++ among others.
When the `-idl` option is used, other options also include:

- The `-always` or `-alwaysgenerate` options force regeneration even when existing stubs/ties/IDL are newer than the input class.
- The `-factory` option uses the `factory` keyword in generated IDL.
- The `-idlModule` from `JavaPackage[.class] toIDLModule` specifies `IDLEntity` package mapping, for example: `-idlModule my.module my::real::idlmod`.
- `-idlFile fromJavaPackage[.class] toIDLFile` specifies `IDLEntity` file mapping, for example: `-idlFile test.pkg.X TEST16.idl`.

`-iiop`
Causes the `rmic` command to generate IIOP stub and tie classes, rather than JRMP stub and skeleton classes. A stub class is a local proxy for a remote object and is used by clients to send calls to a server. Each remote interface requires a stub class, which implements that remote interface. A client reference to a remote object is a reference to a stub. Tie classes are used on the server side to process incoming calls, and dispatch the calls to the proper implementation class. Each implementation class requires a tie class.
If you call the `rmic` command with the `-iiop`, then it generates stubs and ties that conform to this naming convention:

```
_implementationName_stub.class
_interfaceName_tie.class
```

When you use the `-iiop` option, other options also include:

- The `-always` or `-alwaysgenerate` options force regeneration even when existing stubs/ties/IDL are newer than the input class.
- The `-nolocalstubs` option means don't create stubs optimized for same-process clients and servers.
- The `-noValueMethods` option must be used with the `-idl` option. The `-noValueMethods` option prevents the addition of `valuetype` methods and initializers to emitted IDL. These methods and initializers are optional for value types, and are generated unless the `-noValueMethods` option is specified with the `-idl` option.
- The `-poa` option changes the inheritance from `org.omg.CORBA_2_3.portable.ObjectImpl` to `org.omg.PortableServer.Servant`. The `PortableServer` module for the Portable Object Adapter (POA) defines the native

`Servant` type. In the Java programming language, the `Servant` type is mapped to the `Java org.omg.PortableServer.Servant` class. It serves as the base class for all POA servant implementations and provides a number of methods that can be called by the application programmer, and methods that are called by the POA and that can be overridden by the user to control aspects of servant behavior. This behavior is based on the OMG IDL to Java Language Mapping Specification, CORBA V 2.3.1 ptc/00-01-08.pdf .

`-J`*`argument`*
Used with any Java command, the `-J` option passes the *`argument`* that follows it (no spaces between the `-J` and the argument) to the Java interpreter.

`-keep` or `-keepgenerated`
Retains the generated `.java` source files for the stub, skeleton, and tie classes and writes them to the same directory as the `.class` files.

`-nowarn`
Turns off warnings. When the `-nowarn` options is used, the compiler doesn't print warnings.

`-nowrite`
Doesn't write compiled classes to the file system.

`-vcompat` (deprecated)
Generates stub and skeleton classes that are compatible with both the 1.1 and 1.2 JRMP stub protocol versions. This option was the default in releases before 5.0. The generated stub classes use the 1.1 stub protocol version when loaded in a JDK 1.1 virtual machine and use the 1.2 stub protocol version when loaded into a 1.2 (or later) virtual machine. The generated skeleton classes support both 1.1 and 1.2 stub protocol versions. The generated classes are relatively large to support both modes of operation. Note: This option has been deprecated. See Description.

`-verbose`
Causes the compiler and linker to print messages about what classes are being compiled and what class files are being loaded.

`-v1.1` (deprecated)
Generates stub and skeleton classes for the 1.1 JRMP stub protocol version only. The `-v1.1` option is useful only for generating stub classes that are serialization-compatible with existing, statically deployed stub classes generated by the `rmic` command from JDK 1.1 that can't be upgraded (and dynamic class loading isn't being used). Note: This option has been deprecated. See Description.

`-v1.2` (deprecated)
(Default) Generates stub classes for the 1.2 JRMP stub protocol version only. No skeleton classes are generated because skeleton classes aren't used with the 1.2 stub protocol version. The generated stub classes don't work when they're loaded into a JDK 1.1 virtual machine. Note: This option has been deprecated. See Description.

Environment Variables

`CLASSPATH`
Used to provide the system a path to user-defined classes.

- **Oracle Solaris, Linux, OS X**: Directories are separated by colons, for example: `.:/usr/local/java/classes`.
- **Windows**: Directories are separated by colons, for example: `.;C:\usr\local\java\classes`.

rmiregistry

You use the `rmiregistry` command to create and start a remote object registry on the specified port on the current host.

Synopsis

`rmiregistry` *`options port`*

`options`
This represents the option for the `rmiregistry` command. See Options

`port`
The number of a port on the current host at which to start the remote object registry.

Description

The `rmiregistry` command creates and starts a remote object registry on the specified port on the current host. If the port is omitted, then the registry is started on port 1099. The `rmiregistry` command produces no output and is typically run in the background, for example:

```
rmiregistry &
```

A remote object registry is a bootstrap naming service that's used by RMI servers on the same host to bind remote objects to names. Clients on local and remote hosts can then look up remote objects and make remote method invocations.

The registry is typically used to locate the first remote object on which an application needs to call methods. That object then provides application-specific support for finding other objects.

The methods of the `java.rmi.registry.LocateRegistry` class are used to get a registry operating on the local host or local host and port.

The URL-based methods of the `java.rmi.Naming` class operate on a registry and can be used to:

- Bind the specified name to a remote object
- Return an array of the names bound in the registry
- Return a reference, a stub, for the remote object associated with the specified name
- Rebind the specified name to a new remote object
- Destroy the binding for the specified name that's associated with a remote object

Options

-Joption
Used with any Java option to pass the *option* following the `-J` (no spaces between the `-J` and the option) to the Java interpreter.

rmid

You use the `rmid` command to start the activation system daemon that enables objects to be registered and activated in a Java Virtual Machine (JVM).

Synopsis

```
rmid [options]
```

options
This represent the command-line options for the `rmid` command. See Options for rmid.

Description

The `rmid` command starts the activation system daemon. The activation system daemon must be started before objects that can be activated are either registered with the activation system or activated in a JVM.

Start the daemon by executing the `rmid` command and specifying a security policy file, as follows:

```
rmid -J-Djava.security.policy=rmid.policy
```

When you run Oracle's implementation of the `rmid` command, by default you must specify a security policy file so that the `rmid` command can verify whether or not the information in each `ActivationGroupDesc` is allowed to be used to start a JVM for an activation group. Specifically, the command and options specified by the `CommandEnvironment` and any properties passed to an `ActivationGroupDesc` constructor must now be explicitly allowed in the security policy file for the `rmid` command. The value of the `sun.rmi.activation.execPolicy` property dictates the policy that the `rmid` command uses to determine whether or not the information in an `ActivationGroupDesc` can be used to start a JVM for an activation group. For more information see the description of the `-J-Dsun.rmi.activation.execPolicy=policy` option.

Executing the `rmid` command starts the `Activator` and an internal registry on the default port 1098 and binds an `ActivationSystem` to the name `java.rmi.activation.ActivationSystem` in this internal registry.

To specify an alternate port for the registry, you must specify the `-port` option when you execute the `rmid` command. For example, the following command starts the activation system daemon and a registry on the registry's default port, 1099.

```
rmid -J-Djava.security.policy=rmid.policy -port 1099
```

Start RMID on Demand (Oracle Solaris and Linux Only)

An alternative to starting `rmid` from the command line is to configure `inetd` (Oracle Solaris) or `xinetd` (Linux) to start `rmid` on demand.

When RMID starts, it attempts to obtain an inherited channel (inherited from `inetd`/`xinetd`) by calling the `System.inheritedChannel` method. If the inherited channel is null

or not an instance of `java.nio.channels.ServerSocketChannel`, then RMID assumes that it wasn't started by `inetd`/`xinetd`, and it starts as previously described.

If the inherited channel is a `ServerSocketChannel` instance, then RMID uses the `java.net.ServerSocket` obtained from the `ServerSocketChannel` as the server socket that accepts requests for the remote objects it exports: The registry in which the `java.rmi.activation.ActivationSystem` is bound and the `java.rmi.activation.Activator` remote object. In this mode, RMID behaves the same as when it is started from the command line, except in the following cases:

- Output printed to `System.err` is redirected to a file. This file is located in the directory specified by the `java.io.tmpdir` system property (typically `/var/tmp` or `/tmp`) with the prefix `rmid-err` and the suffix `tmp`.
- The `-port` option isn't allowed. If this option is specified, then RMID exits with an error message.
- The `-log` option is required. If this option isn't specified, then RMID exits with an error message

Options for rmid

`-C` *`option`*
Specifies an option that's passed as a command-line argument to each child process (activation group) of the `rmid` command when that process is created. For example, you could pass a property to each virtual machine spawned by the activation system daemon:

```
rmid -C-Dsome.property=value
```

This ability to pass command-line arguments to child processes can be useful for debugging. For example, the following command enables server-call logging in all child JVMs.

```
rmid -C-Djava.rmi.server.logCalls=true
```

`-J` *`option`*
Specifies an option that's passed to the Java interpreter running RMID command. For example, to specify that the `rmid` command use a policy file named `rmid.policy`, the `-J` option can be used to define the `java.security.policy` property on the `rmid` command line, for example:

```
rmid -J-Djava.security.policy-rmid.policy
```

`-J-Dsun.rmi.activation.execPolicy=`*`policy`*
Specifies the policy that the RMID command employs to check commands and command-line options used to start the JVM in which an activation group runs. This option exists only in Oracle's implementation of the Java RMI activation daemon. If this property isn't specified on the command line, then the result is the same as though `-J-Dsun.rmi.activation.execPolicy=default` were specified.
The possible values of *`policy`* can be `default`, *`policyClassName`*, or `none`.

- `default`

  The `default` or unspecified value `execPolicy` allows the `rmid` command to execute commands with specific command-line options only when the `rmid` command was granted permission to execute those commands and options in the security policy

file that the `rmid` command uses. Only the default activation group implementation can be used with the default execution policy.

The `rmid` command starts a JVM for an activation group with the information in the group's registered activation group descriptor, `ActivationGroupDesc`. The group descriptor specifies an optional `ActivationGroupDesc.CommandEnvironment` that includes the command to execute to start the activation group and any command-line options to be added to the command line. By default, the `rmid` command uses the `java` command found in `java.home`. The group descriptor also contains properties overrides that are added to the command line as options defined as: `-D`*`property=value`*. The `com.sun.rmi.rmid.ExecPermission` permission grants the `rmid` command permission to execute a command that's specified in the group descriptor's `CommandEnvironment` to start an activation group. The `com.sun.rmi.rmid.ExecOptionPermission` permission enables the `rmid` command to use command-line options, specified as properties overrides in the group descriptor or as options in the `CommandEnvironment` when starting the activation group. When granting the `rmid` command permission to execute various commands and options, the permissions `ExecPermission` and `ExecOptionPermission` must be granted to all code sources.

`ExecPermission` class: Represents permission for the `rmid` command to execute a specific command to start an activation group.

`ExecPermission` syntax: The name of `ExecPermission` is the path name of a command to grant the `rmid` command permission to execute.

A path name that ends in a slash (`/`) and an asterisk (`*`) indicates that all of the files are contained in that directory where the slash is the file-separator character, `File.separatorChar`.

A path name that ends in a slash (`/`) and a minus sign (`-`) indicates that all files and subdirectories are contained in that directory (recursively).

A path name that consists of the special token `<<ALL FILES>>` matches any file.

A path name that consists of an asterisk (`*`) indicates that all the files are in the current directory.

A path name that consists of a minus sign (`-`) indicates that all the files are in the current directory and (recursively) all files and subdirectories are contained in the current directory.

`ExecOptionPermission` class: Represents permission for the `rmid` command to use a specific command-line option when starting an activation group. The name of `ExecOptionPermission` is the value of a command-line option.

`ExecOptionPermission` syntax: Options support a limited wild card scheme. An asterisk signifies a wild card match, and it can appear as the option name itself (matches any option), or an asterisk (*) can appear at the end of the option name only when the asterisk (`*`) follows a dot (`.`) or an equals sign (`=`).

For example: `*` or `-Dmydir.*` or `-Da.b.c=*` is valid, but `*mydir` or `-Da*b` or `ab*` isn't valid.

Policy file for rmid

When you grant the `rmid` command permission to execute various commands and options, the permissions `ExecPermission` and `ExecOptionPermission` must be

granted to all code sources (universally). It is safe to grant these permissions universally because only the `rmid` command checks these permissions.

An example policy file that grants various execute permissions to the `rmid` command is:

- **Oracle Solaris**:

```
grant {
    permission com.sun.rmi.rmid.ExecPermission
        "/files/apps/java/jdk1.7.0/solaris/bin/java";

    permission com.sun.rmi.rmid.ExecPermission
        "/files/apps/rmidcmds/*";

    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.policy=/files/policies/group.policy";

    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.debug=*";

    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Dsun.rmi.*";
};
```

- **Windows:**

```
grant {
    permission com.sun.rmi.rmid.ExecPermission
        "c:\\files\\apps\\java\\jdk1.7.0\\win\\bin\\java";

    permission com.sun.rmi.rmid.ExecPermission
        "c:\\files\\apps\\rmidcmds\\*";

    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.policy=c:\\files\\policies\\group.policy";

    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.debug=*";

    permission com.sun.rmi.rmid.ExecOptionPermission
        "-Dsun.rmi.*";
};
```

The first permission granted allows the `rmid` command to execute the 1.7.0 release of the `java` command, specified by its explicit path name. By default, the version of the `java` command found in `java.home` is used (the same one that the `rmid` command uses), and doesn't need to be specified in the policy file. The second permission allows the `rmid` command to execute any command in either the directory `/files/apps/rmidcmds` (Oracle Solaris, Linux, and macOS) or the directory `c:\files\apps\rmidcmds\` (Windows).

The third permission granted, `ExecOptionPermission`, allows the `rmid` command to start an activation group that defines the security policy file to be either `/files/policies/group.policy` (Oracle Solaris) or `c:\files\policies\group.policy` (Windows). The next permission allows the `java.security.debug property` to be used by an activation group. The last permission allows any property in the `sun.rmi property` name hierarchy to be used by activation groups.

To start the `rmid` command with a policy file, the `java.security.policy` property needs to be specified on the `rmid` command line, for example:

```
rmid -J-Djava.security.policy=rmid.policy.
```

- *`policyClassName`*

  If the default behavior isn't flexible enough, then an administrator can provide, when starting the `rmid` command, the name of a class whose `checkExecCommand` method is executed to check commands to be executed by the `rmid` command.

  The `policyClassName` specifies a public class with a public, no-argument constructor and an implementation of the following `checkExecCommand` method:

  ```
  public void checkExecCommand(ActivationGroupDesc desc, String[] command)
         throws SecurityException;
  ```

  Before starting an activation group, the `rmid` command calls the policy's `checkExecCommand` method and passes to it the activation group descriptor and an array that contains the complete command to start the activation group. If the `checkExecCommand` throws a `SecurityException`, then the `rmid` command doesn't start the activation group and an `ActivationException` is thrown to the caller attempting to activate the object.

- `none`

  If the `sun.rmi.activation.execPolicy` property value is `none`, then the `rmid` command doesn't perform any validation of commands to start activation groups.

`-log` *`dir`*
Specifies the name of the directory that the activation system daemon uses to write its database and associated information. The log directory defaults to creating a log, in the directory in which the `rmid` command was executed.

`-port` *`port`*
Specifies the port that the registry uses. The activation system daemon binds `ActivationSystem`, with the name `java.rmi.activation.ActivationSystem`, in this registry. The `ActivationSystem` on the local machine can be obtained using the following `Naming.lookup` method call:

```
import java.rmi.*;
    import java.rmi.activation.*;

    ActivationSystem system; system = (ActivationSystem)
    Naming.lookup("//:port/java.rmi.activation.ActivationSystem");
```

`-stop`
Stops the current invocation of the `rmid` command for a port specified by the `-port` option. If no port is specified, then this option stops the `rmid` invocation running on port 1098.

Environment Variables

CLASSPATH
Used to provide the system a path to user-defined classes. Directories are separated by semicolons (;) (Windows) or by colons (:) (Oracle Solaris). For example:

- **Oracle Solaris**:

  ```
  .:/usr/local/java/classes
  ```

- **Windows**:

  ```
  .;C:\usr\local\java\classes
  ```

serialver

You use the `serialver` command to return the `serialVersionUID` for one or more classes in a form suitable for copying into an evolving class.

Synopsis

```
serialver [ options ] [ classnames ]
```

options
This represents the command-line options for the `serialver` command. See Options for serialver.

classnames
The classes for which `serialVersionUID` is to be returned.

Description

The `serialver` command returns the `serialVersionUID` for one or more classes in a form suitable for copying into an evolving class. When called with no arguments, the `serialver` command prints a usage line.

Options for serialver

`-classpath` ***`path-files`***
Sets the search path for application classes and resources. Separate classes and resources with a colon (:).

`-Joption`
Passes the specified *`option`* to the Java Virtual Machine, where *`option`* is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB.

Notes

The `serialver` command loads and initializes the specified classes in its virtual machine, and by default, it doesn't set a security manager. If the `serialver` command is to be run with untrusted classes, then a security manager can be set with the following option:

```
-J-Djava.security.manager
```

When necessary, a security policy can be specified with the following option:

```
-J-Djava.security.policy=policy file
```

6

Java IDL and RMI-IIOP Tools and Commands

You use the Java Interface Definition Language (IDL) and Java Remote Method Invocation interface over the Internet Inter-Orb Protocol (RMI-IIOP) tools and commands to create applications that use OMG-standard IDL and CORBA/IIOP.

The following sections describe the Java IDL and RMI-IIOP tools and commands:

- tnameserv: You use the `tnameserv` command as a substitute for Object Request Broker Daemon (ORBD). It starts the Java Interface Definition Language (IDL) name server.
- idlj: You use the `idlj` command to generate Java bindings for a specified Interface Definition Language (IDL) file.
- orbd: You use the `orbd` command for the client to transparently locate and call persistent objects on servers in the CORBA environment.
- servertool: You use the `servertool` command-line tool to register, unregister, start up, and shut down a persistent server.

tnameserv

You use the `tnameserv` command as a substitute for Object Request Broker Daemon (ORBD).

Synopsis

```
tnameserve -ORBInitialPort [ nameserverport ]
```

`-ORBInitialPort` *`nameserverport`*
The initial port where the naming service listens for the bootstrap protocol used to implement the ORB `resolve_initial_references` and `list_initial_references` methods.

Description

Java Interface Definition Language (IDL) includes the Object Request Broker Daemon (ORBD). ORBD is a daemon process that contains a Bootstrap Service, a Transient Naming Service, a Persistent Naming Service, and a Server Manager. The Java IDL tutorials all use ORBD, but you can substitute the `tnameserv` command for the `orbd` command in any of the examples that use a Transient Naming Service.

The CORBA Common Object Services (COS) Naming Service provides a tree-structure directory for object references similar to a file system that provides a directory structure for files. The Transient Naming Service provided with Java IDL, `tnameserv`, is a simple implementation of the COS Naming Service specification.

Object references are stored in the name space by name and each object reference-name pair is called a name binding. Name bindings can be organized under naming

contexts. Naming contexts are name bindings and serve the same organizational function as a file system subdirectory. All bindings are stored under the initial naming context. The initial naming context is the only persistent binding in the name space. The rest of the name space is lost when the Java IDL naming service process stops and restarts.

For an applet or application to use COS naming, its ORBD must know the port of a host running a naming service or have access to an initial naming context string for that naming service. The naming service can be either the Java IDL naming service or another COS-compliant naming service.

Start the Naming Service

You must start the Java IDL naming service before an application or applet that uses its naming service. Installation of the Java IDL product creates a script (Oracle Solaris, Linux, and OS X: `tnameserv`) or executable file (Windows: `tnameserv.exe`) that starts the Java IDL naming service. Start the naming service so that it runs in the background.

If you specify otherwise, then the Java IDL naming service listens on port 900 for the bootstrap protocol used to implement the Object Request Broker (ORB) `resolve_initial_references` and `list_initial_references methods`, as follows:

```
tnameserv -ORBInitialPort nameserverport&
```

If you don't specify the name server port, then port 900 is used by default. When running Oracle Solaris software, you must become the root user to start a process on a port below 1024. For this reason, it's recommended that you use a port number greater than or equal to 1024. To specify a different port, for example, 1050, and to run the naming service in the background, from an Oracle Solaris, Linux, or OS X command shell, enter:

```
tnameserv -ORBInitialPort 1050&
```

From an MS-DOS system prompt (Windows), enter:

```
start tnameserv -ORBInitialPort 1050
```

Clients of the name server must be made aware of the new port number. Do this by setting the `org.omg.CORBA.ORBInitialPort` property to the new port number when you create the ORB object.

Run the Server and Client on Different Hosts

In most of the Java IDL and RMI-IIOP tutorials, the naming service, server, and client are all running on the development machine. In real-world deployment, the client and server probably run on different host machines from the Naming Service.

For the client and server to find the Naming Service, they must be made aware of the port number and host on which the naming service is running. Do this by setting the `org.omg.CORBA.ORBInitialPort` and `org.omg.CORBA.ORBInitialHost` properties in the client and server files to the machine name and port number on which the Naming Service is running.

You could also use the command-line options `-ORBInitialPort nameserverport#` and `-ORBInitialHost nameserverhostname` to tell the client and server where to find the naming service.

For example, suppose the Transient Naming Service, `tnameserv` is running on port 1050 on host `nameserverhost`. The client is running on host `clienthost`, and the server is running on host `serverhost`.

Start `tnameserv` on the host `nameserverhost`:

```
tnameserv -ORBInitialPort 1050
```

Start the server on the `serverhost`:

```
java Server -ORBInitialPort 1050 -ORBInitialHost nameserverhost
```

Start the client on the `clienthost`:

```
java Client -ORBInitialPort 1050 -ORBInitialHost nameserverhost
```

Stop the Naming Service

To stop the Java IDL naming service, use the relevant operating system command, such as `kill` for an Oracle Solaris, Linux, or OS X process or `Ctrl+C` for a Windows process. The naming service continues to wait for invocations until it's explicitly shut down. Note that names registered with the Java IDL naming service disappear when the service is terminated.

Options

-Joption
Passes `option` to the JVM, where `option` is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. See Overview of Java Options.

Example of Adding Objects to the Name Space

This example shows how to add names to the following simple tree:

```
Initial Naming Context
     plans
     Personal
          calendar
          schedule
```

In the tree, `plans` is an object reference and `Personal` is a naming context that contains two object references: `calendar` and `schedule`.

The following sample program is a self-contained Transient Naming Service client that creates the tree:

```
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class NameClient {

    public static void main(String args[]) {

        try {
```

In Start the Naming Service, the `nameserver` was started on port 1050. The following code example ensures that the client program is aware of this port number.

```
            Properties props = new Properties();
            props.put("org.omg.CORBA.ORBInitialPort", "1050");
            ORB orb = ORB.init(args, props);
```

The following code example obtains the initial naming context and assigns it to `ctx`. The second line copies `ctx` into a dummy object reference `objref` that is attached to various names and added into the name space.

```
            NamingContext ctx =
                NamingContextHelper.narrow(
                    orb.resolve_initial_references("NameService"));
             NamingContext objref = ctx;
```

The following code example creates a name `plans` of type `text` and binds it to the dummy object reference. The `plans` is then added under the initial naming context using the `rebind` method. The `rebind` method enables you to run this program over and over again without getting the exceptions from using the `bind` method.

```
            NameComponent nc1 = new NameComponent("plans", "text");
            NameComponent[] name1 = {nc1};
            ctx.rebind(name1, objref);
            System.out.println("plans rebind successful!");
```

The following code example creates a naming context called `Personal` of type `directory`. The resulting object reference, `ctx2`, is bound to the `name` and added under the initial naming context.

```
            NameComponent nc2 = new NameComponent("Personal", "directory");
            NameComponent[] name2 = {nc2};
            NamingContext ctx2 = ctx.bind_new_context(name2);
            System.out.println("new naming context added..");
```

The remainder of the code binds the dummy object reference using the names `schedule` and `calendar` under the `Personal` naming context (`ctx2`).

```
            NameComponent nc3 = new NameComponent("schedule", "text");
            NameComponent[] name3 = {nc3};
            ctx2.rebind(name3, objref);
            System.out.println("schedule rebind successful!");

            NameComponent nc4 = new NameComponent("calendar", "text");
            NameComponent[] name4 = {nc4};
            ctx2.rebind(name4, objref);
            System.out.println("calendar rebind successful!");
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Example of Browsing the Name Space

The following sample program shows how to browse the name space:

```
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class NameClientList {

    public static void main(String args[]) {
```

```
        try {
```

In Start the Naming Service, the `nameserver` was started on port 1050. The following code example ensures that the client program is aware of this port number:

```
            Properties props = new Properties();
            props.put("org.omg.CORBA.ORBInitialPort", "1050");
            ORB orb = ORB.init(args, props);
```

The following code example obtains the initial naming context:

```
            NamingContext nc =
            NamingContextHelper.narrow(
                orb.resolve_initial_references("NameService"));
```

The `list` method lists the bindings in the naming context. In this case, up to 1000 bindings from the initial naming context will be returned in the `BindingListHolder`; any remaining bindings are returned in the `BindingIteratorHolder`.

```
            BindingListHolder bl = new BindingListHolder();
            BindingIteratorHolder blIt= new BindingIteratorHolder();
            nc.list(1000, bl, blIt);
```

The following code example gets the array of bindings out of the returned `BindingListHolder`. If there are no bindings, then the program ends.

```
            Binding bindings[] = bl.value;
            if (bindings.length == 0) return;
```

The remainder of the code loops through the bindings and prints the names.

```
            for (int i=0; i < bindings.length; i++) {

                // get the object reference for each binding
                org.omg.CORBA.Object obj = nc.resolve(bindings[i].binding_name);
                String objStr = orb.object_to_string(obj);
                int lastIx = bindings[i].binding_name.length-1;

                // check to see if this is a naming context
                if (bindings[i].binding_type == BindingType.ncontext) {
                    System.out.println("Context: " +
                        bindings[i].binding_name[lastIx].id);
                } else {
                    System.out.println("Object: " +
                        bindings[i].binding_name[lastIx].id);
                }
            }
        } catch (Exception e) {
            e.printStackTrace(System.err)
        }
    }
}
```

idlj

You use the `idlj` command to generate Java bindings for a specified Interface Definition Language (IDL) file.

Synopsis

```
idlj [options] idlfile
```

options
The command-line options. Options can appear in any order, but must precede the `idlfile`. See Options for idlj.

idlfile
The name of a file that contains the Interface Definition Language (IDL) definitions. The *`idlfile`* is required and must appear last.

Description

The IDL-to-Java compiler generates the Java bindings for a specified IDL file. Some earlier releases of the IDL-to-Java compiler were named `idltojava`.

Emit Client and Server Bindings

The following `idlj` command generates an IDL file named `My.idl` with client-side bindings:

```
idlj My.idl
```

The previous syntax is equivalent to the following:

```
idlj -fclient My.idl
```

The following example generates the server-side bindings, and includes the client-side bindings plus the skeleton, all of which are Portable Object Adapter (Inheritance Model).

```
idlg -fserver My.idl
```

If you want to generate both client and server-side bindings, then use one of the following (equivalent) commands:

```
idlj -fclient -fserver My.idl
idlj -fall My.idl
```

There are two possible server-side models:

- Portable Servant Inheritance Model
- Tie Model

Portable Servant Inheritance Model

The default server-side model is the Portable Servant Inheritance Model. Given an interface `My` defined in `My.idl`, the file `MyPOA.java` is generated. You must provide the implementation for the `My` interface, and the `My` interface must inherit from the `MyPOA` class. `MyPOA.java` is a stream-based skeleton that extends the class `org.omg.PortableServer.Servant`.

The `My` interface implements the `callHandler` interface and the operations interface associated with the IDL interface that the skeleton implements.

The `PortableServer` module for the Portable Object Adapter (POA) defines the native `Servant` type.

In the Java programming language, the `Servant` type is mapped to the Java `org.omg.PortableServer.Servant` class. It serves as the base class for all POA servant implementations and provides a number of methods that can be called by the application programmer, and methods that are called by the POA and that can be overridden by the user to control aspects of servant behavior.

Another option for the Inheritance Model is to use the `-oldImplBase` flag to generate server-side bindings that are compatible with releases of the Java programming language before Java SE 1.4. The `-oldImplBase` flag is nonstandard, and these APIs are deprecated. You would use this flag only for compatibility with existing servers written in Java SE 1.3. In that case, you would need to modify an existing make file to add the `-oldImplBase` flag to the `idlj` compiler. Otherwise, POA-based server-side mappings are generated. To generate server-side bindings that are backward compatible, do the following:

```
idlj -fclient -fserver -oldImplBase My.idl
idlj -fall -oldImplBase My.idl
```

Given an interface `My` defined in `My.idl`, the file `_MyImplBase.java` is generated. You must provide the implementation for the `My` interface, and the `My` interface must inherit from the `_MyImplBase` class.

Tie Model

The other server-side model is called the Tie Model. This is a delegation model. Because it isn't possible to generate ties and skeletons at the same time, they must be generated separately. The following commands generate the bindings for the Tie Model:

```
idlj -fall My.idl
idlj -fallTIE My.idl
```

For the `My` interface, the second command generates `MyPOATie.java`. The constructor to the `MyPOATie` class takes a delegate. In this example, using the default POA model, the constructor also needs a POA. You must provide the implementation for the delegate. It doesn't have to inherit from any other class, only from the interface `MyOperations`. To use it with the ORB, you must wrap your implementation within the `MyPOATie` class, for example:

```
ORB orb = ORB.init(args, System.getProperties());

// Get reference to rootpoa & activate the POAManager
POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");
rootpoa.the_POAManager().activate();

// create servant and register it with the ORB
MyServant myDelegate = new MyServant();
myDelegate.setORB(orb);

// create a tie, with servant being the delegate.
MyPOATie tie = new MyPOATie(myDelegate, rootpoa);
```

```
// obtain the objectRef for the tie
My ref = tie._this(orb);
```

You might want to use the Tie model instead of the typical Inheritance model when your implementation must inherit from some other implementation. Java allows any number of interface inheritances, but there's only one slot for class inheritance. If you use the inheritance model, then that slot is used up. With the Tie Model, that slot is freed up for your own use. The drawback is that it introduces a level of indirection: One extra method call occurs when a method is called.

For server-side generation, the following Tie Model bindings are compatible with versions of the IDL-to-Java language mapping in versions earlier than Java SE 1.4.

```
idlj -oldImplBase -fall My.idl
idlj -oldImplBase -fallTIE My.idl
```

For the `My` interface, this generates `My_Tie.java`. The constructor to the `My_Tie` class takes an `impl` object. You must provide the implementation for `impl`, but it doesn't have to inherit from any other class, only the interface `HelloOperations`. However to use it with the ORB, you must wrap your implementation within `My_Tie`. For example:

```
ORB orb = ORB.init(args, System.getProperties());

// create servant and register it with the ORB
MyServant myDelegate = new MyServant();
myDelegate.setORB(orb);

// create a tie, with servant being the delegate.
MyPOATie tie = new MyPOATie(myDelegate);

// obtain the objectRef for the tie
My ref = tie._this(orb);
```

Specify Alternate Locations for Emitted Files

If you want to direct the emitted files to a directory other than the current directory, then call the compiler in the following way:

```
idlj -td /altdir My.idl
```

For the `My` interface, the bindings are emitted to `/altdir/My.java`., instead of `./My.java`.

Specify Alternate Locations for Include Files

If the `My.idl` file includes another `idl` file, `MyOther.idl`, then the compiler assumes that the `MyOther.idl` file resides in the local directory. If it resides in `/includes`, for example, then you call the compiler with the following command:

```
idlj -i /includes My.idl
```

If `My.idl` also included in the `Another.idl` that resided in `/moreIncludes`, for example, then you call the compiler with the following command:

```
idlj -i /includes -i /moreIncludes My.idl
```

Because this form of the `include` file can become long, another way to indicate to the compiler where to search for included files is provided. This technique is similar to the idea of an environment variable. Create a file named `idl.config` in a directory that is listed in your `CLASSPATH` variable. Inside `idl.config`, provide a line with the following form:

```
includes=/includes;/moreIncludes
```

The compiler will find this file and read in the includes list. Note that in this example, the separator character between the two directories is a semicolon (;). This separator character is platform-dependent. On the Windows platform, use a semicolon; on the Oracle Solaris, Linux, and OS X platforms, use a colon.

Emit Bindings for Include Files

By default, only those interfaces, structures, and so on, that are defined in the `idl` file on the command line have Java bindings generated for them. The types defined in included files aren't generated. For example, assume the following two `idl` files:

```
My.idl file:

#include <MyOther.idl>
interface My
{
};

MyOther.idl file:

interface MyOther
{
};
```

There's a caveat to the default rule. Any `#include` statements that appear at the global scope are treated as described. These `#include` statements can be thought of as import statements. The `#include` statements that appear within an enclosed scope are treated as true `#include` statements, which means that the code within the included file is treated as though it appeared in the original file and, therefore, Java bindings are emitted for it. For example:

```
My.idl file:

#include <MyOther.idl>
interface My
{
  #include <Embedded.idl>
};

MyOther.idl file:

interface MyOther
{
};

Embedded.idl

enum E {one, two, three};
```

Run `idlj My.idl` to generate the following list of Java files. Notice that `MyOther.java` isn't generated because it's defined in an import-like `#include`. However, `E.java` was generated because it was defined in a true `#include`. Notice that because the `Embedded.idl` file is included within the scope of the interface `My`, it appears within the scope of `My` (in `MyPackage`). If the `-emitAll` flag had been used, then all types in all included files would have been emitted.

```
./MyHolder.java
./MyHelper.java
```

```
./_MyStub.java
./MyPackage
./MyPackage/EHolder.java
./MyPackage/EHelper.java
./MyPackage/E.java
./My.java
```

Insert Package Prefixes

Suppose that you work for a company named ABC that has constructed the following IDL file:

```
Widgets.idl file:

module Widgets
{
  interface W1 {...};
  interface W2 {...};
};
```

If you run this file through the IDL-to-Java compiler, then the Java bindings for W1 and W2 are placed within the `Widgets` package. There's an industry convention that states that a company's packages should reside within a package named `com.<company name>`. To follow this convention, the package name should be `com.abc.Widgets`. To place this package prefix onto the Widgets module, execute the following:

```
idlj -pkgPrefix Widgets com.abc Widgets.idl
```

If you have an IDL file that includes `Widgets.idl`, then the `-pkgPrefix` flag must appear in that command also. If it doesn't, then your IDL file will be looking for a `Widgets` package rather than a `com.abc.Widgets` package.

If you have a number of these packages that require prefixes, then it might be easier to place them into the `idl.config` file described previously. Each package prefix line should be of the form: `PkgPrefix.<type>=<prefix>`. The line for the previous example would be `PkgPrefix.Widgets=com.abc`. This option doesn't affect the Repository ID.

Define Symbols Before Compilation

You might need to define a symbol for compilation that isn't defined within the IDL file, perhaps to include debugging code in the bindings. The command `idlj -d MYDEF My.idl` is equivalent to putting the line `#define MYDEF` inside `My.idl`.

Preserve Preexisting Bindings

If the Java binding files already exist, then the `-keep` flag keeps the compiler from overwriting them. The default is to generate all files without considering that they already exist. If you've customized those files (which you shouldn't do unless you're very comfortable with their contents), then the `-keep` option is very useful. The command `idlj -keep My.idl` emits all client-side bindings that don't already exist.

View Compilation Progress

The IDL-to-Java compiler generates status messages as it progresses through its phases of execution. Use the `-v` option to activate the verbose mode: `idlj -v My.idl`.

By default, the compiler doesn't operate in verbose mode.

Display Version Information

To display the build version of the IDL-to-Java compiler, specify the `-version` option on the command-line: `idlj -version`.

Version information also appears within the bindings generated by the compiler. Any additional options appearing on the command-line are ignored.

Options for idlj

`-d` *`symbol`*
Equivalent to the following line in an IDL file:

```
#define symbol
```

`-emitAll`
Emit all types, including those found in `#included` files.

`-f`*`side`*
Defines what bindings to emit. The *`side`* parameter can be `client`, `server`, `serverTIE`, `all`, or `allTIE`. The `-fserverTIE` and `-fallTIE` options cause delegate model skeletons to be emitted. This defaults to `-fclient` when the flag isn't specified.

`-i` *`include-path`*
By default, the current directory to be scanned for included files. This option adds another directory.

`-keep`
If a file to be generated already exists, then do not overwrite it. By default it is overwritten.

`-noWarn`
Suppress warning messages.

`-oldImplBase`
Generates skeletons compatible with pre-1.4 JDK ORBs. By default, the POA Inheritance Model server-side bindings are generated. This option provides backward-compatibility with earlier releases of the Java programming language by generating server-side bindings that are `ImplBase` Inheritance Model classes.

`-pkgPrefix` *`type prefix`*
Wherever `type` is encountered at file scope, prefix the generated Java package name with `prefix` for all files generated for that type. The type is the simple name of either a top-level module, or an IDL type defined outside of any module.

`-pkgTranslate` *`type package`*
Whenever the module name type is encountered in an identifier, replace it in the identifier with package for all files in the generated Java package. Note that `pkgPrefix` changes are made first. The type value is the simple name of either a top-level module, or an IDL type defined outside of any module and must match the full package name exactly.
If more than one translation matches an identifier, then the longest match is chosen as shown in the following example:
Command:

```
pkgTranslate type pkg -pkgTranslate type2.baz pkg2.fizz
```

Resulting Translation:

```
type => pkg
type.ext => pkg.ext
type.baz => pkg2.fizz
type2.baz.pkg => pkg2.fizz.pkg
```

The following package names `org`, `org.omg`, or any subpackages of `org.omg` can't be translated. Any attempt to translate these packages results in uncompilable code, and the use of these packages as the first argument after `-pkgTranslate` is treated as an error.

`-skeletonName` *`xxx%yyy`*
Use `xxx%yyy` as the pattern for naming the skeleton. The defaults are: `%POA` for the `POA` base class (`-fserver` or `-fall`), and `_%ImplBase` for the `oldImplBase` class (`-oldImplBase`) and (`-fserver` or `-fall`).

`-td` *`dir`*
Use *`dir`* for the output directory instead of the current directory.

`-tieName` *`xxx%yyy`*
Use *`xxx%yyy`* according to the pattern. The defaults are: `%POA` for the `POA` base class (`-fserverTie` or `-fallTie`), and `_%Tie` for the `oldImplBase` tie class (`-oldImplBase`) and (`-fserverTie` or `-fallTie`).

`-v` or `-verbose`
Displays release information and terminates.

`-version`
Displays release information and terminates.

Restrictions

Escaped identifiers in the global scope can't have the same spelling as IDL primitive types, `Object` or `ValueBase`. This is because the symbol table is preloaded with these identifiers. Allowing them to be redefined would overwrite their original definitions. Possible permanent restriction.

The `fixed` IDL type isn' t supported.

Known Problems

No import is generated for global identifiers. If you call an unexported local `impl` object, then you do get an exception, but it seems to be due to a `NullPointerException` in the `ServerDelegate` DSI code.

orbd

You use the `orbd` command for the client to transparently locate and call persistent objects on servers in the CORBA environment.

Synopsis

`orbd [ options ]`

options
Command-line options. See orbd Options.

Description

The `orbd` command enables clients to transparently locate and call persistent objects on servers in the CORBA environment. The Server Manager included with the `orbd` tool is used to enable clients to transparently locate and call persistent objects on servers in the CORBA environment. The persistent servers, while publishing the persistent object references in the naming service, include the port number of the `orbd` in the object reference instead of the port number of the server. The inclusion of an `orbd` port number in the object reference for persistent object references has the following advantages:

- The object reference in the naming service remains independent of the server life cycle. For example, the object reference could be published by the server in the Naming Service when it is first installed, and then, independent of how many times the server is started or shut down, the `orbd` returns the correct object reference to the calling client.
- The client needs to look up the object reference in the naming service only once, and can keep reusing this reference independent of the changes introduced due to server life cycle.

To access the `orbd` Server Manager, the server must be started using `servertool`, which is a command-line interface for application programmers to register, unregister, start up, and shut down a persistent server. See Server Manager.

When `orbd` starts, it also starts a naming service. See **Start and Stop the Naming Service** below.

orbd Options

-ORBInitialPort *nameserverport*
Required. Specifies the port on which the name server should be started. After it's started, `orbd` listens for incoming requests on this port. On Oracle Solaris software, you must become the root user to start a process on a port below 1024. For this reason, Oracle recommends that you use a port number above or equal to 1024.

Nonrequired Options

`-port` *`port`*
Specifies the activation port where `orbd` should be started, and where `orbd` will be accepting requests for persistent objects. The default value for this port is 1049. This port number is added to the port field of the persistent Interoperable Object References (IOR).

`-defaultdb` *`directory`*
Specifies the base where the `orbd` persistent storage directory, `orb.db`, is created. If this option isn't specified, then the default value is `./orb.db`.

`-serverPollingTime` *`milliseconds`*
Specifies how often ORBD checks for the health of persistent servers registered through the `servertool`. The default value is 1000 ms. The value specified for `milliseconds` must be a valid positive integer.

`-serverStartupDelay` *`milliseconds`*
Specifies how long `orbd` waits before sending a location-forward exception after a persistent server that's registered through the `servertool` is restarted. The default

value is 1000 ms. The value specified for `milliseconds` must be a valid positive integer.

-J option
Passes `option` to the Java Virtual Machine, where `option` is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. See Java.

Start and Stop the Naming Service

A naming service is a CORBA service that allows CORBA objects to be named by means of binding a name to an object reference. The name binding can be stored in the naming service, and a client can supply the name to obtain the desired object reference.

Before running a client or a server, you'll start `orbd`. The `orbd` command includes a persistent naming service and a transient naming service, both of which are an implementation of the COS Naming Service.

The Persistent Naming Service provides persistence for naming contexts. This means that this information is persistent across service shutdowns and startups, and is recoverable in the event of a service failure. If ORBD is restarted, then the Persistent Naming Service restores the naming context graph, so that the binding of all clients' and servers' names remains intact (persistent).

For backward compatibility, `tnameserv` a Transient Naming Service that shipped with earlier releases of the JDK, is also included in this release of Java SE. A transient naming service retains naming contexts as long as it is running. If there is a service interruption, then the naming context graph is lost.

The `-ORBInitialPort` argument is a required command-line argument for `orbd`, and is used to set the port number on which the naming service runs. The following instructions assume that you can use port 1050 for the Java IDL Object Request Broker Daemon. When using Oracle Solaris software, you must become a root user to start a process on a port lower than 1024. For this reason, it's recommended that you use a port number above or equal to 1024. You can substitute a different port when necessary.

To start `orbd` from an Oracle Solaris, Linux, or OS X command shell, enter:

```
orbd -ORBInitialPort 1050&
```

From an MS-DOS system prompt (Windows), enter:

```
start orbd -ORBInitialPort 1050
```

Now that `orbd` is running, you can run your server and client applications. When running the client and server applications, they must be made aware of the port number (and machine name, when applicable) where the Naming Service is running. One way to do this is to add the following code to your application:

```
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", "1050");
props.put("org.omg.CORBA.ORBInitialHost", "MyHost");
ORB orb = ORB.init(args, props);
```

In this example, the naming service is running on port 1050 on host `MyHost`. Another way is to specify the port number or machine name, or both, when running the server

or client application from the command line. For example, you would start your `HelloApplication` with the following command-line:

```
java HelloApplication -ORBInitialPort 1050 -ORBInitialHost MyHost
```

To stop the naming service, use the relevant operating system command, such as `pkill orbd` on Oracle Solaris, or `Ctrl+C` in the DOS window in which `orbd` is running. Note that names registered with the naming service can disappear when the service is terminated because of a transient naming service. The Java IDL naming service will run until it's explicitly stopped.

Server Manager

To access the `orbd` Server Manager and run a persistent server, the server must be started with `servertool`, which is a command-line interface for application programmers to register, unregister, start up, and shut down a persistent server. When a server is started using `servertool`, it must be started on the same host and port on which `orbd` is executing. If the server is run on a different port, then the information stored in the database for local contexts will be invalid and the service will not work properly.

In this example, you run the `idlj` compiler and `javac` compiler as shown in the tutorial. To run the `orbd` Server Manager, follow these steps for running the application:

1. Start `orbd`.
   - Oracle Solaris, Linux, or OS X command shell, enter: `orbd -ORBInitialPort 1050`.
   - MS-DOS system prompt (Windows), enter: `start orbd -ORBInitialPort 1050`.
2. Port 1050 is the port on which you want the name server to run. The `-ORBInitialPort` option is a required command-line argument. When using Oracle Solaris software, you must become a root user to start a process on a port below 1024. For this reason, it is recommended that you use a port number above or equal to 1024.
3. Start `servertool`: `servertool -ORBInitialPort 1050`.
4. Make sure the name server (`orbd`) port is the same as in the previous step, for example, `-ORBInitialPort 1050`. The `servertool` must be started on the same port as the name server.
5. In the `servertool` command-line interface, start `Hello` server from the `servertool` prompt:

   ```
   servertool  > register -server HelloServer -classpath . -applicationName
                 HelloServerApName
   ```

6. The `servertool` registers the server, assigns it the name `HelloServerApName`, and displays its server ID with a listing of all registered servers. Run the client application from another terminal window or prompt:

   ```
   java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost
   ```

7. For this example, you can omit `-ORBInitialHost localhost` because the name server is running on the same host as the `Hello` client. If the name server is running on a different host, then use the `-ORBInitialHost nameserverhost` option to specify the host on which the IDL name server is running. Specify the name server (`orbd`) port as done in the previous step, for example, `-ORBInitialPort 1050`. When you finish experimenting with the `orbd` Server Manager, shut down or terminate the

name server (orbd) and servertool. To shut down orbd from an MS-DOS prompt, select the window that's running the server and enter Ctrl+C to shut it down.

8. To shut down orbd from an Oracle Solaris shell, find the process, and terminate with the kill command. The server continues to wait for invocations until it's explicitly stopped. To shut down the servertool, enter quit and press the Enter key.

servertool

You use the servertool command-line tool to register, unregister, start up, and shut down a persistent server.

Synopsis

```
servertool -ORBInitialPort nameserverport [ options ] [ commands ]
```

options
The command-line options. See Options for servertool.

commands
The command-line commands. See Using servertool Commands.

If you didn't enter a command when starting servertool, then command-line tool displays with a servertool > prompt. Enter commands at the servertool > prompt.

If you enter a command when starting servertool, then Java IDL Server Tool starts, runs the command, and exits.

The -ORBInitialPort nameserverport option is required. The value for nameserverport must specify the port on which orbd is running and listening for incoming requests.

> **Note:**
>
> On Oracle Solaris, you must become a root user to start a process on a port below 1024. Oracle recommends that you use a port number above or equal to 1024 for the nameserverport value.

Description

The servertool command provides the command-line interface for developers to register, unregister, start up, and shut down a persistent server. Command-line commands let you obtain various statistical information about the server. See Using servertool Commands.

Options for servertool

-ORBInitialHost *nameserverhost*
This option is required to secify the host machine on which the name server runs and listens for incoming requests. The nameserverhost value must specify the port on which the orb is running and listening for requests. The value defaults to localhost when this option isn't specified. If orbd and servertool are running on different

machines, then you must specify the name or IP address of the host on which `orbd` is running.

> **Note:**
>
> On Oracle Solaris, you must become a root user to start a process on a port below 1024. Oracle recommends that you use a port number above or equal to 1024 for the `nameserverport` value.

-Joption
Passes `option` to the Java Virtual Machine, where `option` is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. See Java.

Using servertool Commands

You can start the `servertool` command with or without a command-line command.

- If you don't specify a command when you start `servertool`, then the command-line tool displays the `servertool` prompt where you can enter commands: `servertool >`.
- If you specify a command when you start `servertool`, then the Java IDL Server Tool starts, executes the command, and exits.

`register -server` *`server-class-name`* `-classpath` *`classpath-to-server`* `[ -applicationName` *`application-name`* `-args` *`args-to-server`* `-vmargs` *`flags-for-JVM`* `]`
Registers a new persistent server with the Object Request Broker Daemon (ORBD). If the server isn't already registered, then it's registered and activated. This command causes an installation method to be called in the `main` class of the server identified by the `-server` option. The installation method must be `public static void install(org.omg.CORBA.ORB)`. The install method is optional and lets developers provide their own server installation behavior, such as creating a database schema.

`unregister -serverid` *`server-id`* `| -applicationName` *`application-name`*
Unregisters a server from the ORBD with either its server ID or its application name. This command causes an uninstallation method to be called in the `main` class of the server identified by the `-server` option. The `uninstall` method must be `public static void uninstall(org.omg.CORBA.ORB)`. The `uninstall` method is optional and lets developers provide their own server uninstallation behavior, such as undoing the behavior of the `install` method.

`getserverid -applicationName` *`application-name`*
Returns the server ID that corresponds to the `application-name` value.

`list`
Lists information about all persistent servers registered with the ORBD.

`listappnames`
Lists the application names for all servers currently registered with the ORBD.

`listactive`
Lists information about all persistent servers that were started by the ORBD and are currently running.

`locate -serverid server-id | -applicationName application-name [ -endpointTypeendpointType ]`
Locates the endpoints (ports) of a specific type for all ORBs created by a registered server. If a server isn't already running, then it's activated. If an `endpointType` value isn't specified, then the plain/non-protected endpoint associated with each ORB in a server is returned.

`locateperorb -serverid server-id | -applicationName application-name [ -orbidORB-name ]`
Locates all the endpoints (ports) registered by a specific Object Request Broker (ORB) of a registered server. If a server isn't already running, then it's activated. If an `orbid` isn't specified, then the default value of `""` is assigned to the `orbid`. If any ORBs are created with an `orbid` of an empty string, then all ports registered by it are returned.

`orblist -serverid server-id | -applicationName application-name`
Lists the `ORBId` of the ORBs defined on a server. An `ORBId` is the string name for the ORB created by the server. If the server isn't already running, then it's activated.

`shutdown -serverid server-id | -applicationName application-name`
Shut down an active server that's registered with ORBD. During execution of this command, the `shutdown` method defined in the class specified by either the `-serverid` or `-applicationName` parameter is also called to shut down the server process.

`startup -serverid server-id | -applicationName application-name`
Starts up or activate a server that is registered with ORBD. If the server isn't running, then this command starts the server. If the server is already running, then an error message is displayed.

`help`
Lists all the commands available to the server through the `servertool` command.

`quit`
Exits the `servertool` command.

7

Java Deployment Tools and Commands

You use Java deployment tools and commands to package Java and JavaFX applications for deployment.

The following sections describe the deployment tools and commands:

- pack200: You use the `pack200` command to transform a Java Archive (JAR) file into a compressed pack200 file with the Java gzip compressor.
- unpack200: You use the `unpack200` command to transform a packed file into a JAR file for web deployment.
- javapackager: You use the `javapackager` command to perform tasks related to packaging Java and JavaFX applications.

pack200

You use the `pack200` command to transform a Java Archive (JAR) file into a compressed pack200 file with the Java gzip compressor.

Synopsis

```
pack200 [-opt... | --option=value] x.pack[.gz] JAR-file
```

-opt... | --option=value
Options can be in any order. The last option on the command line or in a properties file supersedes all previously specified options. See Options for the pack200 Command.

***x.pack*[.gz]**
Name of the output file.

***file*.jar**
Name of the input file.

Description

The `pack200` command is a Java application that transforms a JAR file into a compressed `pack200` file with the Java gzip compressor. This command packages a JAR file into a compressed `pack200` file for web deployment. The `pack200` files are highly compressed files that can be directly deployed to save bandwidth and reduce download time.

Typical usage is shown in the following example, where `myarchive.pack.gz` is produced with the default `pack200` command settings:

```
pack200 myarchive.pack.gz myarchive.jar
```

> **Note:**
>
> This command shouldn't be confused with `pack`. The `pack` and `pack200` commands are separate products. The Java SE API Specification provided with the JDK is the superseding authority, when there are discrepancies.

Exit Status

The following exit values are returned: 0 for successful completion and a number greater than 0 when an error occurs.

Options for the pack200 Command

The `pack200` command has several options to fine-tune and set the compression engine. The typical usage is shown in the following example, where `myarchive.pack.gz` is produced with the default `pack200` command settings:

```
pack200 myarchive.pack.gz myarchive.jar
```

`-r` or `--repack`
Produces a JAR file by packing and unpacking a JAR file. The resulting file can be used as an input to the `jarsigner` tool. The following example packs and unpacks the `myarchive.jar` file:

```
pack200 --repack myarchive-packer.jar myarchive.jar
pack200 --repack myarchive.jar
```

`-g` or `--no-gzip`
Produces a `pack200` file. With this option, a suitable compressor must be used, and the target system must use a corresponding decompresser.

```
pack200 --no-gzip myarchive.pack myarchive.jar
```

`--gzip`
(Default) Post-compresses the pack output with `gzip`.

`-G` or `-–strip-debug`
Strips debugging attributes from the output. These include `SourceFile`, `LineNumberTable`, `LocalVariableTable` and `LocalVariableTypeTable`. Removing these attributes reduces the size of both downloads and installations, also reduces the usefulness of debuggers.

`--keep-file-order`
Preserves the order of files in the input file. This is the default behavior.

`-O` or `--no-keep-file-order`
Reorders and transmits all elements. The packer can also remove JAR directory names to reduce the download size. However, certain JAR file optimizations, such as indexing, might not work correctly.

`-S`*`N`* or `--segment-limit=`*`N`*
The value is the estimated target size *`N`* (in bytes) of each archive segment. If a single input file requires more than *`N`*bytes, then its own archive segment is provided. As a special case, a value of `-1` produces a single large segment with all input files, while a value of 0 produces one segment for each class. Larger archive segments result in

less fragmentation and better compression, but processing them requires more memory.
The size of each segment is estimated by counting the size of each input file to be transmitted in the segment with the size of its name and other transmitted properties. The default is -1, which means that the packer creates a single segment output file. In cases where extremely large output files are generated, users are strongly encouraged to use segmenting or break up the input file into smaller JAR file.
A 10 MB JAR packed without this limit typically packs about 10 percent smaller, but the packer might require a larger Java heap (about 10 times the segment limit).

`-E`*`value`* or `--effort=`*`value`*
If the value is set to a single decimal digit, then the packer uses the indicated amount of effort in compressing the archive. Level 1 might produce somewhat larger size and faster compression speed, while level 9 takes much longer, but can produce better compression. The special value 0 instructs the `pack200` command to copy through the original JAR file directly with no compression. The JSR 200 standard requires any unpacker to understand this special case as a pass-through of the entire archive.
The default is 5, to invest a modest amount of time to produce reasonable compression.

`-H`*`value`* or `--deflate-hint=`*`value`*
Overrides the default, which preserves the input information, but can cause the transmitted archive to be larger. The possible values are: `true`, `false`, or `keep`.
If the `value` is `true` or false, then the `packer200` command sets the deflation hint accordingly in the output archive and doesn't transmit the individual deflation hints of archive elements.
The `keep` value preserves deflation hints observed in the input JAR. This is the default.

`-m`*`value`* or `--modification-time=`*`value`*
The possible values are `latest` and `keep`.
If the value is `latest`, then the packer attempts to determine the latest modification time, among all the available entries in the original archive, or the latest modification time of all the available entries in that segment. This single value is transmitted as part of the segment and applied to all the entries in each segment. This can marginally decrease the transmitted size of the archive at the expense of setting all installed files to a single date.
If the value is `keep`, then modification times observed in the input JAR are preserved. This is the default.

`-P`*`file`* or `--pass-file=`*`file`*
Indicates that a file should be passed through bytewise with no compression. By repeating the option, multiple files can be specified. There is no path name transformation, except that the system file separator is replaced by the JAR file separator forward slash (/). The resulting file names must match exactly as strings with their occurrences in the JAR file. If `file` is a directory name, then all files under that directory are passed.

`-U` *`action`* or `--unknown-attribute=`*`action`*
Overrides the default behavior, which means that the class file that contains the unknown attribute is passed through with the specified `action`. The possible values for actions are `error`, `strip`, or `pass`.
If the value is `error`, then the entire `pack200` command operation fails with a suitable explanation.
If the value is `strip`, then the attribute is dropped. Removing the required Java Virtual Machine (JVM) attributes can cause class loader failures.

If the value is `pass`, then the entire class is transmitted as though it is a resource.

`-C attribute-name=layout` or `--class-attribute=attribute-name=action`
(user-defined attribute) See the description for `–Dattribute-name=layout`.

`-F attribute-name=layout` or `--field-attribute=attribute-name=action`
(user-defined attribute) See the description for `–Dattribute-name=layout`

`-Mattribute-name=layout` or `--method-attribute=attribute-name=action`
(user-defined attribute) See the description for `–Dattribute-name=layout`

`–D attribute-name=layout` or `--code-attribute=attribute-name=action`
(user-defined attribute) The attribute layout can be specified for a class entity, such as `class-attribute`, `field-attribute`, `method-attribute`, and `code-attribute`. The *`attribute-name`* is the name of the attribute for which the layout or action is being defined. The possible values for *`action`* are `some-layout-string`, `error`, `strip`, `pass`.
`some-layout-string`: The layout language is defined in the JSR 200 specification, for example: `--class-attribute=SourceFile=RUH`.
If the value is `error`, then the `pack200` operation fails with an explanation.
If the value is `strip`, then the attribute is removed from the output. Removing JVM-required attributes can cause class loader failures. For example, `--class-attribute=CompilationID=pass` causes the class file that contains this attribute to be passed through without further action by the packer.
If the value is `pass`, then the entire class is transmitted as though it's a resource.

`-f pack.properties` or `--config-file=pack.properties`
Indicates a configuration file, containing Java properties to initialize the packer, can be specified on the command line.

```
pack200 -f pack.properties myarchive.pack.gz myarchive.jar
more pack.properties
# Generic properties for the packer.
modification.time=latest
deflate.hint=false
keep.file.order=false
# This option will cause the files bearing new attributes to
# be reported as an error rather than passed uncompressed.
unknown.attribute=error
# Change the segment limit to be unlimited.
segment.limit=-1
```

`-v` or `--verbose`
Outputs minimal messages. Multiple specification of this option will create more verbose messages.

`-q` or `--quiet`
Specifies quiet operation with no messages.

`-l filename`or `--log-file=filename`
Specifies a log file to output messages.

`-?` , `-h` , or`--help`
Prints help information about this command.

`-V` or `--version`
Prints version information about this command.

-Joption
Passes the specified option to the Java Virtual Machine. For example, `-J-Xms48m` sets the startup memory to 48 MB.

unpack200

You use the `unpack200` command to transform a packed file into a JAR file for web deployment.

Synopsis

`unpack200` [*`options`*] *`input-file JAR-file`*

options
The command-line options. See Options for the unpack200 Command.

input-file
Name of the input file, which can be a `pack200 gzip` file or a `pack200` file. The input can also be a JAR file produced by `pack200` with an effort of `0`, in which case the contents of the input file are copied to the output JAR file with the `pack200` marker.

JAR-file
Name of the output JAR file.

Description

The `unpack200` command is a native implementation that transforms a packed file produced by the `pack200` into a JAR file for web deployment. An example of typical usage follows. In the following example, the `myarchive.jar` file is produced from `myarchive.pack.gz` with the default `unpack200` command settings.

```
unpack200 myarchive.pack.gz myarchive.jar
```

Options for the unpack200 Command

`-H` or `--deflate-hint=`*`value`*
Sets the deflation to be `true`, `false`, or `keep` on all entries within a JAR file. The default mode is `keep`. If the value is `true` or `false`, then the `--deflate=hint` option overrides the default behavior and sets the deflation mode on all entries within the output JAR file.

`-r` or`--remove-pack-file`
Removes the input pack file.

`-v` or `--verbose`
Displays minimal messages. Multiple specifications of this option displays more verbose messages.

`-q` or `--quiet`
Specifies quiet operation with no messages.

`-l` *filename* or `--log-file=`*filename*
Specifies a log file where output messages are logged.

`-?` or `-h` or `--help`
Prints help information about the `unpack200` command.

`-V` or `--version`
Prints version information about the `unpack200` command.

`-J`option
Passes option to the Java Virtual Machine, where `option` is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB.

Notes

This command shouldn't be confused with the `unpack` command. They're distinctly separate products.

The Java SE API Specification provided with the JDK is the superseding authority in case of discrepancies.

Exit Status

The following exit values are returned: 0 for successful completion, and a value that is greater than 0 when an error occurred.

javapackager

You use the `javapackager` command to perform tasks related to packaging Java and JavaFX applications.

Synopsis

`javapackager` *`command`* [*`options`*]

`command`
The task that you want to perform. See Commands for the javapackager Command.

`options`
One or more options for the command, separated by spaces. See Options for the createbss Command, Options for the createjar Command, Options for the deploy Command, Options for the makeall Command, and Options for the signjar Command.

> **Note:**
>
> The `javapackager` command isn't available on Oracle Solaris.

Description

The Java Packager tool compiles, packages, and prepares Java and JavaFX applications for distribution. The `javapackager` command is the command-line version. For available Ant tasks, see JavaFX Ant Tasks in *Java Platform, Standard Edition Deployment Guide*.

For self-contained applications, the Java Packager for JDK 9 packages applications with a JDK 9 runtime image generated by the `jlink` tool. To package a JDK 8 or JDK 7 JRE with your application, use the JDK 8 Java Packager.

Commands for the javapackager Command

You can run the following commands from the command line, followed by the options for the command.

`-createbss`
Converts CSS files into binary form. See Options for the createbss Command for the options used with this command.

`-createjar`
Produces a JAR according to other parameters. See Options for the createjar Command for the options used with this command.

`-deploy`
Assembles the application package for distribution. Modular and nonmodular applications are supported. By default, the deploy task generates the base application package. It can also generate a self-contained application package, if requested. See Options for the deploy Command for the options used with this command.
The bundle for a self-contained application includes a custom runtime created by calling `jlink`. The Java Packager for JDK 9 packages applications with a JDK 9 runtime image. To package a JDK 8 or JDK 7 JRE with your application, use the JDK 8 Java Packager.

`-makeall`

> **Note:**
>
> The `-makeall` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release.

Performs compilation, `createjar`, and `deploy` steps as one call, with most arguments predefined, and attempts to generate all applicable self-contained application packages. The source files must be located in a folder called `src`, and the resulting files (JAR, JNLP, HTML, and self-contained application packages) are put in a folder called `dist`. This command can be configured only in a minimal way and is as automated as possible. See Options for the makeall Command for the options used with this command.

`-signjar`

> **Note:**
>
> The `-signjar` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release. It also doesn't work with multirelease JAR file. Instead, use the jarsigner tool to sign the JAR file.

Signs JAR files with a provided certificate. See Options for the signjar Command for the options used with this command.

Options for the createbss Command

`-outdir` *`dir`*
Name of the directory that receives the generated output files.

`-srcdir` *`dir`*
Base directory of the files to pack.

`-srcfiles` *`files`*
List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be used.

Options for the createjar Command

`-appclass` *`app-class`*
Qualified name of the application class to be executed.

`-argument` *`arg`*
An unnamed argument to be inserted into the JNLP file as an `<fx:argument>` element.

`-classpath` *`files`*
List of dependent JAR file names.

`-manifestAttrs` *`manifest-attributes`*
List of names and values for additional manifest attributes. Syntax:

`"name1=value1,name2=value2,..."`

`-nocss2bin`
The packager doesn't convert CSS files to binary form before copying to JAR file.

`-noembedlauncher`
If present, the packager will not add the JavaFX launcher classes to the jarfile.

`-outdir` *`dir`*
Name of the directory that receives the generated output files.

`-outfile` *`filename`*
Name (without the extension) of the file that's generated.

`-paramfile` *`file`*
Properties file with named parameters and their default values to pass to the application.

`-preloader` *`preloader-class`*
Qualified name of the JavaFX preloader class to be executed. Use this option only for JavaFX applications. Don't use for Java applications, including headless applications.

`-runtimeversion` *`version`*
Specifies the version of the required JavaFX Runtime.

`-srcdir` *`dir`*
Base directory of the files to pack.

`-srcfiles` *`files`*
List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be packed.

Options for the deploy Command

`--add-modules` *`modulename`*`[,`*`modulename`*`...]`
Specifies the root modules to resolve in addition to the initial module.

`-allpermissions`
If present, the application requires all security permissions in the JNLP file.

`-appclass` *`app-class`*
Qualified name of the application class to be executed.

`-argument` *`arg`*
An unnamed argument to be inserted into an `<fx:argument>` element in the JNLP file.

`-B`*`bundler-argument=value`*
Provides information to the bundler that's used to package a self-contained application. See Arguments for Self-Contained Application Bundles for information about the arguments for each bundler.

`-callbacks` *`callback-methods`*
Specifies one or more user callback methods in generated HTML. The format is the following:

`"`*`name1`*`:`*`value1`*`,`*`name2`*`:`*`value2`*`,..."`

`-description` *`description`*
Description of the application.

`-embedjnlp`
If present, the JNLP file embedded in the HTML document.

`-embedCertificates`
If present, the certificates will be embedded in the jnlp file.

`-height` *`height`*
Height of the application.

`-htmlparamfile` *`file`*
Properties file with parameters for the resulting application when it is run in the browser.

`-isExtension`
If present, the `srcfiles` as extensions.

`--limit-modules` *`modulename`*`[,`*`modulename`*`...]`
Limits the universe of observable modules.

`-m` *`modulename`* `[/`*`mainclass`*`]` or `--module` *`modulename`* `[/`*`mainclass`*`]`
Specifies the initial module to resolve, and the name of the main class to execute if not specified by the module.

`-p` *`module path`* or `--module-path` *`module path`*
A : separated list of directories, each directory is a directory of modules.

`-name` *`name`*
Name of the application.

`-native` *`type`*
Generate the files needed for a Java Web Start application when *`type`* is set to `jnlp`. Otherwise, generate self-contained application bundles, if possible. Use the `-B` option to provide arguments to the bundlers being used. If *`type`* is specified, then only a bundle of this type is created. If no type is specified, then `all` is used.
The following values are valid for *type*:

- `jnlp`: Generates the `.jnlp` and `.html` files for a Java Web Start application.
- `all`: Runs all of the installers for the platform on which it's running, and creates a disk image for the application. This value is used if *`type`* isn't specified.
- `installer`: Runs all of the installers for the platform on which it's running.
- `image`: Creates a disk image for the application.

  Linux and Windows: The image is the directory that gets installed.

  macOS: The image is the `.app` file.
- `exe`: Generates a Windows `.exe` package.
- `msi`: Generates a Windows Installer package.
- `dmg`: Generates a DMG file for macOS.
- `pkg`: Generates a `.pkg` package for macOS.
- `mac.appStore`: Generates a package for the Mac App Store.
- `rpm`: Generates an RPM package for Linux.
- `deb`: Generates a Debian package for Linux.

`-nosign`
Linux and macOS: If present, the bundle generated for self-contained applications isn't signed by the bundler. The default for bundlers that support signing is to sign the bundle if signing keys are properly configured. This attribute is ignored by bundlers that don't support signing.

`-outdir` *`dir`*
Name of the directory that receives the generated output files.

`-outfile` *`filename`*
Name (without the extension) of the file that is generated.

`-paramfile` *`file`*
Properties file with named parameters and their default values to pass to the application.

`-preloader` *`preloader-class`*
Qualified name of the JavaFX preloader class to be executed. Use this option only for JavaFX applications. Don't use for Java applications, including headless applications.

`-srcdir` *`dir`*
Base directory of the files to pack.

`-srcfiles files`
List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be used.

`--strip-native-commands [true|false]`
Remove command-line tools such as `java.exe` from the Java runtime that's generated for packaging with self-contained applications. The default is `true`. To keep the tools in the runtime, specify `false`.

`-templateId`
Application ID of the application for template processing.

`-templateInFilename`
Name of the HTML template file. Placeholders are in the following form:

`#XXXX.YYYY(APPID)#`

`APPID` is the identifier of an application and `XXXX` is one of following:

- `DT.SCRIPT.URL`

  Location of `dtjava.js` in the Deployment Toolkit. By default, the location is `http://java.com/js/dtjava.js`.

- `DT.SCRIPT.CODE`

  Script element to include `dtjava.js` of the Deployment Toolkit.

- `DT.EMBED.CODE.DYNAMIC`

  Code to embed the application into a given placeholder. It is expected that the code is wrapped in the `function()` method.

- `DT.EMBED.CODE.ONLOAD`

  All of the code needed to embed the application into a web page using the `onload` hook (except inclusion of `dtjava.js`).

- `DT.LAUNCH.CODE`

  Code needed to launch the application. It's expected that the code is wrapped in the `function()` method.

`-templateOutFilename`
Name of the HTML file generated from the template.

`-title title`
Title of the application.

`-updatemode update-mode`
Sets the update mode for the JNLP file.

`-vendor vendor`
Vendor of the application.

`-width width`
Width of the application.

Options for the makeall Command

> **Note:**
>
> The `-makeall` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release.

`-appclass` *`app-class`*
Qualified name of the application class to be executed.

`-classpath` *`files`*
List of dependent JAR file names.

`-height` *`height`*
Height of the application.

`-name` *`name`*
Name of the application.

`-preloader` *`preloader-class`*
Qualified name of the JavaFX preloader class to be executed. Use this option only for JavaFX applications. Don't use for Java applications, including headless applications.

`-v`
Enables verbose output.

`-width` *`width`*
Width of the application.

Options for the signjar Command

> **Note:**
>
> The `-signjar` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release. It also doesn't work with multirelease JAR files. Use the jarsigner tool to sign the JAR file.

`-alias` *`key-alias`*
Alias for the key.

`-keyPass` *`password`*
Password for recovering the key.

`-keyStore` *`file`*
Keystore file name.

`-outdir` *`dir`*
Name of the directory that receives the generated output files.

`-storePass` *`password`*
Password to check the integrity of the keystore or unlock the keystore.

`-storeType type`
Keystore type. The default value is `jks`.

`-srcdir dir`
Base directory of the files to pack.

`-srcfiles files`
List of files in `srcdir`. If omitted, all files in `srcdir` (which is a mandatory argument in this case) will be packed.

Arguments for Self-Contained Application Bundles

The `-B`*`bundler-argument=value`* option for the `-deploy` command is used when generating self-contained applications. This option enables you to set an argument for the bundler that's used to create self-contained applications. To set more than one argument, pass an instance of this option for each argument. Each type of bundler has its own set of arguments.

The following sections describe the valid arguments for the available bundlers:

- General Bundler Arguments
- macOS Application Bundler Arguments
- macOS DMG (Disk Image) Bundler Arguments
- macOS PKG Bundler Arguments
- Mac App Store Bundler Arguments
- Linux Debian Bundler Arguments
- Linux RPM Bundler Arguments
- Windows EXE Bundler Arguments
- Windows MSI Bundler Arguments

General Bundler Arguments

General bundler arguments are valid for all bundlers.

`appVersion=`*`version`*
Version of the application package. Some bundlers restrict the format of the version string.

`arguments=`*`option=value`*
Arguments to pass to the application when it is started. Enclose the argument list in quotes. To pass multiple options, separate the option-value pairs with spaces, for example:

```
-Barguments="this.is.a.test=tru one.more.arg=affirmative"
```

`classPath=`*`path`*
Class path relative to the assembled application directory. The path is typically extracted from the JAR file manifest, and doesn't need to be set if you're using the other `javapackager` commands.

`dropinResourcesRoot=`*`directory`*
Directory in which to look for bundler-specific drop-in resources. For example, on macOS, to look in the current directory for the `Info.plist` file, use the following:

```
-BdropinResourcesRoot=.
```

The file is then found in the current directory: `package/macosx/Info.plist`.

`icon=`*`path`*
Location of the default icon to be used for application launchers and other assists.
Linux: The format must be `.png`.
macOS: The format must be `.icns`.
Windows: The format must be `.ico`.

`identifier=`*`value`*
Default value that is used for other platform-specific values such as `mac.CFBundleIdentifier`. Reverse DNS order is recommended, for example, `com.example.application.my-application`.

`jvmOptions=`*`option`*
Option to be passed to the JVM when the application is run. Any option that is valid for the `java` command can be used. To pass more than one option, use multiple instances of the `-B` option, as shown in the following example:

```
-BjvmOptions=-Xmx128m -BjvmOptions=-Xms128m
```

`jvmProperties=`*`property=value`*
Java system property to be passed to the VM when the application is run. Any property that's valid for the `-D` option of the `java` command can be used. Specify both the property name and the value for the property. To pass more than one property, use multiple instances of the `-B` option, as shown in the following example:

```
-BjvmProperties=apiUserName=example -BjvmProperties=apiKey=abcdef1234567890
```

`mainJar=`*`filename`*
Name of the JAR file that contains the main class for the application. The file name is typically extracted from the JAR file manifest, and doesn't need to be set if you're using the other `javapackager` commands.

`preferencesID=`*`node`*
Preferences node to examine to check for JVM options that the user can override. The node specified is passed to the application at runtime as the option `-Dapp.preferences.id`. This argument is used with the `userJVMOptions` argument.

`runtime=`*`path`*
Location of the JRE or JDK to use with a Java Web Start application, valid only when the `-native` option is set to `jnlp`.

`userJvmOptions=`*`option=value`*
JVM options that users can override. Any option that's valid for the `java` command can be used. Specify both the option name and the value for the option. To pass more than one option, use multiple instances of the `-B` option, as shown in the following example:

```
-BuserJvmOptions=-Xmx=128m -BuserJvmOptions=-Xms=128m
```

macOS Application Bundler Arguments

`mac.category=`*`category`*
Category for the application. The category must be in the list of categories found on the Apple Developer website.

`mac.CFBundleIdentifier=value`
Value stored in the info plist for `CFBundleIdentifier`. This value must be globally unique and contain only letters, numbers, dots, and dashes. Reverse DNS order is recommended, for example, `com.example.application.my-application`.

`mac.CFBundleName=name`
Name of the application as it appears on the macOS menu bar. A name of fewer than 16 characters is recommended. The default is the `name` attribute.

`mac.CFBundleVersion=value`
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `appVersion` argument. If the `appVersion` argument is specified with a valid value and the `mac.CFBundleVersion` argument isn't specified, then the `appVersion` value is used. If neither argument is specified, then `100` is used as the version number.

`mac.signing-key-developer-id-app=key`
Name of the signing key used for Developer ID or Gatekeeper signing. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application isn't signed.

`mac.bundle-id-signing-prefix=prefix`
Prefix that is applied to the signed binary when binaries that lack plists or existing signatures are found inside the bundles.

macOS DMG (Disk Image) Bundler Arguments

The macOS DMG installer shows the license file specified by `licenseFile`, if provided, before allowing the disk image to be mounted.

`licenseFile=path`
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources, for example, `-BlicenseFile=COPYING`.

`systemWide=boolean`
Flag that indicates which drag-to-install target to use. Set to `true` to show the Applications folder. Set to `false` to show the Desktop folder. The default is `true`.

`mac.CFBundleVersion=value`
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `appVersion` argument. If the `appVersion` argument is specified with a valid value and the `mac.CFBundleVersion` argument isn't specified, then the `appVersion` value is used. If neither argument is specified, then `100` is used as the version number.

`mac.dmg.simple=boolean`
Flag that indicates if DMG customization steps that depend on executing AppleScript code are skipped. Set to `true` to skip the steps. When set to `true`, the disk window doesn't have a background image, and the icons aren't moved into place. If the `systemWide` argument is also set to `true`, then a symbolic link to the root Applications folder is added to the DMG file. If the `systemWide` argument is set to `false`, then only the application is added to the DMG file, no link to the desktop is added.

macOS PKG Bundler Arguments

The macOS PKG installer presents a wizard and shows the license file specified by the `licenseFile` argument as one of the pages in the wizard. The user must accept the terms before installing the application.

`licenseFile=`*`path`*
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources, for example, `-BlicenseFile=COPYING`.

`mac.signing-key-developer-id-installer=`*`key`*
Name of the signing key used for Developer ID or Gatekeeper signing. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application isn't signed.

`mac.CFBundleVersion=`*`value`*
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `appVersion` argument. If the `appVersion` argument is specified with a valid value and the `mac.CFBundleVersion` argument isn't specified, then the `appVersion` value is used. If neither argument is specified, `100` is used as the version number.

Mac App Store Bundler Arguments

`mac.app-store-entitlements=`*`path`*
Location of the file that contains the entitlements that the application operates under. The file must be in the format specified by Apple. The path to the file can be specified in absolute terms, or relative to the invocation of `javapackager`. If no entitlements are specified, then the application operates in a sandbox that's stricter than the typical applet sandbox, and access to network sockets and all files is prevented.

`mac.signing-key-app=`*`key`*
Name of the application signing key for the Mac App Store. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application isn't signed.

`mac.signing-key-pkg=`*`key`*
Name of the installer signing key for the Mac App Store. If you imported a standard key from the Apple Developer Website, then that key is used by default. If no key can be identified, then the application isn't signed.

`mac.CFBundleVersion=`*`value`*
Version number for the application, used internally. The value must be at least one integer and no more than three integers separated by periods (.) for example, 1.3 or 2.0.1. The value can be different than the value for the `appVersion` argument. If the `appVersion` argument is specified with a valid value and the `mac.CFBundleVersion` argument isn't specified, then the `appVersion` value is used. If neither argument is specified, then `100` is used as the version number. If this version is an upgrade for an existing application, then the value must be greater than previous version number.

Linux Debian Bundler Arguments

The license file specified by the `licenseFile` argument isn't presented to the user in all cases, but the file is included in the application metadata.

`category=category`
Category for the application. See Registered Categories in *Desktop Menu Specification* for examples.

`copyright=string`
Copyright string for the application. This argument is used in the Debian metadata.

`email=address`
Email address used in the Debian Maintainer field.

`licenseFile=path`
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources, for example, `-BlicenseFile=COPYING`.

`licenseType=type`
Short name of the license type, such as `-BlicenseType=Proprietary`, or `"-BlicenseType=GPL v2 + Classpath Exception"`.

`vendor=value`
Corporation, organization, or individual providing the application. This argument is used in the Debian Maintainer field.

Linux RPM Bundler Arguments

`category=category`
Category for the application. See Registered Categories in *Desktop Menu Specification* for examples.

`licenseFile=path`
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources, for example, `-BlicenseFile=COPYING`.

`licenseType=type`
Short name of the license type, such as `-BlicenseType=Proprietary`, or `"-BlicenseType=GPL v2 + Classpath Exception"`.

`vendor=value`
Corporation, organization, or individual providing the application.

Windows EXE Bundler Arguments

`copyright=string`
Copyright string for the application. The string must be a single line no longer than 100 characters. This argument is used in various executable file and registry metadata.

`installdirChooser=boolean`
Flag that indicates if the user can choose the directory in which the application is installed. Set to `true` to show a dialog box for the user to choose the directory. Set to `false` to install the application in the directory indicated by the `systemWide` argument. The default is `false`.

licenseFile=*path*
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources, for example, `-BlicenseFile=COPYING`.

menuHint=*boolean*
Flag that indicates if a shortcut is installed on the Start menu or Start screen. Set to `true` to install the shortcut. The default is `true`.

shortcutHint=*boolean*
Flag that indicates if a shortcut is placed on the desktop. Set to `true` to add a shortcut to the desktop. The default is `false`.

systemWide=*boolean*
Flag that indicates if the application is installed in the `Program Files` directory or in the standard location in the users home directory. Set to `true` to install the application in `Program Files`. Set to `false` to install the application in the user's home directory. The default is `false`.

win.menuGroup=*group*
Menu group in which to install the application when `menuHint` is `true`. This argument is ignored when `menuHint` is `false`.

vendor=*value*
Corporation, organization, or individual providing the application. This argument is used in various executable file and registry metadata.

Windows MSI Bundler Arguments

installdirChooser=*boolean*
Flag that indicates if the user can choose the directory in which the application is installed. Set to `true` to show a dialog box for the user to choose the directory. Set to `false` to install the application in the directory indicated by the `systemWide` argument. The default is `false`.

licenseFile=*path*
Location of the End User License Agreement (EULA) to be presented or recorded by the bundler. The path is relative to the packaged application resources, for example, `-BlicenseFile=COPYING`.

menuHint=*boolean*
Flag that indicates if a shortcut is installed on the Start menu or Start screen. Set to `true` to install the shortcut. The default is `true`.

shortcutHint=*boolean*
Flag that indicates if a shortcut is placed on the desktop. Set to `true` to add a shortcut to the desktop. The default is `false`.

systemWide=*boolean*
Flag that indicates if the application is installed in the Program Files directory or in the standard location in the users home directory. Set to `true` to install the application in Program Files. Set to `false` to install the application in the user's home directory. The default is `true`.

win.menuGroup=*group*
Menu group in which to install the application when `menuHint` is `true`. This argument is ignored when `menuHint` is `false`.

vendor=*value*
Corporation, organization, or individual providing the application. This argument is used in various executable file and registry metadata.

Deprecated Options

The following options are no longer used by the packaging tool and are ignored if present.

-embedCertificates
If present, the certificates will be embedded in the JNLP file. Deprecated `–deploy` option.

-noembedlauncher
If present, the packager will not add the JavaFX launcher classes to the JAR file. Deprecated.

Notes

- A `-v` option can be used with any task command to enable verbose output.
- When the `-srcdir` option is allowed in a command, it can be used more than once. If the `-srcfiles` option is specified, then the files named in the argument are looked for in the location specified in the preceding `-srcdir` option. If there is no `-srcdir` preceding `-srcfiles`, then the directory from which the `javapackager` command is executed is used.

Examples

Example 1 Using the -createjar Command

```
javapackager -createjar -appclass package.ClassName
  -srcdir classes -outdir out -outfile outjar -v
```

Packages the contents of the `classes` directory to `outjar.jar`, and sets the application class to `package.ClassName`.

Example 2 Using the -deploy Command

```
javapackager -deploy -outdir outdir -outfile outfile -width 34 -height 43
  -name AppName -appclass package.ClassName -v -srcdir compiled
```

Generates `outfile.jnlp` and the corresponding `outfile.html` files in `outdir` for the application `AppName`, which is started by `package.ClassName` and has dimensions of 34 by 43 pixels.

Example 3 Using the -makeall Command

> **Note:**
>
> The `-makeall` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release.

```
javapackager -makeall -appclass brickbreaker.Main -name BrickBreaker -width 600
-height 600
```

Does all the packaging work including compilation, `createjar`, and `deploy`.

Example 4 Using the -signjar Command

> **Note:**
>
> The `-signjar` command for the Java Packager tool is deprecated in JDK 9 in preparation for removal in a future release. It also doesn't work with multirelease JAR files. Use the jarsigner tool to sign the JAR file.

```
javapackager -signJar -outdir dist -keyStore sampleKeystore.jks -storePass ****
-alias duke -keypass **** -srcdir dist
```

Signs all of the JAR files in the `dist` directory, attaches a certificate with the specified alias, `keyStore` and `storePass`, and puts the signed JAR files back into the `dist` directory.

Example 5 Using the -deploy Command with Bundler Arguments
Linux:
Generates the native Linux Debian package for running the `BrickBreaker` application as a self-contained application.

```
javapackager -deploy -native deb -Bcategory=Education -BjvmOptions=-Xmx128m
    -BjvmOptions=-Xms128m -outdir packages -outfile BrickBreaker -srcdir dist
    -srcfiles BrickBreaker.jar -appclass brickbreaker.Main -name BrickBreaker
    -title "BrickBreaker demo"
```

Windows:
Generates the native Windows EXE package for running the `BrickBreaker` application as a self-contained application.

```
javapackager -deploy -native exe -BsystemWide=true -BjvmOptions=-Xmx128m
    -BjvmOptions=-Xms128m -outdir packages -outfile BrickBreaker -srcdir dist
    -srcfiles BrickBreaker.jar -appclass brickbreaker.Main -name BrickBreaker
    -title "BrickBreaker demo"
```

8
Java Web Start Tool

You use the Java Web Start command and options to start the reference implementation that starts Java applications and applets hosted on a network.

The following section describes Java Web Start command and options:

- javaws: You use the `javaws` tool command and its options to start Java Web Start.

javaws

You use the `javaws` tool command and its options to start Java Web Start.

Synopsis

```
javaws [run-options] jnlp


javaws [control-options]
```

run-options
The *`run-options`* can be in any order. See Run-Options for the javaws Command.

jnlp
This represents either the path of or the URL of the Java Network Launching Protocol (JNLP) file.

control-options
The *`control-options`* can be in any order. See Control-Options for the javaws Command.

Description

> **Note:**
>
> The `javaws` command isn't available on Oracle Solaris.

The `javaws` command starts Java Web Start, which is the reference implementation of the JNLP file. Java Web Start starts Java applications and applets hosted on a network.

If a JNLP file is specified, then the `javaws` command starts the Java application or applet specified in the JNLP file.

The `javaws` command has a set of options that are supported in the current release. However, the options may be removed in a future release.

See *Java Platform, Standard Edition Deployment Guide* for information about the user and system cache and `deployment.properties` files.

Run-Options for the javaws Command

`-verbose`
Displays additional output.

`-offline`
Runs the application in offline mode.

`-system`
Runs the application from the system cache only.

`-Xnosplash`
Runs without displaying a splash screen.

`-J`*`option`*
Passes the option to the Java Virtual Machine (JVM), where *option* is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. See java.

`-wait`
Starts the `java` process and waits for its exit. The `javaws` tool process does not exit until the application exits. This option doesn't function as described on Windows platforms.

`-open` ***`arguments`***
Replaces the arguments in the JNLP file with `-open` *`arguments`* .

`–print` ***`arguments`***
Replaces the arguments in the JNLP file with `-print` *`arguments`* .

Control-Options for the javaws Command

`-viewer`
Shows the cache viewer in the Java Control Panel.

`-userConfig` ***`property-name`***
Clears the specified deployment property.

`-userConfig` ***`property-name property-value`***
Sets the specified deployment property to the specified value.

`-clearcache`
Removes all noninstalled applications from the cache.

`-uninstall`
Removes all applications from the cache.

`-uninstall` ***`jnlp file`***
Removes the application from the cache.

`-import` ***`import-options jnlp-file`***
Imports the application to the cache. See **Import-Options for the javaws Command** for the list and description of available options.

Import-Options for the javaws Command

`-silent`
Imports silently without the user interface.

`-system`
Imports the application to the system cache.

`-codebase` *`url`*
Retrieves resources from the specified codebase.

`-shortcut`
Installs shortcuts if the user allows a prompt. This option has no effect unless the `-silent` option is also used.

`-association`
Installs associations if the user allows a prompt. This option has no effect unless the `-silent` option is also used.

> **Note:**
>
> The command, `javaws -shortcut -uninstall`, removes both the association as well as the implementation.

9
Monitoring Tools and Commands

You use Java Virtual Machine (JVM) monitoring tools and commands to monitor and manage Java applications and the JVM.

> **Note:**
>
> Tools identified as **Experimental** are unsupported and might not be available in future JDK releases.

The following sections describe the JDK tools and commands used to monitor and manage Java applications and the JVM:

- jconsole: You use the `jconsole` command to start a graphical console to monitor and manage Java applications.
- jps: **Experimental** You use the `jps` command to list the instrumented JVMs on the target system.
- jstat: **Experimental** You use the `jstat` command to monitor JVM statistics. This command is experimental and unsupported.
- jstatd: **Experimental** You use the `jstatd` command to monitor the creation and termination of instrumented Java HotSpot VMs. This command is experimental and unsupported.
- jmc: You use the `jmc` command and its options to launch Java Mission Control. Java Mission Control is a profiling, monitoring, and diagnostics tools suite.

jconsole

You use the `jconsole` command to start a graphical console to monitor and manage Java applications.

Synopsis

```
jconsole [-interval=n] [-notile] [-plugin path] [-version] [connection ... ] [-J
input arguments]

jconsole -help
```

`-interval`
Sets the update interval to *n* seconds (default is 4 seconds).

`-notile`
Doesn't tile the windows for two or more connections.

`-pluginpath` *`path`*
Specifies the path that `jconsole` uses to look up plug-ins. The plug-in *`path`* should contain a provider-configuration file named `META-INF/services/com.sun.tools.jconsole.JConsolePlugin` that contains one line for each plug-in. The line specifies the fully qualified class name of the class implementing the `com.sun.tools.jconsole.JConsolePlugin` class.

`-version`
Prints the program version.

`connection = pid | host:port | jmxURL`

- The `pid` value is the process ID of a target process. The JVM must be running with the same user ID as the user ID running the `jconsole` command.
- The `host:port` values are the name of the host system on which the JVM is running, and the port number specified by the system property `com.sun.management.jmxremote.port` when the JVM was started.
- The `jmxUrl` value is the address of the JMX agent to be connected to as described in JMXServiceURL.

`-J input arguments`
Passes *`input arguments`* to the JVM on which the `jconsole` command is run.

`-help` or `--help`
Displays the help message for the command.

Description

The `jconsole` command starts a graphical console tool that lets you monitor and manage Java applications and virtual machines on a local or remote machine.

On Windows, the `jconsole` command doesn't associate with a console window. It does, however, display a dialog box with error information when the `jconsole` command fails.

jps

You use the `jps` command to list the instrumented JVMs on the target system. This command is experimental and unsupported.

Synopsis

```
jps [ -q ] [ -mlvV ][hostid ]

jps [ -help ]
```

`-q`
Suppresses the output of the class name, JAR file name, and arguments passed to the `main` method, producing a list of only local JVM identifiers.

`-mlvV`

- `-m` displays the arguments passed to the `main` method. The output may be `null` for embedded JVMs.
-

`-l` displays the full package name for the application's `main` class or the full path name to the application's JAR file.

- `-v` displays the arguments passed to the JVM.
- `-V` suppresses the output of the class name, JAR file name, and arguments passed to the `main` method, producing a list of only local JVM identifiers.

`hostid`
The identifier of the host for which the process report should be generated. The `hostid` can include optional components that indicate the communications protocol, port number, and other implementation specific data. See Host Identifier.

`-help`
Displays the help message for the `jps` command.

Description

The `jps` command lists the instrumented Java HotSpot VMs on the target system. The command is limited to reporting information on JVMs for which it has the access permissions.

If the `jps` command is run without specifying a `hostid`, then it searches for instrumented JVMs on the local host. If started with a `hostid`, then it searches for JVMs on the indicated host, using the specified protocol and port. A `jstatd` process is assumed to be running on the target host.

The `jps` command reports the local JVM identifier, or `lvmid`, for each instrumented JVM found on the target system. The `lvmid` is typically, but not necessarily, the operating system's process identifier for the JVM process. With no options, the `jps` command lists each Java application's `lvmid` followed by the short form of the application's class name or jar file name. The short form of the class name or JAR file name omits the class's package information or the JAR files path information.

The `jps` command uses the Java launcher to find the class name and arguments passed to the main method. If the target JVM is started with a custom launcher, then the class or JAR file name, and the arguments to the `main` method aren't available. In this case, the `jps` command outputs the string `Unknown` for the class name, or JAR file name, and for the arguments to the `main` method.

The list of JVMs produced by the `jps` command can be limited by the permissions granted to the principal running the command. The command lists only the JVMs for which the principal has access rights as determined by operating system-specific access control mechanisms.

Host Identifier

The host identifier, or `hostid`, is a string that indicates the target system. The syntax of the `hostid` string corresponds to the syntax of a URI:

```
[protocol:][[//]hostname][:port][/servername]
```

`protocol`
The communications protocol. If the `protocol` is omitted and a `hostname` isn't specified, then the default protocol is a platform-specific, optimized, local protocol. If the protocol is omitted and a host name is specified, then the default protocol is `rmi`.

hostname
A host name or IP address that indicates the target host. If you omit the `hostname` parameter, then the target host is the local host.

port
The default port for communicating with the remote server. If the `hostname` parameter is omitted or the `protocol` parameter specifies an optimized, local protocol, then the `port` parameter is ignored. Otherwise, treatment of the `port` parameter is implementation-specific. For the default `rmi` protocol, the `port` parameter indicates the port number for the `rmiregistry` on the remote host. If the `port` parameter is omitted, and the `protocol` parameter indicates `rmi`, then the default `rmiregistry` port (`1099`) is used.

servername
The treatment of this parameter depends on the implementation. For the optimized, local protocol, this field is ignored. For the `rmi` protocol, this parameter is a string that represents the name of the RMI remote object on the remote host. See the jstatdcommand `-n` option.

Output Format of the jps Command

The output of the `jps` command has the following pattern:

```
lvmid [ [ classname | JARfilename | "Unknown"] [ arg* ] [ jvmarg* ] ]
```

All output tokens are separated by white space. An `arg` value that includes embedded white space introduces ambiguity when attempting to map arguments to their actual positional parameters.

> **Note:**
>
> It's recommended that you don't write scripts to parse `jps` output because the format might change in future releases. If you write scripts that parse `jps` output, then expect to modify them for future releases of this tool.

Examples

This section provides examples of the `jps` command.

List the instrumented JVMs on the local host:

```
jps
18027 Java2Demo.JAR
18032 jps
18005 jstat
```

The following example lists the instrumented JVMs on a remote host. This example assumes that the `jstat` server and either the its internal RMI registry or a separate external `rmiregistry` process are running on the remote host on the default port (port `1099`). It also assumes that the local host has appropriate permissions to access the remote host. This example includes the `-l` option to output the long form of the class names or JAR file names.

```
jps -l remote.domain
3002 /opt/jdk1.7.0/demo/jfc/Java2D/Java2Demo.JAR
2857 sun.tools.jstatd.jstatd
```

The following example lists the instrumented JVMs on a remote host with a nondefault port for the RMI registry. This example assumes that the `jstatd` server, with an internal RMI registry bound to port `2002`, is running on the remote host. This example also uses the `-m` option to include the arguments passed to the `main` method of each of the listed Java applications.

```
jps -m remote.domain:2002
3002 /opt/jdk1.7.0/demo/jfc/Java2D/Java2Demo.JAR
3102 sun.tools.jstatd.jstatd -p 2002
```

jstat

You use the `jstat` command to monitor JVM statistics. This command is experimental and unsupported.

Synopsis

```
jstat generalOptions

jstat -outputOptions [ -t] [-hlines] vmid [interval [count] ]
```

generalOptions
A single general command-line option. See General Options.

outputOptions
An option reported by the `-options` option. One or more output options that consist of a single `statOption`, plus any of the `-t`, `-h`, and `-J` options. See Output Options for the jstat Command.

`-t`
Displays a time-stamp column as the first column of output. The time stamp is the time since the start time of the target JVM.

`-h` *`n`*
Displays a column header every *`n`* samples (output rows), where *`n`* is a positive integer. The default value is `0`, which displays the column header of the first row of data.

vmid
A virtual machine identifier, which is a string that indicates the target JVM. See Virtual Machine Identifier.

interval
The sampling interval in the specified units, seconds (s) or milliseconds (ms). Default units are milliseconds. This must be a positive integer. When specified, the `jstat` command produces its output at each interval.

count
The number of samples to display. The default value is infinity, which causes the `jstat` command to display statistics until the target JVM terminates or the `jstat` command is terminated. This value must be a positive integer.

Description

The `jstat` command displays performance statistics for an instrumented Java HotSpot VM. The target JVM is identified by its virtual machine identifier, or `vmid` option.

The `jstat` command supports two types of options, general options and output options. General options cause the `jstat` command to display simple usage and version information. Output options determine the content and format of the statistical output.

All options and their functionality are subject to change or removal in future releases.

General Options

If you specify one of the general options, then you can't specify any other option or parameter.

`-help`
Displays a help message.

`-options`
Displays a list of static options. See Output Options for the jstat Command.

Output Options for the jstat Command

If you don't specify a general option, then you can specify output options. Output options determine the content and format of the `jstat` command's output, and consist of a single `statOption`, plus any of the other output options (`-h`, `-t`, and `-J`). The `statOption` must come first.

Output is formatted as a table, with columns that are separated by spaces. A header row with titles describes the columns. Use the `-h` option to set the frequency at which the header is displayed. Column header names are consistent among the different options. In general, if two options provide a column with the same name, then the data source for the two columns is the same.

Use the `-t` option to display a time-stamp column, labeled Timestamp as the first column of output. The Timestamp column contains the elapsed time, in seconds, since the target JVM started. The resolution of the time stamp is dependent on various factors and is subject to variation due to delayed thread scheduling on heavily loaded systems.

Use the interval and count parameters to determine how frequently and how many times, respectively, the `jstat` command displays its output.

> **Note:**
>
> Don't write scripts to parse the `jstat` command's output because the format might change in future releases. If you write scripts that parse the `jstat` command output, then expect to modify them for future releases of this tool.

`-statOption`
Determines the statistics information that the `jstat` command displays. The following lists the available options. Use the `-options` general option to display the list of options for a particular platform installation. See Stat Options and Output.
`class`: Displays statistics about the behavior of the class loader.
`compiler`: Displays statistics about the behavior of the Java HotSpot VM Just-in-Time compiler.
`gc`: Displays statistics about the behavior of the garbage collected heap.

`gccapacity`: Displays statistics about the capacities of the generations and their corresponding spaces.
`gccause`: Displays a summary about garbage collection statistics (same as `-gcutil`), with the cause of the last and current (when applicable) garbage collection events.
`gcnew`: Displays statistics about the behavior of the new generation.
`gcnewcapacity`: Displays statistics about the sizes of the new generations and their corresponding spaces.
`gcold`: Displays statistics about the behavior of the old generation and metaspace statistics.
`gcoldcapacity`: Displays statistics about the sizes of the old generation.
`gcmetacapacity`: Displays statistics about the sizes of the metaspace.
`gcutil`: Displays a summary about garbage collection statistics.
`printcompilation`: Displays Java HotSpot VM compilation method statistics.

-JjavaOption
Passes `javaOption` to the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. For a complete list of options, see java.

Stat Options and Output

The following information summarizes the columns that the `jstat` command outputs for each *`statOption`*.

`-class` ***`option`***
Class loader statistics.
`Loaded`: Number of classes loaded.
`Bytes`: Number of KB loaded.
`Unloaded`: Number of classes unloaded.
`Bytes`: Number of KB loaded.
`Time`: Time spent performing class loading and unloading operations.

`-compiler` ***`option`***
Java HotSpot VM Just-in-Time compiler statistics.
`Compiled`: Number of compilation tasks performed.
`Failed`: Number of compilations tasks failed.
`Invalid`: Number of compilation tasks that were invalidated.
`Time`: Time spent performing compilation tasks.
`FailedType`: Compile type of the last failed compilation.
`FailedMethod`: Class name and method of the last failed compilation.

`-gc` ***`option`***
Garbage collected heap statistics.
`S0C`: Current survivor space 0 capacity (KB).
`S1C`: Current survivor space 1 capacity (KB).
`S0U`: Survivor space 0 utilization (KB).
`S1U`: Survivor space 1 utilization (KB).
`EC`: Current eden space capacity (KB).
`EU`: Eden space utilization (KB).
`OC`: Current old space capacity (KB).
`OU`: Old space utilization (KB).
`MC`: Metaspace capacity (KB).
`MU`: Metaspace utilization (KB).
`CCSC`: Compressed class space capacity (KB).
`CCSU`: Compressed class space used (KB).
`YGC`: Number of young generation garbage collection (GC) events.

`YGCT`: Young generation garbage collection time.
`FGC`: Number of full GC events.
`FGCT`: Full garbage collection time.
`GCT`: Total garbage collection time.

`-gccapacity` *`option`*
Memory pool generation and space capacities.
`NGCMN`: Minimum new generation capacity (KB).
`NGCMX`: Maximum new generation capacity (KB).
`NGC`: Current new generation capacity (KB).
`S0C`: Current survivor space 0 capacity (KB).
`S1C`: Current survivor space 1 capacity (KB).
`EC`: Current eden space capacity (KB).
`OGCMN`: Minimum old generation capacity (KB).
`OGCMX`: Maximum old generation capacity (KB).
`OGC`: Current old generation capacity (KB).
`OC`: Current old space capacity (KB).
`MCMN`: Minimum metaspace capacity (KB).
`MCMX`: Maximum metaspace capacity (KB).
`MC`: Metaspace capacity (KB).
`CCSMN`: Compressed class space minimum capacity (KB).
`CCSMX`: Compressed class space maximum capacity (KB).
`CCSC`: Compressed class space capacity (KB).
`YGC`: Number of young generation GC events.
`FGC`: Number of full GC events.

`-gccause` *`option`*
This option displays the same summary of garbage collection statistics as the `-gcutil` option, but includes the causes of the last garbage collection event and (when applicable), the current garbage collection event. In addition to the columns listed for `-gcutil`, this option adds the following columns:
`LGCC`: Cause of last garbage collection
`GCC`: Cause of current garbage collection

`-gcnew` *`option`*
New generation statistics.
`S0C`: Current survivor space 0 capacity (KB).
`S1C`: Current survivor space 1 capacity (KB).
`S0U`: Survivor space 0 utilization (KB).
`S1U`: Survivor space 1 utilization (KB).
`TT`: Tenuring threshold.
`MTT`: Maximum tenuring threshold.
`DSS`: Desired survivor size (KB).
`EC`: Current eden space capacity (KB).
`EU`: Eden space utilization (KB).
`YGC`: Number of young generation GC events.
`YGCT`: Young generation garbage collection time.

`-gcnewcapacity` *`option`*
New generation space size statistics.
`NGCMN`: Minimum new generation capacity (KB).
`NGCMX`: Maximum new generation capacity (KB).
`NGC`: Current new generation capacity (KB).
`S0CMX`: Maximum survivor space 0 capacity (KB).

S0C: Current survivor space 0 capacity (KB).
S1CMX: Maximum survivor space 1 capacity (KB).
S1C: Current survivor space 1 capacity (KB).
ECMX: Maximum eden space capacity (KB).
EC: Current eden space capacity (KB).
YGC: Number of young generation GC events.
FGC: Number of full GC events.

-gcold *option*
Old generation size statistics.
MC: Metaspace capacity (KB).
MU: Metaspace utilization (KB).
CCSC: Compressed class space capacity (KB).
CCSU: Compressed class space used (KB).
OC: Current old space capacity (KB).
OU: Old space utilization (KB).
YGC: Number of young generation GC events.
FGC: Number of full GC events.
FGCT: Full garbage collection time.
GCT: Total garbage collection time.

-gcoldcapacity *option*
Old generation statistics.
OGCMN: Minimum old generation capacity (KB).
OGCMX: Maximum old generation capacity (KB).
OGC: Current old generation capacity (KB).
OC: Current old space capacity (KB).
YGC: Number of young generation GC events.
FGC: Number of full GC events.
FGCT: Full garbage collection time.
GCT: Total garbage collection time.

-gcmetacapacity *option*
Metaspace size statistics.
MCMN: Minimum metaspace capacity (KB).
MCMX: Maximum metaspace capacity (KB).
MC: Metaspace capacity (KB).
CCSMN: Compressed class space minimum capacity (KB).
CCSMX: Compressed class space maximum capacity (KB).
YGC: Number of young generation GC events.
FGC: Number of full GC events.
FGCT: Full garbage collection time.
GCT: Total garbage collection time.

-gcutil *option*
Summary of garbage collection statistics.
S0: Survivor space 0 utilization as a percentage of the space's current capacity.
S1: Survivor space 1 utilization as a percentage of the space's current capacity.
E: Eden space utilization as a percentage of the space's current capacity.
O: Old space utilization as a percentage of the space's current capacity.
M: Metaspace utilization as a percentage of the space's current capacity.
CCS: Compressed class space utilization as a percentage.
YGC: Number of young generation GC events.
YGCT: Young generation garbage collection time.

`FGC`: Number of full GC events.
`FGCT`: Full garbage collection time.
`GCT`: Total garbage collection time.

`-printcompilation` *`option`*
Java HotSpot VM compiler method statistics.
`Compiled`: Number of compilation tasks performed by the most recently compiled method.
`Size`: Number of bytes of byte code of the most recently compiled method.
`Type`: Compilation type of the most recently compiled method.
`Method`: Class name and method name identifying the most recently compiled method. Class name uses a slash (/) instead of a dot (.) as a name space separator. The method name is the method within the specified class. The format for these two fields is consistent with the HotSpot `-XX:+PrintCompilation` option.

Virtual Machine Identifier

The syntax of the `vmid` string corresponds to the syntax of a URI:

```
[protocol:][//]lvmid[@hostname[:port]/servername]
```

The syntax of the `vmid` string corresponds to the syntax of a URI. The `vmid` string can vary from a simple integer that represents a local JVM to a more complex construction that specifies a communications protocol, port number, and other implementation-specific values.

`protocol`
The communications protocol. If the *`protocol`* value is omitted and a host name isn't specified, then the default protocol is a platform-specific optimized local protocol. If the *`protocol`* value is omitted and a host name is specified, then the default protocol is `rmi`.

`lvmid`
The local virtual machine identifier for the target JVM. The `lvmid` is a platform-specific value that uniquely identifies a JVM on a system. The `lvmid` is the only required component of a virtual machine identifier. The `lvmid` is typically, but not necessarily, the operating system's process identifier for the target JVM process. You can use the `jps` command to determine the `lvmid`. Also, you can determine the `lvmid` on Oracle Solaris, Linux, and OS X platforms with the `ps` command, and on Windows with the Windows Task Manager.

`hostname`
A host name or IP address that indicates the target host. If the *`hostname`* value is omitted, then the target host is the local host.

`port`
The default port for communicating with the remote server. If the *`hostname`* value is omitted or the *`protocol`* value specifies an optimized, local protocol, then the *`port`* value is ignored. Otherwise, treatment of the *`port`* parameter is implementation-specific. For the default `rmi` protocol, the port value indicates the port number for the `rmiregistry` on the remote host. If the *`port`* value is omitted and the *`protocol`* value indicates `rmi`, then the default rmiregistry port (1099) is used.

servername
The treatment of the *servername* parameter depends on implementation. For the optimized local protocol, this field is ignored. For the `rmi` protocol, it represents the name of the RMI remote object on the remote host.

Examples

This section presents some examples of monitoring a local JVM with an *lvmid* of 21891.

The gcutil Option

This example attaches to lvmid 21891 and takes 7 samples at 250 millisecond intervals and displays the output as specified by the `-gcutil` option.

The output of this example shows that a young generation collection occurred between the third and fourth sample. The collection took 0.078 seconds and promoted objects from the eden space (E) to the old space (O), resulting in an increase of old space utilization from 66.80% to 68.19%. Before the collection, the survivor space was 97.02% utilized, but after this collection it's 91.03% utilized.

```
jstat -gcutil 21891 250 7
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00  97.02  70.31  66.80  95.52  89.14      7    0.300     0    0.000    0.300
  0.00  97.02  86.23  66.80  95.52  89.14      7    0.300     0    0.000    0.300
  0.00  97.02  96.53  66.80  95.52  89.14      7    0.300     0    0.000    0.300
 91.03   0.00   1.98  68.19  95.89  91.24      8    0.378     0    0.000    0.378
 91.03   0.00  15.82  68.19  95.89  91.24      8    0.378     0    0.000    0.378
 91.03   0.00  17.80  68.19  95.89  91.24      8    0.378     0    0.000    0.378
 91.03   0.00  17.80  68.19  95.89  91.24      8    0.378     0    0.000    0.378
```

Repeat the Column Header String

This example attaches to lvmid 21891 and takes samples at 250 millisecond intervals and displays the output as specified by `-gcnew` option. In addition, it uses the `-h3` option to output the column header after every 3 lines of data.

In addition to showing the repeating header string, this example shows that between the second and third samples, a young GC occurred. Its duration was 0.001 seconds. The collection found enough active data that the survivor space 0 utilization (S0U) would have exceeded the desired survivor size (DSS). As a result, objects were promoted to the old generation (not visible in this output), and the tenuring threshold (TT) was lowered from 31 to 2.

Another collection occurs between the fifth and sixth samples. This collection found very few survivors and returned the tenuring threshold to 31.

```
jstat -gcnew -h3 21891 250
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
  64.0   64.0    0.0   31.7 31  31   32.0    512.0    178.6    249    0.203
  64.0   64.0    0.0   31.7 31  31   32.0    512.0    355.5    249    0.203
  64.0   64.0   35.4    0.0  2  31   32.0    512.0     21.9    250    0.204
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
  64.0   64.0   35.4    0.0  2  31   32.0    512.0    245.9    250    0.204
  64.0   64.0   35.4    0.0  2  31   32.0    512.0    421.1    250    0.204
  64.0   64.0    0.0   19.0 31  31   32.0    512.0     84.4    251    0.204
 S0C    S1C    S0U    S1U   TT MTT  DSS      EC       EU     YGC     YGCT
  64.0   64.0    0.0   19.0 31  31   32.0    512.0    306.7    251    0.204
```

Include a Time Stamp for Each Sample

This example attaches to *lvmid* 21891 and takes 3 samples at 250 millisecond intervals. The `-t` option is used to generate a time stamp for each sample in the first column.

The Timestamp column reports the elapsed time in seconds since the start of the target JVM. In addition, the `-gcoldcapacity` output shows the old generation capacity (OGC) and the old space capacity (OC) increasing as the heap expands to meet allocation or promotion demands. The old generation capacity (OGC) has grown from 11,696 KB to 13,820 KB after the eighty-first full garbage collection (FGC). The maximum capacity of the generation (and space) is 60,544 KB (OGCMX), so it still has room to expand.

```
Timestamp      OGCMN    OGCMX     OGC       OC       YGC   FGC    FGCT    GCT
          150.1   1408.0  60544.0  11696.0  11696.0   194    80    2.874   3.799
          150.4   1408.0  60544.0  13820.0  13820.0   194    81    2.938   3.863
          150.7   1408.0  60544.0  13820.0  13820.0   194    81    2.938   3.863
```

Monitor Instrumentation for a Remote JVM

This example attaches to *lvmid* 40496 on the system named `remote.domain` using the `-gcutil` option, with samples taken every second indefinitely.

The *lvmid* is combined with the name of the remote host to construct a *vmid* of `40496@remote.domain`. This *vmid* results in the use of the `rmi` protocol to communicate to the default `jstatd` server on the remote host. The `jstatd` server is located using the `rmiregistry` command on `remote.domain` that's bound to the default port of the `rmiregistry` command (port 1099).

```
jstat -gcutil 40496@remote.domain 1000
... output omitted
```

jstatd

You use the `jstatd` command to monitor the creation and termination of instrumented Java HotSpot VMs. This command is experimental and unsupported.

Synopsis

```
jstatd [options ]
```

options
This represents the `jstatd` command-line options. See Options for the jstatd Command.

Description

The `jstatd` command is an RMI server application that monitors for the creation and termination of instrumented Java HotSpot VMs and provides an interface to enable remote monitoring tools to attach to JVMs that are running on the local host.

The `jstatd` server requires an RMI registry on the local host. The `jstatd` server attempts to attach to the RMI registry on the default port, or on the port you specify with the `-p port` option. If an RMI registry is not found, then one is created within the `jstatd` application that's bound to the port that's indicated by the `-p port` option or to

the default RMI registry port when the `-p port` option is omitted. You can stop the creation of an internal RMI registry by specifying the `-nr` option.

Options for the jstatd Command

`-nr`
This option does not attempt to create an internal RMI registry within the `jstatd` process when an existing RMI registry isn't found.

`-p` *`port`*
This option sets the port number where the RMI registry is expected to be found, or when not found, created if the `-nr` option isn't specified.

`-n` *`rminame`*
This option sets the name to which the remote RMI object is bound in the RMI registry. The default name is `JStatRemoteHost`. If multiple `jstatd` servers are started on the same host, then the name of the exported RMI object for each server can be made unique by specifying this option. However, doing so requires that the unique server name be included in the monitoring client's `hostid` and `vmid` strings.

`-Joption`
This option passes a Java `option` to the JVM, where the option is one of those described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. See java.

Security

The `jstatd` server can monitor only JVMs for which it has the appropriate native access permissions. Therefore, the `jstatd` process must be running with the same user credentials as the target JVMs. Some user credentials, such as the root user in Oracle Solaris, Linux, and OS X operating systems, have permission to access the instrumentation exported by any JVM on the system. A `jstatd` process running with such credentials can monitor any JVM on the system, but introduces additional security concerns.

The `jstatd` server doesn't provide any authentication of remote clients. Therefore, running a `jstatd` server process exposes the instrumentation export by all JVMs for which the `jstatd` process has access permissions to any user on the network. This exposure might be undesirable in your environment, and therefore, local security policies should be considered before you start the `jstatd` process, particularly in production environments or on networks that aren't secure.

The `jstatd` server installs an instance of `RMISecurityPolicy` when no other security manager is installed, and therefore, requires a security policy file to be specified. The policy file must conform to Default Policy Implementation and Policy File Syntax.

If your security concerns can't be addressed with a customized policy file, then the safest action is to not run the `jstatd` server and use the `jstat` and `jps` tools locally.

Remote Interface

The interface exported by the `jstatd` process is proprietary and guaranteed to change. Users and developers are discouraged from writing to this interface.

Examples

The following are examples of the `jstatd` command. The `jstatd` scripts automatically start the server in the background.

Internal RMI Registry

This example shows how to start a `jstatd` session with an internal RMI registry. This example assumes that no other server is bound to the default RMI registry port (port `1099`).

```
jstatd -J-Djava.security.policy=all.policy
```

External RMI Registry

This example starts a `jstatd` session with an external RMI registry.

```
rmiregistry&
jstatd -J-Djava.security.policy=all.policy
```

This example starts a `jstatd` session with an external RMI registry server on port `2020`.

```
jrmiregistry 2020&
jstatd -J-Djava.security.policy=all.policy -p 2020
```

This example starts a `jstatd` session with an external RMI registry on port 2020 that's bound to `AlternateJstatdServerName`.

```
rmiregistry 2020&
jstatd -J-Djava.security.policy=all.policy -p 2020
    -n AlternateJstatdServerName
```

Stop the Creation of an In-Process RMI Registry

This example starts a `jstatd` session that doesn't create an RMI registry when one isn't found. This example assumes an RMI registry is already running. If an RMI registry isn't running, then an error message is displayed.

```
jstatd -J-Djava.security.policy=all.policy -nr
```

Enable RMI Logging

This example starts a `jstatd` session with RMI logging capabilities enabled. This technique is useful as a troubleshooting aid or for monitoring server activities.

```
jstatd -J-Djava.security.policy=all.policy
    -J-Djava.rmi.server.logCalls=true
```

jmc

You use the `jmc` command to launch Java Mission Control. Java Mission Control is a profiling, monitoring, and diagnostics tools suite.

Synopsis

```
jmc
```

Description

Java Mission Control is a tool for production time profiling and diagnostics for the Java HotSpot JVM. The two main features of Java Mission Control are the Management Console and Java Flight Recorder, but several more features are offered as plug-ins, which can be downloaded from the tool. Java Mission Control is also available as a set of plug-ins for the Eclipse IDE.

10
Web Services Tools and Commands

You can use JDK web services tools and commands to create and manage web service resources.

The following sections describe the JDK web services tools and commands:

- schemagen: You can use the `schemagen` tool and commands to generate a schema for every namespace that's referenced in your Java classes.
- wsgen: You use the `wsgen` command to generate Java API for XML Web Services (JAX-WS) portable artifacts used in JAX-WS web services.
- wsimport: You use the `wsimport` command to generate Java API for XML Web Services (JAX-WS) portable artifacts.
- xjc: You use the `xjc` shell script to compile an XML schema file into fully annotated Java classes.

schemagen

You can use the `schemagen` tool and commands to generate a schema for every namespace that's referenced in your Java classes.

Synopsis

```
schemagen [ options ] java-files
```

options
The command-line options. See Options for the schemagen Tool.

java-files
The Java class files to be processed.

Description

The schema generator creates a schema file for each name space referenced in your Java classes. Currently, you can't control the name of the generated schema files.

Start the schema generator with the appropriate `schemagen` shell script in the `bin` directory for your platform. The current schema generator can process either Java source files or class files.

```
schemagen.sh Foo.java Bar.java ...
Note: Writing schema1.xsd
```

If your Java files reference other classes, then those classes must be accessible on your system `CLASSPATH` environment variable, or they need to be specified in the `schemagen` command line with the class path options. If the referenced files aren't accessible or specified, then you get errors when you generate the schema.

Options for the schemagen Tool

`-d` *`path`*
This option sets the location where the `schemagen` command places processor-generated and `javac`-generated class files.

`-cp` *`path`* or `-classpath` *`path`*
This option sets the location where the `schemagen` command places user-specified files.

`-encoding` *`encoding`*
This option specifies the encoding to use for `apt` or `javac` command invocations.

`-episode` *`file`*
This option generates an episode file for separate compilation.

`-disableXmlSecurity`
This option disables XML security features for usage on XML parsing APIs.

`-version`
This option displays release information.

`-fullversion`
This option displays full version information.

`-help`
This option displays a help message.

wsgen

You use the `wsgen` command to generate Java API for XML Web Services (JAX-WS) portable artifacts used in JAX-WS web services.

Synopsis

```
wsgen [options] SEI
```

`options`
This represents the `wsgen` command-line options. See Options for wsgen.

`SEI`
The web service endpoint implementation (SEI) class to be read.

Description

The `wsgen` command generates JAX-WS portable artifacts used in JAX-WS web services. The tool reads a web service endpoint class and generates all the required artifacts for web service deployment and invocation.

To start the `wsgen` tool, enter the following commands:

- **Oracle Solaris, Linux, and OS X**:

  ```
  export JAXWS_HOME=/pathto/jaxws-ri
  $JAXWS_HOME/bin/wsgen.sh -help
  ```

- **Windows**:

```
 set JAXWS_HOME=c:\pathto\jaxws-ri
%JAXWS_HOME%\bin\wsgen.bat -help
```

Options for wsgen

`-classpath` *`path`* or `-cp` *`path`*
This option sets the location of the input class files.

`-d` *`directory`*
This option sets the location for where to place generated output files.

`-encoding` *`encoding`*
This option specifies the character encoding used by source files.

`-extension`
This option allows the use of vendor extensions. Use of extensions can result in applications that aren't portable or that don't work with other implementations.

`-help`
This option displays a help message about the `wsgen` command.

`-Joption`

`-keep`
This option keeps the generated files.

`-r` *`directory`*
This option with the `-wsdl` option is used to specify where to place generated resource files such as web Services Definition Language (WSDL) files.

`-s` *`directory`*
This option sets the location for where to place generated source files.

`-verbose`
This option displays compiler messages.

`-version`
This option prints release information.

`-fullversion`
This option prints full version information.

`-wsdl[:`*`protocol`*`]`
This is an optional command that generates a WSDL file to review before endpoint deployment. The WSDL file contains a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns.

> **Note:**
>
> You don't have to generate WSDL at development time because the JAX-WS run time environment generates a WSDL file for you when you deploy your service.

By default, the `wsgen` command doesn't generate a WSDL file. The `protocol` value is optional and is used to specify what protocol should be used for the WSDL binding (`wsdl:binding`). Valid protocols are `soap1.1` and `Xsoap1.2`. The default is `soap1.1`. The `Xsoap1.2` protocol isn't standard and can be used only with the `-extension` option.

`-inlineSchemas`
This option produces inline schemas in the generated `wsdl`. This must be used in conjunction with the `-wsdl` option.

`-servicename` *`name`*
This option is used only with the `-wsdl` option to specify a particular WSDL service (`wsdl:service`) name to be generated in the WSDL file, for example: `-servicename "{http://mynamespace/}MyService"`.

`-portname` *`name`*
This option is used only with the `-wsdl` option to specify a particular WSDL port (`wsdl:port`) name to be generated in the WSDL file, for example: `-portname "{http://mynamespace/}MyPort"`.

`-x` *`file`*
This option specifies the External Web Service Metadata XML descriptor.

Extensions of wsgen

`-Xnocompile`
This option doesn't compile generated Java files.

Examples

The following example generates the wrapper classes for `StockService` with `@WebService` annotations inside the stock directory.

```
wsgen -d stock -cp myclasspath stock.StockService
```

The following example generates a Simple Object Access Protocol (SOAP) 1.1 WSDL file and schema for the `stock.StockService` class with `@WebService` annotations.

```
wsgen -wsdl -d stock -cp myclasspath stock.StockService
```

The following example generates a SOAP 1.2 WSDL file.

```
wsgen -wsdl:Xsoap1.2 -d stock -cp myclasspath stock.StockService
```

wsimport

You use the `wsimport` command to generate Java API for XML Web Services (JAX-WS) portable artifacts.

Synopsis

```
wsimport [ options ] wsdl_URI
```

`options`
This represents the `wsimport` command-line options. See Options for the wsimport Command.

wsdl_URI
The file that contains the machine-readable description of how the web service can be called, what parameters it expects, and what data structures it returns.

Description

The `wsimport` command generates the following JAX-WS portable artifacts. These artifacts can be packaged in a WAR file with the Web Services Description Language (WSDL) file and schema documents and the endpoint implementation to be deployed. The `wsimport` command also provides a `wsimport` Ant task.

- Service Endpoint Interface (SEI)
- Service
- Exception class is mapped from `wsdl:fault` (if any)
- Async Response Bean is derived from response `wsdl:message` (if any)
- Java Architecture for XML Binding (JAXB) generated value types (mapped Java classes from schema types)

To start the `wsgen` command, enter the following commands:

- **Oracle Solaris/Linux**:

  ```
  /bin/wsimport.sh -help
  ```

- **Windows**:

  ```
  \bin\wsimport.bat -help
  ```

Options for the wsimport Command

`-b` *`path`*
Specifies external JAX-WS or JAXB binding files. Multiple JAX-WS and Java Architecture for XML Binding (JAXB) binding files can be specified with the `-b` option. You can use these files to customize package names, bean names, and so on.

`-B` *`jaxbOption`*
Passes the `jaxbOption` option to the JAXB schema compiler.

`-catalog` *`file`*
Specifies a catalog file to resolve external entity references. The `-catalog` option supports the TR9401, XCatalog, and OASIS XML Catalog formats.

`-classpath` *`path`* or `-cp` *`path`*
Specifies where to find user class files and `wsimport` extensions.

`-d` *`directory`*
Specifies where to place generated output files.

`-encoding` *`encoding`*
Specifies the character encoding used by the source files.

`-extension`
Allows vendor extensions. Use of extensions can result in applications that aren't portable or that don't work with other implementations.

`-help`
Displays a help message for the `wsimport` command.

`-httpproxy:`*`proxy`*
Specifies an HTTP proxy server. The format is:
`[`*`user`*`[:`*`password`*`]@]`*`proxyHost`*`:`*`proxyPort`*

`-J`*`javacOption`*
Passes this option to `javac`.

`-keep`
Keeps generated files.

`-p` ***`name`***
Specifies a target package *`name`* to override the WSDL file and schema binding customizations, and the default algorithm defined in the specification.

`–m` ***`name`***
Generates `module-info.java` with the given Java module name.

`-quiet`
Suppresses the `wsimport` command output.

`-s` ***`directory`***
Specifies where to place generated source files.

`-target` ***`version`***
Generates code according to the specified JAX-WS specification version. Version 2.0 generates compliant code for the JAX-WS 2.0 specification.

`-verbose`
Displays compiler messages.

`-version`
Prints version information.

`-fullversion`
Prints full version information.

`-wsdllocation` ***`location`***
Specifies the `@WebServiceClient.wsdlLocation` value.

`-clientjar` ***`jarfile`***
Creates the `jar` file of the generated artifacts along with the WSDL metadata required for invoking the web service.

`-generateJWS`
Generates a stubbed Java Web Start (JWS) implementation file.

`-implDestDir` ***`directory`***
Specifies where to generate the JWS implementation file.

`-implServiceName` ***`name`***
Specifies the local portion of service name for generated JWS implementations.

`-implPortName` ***`name`***
Specifies the local portion of the port name for generated JWS implementations.

Multiple JAX-WS and JAXB binding files can be specified using the -b option, and they can be used to customize various things such as package names and bean names.

Extensions for the wsimport Command

-XadditionalHeaders
Maps headers not bound to a request or response message to Java method parameters.

-Xauthfile *file*
Specifies the WSDL URI that identifies the file that contains authorization information. This URI is in the following format:

`http://user-name:password@host-name/web-service-name>?wsdl.`

-Xdebug
Prints debugging information.

-Xno-addressing-databinding
Enables binding of W3C EndpointReferenceType to Java.

-Xnocompile
Doesn't compile the generated Java files.

-XdisableAuthenticator
Disables Authenticator used by the JAX-WS reference implementation. -Xauthfile option will be ignored if set.

-XdisableSSLHostnameVerification
Disables the SSL Hostname verification while fetching wsdl files.

Examples

The following are examples of using the wsimport command:

```
wsimport stock.wsdl -b stock.xml -b stock.xjb

wsimport -d generated http://example.org/stock?wsdl
```

xjc

You use the xjc shell script to compile an XML schema file into fully annotated Java classes.

Synopsis

`xjc [-options] schema file/URL/dir/jar ... [-b bindinfo] ...`

options
This represents the xjc command-line options. See Options for the xjc Command.

schema file/URL/dir/jar
This represents the location of the XML schema file. If dir is specified, then all schema files in it are compiled. If jar is specified, then the /META-INF/sun-jaxb.episode binding file is compiled.
This specifies one or more schema files to compile. If you specify a directory, then the xjc command scans it for all schema files and compiles them.

-b *bindinfo*
The location of the binding files.

> **Note:**
>
> If *dir* is specified, all schema files in it will be compiled. If *jar* is specified, the `/META-INF/sun-jaxb.episode` binding file will be compiled.

Description

Start the binding compiler with the appropriate `xjc` shell script in the `bin` directory for your platform. There's also an Ant task to run the binding compiler.

Options for the xjc Command

-nv
This option disables strict schema validation. This doesn't mean that the binding compiler won't perform any validation, but means that it will perform a less strict validation.
By default, the `xjc` binding compiler performs strict validation of the source schema before processing it.

-extension
This option allows vendor extensions to be used. By default, the `xjc` binding compiler strictly enforces the rules outlined in the Compatibility Rules chapter and Appendix E. 2 of the JAXB Specification. Appendix E.2 defines a set of W3C XML Schema features that aren't completely supported by JAXB v1.0. In some cases, you may be allowed to use them in the `-extension` mode enabled by this switch. In the default (strict) mode, you're also limited to using only the binding customization defined in the specification. By using the `-extension` switch, you'll be allowed to use the JAXB Vendor Extensions.

-b *file/dir*
This option specifies one or more external binding files to process. Each binding file must have its own `-b` switch. The syntax of the external binding files is flexible. You can have a single binding file that contains customization for multiple schemas or you can break the customization into multiple bindings files. For example:
`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings123.xjb`
`xjc schema1.xsd schema2.xsd schema3.xsd -b bindings1.xjb -b bindings2.xjb -b bindings3.xjb`.
In addition, the ordering of the schema files and binding files on the command line doesn't matter.

-d *dir*
This option specifies an alternate output directory instead of the default. The output directory must already exist. The`xjc` binding compiler doesn't create it for you.
By default, the `xjc` binding compiler generates the Java content classes in the current directory.

-p *pkg*
When you specify a target package with this command-line option, it overrides any binding customization for the package name and the default package name algorithm defined in the specification.

-m *name*
This option generates `module-info.java` using the specified Java module name.

-httpproxy *proxy*
This specifies the HTTP or HTTPS proxy in the format *[user[:password]@]proxyHost[:proxyPort]*. The old `-host` and `-port` options are still supported by the RI for backward compatibility, but they are deprecated. The password specified with this option is an argument that's visible to other users who use the top command. For greater security, use the `-httpproxyfile` option.

-httpproxyfile *file*
This option specifies the HTTP or HTTPS proxy with a file. This is same format as the `-httpproxy` option, but the password specified in the file isn't visible to other users.

-classpath *arg*
This option specifies where to find client application class files used by the *jxb:javaType* and `xjc:`*superClass* customization.

-catalog *file*
This option specifies catalog files to resolve external entity references. It supports the TR9401, XCatalog, and OASIS XML Catalog formats.

-readOnly
This option forces the `xjc` binding compiler to mark the generated Java sources as read-only.
By default, the `xjc` binding compiler doesn't write-protect the Java source files that it generates.

-npa
This option suppresses the generation of package-level annotations into `**/package-info.java`. Using this switch causes the generated code to internalize those annotations into the other generated classes.

-no-header
This option suppresses the generation of a file header comment that includes some note and time stamp. Using this makes the generated code more compatible with the `diff` command.

-target [2.0|2.1]
This option generates code in accordance with the specified JAXWS specification version. Defaults to 2.2. The accepted values are `2.0`, `2.1`, and `2.2`.

-encoding *encoding*
This option specifies character encoding for generated source files.

-enableIntrospection
This option enables the correct generation of Boolean getters and setters to enable Bean Introspection APIs.

-disableXmlSecurity
This option disables XML security features when parsing XML documents.

-contentForWildcard
This option generates content property for types with multiple `xs:any` derived elements.

-xmlschema
This option treats input schemas as W3C XML Schema (default). If you don't specify this switch, then your input schemas are treated as though they're W3C XML Schemas.

-dtd
This option treats input schemas as XML DTD (experimental and unsupported). Support for `RELAX NG` schemas is provided as a JAXB Vendor Extension.

-wsdl
This option treats input as WSDL and compiles schemas inside it (experimental and unsupported).

-verbose
This option generates extra verbose output, such as printing informational messages or displaying stack traces upon some errors.

-quiet
This option suppresses compiler output, such as progress information and warnings.

-help
This option displays a brief summary of the compiler switches.

-version
This option displays the compiler version information.

-fullversion
This option displays full version information.

Extensions for the xjc Command

-Xpropertyaccessors
This option uses `XmlAccessType PROPERTY` instead of `FIELD` for generated classes.

-mark-generated
This option marks the generated code with the annotation `@javax.annotation.Generated`.

-Xinject-code
This option injects the specified Java code fragments into the generated code.

-episode *file*
This option generates the specified episode file for separate compilation.

-XLocator
This option causes the generated code to expose Simple API for XML (SAX) Locator information about the source XML in the Java bean instances after unmarshalling.

-Xsync-methods
This option causes all of the generated method signatures to include the `synchronized` keyword.

Deprecated and Removed Options for the xjc Command

`-host` and `-port`
These options are replaced with the `-httpproxy` option. For backward compatibility, these options are supported, but won't be documented and might be removed from future releases.

`-use-runtime`
Because the JAXB 2.0 specification has defined a portable runtime environment, it's no longer necessary for the JAXB reference implementation to generate `**/impl/runtime` packages. Therefore, this switch is obsolete and was removed.

`-source`
The `-source` compatibility switch was introduced in the first JAXB 2.0 Early Access release. This switch is removed from future releases of JAXB 2.0. If you need to generate 1.0.*n* code, then use an installation of the 1.0.*n* codebase.

Compiler Restrictions for the xjc Command

In general, it's safest to compile all related schemas as a single unit with the same binding compiler switches. Keep the following list of restrictions in mind when running the `xjc` command. Most of these issues apply only when you compile multiple schemas with multiple invocations of the `xjc` command.

To compile multiple schemas at the same time, remember the following precedence rules for the target Java package name:

1. The `-p` option has the highest precedence.
2. If there are *`jaxb:package`* customizations.
3. If `targetNamespace` is declared, then apply the `targetNamespace` to the Java package name algorithm defined in the specification.
4. If no `targetNamespace` is declared, then use a hard coded package named `generated`.

You can't have more than one *`jaxb:schemaBindings`* per name space, so it's impossible to have two schemas in the same target name space compiled into different Java packages.

All schemas being compiled into the same Java package must be submitted to the XJC binding compiler at the same time. They can't be compiled independently and work as expected.

Element substitution groups that are spread across multiple schema files must be compiled at the same time.

11
Java Accessibility Utilities and Commands

Java Access Bridge 2.0.2 includes Java accessibility utilities for examining accessible information about the objects in the Java Virtual Machine (JVM) and the component trees in a particular Java Virtual Machine.

The following topics describe the Java accessibility utilities and their commands:

- jaccessinspector: You use the `jaccessinspector` accessibility evaluation tool for the Java Accessibility Utilities API to examine accessible information about the objects in the Java Virtual Machine.
- jaccesswalker: You use the `jaccesswalker` to navigate through the component trees in a particular Java Virtual Machine and presents the hierarchy in a tree view.

jaccessinspector

You use the `jaccessinspector` accessibility evaluation tool for the Java Accessibility Utilities API to examine accessible information about the objects in the Java Virtual Machine.

Description

The `jaccessinspector` tool lets you select different methods for examining the object accessibility information::

- When events occur such as a change of focus, mouse movement, property change, menu selection, and the display of a popup menu
- When you press the F1 key when the mouse is over an object, or F2 when the mouse is over a window

After an object has been selected for examination, the `jaccessinspector` tool displays the results of calling Java Accessibility API methods on that object.

Running the jaccessinspector Tool

To use the `jaccessinspector` tool, launch the `jaccessinspector` Windows application after launching a Java application. For example, to launch `jaccessinspector` , run one of the following Windows applications:

> **Note:**
>
> The no-suffix version is installed with 64 bit Java.
>
> `JAVA_HOME` is an environment variable and it should be set to the path of the JDK or JRE, for example, `c:\Program Files\Java\jdk-9`.

- 64-bit Windows:

  - `%JAVA_HOME%\bin\jaccessinspector.exe`: Inspects a Java application as if `jaccessinspector` were a 64-bit assistive technology application
  - `%JAVA_HOME%\bin\jaccessinspector-32.exe`: Inspects a Java application as if `jaccessinspector` were a 32-bit assistive technology application

You now have two windows open: The Java application window and the `jaccessinspector` window. The `jaccessinspector` window contains five menus:

- File Menu
- UpdateSettings Menu
- JavaEvents Menu
- AccessibilityEvents Menu
- Options Menu

The items in **UpdateSettings**, **JavaEvents**, and **AccessibilityEvents** menus let you query Java applications in a variety of ways.

File Menu

This section describes the **File** menu items.

AccessBridge DLL Loaded
Enables and disables AccessBridge DLL Loaded.

Exit
Exits from the tool.

UpdateSettings Menu

This section describes the **UpdateSettings** menu items.

Update from Mouse
Determines the x- and y-coordinates of the mouse (assuming the *jaccessinspector* tool window is topmost) when the mouse has stopped moving, and then queries the Java application for the accessible object underneath the mouse, dumping the output into the `jaccessinspector` window.

Update with F2 (Mouse HWND)
Determines the x- and y-coordinates of the mouse (assuming the *jaccessinspector* tool window is topmost), and then queries the Java application for the accessible object of the HWND underneath the mouse, dumping the output into the `jaccessinspector` window.

Update with F1 (Mouse Point)
Determines the x- and y-coordinates of the mouse (assuming the *jaccessinspector* tool window is topmost), and then queries the Java application for the accessible object underneath the cursor, dumping the output into the `jaccessinspector` window.

JavaEvents Menu

This section describes the **JavaEvents** menu items.

Track Mouse Events
Registers with the Java application all Java Mouse Entered events, and upon receiving one, queries the object that was entered by the cursor and dumps the output into the `jaccessinspector` window.

> **Note:**
>
> If the mouse is moved quickly, then there may be some delay before the displayed information is updated.

Track Focus Events
Registers with the Java application all Java Focus Gained events, and upon receiving an event, queries the object that received the focus and dumps the output into the `jaccessinspector` window.

Track Caret Events
Register with the Java application all Java Caret Update events, and upon receiving an event, queries the object in which the caret was updated, and dumps the output into the `jaccessinspector` window.

> **Note:**
>
> Because objects that contain carets are almost by definition objects that are rich text objects, this won't seem as responsive as the other event tracking options. In real use, one would make fewer accessibility calls in Caret Update situations (for example, just get the new letter, word, sentence at the caret location), which would be significantly faster.

Track Menu Selected | Deselected | Canceled Events
Registers with the Java application all Menu events, and upon receiving an event, queries the object in which the caret was updated, and dumps the output into the `jaccessinspector` window.

Track Popup Visible | Invisible | Cancelled Events
Registers with the Java application all Popup Menu events, and upon receiving an event, queries the object in which the caret was updated, and dumps the output into the `jaccessinspector` window.

Track Shutdown Events
Registers with the Java application to receive a Property Changed event when a Java application terminates.

AccessibilityEvents Menu

This section describes the **AccessibilityEvents** menu items.

> **Note:**
>
> The items listed in the **AccessibilityEvents** menu are the most important for testing applications, especially for assistive technology applications.

Track Name Property Events
Registers with the Java application all Java Property Changed events specifically on accessible objects in which the Name property has changed, and upon receiving an event, dumps the output into the scrolling window, along with information about the property that changed.

Track Description Property Events
Register with the Java application for all Java Property Changed events specifically on accessible objects in which the Description property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track State Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the State property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track Value Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the Value property has changed, and upon receiving an event, dumps the output into the scrolling window, along with information about the property that changed.

Track Selection Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the Selection property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track Text Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the Text property has changed, and upon receiving one event, dump the output into the `jaccessinspector` window, along with information about the property that changed.

Track Caret Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the Caret property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track VisibleData Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the VisibleData property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track Child Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the Child property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track Active Descendent Property Events
Register with the Java application all Java Property Changed events specifically on accessible objects in which the Active Descendent property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Track Table Model Change Property Events
Register with the Java application all Property Changed events specifically on accessible objects in which the Table Model Change property has changed, and upon receiving an event, dumps the output into the `jaccessinspector` window, along with information about the property that changed.

Options Menu

This section describes the **Options** menu items.

Monitor the same events as JAWS
Enables monitoring of only the events also monitored by JAWS.

Monitor All Events
Enables monitoring of all events in the `jaccessinspector` window.

Reset All Events
Resets the selected Options to the default settings.

Go To Message
Opens the Go To Message dialog that lets you display a logged message by entering its message number.

Clear Message History
Clears the history of logged messages from the `jaccessinspector` window.

jaccesswalker

You use the `jaccesswalker` to navigate through the component trees in a particular Java Virtual Machine and presents the hierarchy in a tree view.

Description

You select a node in the tree, and from the **Panels** menu, you select **Accessibility API Panel**. The `jaccesswalker` tool shows you the accessibility information for the object in the window.

Running the jaccesswalker Tool

To use `jaccesswalker`, launch the `jaccesswalker` Windows application after launching a Java application. For example, to launch `jaccesswalker`, run one of the following Windows applications:

> **Note:**
>
> - The no-suffix version is installed with 64 bit Java.
> - `JAVA_HOME` is an environment variable and it should be set to the path of the JDK or JRE, for example `c:\Program Files\Java\jdk-9` .

- 64-bit Windows:
  - `%JAVA_HOME%\bin\jaccesswalker.exe:` Inspects a Java application as if `jaccesswalker` were a 64-bit assistive technology application
  - `%JAVA_HOME%\bin\jaccesswalker-32.exe:` Inspects a Java application as if `jaccesswalker` were a 32-bit assistive technology application

You now have two windows open: The Java application window, and the window for the `jaccesswalker` tool. There are two tasks that you can do with `jaccesswalker`. You can build a tree view of the Java applications' GUI hierarchy, and you can query the Java Accessibility API information of a particular element in the GUI hierarchy.

Building the GUI Hierarchy

From the **File** menu, select **Refresh Tree** menu. The `jaccesswalker` tool builds a list of the top-level windows belonging to Java applications and applets. The tool then recursively queries the elements in those windows, and builds a tree of all of the GUI components in all of the Java applications and applets in all of the JVMs running in the system.

Examining a GUI Component

After a GUI tree is built, you can view detailed accessibility information about an individual GUI component by selecting it in the tree, then selecting **Panels,** and then **Display Accessibility Information**.

12

Troubleshooting Tools and Commands

You use JDK troubleshooting tools and commands to troubleshoot Java applications and the Java Virtual Machine (JVM).

> **Note:**
>
> Tools identified as **Experimental** are unsupported and might not be available in future JDK releases.

The following sections describe the JDK troubleshooting tools and commands:

- jcmd: You use the `jcmd` utility to send diagnostic command requests to a running Java Virtual Machine (JVM).
- jdb: You use the `jdb` command and it's options to find and fix bugs in Java platform programs.
- jhsdb: You use the `jhsdb` tool to attach to a Java process or to launch a postmortem debugger to analyze the content of a core dump from a crashed Java Virtual Machine (JVM).
- jinfo: **Experimental** You use the `jinfo` command to generate Java configuration information for a specified Java process. This command is experimental and unsupported.
- jmap: **Experimental** You use the `jmap` command to print details of a specified process. This command is experimental and unsupported.
- jstack: **Experimental** You use the `jstack` command to print Java stack traces of Java threads for a specified Java process. This command is experimental and unsupported.

jcmd

You use the `jcmd` utility to send diagnostic command requests to a running Java Virtual Machine (JVM).

Synopsis

```
jcmd [pid | main-class] command...|PerfCounter.print|
-f filename

jcmd -l

jcmd -h
```

> **Note:**
>
> The Java Flight Recorder (JFR) used with the `jcmd` utility is a commercial product and must be enabled before it is used. Once the JVM is running, the `jcmd` command `VM.unlock_commercial_features` is used to unlock commercial features and enable use of the JFR commands described in Commands for jcmd.

pid
When used, the `jcmd` utility sends the diagnostic command request to the process ID for the Java process.

main-class
When used, the `jcmd` utility sends the diagnostic command request to all Java processes with the specified name of the main class.

command
The *command* must be a valid `jcmd` command for the selected JVM. The list of available commands for `jcmd` is obtained by running the `help` command (`jcmd` *pid* `help`) where *pid* is the process ID for the running Java process. If the *pid* is `0`, commands will be sent to all Java processes. The main class argument will be used to match, either partially or fully, the class used to start Java. If no options are given, it lists the running Java process identifiers with the main class and command-line arguments that were used to launch the process (the same as using `-l`).

`Perfcounter.print`
Prints the performance counters exposed by the specified Java process.

`–f` *filename*
Reads and executes commands from a specified file,*filename* .

`-l`
Displays the list of running Java Virtual Machine process identifiers with the main class and command-line arguments that were used to launch the process.

`-h`
Displays the`jcmd` utility's command-line help.

Description

The `jcmd` utility is used to send diagnostic command requests to the JVM. It must be used on the same machine on which the JVM is running, and have the same effective user and group identifiers that were used to launch the JVM. Each diagnostic command has its own set of arguments. To display the description, syntax, and a list of available arguments for a diagnostic command, use the name of the command as the argument. For example

```
jcmd pid help command
```

If arguments contain spaces, then you must surround them with single or double quotation marks (`'` or `"`). In addition, you must escape single or double quotation marks with a backslash (\) to prevent the operating system shell from processing quotation marks. Alternatively, you can surround these arguments with single

quotation marks and then with double quotation marks (or with double quotation marks and then with single quotation marks).

If you specify the process identifier (*pid*) or the main class (*main-class*) as the first argument, then the `jcmd` utility sends the diagnostic command request to the Java process with the specified identifier or to all Java processes with the specified name of the main class. You can also send the diagnostic command request to all available Java processes by specifying `0` as the process identifier.

If you run `jcmd` without arguments or with the `-l` option, it prints the list of running Java process identifiers with the main class and command-line arguments that were used to launch the process. Running `jcmd` with the `-h` or `-help` option prints the tool's help message.

Use one of the following as the diagnostic command request:

- `Perfcounter.print`
- `-f` *filename*
- *command* [*arguments*]

Commands for jcmd

The *command* must be a valid `jcmd` diagnostic command for the selected JVM. The list of available commands for `jcmd` is obtained by running the `help` command (`jcmd` *pid* `help`) where *pid* is the process ID for the running Java process. If the *pid* is `0`, commands will be sent to all Java processes. The main class argument will be used to match, either partially or fully, the class used to start Java. If no options are given, it lists the running Java process identifiers with the main class and command-line arguments that were used to launch the process (the same as using `-l`).

The following commands are available:

`help [`*`options`*`][`*`arguments`*`]`
For more information about a specific command.
arguments:

- *command name* : The name of the command for which we want help (STRING, no default value)

> **Note:**
>
> The following *options* must be specified using either *key* or *key=value* syntax.

options:

- –all: (Optional) Show help for all commands (BOOLEAN, false) .

`Compiler.codecache`
Prints code cache layout and bounds.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`Compiler.codelist`
Prints all compiled methods in code cache that are alive.

Impact: Medium
Permission: `java.lang.management.ManagementPermission(monitor)`

`Compiler.queue`
Prints methods queued for compilation.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`Compiler.directives_add <filename> arguments`
Adds compiler directives from a file.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`
`arguments`:
filename : The name of the directives file (STRING, no default value)

`Compiler.directives_clear`
Remove all compiler directives.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`Compiler.directives_print`
Prints all active compiler directives.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`Compiler.directives_remove`
Remove latest added compiler directive.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`GC.class_histogram [options]`
Provides statistics about the Java heap usage.
Impact: High — depends on Java heap size and content.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `-all` : (Optional) Inspects all objects, including unreachable objects (BOOLEAN, false)

`GC.class_stats [options] [arguments]`
Provide statistics about Java class meta data.
Impact: High — depends on Java heap size and content.

> **Note:**
>
> The *`options`* must be specified using either *`key`* or *`key=value`* syntax.

options:

- `-all`: (Optional) Shows all columns (BOOLEAN, false)
- `-csv`: (Optional) Prints in CSV (comma-separated values) format for spreadsheets (BOOLEAN, false)
- `-help`: (Optional) Shows the meaning of all the columns (BOOLEAN, false)

arguments

- *columns*: (Optional) Comma-separated list of all the columns to be shown. If not specified, the following columns are shown:
  - InstBytes
  - KlassBytes
  - CpAll
  - annotations
  - MethodCount
  - Bytecodes
  - MethodAll
  - ROAll
  - RWAll
  - Total

  (STRING, no default value)

`GC.finalizer_info`
Provides information about the Java finalization queue.
Impact: Medium
Permission: `java.lang.management.ManagementPermission(monitor)`

`GC.heap_dump [options] [arguments]`
Generates a HPROF format dump of the Java heap.
Impact: High — depends on the Java heap size and content. Request a full GC unless the `-all` option is specified.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *options* must be specified using either *key* or *key=value* syntax.

options:

- `-all` : [optional] Dump all objects, including unreachable objects (BOOLE AN, false)

arguments:

- *filename*: The name of the dump file (STRING, no default value)

`GC.heap_info`
Provides generic Java heap information.

Impact: Medium
Permission: `java.lang.management.ManagementPermission(monitor)`

`GC.run`
Calls `java.lang.System.gc()`.
Impact: Medium — depends on the Java heap size and content.

`GC.run_finalization`
Calls `java.lang.System.runFinalization()`.
Impact: Medium — depends on the Java content.

`JFR.check [options]`
Checks running JFR recording(s).
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `name` : (Optional) Recording text, e.g. \"My Recording\" or omit to see all recordings (STRING, no default value)
- `verbose` : (Optional) Print event settings for the recording(s) (BOOLEAN, false)

`JFR.configure [options]`
Configures the Java Flight Recorder (JFR)
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `repositorypath`: (Optional) Sets the path to the repository, such as `\My Repository\` (STRING, no default value)
- `dumppath`: (Optional) Sets the path to dump, such as, `\"My Dump path\"` (STRING, no default value)
- `stackdepth`: (Optional) Sets the stack Depth (JLONG, 64)
- `globalbuffercount`: (Optional) Sets the number of global buffers, (JLONG, 32)
- `globalbuffersize`: (Optional) Sets the size of a global buffers, (JLONG, 524288)
- `thread_buffer_size`: (Optional) Sets the size of a thread buffer (JLONG, 8192)
- `memorysize`: (Optional) Sets the overall memory size, (JLONG, 16777216)

- `threadbufferstodisk`: (Optional) Sets the thread buffers to be written directly to disk (BOOLEAN, false)
- `maxchunksize`: (Optional) Sets the size of an individual disk chunk (JLONG, 12582 912)
- `samplethreads`: (Optional) Activates thread sampling (BOOLEAN, true)

`JFR.dump [options]`
Copies contents of a JFR recording to file. Either the name or the recording id must be specified.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `name` : (Optional) Recording name, e.g. \"My Recording\" (STRING, no default value)
- `filename` : Copy recording data to file, i.e \"C:\Users\user\My Recording.jfr\" (STRING, no default value)
- `path-to-gc-roots`: (Optional) Collects path to GC roots (BOOLEAN, false)

`JFR.start [options]`
Starts a new JFR recording.
Impact: Medium: Depending on the settings for a recording, the impact can range from low to high.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `name` : (Optional) Name that can be used to identify recording, e.g. \"My Recording \" (STRING, no default value)
- `settings` : (Optional) Settings file(s), e.g. profile or default. See JRE _HOME/lib/jfr (STRING SET, no default value)
- `delay` : (Optional) Delay recording start with (s)econds, (m)inutes), (h) ours), or (d)ays, e.g. 5h. (NANOTIME, 0)
- `duration` : (Optional) Duration of recording in (s)econds, (m)inutes, (h) ours, or (d)ays, e.g. 300s. (NANOTIME, 0)
- `disk` : (Optional) Recording should be persisted to disk (BOOLEAN, no def ault value)

- `filename` : (Optional) Resulting recording filename, e.g. \"C:\Users\user \My Recording.jfr\" (STRING, no default value)
- `maxage` : (Optional) Maximum time to keep recorded data (on disk) in (s)e conds, (m)inutes, (h)ours, or (d)ays, e.g. 60m, or 0 for no limit (NANOTIME, 0)
- `maxsize` : (Optional) Maximum amount of bytes to keep (on disk) in (k)B, (M)B or (G)B, e.g. 500M, or 0 for no limit (MEMORY SIZE, 0)
- `dumponexit` : (Optional) Dump running recording when JVM shuts down (BOOL EAN, no default value)
- `path-to-gc-roots`: (Optional) Collects path to GC roots (BOOLEAN, false)

`JFR.stop [`*`options`*`]`
Stops a JFR recording
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `name`: (Optional) Recording text, such as,

  `\"My Recording\"` (STRING, no default value)
- `filename`: (Optional) Copy recording data to file, such as,

  `. \"C:\Users\user\ My Recording.jfr\"` (STRING, no default value)

`JVMTI.agent_load [`*`arguments`*`]`
Loads JVMTI native agent.
Impact: Low
Permission: `java.lang.management.ManagementPermission(control)`
`arguments`:

- *library path* : Absolute path of the JVMTI agent to load. (STRING, no default value)
- *agent option* : (Optional) Option string to pass the agent. (STRING, no default value)

`JVMTI.data_dump`
Signals the JVM to do a data-dump request for JVMTI.
Impact: High
Permission: `java.lang.management.ManagementPermission(monitor)`

`ManagementAgent.start [`*`options`*`]`
Starts remote management agent.
Impact: Low — no impact

> **Note:**
>
> The following *options* must be specified using either *key* or *key=value* syntax.

options:

- config.file : (Optional) Sets com.sun.management.config.file (STRING, no default value)
- jmxremote.host : (Optional) Sets com.sun.management.jmxremote.host (STRIN G, no default value)
- jmxremote.port : (Optional) Sets com.sun.management.jmxremote.port (STRIN G, no default value)
- jmxremote.rmi.port : (Optional) Sets com.sun.management.jmxremote.rmi.port (STRING, no default value)
- jmxremote.ssl : (Optional) Sets com.sun.management.jmxremote.ssl (STRING, no default value)
- jmxremote.registry.ssl : (Optional) Sets com.sun.management.jmxremote.registry.ssl (STRING, no default value)
- jmxremote.authenticate : (Optional) Sets com.sun.management.jmxremote.authenticate (STRING, no default value)
- jmxremote.password.file : (Optional) Sets com.sun.management.jmxremote.password.file (STRING, no default value)
- jmxremote.access.file : (Optional) Sets com.sun.management.jmxremote.acce ss.file (STRING, no default value)
- jmxremote.login.config : (Optional) Sets com.sun.management.jmxremote.log in.config (STRING, no default value)
- jmxremote.ssl.enabled.cipher.suites : (Optional) Sets com.sun.management.
- jmxremote.ssl.enabled.cipher.suite: (STRING, no default value)
- jmxremote.ssl.enabled.protocols : (Optional) Sets com.sun.management.jmxr emote.ssl.enabled.protocols (STRING, no default value)
- jmxremote.ssl.need.client.auth : (Optional) Sets com.sun.management.jmxre mote.need.client.auth (STRING, no default value)
- jmxremote.ssl.config.file : (Optional) Sets com.sun.management.jmxremote. ssl_config_file (STRING, no default value)
- jmxremote.autodiscovery : (Optional) Sets com.sun.management.jmxremote.au todiscovery (STRING, no default value)
- jdp.port : (Optional) Sets com.sun.management.jdp.port (INT, no default v alue)
- jdp.address : (Optional) Sets com.sun.management.jdp.address (STRING, no default value)
- jdp.source_addr : (Optional) Sets com.sun.management.jdp.source_addr (STR ING, no default value)
- jdp.ttl : (Optional) Sets com.sun.management.jdp.ttl (INT, no default value)

- `jdp.pause` : (Optional) Sets `com.sun.management.jdp.pause` (INT, no default value)
- `jdp.name` : (Optional) Sets `com.sun.management.jdp.name` (STRING, no defaul t value)

`ManagementAgent.start_local`
Starts the local management agent.
Impact: Low —no impact

`ManagementAgent.status`
Print the management agent status.
Impact: Low — no impact
Permission: `java.lang.management.ManagementPermission(monitor)`

`ManagementAgent.stop`
Stops the remote management agent.
Impact: Low — no impact

`Thread.print [options]`
Prints all threads with stacktraces.
Impact: Medium — depends on the number of threads.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `-l` : (Optional) Prints `java.util.concurrent` locks (BOOLEAN, false)

`VM.check_commercial_features`
Display status of commercial features
Impact: Low — no impact

`VM.unlock_commercial_features`
Unlock commercial features
Impact: Low — no impact
Permission: `java.lang.management.ManagementPermission(control)`

`VM.classloader_stats`
Prints statistics about all ClassLoaders.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`VM.class_hierarchy [options] [arguments]`
Prints a list of all loaded classes, indented to show the class hierarchy. The name of each class is followed by the ClassLoaderData* of its ClassLoader, or "null " if it is loaded by the bootstrap class loader.
Impact: Medium — depends on the number of loaded classes.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *options* must be specified using either *key* or *key=value* syntax.

options:

- `-i`: (Optional) Inherited interfaces should be printed. (BOOLEAN, false)
- `-s`: (Optional) If a class name is specified, it prints the subclasses. If the class name is not specified, only the superclasses are printed. (BOOLEAN, false)

arguments

- *classname*: (Optional) The name of the class whose hierarchy should be printed. If not specified, all class hierarchies are printed. (STRING, no default value)

`VM.command_line`
Prints the command line used to start this VM instance.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`VM.dynlibs`
Prints the loaded dynamic libraries.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`VM.info`
Prints information about the JVM environment and status.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

`VM.log [options]`
Lists current log configuration, enables/disables/configures a log output, or ro tates all logs.
Impact: Low
`Permission: java.lang.management.ManagementPermission(control)`
options:

> **Note:**
>
> The following *options* must be specified using either *key* or *key=value* syntax.

- `output`: (Optional) The name or index (#) of output to configure. (STRING, no default value)
- `output_options`: (Optional) Options for the output. (STRING, no default value)
- `what`: (Optional) Configures what tags to log. (STRING, no default value)
- `decorators`: (Optional) Configures which decorators to use. Use 'none' o r an empty value to remove all. (STRING, no default value)
- `disable`: (Optional) Turns off all logging and clears the log configurat ion. (BOOLEAN, no default value)

- `list`: (Optional) Lists current log configuration. (BOOLEAN, no default value)
- `rotate`: (Optional) Rotates all logs. (BOOLEAN, no default value)

`VM.flags [options]`
Prints the VM flag options and their current values.
Impact: Low
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `-all` : (Optional) Prints all flags supported by the VM (BOOLEAN, false).

`VM.native_memory [options]`
Prints native memory usage
Impact: Medium
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `summary`: (Optional) Requests runtime to report current memory summary, which includes total reserved and committed memory, along with memory usage summary by each subsystem. (BOOLEAN, false)
- `detail`: (Optional) Requests runtime to report memory allocation >= 1K by each callsite. (BOOLEAN, false)
- `baseline`: (Optional) Requests runtime to baseline current memory usage, so it can be compared against in later time. (BOOLEAN, false)
- `summary.diff`: (Optional) Requests runtime to report memory summary comparison against previous baseline. (BOOLEAN, false)
- `detail.diff` : (Optional) Requests runtime to report memory detail comparison against previous baseline, which shows the memory allocation activities at different callsites. (BOOLEAN, false)
- `shutdown`: (Optional) Requests runtime to shutdown itself and free the memory used by runtime. (BOOLEAN, false)
- `statistics`: (Optional) Prints tracker statistics for tuning purpose. (BOOLEAN, false)
- `scale` : (Optional) Memory usage in which scale, KB, MB or GB (STRING, KB)

`VM.print_touched_methods`
Prints all methods that have ever been touched during the lifetime of this JVM.

Impact: Medium — depends on Java content.

`VM.set_flag [arguments]`
Sets the VM flag option by using the provided value.
Impact: Low
Permission: `java.lang.management.ManagementPermission(control)`
`arguments`:

- *flag name* : The name of the flag that you want to set (STRING, no default value)
- *string value* : (Optional) The value that you want to set (STRING, no default value)

`VM.stringtable [options]`
Dumps the string table.
Impact: Medium — depends on the Java content.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `-verbose` : (Optional) Dumps the content of each string in the table (BOOLEAN, false)

`VM.symboltable [options]`
Dumps the symbol table.
Impact: Medium — depends on the Java content.
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax).

`options`:

- `-verbose` : (Optional) Dumps the content of each symbol in the table (BOOLEAN, false)

`VM.systemdictionary`
Prints the statistics for dictionary hashtable sizes and bucket length.
Impact: Medium
Permission: `java.lang.management.ManagementPermission(monitor)`

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `verbose` : (Optional) Dump the content of each dictionary entry for all class loaders (BOOLEAN, false) .

`VM.system_properties`
Prints the system properties.
Impact: Low
Permission: `java.util.PropertyPermission(*, read)`

`VM.uptime [options]`
Prints the VM uptime.
Impact: Low

> **Note:**
>
> The following *`options`* must be specified using either *`key`* or *`key=value`* syntax.

`options`:

- `-date` : (Optional) Adds a prefix with the current date (BOOLEAN, false)

`VM.version`
Prints JVM version information.
Impact: Low
Permission: `java.util.PropertyPermission(java.vm.version, read)`

jdb

You use the `jdb` command and it's options to find and fix bugs in Java platform programs.

Synopsis

`jdb [options] [classname] [arguments]`

`options`
This represents the `jdb` command-line options. See Options for the jdb command.

`classname`
This represents the name of the main class to debug.

`arguments`
This represents the arguments that are passed to the `main()` method of the class.

Description

The Java Debugger (JDB) is a simple command-line debugger for Java classes. The `jdb` command and its options call the JDB. The `jdb` command demonstrates the Java Platform Debugger Architecture and provides inspection and debugging of a local or remote JVM.

Start a JDB Session

There are many ways to start a JDB session. The most frequently used way is to have the JDB launch a new JVM with the main class of the application to be debugged. Do this by substituting the `jdb` command for the `java` command in the command line. For

example, if your application's main class is `MyClass`, then use the following command to debug it under the JDB:

```
jdb MyClass
```

When started this way, the `jdb` command calls a second JVM with the specified parameters, loads the specified class, and stops the JVM before executing that class's first instruction.

Another way to use the `jdb` command is by attaching it to a JVM that's already running. Syntax for starting a JVM to which the `jdb` command attaches when the JVM is running is as follows. This loads in-process debugging libraries and specifies the kind of connection to be made.

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n MyClass
```

You can then attach the `jdb` command to the JVM with the following command:

```
jdb -attach 8000
```

8000 is the address of the running JVM.

The `MyClass` argument isn't specified in the `jdb` command line in this case because the `jdb` command is connecting to an existing JVM instead of launching a new JVM.

There are many other ways to connect the debugger to a JVM, and all of them are supported by the `jdb` command. The Java Platform Debugger Architecture has additional documentation on these connection options.

Breakpoints

Breakpoints can be set in the JDB at line numbers or at the first instruction of a method, for example:

- The command `stop at MyClass:22` sets a breakpoint at the first instruction for line 22 of the source file containing `MyClass`.
- The command `stop in java.lang.String.length` sets a breakpoint at the beginning of the method `java.lang.String.length`.
- The command `stop in MyClass.<clinit>` uses `<clinit>` to identify the static initialization code for `MyClass`.

When a method is overloaded, you must also specify its argument types so that the proper method can be selected for a breakpoint. For example, `MyClass.myMethod(int,java.lang.String)` or `MyClass.myMethod()`.

The `clear` command removes breakpoints using the following syntax: `clear MyClass:45`. Using the `clear` or `stop` command with no argument displays a list of all breakpoints currently set. The `cont` command continues execution.

Stepping

The `step` command advances execution to the next line whether it's in the current stack frame or a called method. The `next` command advances execution to the next line in the current stack frame.

Exceptions

When an exception occurs for which there isn't a `catch` statement anywhere in the throwing thread's call stack, the JVM typically prints an exception trace and exits.

When running under the JDB, however, control returns to the JDB at the offending throw. You can then use the `jdb` command to diagnose the cause of the exception.

Use the `catch` command to cause the debugged application to stop at other thrown exceptions, for example: `catch java.io.FileNotFoundException` or `catch mypackage.BigTroubleException`. Any exception that's an instance of the specified class or subclass stops the application at the point where the exception is thrown.

The `ignore` command negates the effect of an earlier `catch` command. The `ignore` command doesn't cause the debugged JVM to ignore specific exceptions, but only to ignore the debugger.

Options for the jdb command

When you use the `jdb` command instead of the `java` command on the command line, the `jdb` command accepts many of the same options as the `java` command..

The following options are accepted by the `jdb` command:

`-help`
Displays a help message.

`-sourcepath` ***`dir1:dir2: . . .`***
Uses the specified path to search for source files in the specified path. If this option is not specified, then use the default path of dot (.).

`-attach` ***`address`***
Attaches the debugger to a running JVM with the default connection mechanism.

`-listen` ***`address`***
Waits for a running JVM to connect to the specified address with a standard connector.

`-listenany`
Waits for a running JVM to connect at any available address using a standard connector.

`-launch`
Starts the debugged application immediately upon startup of the `jdb` command. The `-launch` option removes the need for the `run` command. The debugged application is launched and then stopped just before the initial application class is loaded. At that point, you can set any necessary breakpoints and use the `cont` command to continue execution.

`-listconnectors`
Lists the connectors available in this JVM.

`-connect` ***`connector-name:name1=value1...`***
Connects to the target JVM with the named connector and listed argument values.

`-dbgtrace [`*`flags`*`]`
Prints information for debugging the `jdb` command.

`-tclient`
Runs the application in the Java HotSpot VM client.

-tserver
Runs the application in the Java HotSpot VM server.

-Joption
Passes `option` to the JVM, where option is one of the options described on the reference page for the Java application launcher. For example, `-J-Xms48m` sets the startup memory to 48 MB. See Overview of Java Options.

The following options are forwarded to the debuggee process:

-v -verbose[:*class*|gc|jni]
Turns on the verbose mode.

-Dname=value
Sets a system property.

-classpath *dir*
Lists directories separated by colons in which to look for classes.

-X *option*
A nonstandard target JVM option.

Other options are supported to provide alternate mechanisms for connecting the debugger to the JVM that it's to debug.

jhsdb

You use the `jhsdb` tool to attach to a Java process or to launch a postmortem debugger to analyze the content of a core dump from a crashed Java Virtual Machine (JVM).

Synopsis

```
jhsdb clhsdb [--pid pid | --exe executable --core coredump]

jhsdb debugd [options] pid [server-id]|[option] executable core [server-id]

jhsdb hsdb [--pid pid | --exe executable --core coredump]

jhsdb jstack [--pid pid | --exe executable --core coredump] [options]

jhsdb jmap [--pid pid | --exe executable --core coredump] [options]

jhsdb jinfo [--pid pid | --exe executable --core coredump] [options]

jhsdb jsnap [options] [--pid pid | --exe executable --core coredump]
```

pid
The process ID to which the `jhsdb` tool should attach. The process must be a Java process. To get a list of Java processes running on a machine, use the jps command.

server-id
An optional unique ID to use when multiple debug servers are running on the same remote host.

executable
The Java executable file from which the core dump was produced.

core
The core file to which the `jhsdb` tool should attach.

options
The command-line options for a `jhsdb` mode. See Common Options for jhsdb Modes, Options for the debugd Mode, Options for the jinfo Mode, Options for the jmap Mode, Options for the jmap Mode, Options for the jstack Mode, and Options for the jsnap Mode.

> **Note:**
>
> Either the *pid* or the pair of *executable* and *core* files must be provided.

Description

You can use the `jhsdb` tool to attach to a Java process or to launch a postmortem debugger to analyze the content of a core-dump from a crashed Java Virtual Machine (JVM). This command is experimental and unsupported.

> **Note:**
>
> Attaching the `jhsdb` tool to a live process will cause the process to hang and the process will probably crash when the debugger detaches.

The `jhsdb` tool can be launched in any one of the following modes:

`jhsdb clhsdb`
Starts the interactive command-line debugger.

`jhsdb debugd`
Starts the remote debug server.

`jhsdb hsdb`
Starts the interactive GUI debugger.

`jhsdb jstack`
Prints stack and locks information.

`jhsdb jmap`
Prints heap information.

`jhsdb jinfo`
Prints basic JVM information.

`jhsdb jsnap`
Prints performance counter information.

Common Options for jhsdb Modes

In addition to any required `jstack`, `jmap`, `jinfo` or `jsnap` mode specific options, the `pid`, `exe`, or `core` options must be provided for all modes. The following options are available for all modes.

`--pid`
The process ID of the hanging process.

`--exe`
The executable file name.

`--core`
The core dump file name.

`--help`
Displays the options available for the command.

Options for the debugd Mode

`server-id`
An optional unique ID for this debug server. This is required if multiple debug servers are run on the same machine.

Options for the jinfo Mode

Without specified options, the `jhsdb jinfo` prints both flags and properties.

`--flags`
Prints the VM flags.

`--sysprops`
Prints the Java system properties.

no option
Prints the VM flags and the Java system properties.

Options for the jmap Mode

In addition to the following mode specific options, the `pid`, `exe`, or `core` options described in Common Options for jhsdb Modes must be provided.

no option
Prints the same information as Solaris `pmap`.

`--heap`
Prints the `java` heap summary.

`--binaryheap`
Dumps the `java` heap in `hprof` binary format.

`--dumpfile`
Prints the name of the dumpfile.

`--histo`
Prints the histogram of `java` object heap.

`--clstats`
Prints the class loader statistics.

`--finalizerinfo`
Prints the information on objects awaiting finalization.

Options for the jstack Mode

In addition to the following mode specific options, the `pid`, `exe`, or `core` options described in Common Options for jhsdb Modes must be provided.

`--locks`
Prints the `java.util.concurrent` locks information.

`--mixed`
Attempts to print both `java` and native frames if the platform allows it.

Options for the jsnap Mode

In addition to the following mode specific option, the `pid`, `exe`, or `core` options described in Common Options for jhsdb Modes must be provided.

`--all`
Prints all performance counters.

jinfo

You use the `jinfo` command to generate Java configuration information for a specified Java process. This command is experimental and unsupported.

Synopsis

```
jinfo [option] pid
```

`option`
This represents the `jinfo` command-line options. See Options for the jinfo Command.

`pid`
The process ID for which the configuration information is to be printed. The process must be a Java process. To get a list of Java processes running on a machine, use the jps command.

Description

The `jinfo` command prints Java configuration information for a specified Java process. The configuration information includes Java system properties and JVM command-line flags. If the specified process is running on a 64-bit JVM, then you might need to specify the `-J-d64` option, for example:

```
jinfo -J-d64 -sysprops pid
```

This command is unsupported and might not be available in future releases of the JDK. In Windows Systems where `dbgeng.dll` is not present, the Debugging Tools for Windows must be installed to have these tools work. The `PATH` environment variable should contain the location of the `jvm.dll` that's used by the target process or the location from which the core dump file was produced.

Options for the jinfo Command

> **Note:**
>
> If none of the following options are used, both the command-line flags and the system property name-value pairs are printed.

`-flag` *`name`*
Prints the name and value of the specified command-line flag.

`-flag` *`[+|-]name`*
Enables or disables the specified Boolean command-line flag.

`-flag` *`name=value`*
Sets the specified command-line flag to the specified value.

`-flags`
Prints command-line flags passed to the JVM.

`-sysprops`
Prints Java system properties as name-value pairs.

`-h` or `-help`
Prints a help message.

jmap

You use the `jmap` command to print details of a specified process. This command is experimental and unsupported.

Synopsis

```
jmap [options] pid
```

`options`
This represents the `jmap` command-line options. See Options for the jmap Command.

`pid`
The process ID for which the information specified by the *`options`* is to be printed. The process must be a Java process. To get a list of Java processes running on a machine, use the jps command.

Description

The `jmap` command prints details of a specified running process.

> **Note:**
>
> This command is unsupported and might not be available in future releases of the JDK. On Windows Systems where the `dbgeng.dll` file isn't present, the Debugging Tools for Windows must be installed to make these tools work. The `PATH` environment variable should contain the location of the `jvm.dll` file that's used by the target process or the location from which the core dump file was produced.

Options for the jmap Command

`-clstats` *`pid`*
Connects to a running process and prints class loader statistics of Java heap.

`-finalizerinfo` *`pid`*
Connects to a running process and prints information on objects awaiting finalization.

`-histo[:live]` *`pid`*
Connects to a running process and prints a histogram of the Java object heap. If the *`live`* suboption is specified, it then counts only live objects.

`-dump:`*`dump options pid`*
Connects to a running process and dumps the Java heap. The *`dump options`* include:

- `live` — When specified, dumps only the live objects; if not specified, then dumps all objects in the heap.

  `format=b` — Dumps the Java heap,. in `hprof` binary format
- `file=`*`filename`* — Dumps the heap to *`filename`*

Example: `jmap -dump:live,format=b,file=heap.bin` *`pid`*

jstack

You use the `jstack` command to print Java stack traces of Java threads for a specified Java process. This command is experimental and unsupported.

Synopsis

`jstack [`*`options`*`]` *`pid`*

`options`
This represents the `jstack` command-line options. See Options for the jstack Command.

`pid`
The process ID for which the stack trace is printed. The process must be a Java process. To get a list of Java processes running on a machine, use the jps command.

Description

The `jstack` command prints Java stack traces of Java threads for a specified Java process. For each Java frame, the full class name, method name, byte code index (BCI), and line number, when available, are printed. C++ mangled names aren't demangled. To demangle C++ names, the output of this command can be piped to `c+`

`+filt`. When the specified process is running on a 64-bit JVM, you might need to specify the `-J-d64` option, for example: `jstack -J-d64`*`pid`* .

> **Note:**
>
> This command is unsupported and might not be available in future releases of the JDK. In Windows Systems where the `dbgeng.dll` file isn't present, the Debugging Tools for Windows must be installed so that these tools work. The `PATH` environment variable needs to contain the location of the `jvm.dll` that is used by the target process, or the location from which the core dump file was produced.

Options for the jstack Command

`-l`
The long listing option prints additional information about locks.

`-h` or `–help`
Prints a help message.

13
Script Commands

You use JDK commands to run scripts that interact with the Java platform.

> **Note:**
>
> Commands identified as **Experimental** are unsupported and might not be available in future JDK releases.

The following sections describe the commands used to run scripts:

- jjs: You use the `jjs` command-line tool to invoke the Nashorn engine.
- jrunscript: **Experimental** You use the `jrunscript` command to run a command-line script shell that supports interactive and batch modes.

jjs

You use the `jjs` command-line tool to invoke the Nashorn engine.

Synopsis

```
jjs [options] script-files [-- arguments]
```

options
This represents one or more options of the `jjs` command, separated by spaces. See Options for the jjs Command.

script-files
This represents one or more script files that you want to interpret using the Nashorn engine, separated by spaces. If no files are specified, then an interactive shell is started.

arguments
All values after the double hyphen marker (`--`) are passed through to the script or the interactive shell as arguments. These values can be accessed by using the *`arguments`* property.

Description

The `jjs` command-line tool is used to invoke the Nashorn engine. You can use it to interpret one or several script files, or to run an interactive shell.

Options for the jjs Command

The options of the `jjs` command control the conditions under which scripts are interpreted by Nashorn engine.

`-Dname=value`
Sets a system property to be passed to the script by assigning a value to a property name. The following example shows how to invoke Nashorn engine in interactive mode and assign `myValue` to the property named `myKey`:

```
>> jjs -DmyKey=myValue
jjs> java.lang.System.getProperty("myKey")
myValue
jjs>
```

This option can be repeated to set multiple properties.

`--add-modules modules`
Specifies the root user Java modules.

`-cp path` or `-classpath path`
Specifies the path to the supporting class files. To set multiple paths, the option can be repeated, or you can separate each path with the following character:

- **Oracle Solaris, Linux, and OS X**: Colon (:)
- **Windows**; Semicolon (;)

`-doe=[true|false]` or `-dump-on-error=[true|false]`
Provides a full stack trace when an error occurs. By default, only a brief error message is printed. The default parameter is `false`.

`-fv=[true|false]` or `-fullversion=[true|false]`
Prints the full Nashorn version string. The default parameter is `false`.

`-fx=[true|false]`
Launches the script as a JavaFX application. The default parameter is `false`.

`-h` or `-help`
Prints the list of options and their descriptions.

`--language=[es5|es6]`
Specifies the ECMAScript language version. The default version is ES5.

`--module-path path`
Specifies where to find user Java modules.

`-ot=[true|false]` or `-optimistic-types=[true|false]`
Enables or disables optimistic type assumptions with deoptimizing recompilation. This makes the compiler try, for any program symbol whose type can't be proven at compile time, to type it as narrowly and primitively as possible. If the runtime encounters an error because the symbol type is too narrow, then a wider method is generated until a steady stage is reached. While this produces as optimal Java bytecode as possible, erroneous type guesses will lead to longer warmup. Optimistic typing is currently enabled by default, but it can be disabled for faster startup performance. The default parameter is `true`.

`-scripting=[true|false]`
Enables a shell scripting features. The default parameter is `true`.

`-strict=[true|false]`
Enables a strict mode, which enforces stronger adherence to the standard (ECMAScript Edition 5.1), making it easier to detect common coding errors. The default parameter is `false`.

`-t=`*`zone`* or `-timezone=`*`zone`*
Sets the specified time zone for script execution. It overrides the time zone set in the OS and used by the `Date` object. The default *`zone`* is `America/Los_Angeles`.

`-v=[true|false]` or `-version=[true|false]`
Prints the Nashorn version string. The default parameter is `false`.

Example of Running a Script with Nashorn

```
jjs script.js
```

Example of Running Nashorn in Interactive Mode

```
>> jjs
jjs> println("Hello, World!")
Hello, World!
jjs> quit()
>>
```

Example of Passing Arguments to Nashorn

```
>> jjs -- a b c
jjs> arguments.join(", ")
a, b, c
jjs>
```

jrunscript

You use the `jrunscript` command to run a command-line script shell that supports interactive and batch modes. **Note**: This tool is **experimental** and unsupported.

Synopsis

```
jrunscript [options] [arguments]
```

`options`
This represents the `jrunscript` command-line options that can be used. See Options for the jrunscript Command.

`arguments`
Arguments, when used, follow immediately after options or the command name. See Arguments.

Description

The `jrunscript` command is a language-independent command-line script shell. The `jrunscript` command supports both an interactive (read-eval-print) mode and a batch (`-f` option) mode of script execution. By default, JavaScript is the language used, but the `-l` option can be used to specify a different language. By using Java to scripting language communication, the `jrunscript` command supports an exploratory programming style.

If JavaScript is used, then before it evaluates a user defined script, the `jrunscript` command initializes certain built-in functions and objects, which are documented in the API Specification for `jrunscript` JavaScript built-in functions.

Options for the jrunscript Command

`-cp` *`path`* or `-classpath` *`path`*
Indicates where any class files are that the script needs to access.

`-D`*`name=value`*
Sets a Java system property.

`-J`*`flag`*
Passes *`flag`* directly to the Java Virtual Machine where the `jrunscript` command is running.

`-l` *`language`*
Uses the specified scripting language. By default, JavaScript is used. To use other scripting languages, you must specify the corresponding script engine's JAR file with the `-cp` or `-classpath` option.

`-e` *`script`*
Evaluates the specified script. This option can be used to run one-line scripts that are specified completely on the command line.

`-encoding` *`encoding`*
Specifies the character encoding used to read script files.

`-f` *`script-file`*
Evaluates the specified script file (batch mode).

`-f -`
Enters interactive mode to read and evaluate a script from standard input.

`-help` or `-?`
Displays a help message and exits.

`-q`
Lists all script engines available and exits.

Arguments

If arguments are present and if no `-e` or `-f` option is used, then the first argument is the script file and the rest of the arguments, if any, are passed as script arguments. If arguments and the `-e` or the `-f` option are used, then all arguments are passed as script arguments. If arguments `-e` and `-f` are missing, then the interactive mode is used.

Example of Executing Inline Scripts

```
jrunscript -e "print('hello world')"
jrunscript -e "cat('http://www.example.com')"
```

Example of Using Specified Language and Evaluate the Script File

```
jrunscript -l js -f test.js
```

Example of Interactive Mode

```
jrunscript
js> print('Hello World\n');
Hello World
js> 34 + 55
89.0
js> t = new java.lang.Thread(function() { print('Hello World\n'); })
Thread[Thread-0,5,main]
js> t.start()
js> Hello World

js>
```

Run Script File with Script Arguments

In this example, the `test.js` file is the script file. The `arg1`, `arg2`, and `arg3` arguments are passed to the script. The script can access these arguments with an arguments array.

```
jrunscript test.js arg1 arg2 arg3
```