

Java Platform, Standard Edition

Javadoc Guide



Release 9
E75727-02
September 2017

The Oracle logo, consisting of a solid red square with the word "ORACLE" in white, uppercase, sans-serif font centered within it.

ORACLE®

Java Platform, Standard Edition Javadoc Guide, Release 9

E75727-02

Copyright © 2014, 2017, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

	Preface	
	<hr/>	
	Audience	iv
	Documentation Accessibility	iv
	Related Documents	iv
	Conventions	iv
1	Javadoc	
	<hr/>	
	Javadoc Features	1-1
2	Source Files	
	<hr/>	
3	Javadoc Command	
	<hr/>	
	Javadoc Doclets	3-1
	Javadoc Doclet Options	3-1
	Using the link Option	3-11
	Using the linkoffline Option	3-13
	Using the Tag Option	3-15
	javadoc Command-Line Argument Files	3-17
	The Standard Doclet	3-18
	Javadoc Standard Doclet	3-18
	Generated Files	3-28
	Examples of Running the javadoc Command	3-31

Preface

This guide provides information about using the `javadoc` command, its options, and the Standard Doclet.

Audience

This document is intended for Javadoc tool users. Users who are developing Javadoc content should also see the [Javadoc specification](#) for detailed information required to create javadoc content.

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

- See [Oracle JDK 9 Documentation](#) for other JDK 9 guides.
- The Javadoc developers can refer to the Javadoc specifications.

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.
<i>italic</i>	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

Javadoc

The `javadoc` tool is a program that reads a collection of source files into an internal form.

The structure is: `(source-files)->[javadoc-tool:doclet]->(generated files)`.

The [Javadoc doclet](#) is like a pluggable back end that analyzes this internal form with some goal in mind. It can generate HTML, MIF, or XML files, depending on the doclet.

The content of the generated files is specific to the doclet. The standard doclet generates HTML documentation, but a different doclet, for example, could generate a report of misspelled words or grammatical errors.

If you specify a doclet other than the standard doclet, then the descriptions in this guide might not apply to the operation of that doclet or the files (if any) that are generated.

To use the `javadoc` tool, you must:

- Write source code, containing documentation comments. Documentation comments are formatted in HTML and are interspersed with the `javadoc` tool.
- Run the `javadoc` tool. You need to specify a doclet to analyze the documentation comments and any other special tags. However, if you don't specify any doclet, by default, the [Standard Doclet](#) is included. You specify a number of command-line options, some of which are directed at the `javadoc` tool itself, and some of which are specific to the selected doclet. The command-line help shows and distinguishes the options for the tool that apply to the currently selected doclet. When the standard doclet is used, the output generated by the standard doclet consists of a series of HTML pages. If you specify a different doclet, then the operation of that doclet and what files (if any) are generated may or may not be equivalent to the standard doclet described in this guide.

Javadoc Features

Javadoc features include the following: Javadoc search, support for generating HTML5 output, support for documentation comments in module systems, and simplified Doclet API.

Search

The `javadoc` tool runs the doclet that may generate output. The standard doclet generates output that lets you search the generated documentation. A search box is available on the generated APIs and provides the following:

- You can search for elements and additional key phrases defined in the API
- Results, including results that exactly match the entered characters followed by results that contains the entered characters anywhere in the string. Multiple results are displayed as simple scrolling lists below the search box. Results are categorized as follows, for easier classification and appropriate user selection:

- Modules
- Packages
- Types
- Members
- Search Tags

Multiple results with different program element names are displayed if the search term or a phrase is inherited using the `@inheritDoc` tag.

- Page redirection based on user selection.

You can search for the following:

- **Declared names of modules, packages, types, and members:** Because methods can be overloaded, the simple names of method parameter types are also indexed and can be searched for. The method parameter names can't be indexed.
- **A search term or a phrase indexed using a new inline tag, `@index`:** Other inline tags cannot be nested inside `@index`. You can only search a phrase or search term marked with `@index` within a declaration's `javadoc` comment. For example, the domain-specific term `ulps` is used throughout the `java.lang.Math` class, but doesn't appear in any class or method declaration names. To help users of the Math API, the API designer could tag various occurrences of `ulps` in a class-level `javadoc` comment or a method-level `javadoc` comment. Tagging is achieved using `{@index ulps}`. The term `ulps` is indexed by the `javadoc` tool.

Module System

The `javadoc` tool supports documentation comments in module declarations. Some Javadoc command-line options enable you to specify the set of modules to document and generate a new summary page for any modules being documented. It has new command-line options to configure the set of modules to be documented and generates a new summary page for any modules being documented. See [Javadoc Doclet Options](#).

HTML 5 Support

You can generate HTML5 output. To get fully-compliant HTML5 output, ensure that any HTML content provided in documentation comments are compliant with HTML5.

Simplified Doclet API

The Doclet API uses powerful APIs that can better represent all the language features. See [Standard Doclet Options](#).

2

Source Files

The `javadoc` tool generates output that originates from the following types of source files: Java language source files for classes (`.java`), package comment files, overview comment files, and miscellaneous unprocessed files.

This topic describes source files, test files, and template files that can also be in the source tree, but that must be sure not to document.

Class Source Files

The source file of each class can have their own documentation comments.

Overview Comment Files

Each application or set of packages that you're documenting can have its own overview documentation comment that's kept in its own source file, which the `javadoc` tool then merges into the generated overview page. You typically include in this comment any documentation that applies to the entire application or set of packages. You can name the file anything that you want, such as `overview.html` and place it anywhere. A typical location is at the top of the source tree.

Oracle Solaris, Linux, and macOS: For example, if the source files for the `java.math` package are contained in the `/home/user/src/java/math` directory, then you could create an overview comment file in `/home/user/src/overview.html`.

Windows: For example, if the source files for the `java.math` package are contained in the `C:\user\src\java\math` directory, then you could create an overview comment file in `C:\user\src\overview.html`.

You can have multiple overview comment files for the same set of source files in case you want to run the `javadoc` tool multiple times on different sets of packages. For example, you could run the `javadoc` tool once with `-private` option for internal documentation and again without that option for public documentation. In this case, you could describe the documentation as public or internal in the first sentence of each overview comment file.

The content of the overview comment file is one big documentation comment that's written in HTML. Make the first sentence a summary about the application or set of packages. Don't put a title or any other text between the `<body>` tag and the first sentence. All tags, except inline tags, such as an `{@link}` tag, must appear after the main description. If you add an `@see` tag, then it must have a fully qualified name. When you run the `javadoc` tool, specify the overview comment file name with the `-overview` option. The file is then processed similarly to that of a package comment file. The `javadoc` tool does the following:

- Copies all content between the `<body>` and `</body>` tags for processing.
- Processes the overview tags that are present.
- Inserts the processed text at the bottom of the generated overview page.
- Copies the first sentence of the overview comment to the top of the overview summary page.

Unprocessed Files

Your source files can include any files that you want the `javadoc` tool to copy to the destination directory. These files usually include graphic files, example Java source and class files, and self standing HTML files with a lot of content that would overwhelm the documentation comment of a typical Java source file.

To include unprocessed files, put them in a directory called `doc-files`. The `doc-files` directory can be a subdirectory of any package directory that contains source files. You can have one `doc-files` subdirectory for each package.

Oracle Solaris, Linux, and macOS: For example, if you want to include the image of a button in the `java.awt.Button` class documentation, then place the image file in the `/home/user/src/java/awt/doc-files/` directory. Don't place the `doc-files` directory at `/home/user/src/java/doc-files`, because `java` isn't a package. It doesn't contain any source files.

Windows: For example, if you want to include the image of a button in the `java.awt.Button` class documentation, then place the image file in the `\src\java\awt\doc-files` directory. Don't place the `doc-files` directory at `\src\java\doc-files`, because `java` is not a package. It doesn't contain any source files.

All links to the unprocessed files must be included in the code because the `javadoc` tool doesn't look at the files. The `javadoc` tool copies the directory and all of its contents to the destination. The following example shows how the link in the `Button.java` documentation comment might look:

```
/**
 * <p> This button looks like this:</p>
 * 
 */
```

Test and Template Files

You can store test and template files in the source tree in the same directory with or in a subdirectory of the directory where the source files reside. To prevent test and template files from being processed, run the `javadoc` tool and explicitly pass in individual source file names.

Test files are valid, compilable source files. Template files aren't valid, compatible source files, but they often have the `.java` suffix.

- **Test Files** : If you want your test files to belong to either an unnamed package or to a package other than the package that the source files are in, then put the test files in a subdirectory underneath the source files and give the directory an invalid name. If you put the test files in the same directory with the source files and call the `javadoc` tool with a command-line argument that indicates its package name, then the test files cause warnings or errors. If the files are in a subdirectory with an invalid name, then the test file directory is skipped and no errors or warnings are issued. For example, to add test files for source files in `com.package1`, put them in a subdirectory in an invalid package name. The following directory name is invalid because it contains a hyphen:

- **Oracle Solaris, Linux, and macOS:** `com/package1/test-files/`
- **Windows:** `com\package1\test-files\`

If your test files contain documentation comments, then you can set up a separate run of the `javadoc` tool to produce test file documentation by passing in their test source file names with wild cards, such as `com/package1/test-files/*.java`.

- **Template Files** : If you want a template file to be in the source directory, but not generate errors when you execute the `javadoc` tool, then give it an invalid file name such as `Buffer-Template.java` to prevent it from being processed. The `javadoc` tool processes only source files with names, when stripped of the `.java` suffix, that are valid class names.

Processing the Package Comment File

When the `javadoc` tool runs, it searches for the package comment file. If the package comment file is found, then the `javadoc` tool does the following:

- Copies the comment for processing. For `package.html`, the `javadoc` tool copies all content between the `<body>` and `</body>` HTML tags. You can include a `<head>` section to put a `<title>` tag, source file copyright statement, or other information, but none of these appears in the generated documentation.
- Processes the package tags.
- Inserts the processed text at the bottom of the generated package summary page.
- Copies the first sentence of the package comment to the top of the package summary page. The `javadoc` tool also adds the package name and this first sentence to the list of packages on the overview page.

The end of the sentence is determined by the rules used for the end of the first sentence of class and member main descriptions.

3

Javadoc Command

The `javadoc` command-line synopsis is `javadoc [options] [packagenames] [sourcefiles] [@files]`. The options can either be Doclet options or Standard Doclet options. The `javadoc` command can also be run programmatically.

This topic contains the following sections:

- [Javadoc Doclet](#)
- [Standard Doclet](#)
- [Examples of Running the Javadoc Command](#)

Javadoc Doclets

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

Javadoc Doclet Options

The `javadoc` command has options for doclets. The Standard Doclet provides additional options.

The `javadoc` command uses doclets to determine its output and the default Standard Doclet unless a custom doclet is specified with the `-doclet` option. While option names are not case-sensitive, their arguments are.

Synopsis

```
javadoc [packages/source-files] [options][@files]
```

packages

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 them.

By default, the `javadoc` command 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.

source-files

Names of Java source files that you want to document, separated by spaces, for example, `Class.java Object.java Button.java`. By default, the `javadoc` command 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.

options

Command-line options, separated by spaces.

@files

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 a corresponding set of 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 either the `-subpackages` option 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.

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.

Process Source Files

The `javadoc` command processes files that end in the source file extension and other files described in [Source Files](#). If you run the `javadoc` command by passing in individual source file names, then you can determine exactly which source files are processed. However, that isn't how most developers want to work because it's simpler to pass in package names. The `javadoc` command can be run three ways, without explicitly specifying the source file names. You can pass in package names, use the `-subpackages` option, or use wild cards with source file names. In these cases, the `javadoc` command processes a source file only when the file fulfills all of the following requirements:

- The file name prefix (with `.java` removed) is a valid class name.
- The path name relative to the root of the source tree is a valid package name after the separators are converted to dots.
- The package statement contains the valid package name.

Processing Links

During a run, the `javadoc` command adds cross-reference links to package, class, and member names that are being documented as part of that run. Links appear in the following places:

- Declarations (return types, argument types, and field types)
- See Also sections that are generated from `@see` tags
- Inline text generated from `{@link}` tags
- Exception names generated from `@throws` tags
- *Specified by* links to interface members and *Overrides* links to class members
- Summary tables listing packages, classes, and members
- Package and class inheritance trees
- The index

Processing Details

The `javadoc` command produces one complete document every time it runs. It doesn't perform incremental builds that modify or directly incorporate the results from earlier runs. However, the `javadoc` command can link to results from other runs.

The `javadoc` command implementation requires and relies on the Java compiler. The `javadoc` command calls part of the `javac` command to compile the declarations and ignore the member implementations. The `javadoc` command builds a rich internal representation of the classes that includes the class hierarchy and use relationships to generate the HTML documentation. The `javadoc` command also picks up user-supplied documentation from documentation comments in the source code.

The `javadoc` command can run on source files that are pure stub files with no method bodies. This means that you can write documentation comments and run the `javadoc` command in the early stages of design before API implementation.

Relying on the compiler ensures that the HTML output corresponds exactly with the actual implementation, which may rely on implicit, rather than explicit, source code. For example, the `javadoc` command documents default constructors that are present in the compiled class files but not in the source code.

In many cases, the `javadoc` command lets you generate documentation for source files with incomplete or erroneous code. You can generate documentation before any debugging and troubleshooting is done. The `javadoc` command does primitive checking of documentation comments.

When the `javadoc` command builds its internal structure for the documentation, it loads all referenced classes. Because of this, the `javadoc` command must be able to find all referenced classes, and whether they're bootstrap classes, extensions, or user classes.

Javadoc Doclets

You can customize the content and format of the `javadoc` command output with doclets. The `javadoc` command has a default built-in doclet, called the Standard Doclet, that generates HTML-formatted API documentation. You can write your own doclet to generate HTML, XML, MIF, RTF or whatever output format you want.

When a custom doclet isn't specified with the `-doclet` option, the `javadoc` command uses the default Standard Doclet. The `javadoc` command has several options that are available regardless of which doclet is being used. The Standard Doclet adds a supplementary set of command-line options.

Javadoc Doclet Options

The `javadoc` tool supports documentation comments in module declarations. It has command-line options, such as `--module-path`, `--upgrade-module-path`, or `--module-source-path` to configure the set of modules to be documented, and generates a new summary page for any modules being documented. The module-related options are available for generating documentation. The following options are the core options that are available to all doclets:

`--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`.

`--add-exports` *module/package=other-module(,other-module)**

Specifies a package that's to be considered as exported 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. *other-module* is `ALL-UNNAMED` to require the unnamed module.

`-bootclasspath` *classpathlist*

Specifies the paths where the boot classes reside. These are typically the Java platform classes. The `bootclasspath` is part of the search path that the `javadoc` command uses to look up source and class files.

To separate directories in the `classpathlist` parameters, use one of the following delimiters:

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

 **Note:**

The `-bootclasspath` option is only allowed if you are working to generate documentation for older versions of the Java platform, such as using JDK 6, 7, or 8 for `-source`, `-target`, or `--release`.

`-breakiterator`

Uses the internationalized sentence boundary of `java.text.BreakIterator` to determine the end of the first sentence in the main description of a package, class, or member for English. All other locales already use the `BreakIterator` class, rather than an English language, locale-specific algorithm. 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 as follows:

- English default sentence-break algorithm: Stops at a period followed by a space or an HTML block tag, such as `<P>`.

- Break iterator 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 won't 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 it's embedded in an HTML tag.
- A question mark always ends the first sentence: If a double quotation mark follows the question mark, then the double quotation mark also gets included in the first sentence, but that character then ends the sentence.

-classpath path OR --class-path 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.

`CLASSPATH` is the environment variable that provides the path that the `javadoc` command uses to find user class files. This environment variable is overridden by the `-classpath` option. To separate directories, use one of the following delimiters: a semicolon for Windows or a colon for Oracle Solaris.

Oracle Solaris, Linux, and macOS: To separate multiple paths, use a colon (:). For example, `./home/classes:/usr/local/java/classes`

Windows: To separate multiple paths, use a semicolon (;). For example, `./C:\classes;C:\home\java\classes`

The `javadoc` command searches all subdirectories of the specified paths.

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 macOS: 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 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's 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 the directory `mydir` contains `a.jar` and `b.jar`, then the class path element `foo/*` 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 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

Specifies the class file that starts the doclet used in generating the documentation. 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 implement the `jdk.javadoc.doclet.Doclet` interface. The path to this starting class is defined by the `-docletpath` option.

`-docletpath path`

Specifies the path to the doclet-starting class file (specified with the `-doclet` option) and any JAR files that 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. 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 macOS:

```
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` command to expand the set of modules to be documented. By default, only the modules given explicitly on the command line are documented. This option supports the following values:

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

`-extdirs dirlist`

Overrides the location of installed extensions. These are any classes that use the Java Extension mechanism. The `extdirs` option is part of the search path that the `javadoc` command uses to look up source and class files. See the `-classpath` option for more information. To separate directories in `dirlist`, use a semicolon (;) for Windows and a colon (:) for Oracle Solaris, Linux, and macOS.

`-help` OR `--help`

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

`-J flag`

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 to generate the documentation, then you need to 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 only initial memory, which is useful when you know the minimum amount of memory required.

There's 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.

--limit-modules *module(,module)**

Limits the number 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 the `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.

The application module path (`--module-path`, or `-mp` for short) contains compiled definitions of library and application modules (all phases). At link time, this path can also contain Java SE and Java Development Kit (JDK) modules.

--module-source-path *path*

Specifies where to find input source files for multiple modules. The `--module-source-path` option contains module definitions in source form (compile time only).

-package

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

--patch-module *module=file(file*)*

Overrides arguments in a module with classes and resources in JAR files or directories.

**Note:**

`file(:file)*` is the same as `path` in other options, such as `--module-path`, or `--module-source-path`.

-private

Shows all classes and members.

-protected

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

-public

Shows only 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 a specified release.

--show-members *value*

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

protected

Shows public and protected declarations. This is the default.

public

Shows only public values.

package

Shows public, protected, and package declarations.

private

Shows all declarations.

--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 declarations (fields or methods) are documented, where *value* can be one of the following:

protected

Shows public and protected declarations. This is the default.

public

Shows only public values.

package

Shows public, protected, and package declarations.

private

Shows all declarations.

-source *release*

Specifies the release of source code accepted. Use the value of *release* that corresponds to the value used when you compile code with the `javac` command. The oldest of the releases is JDK 6 and the latest is JDK 9.

--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 macOS: To separate multiple paths, use a colon (:).

Windows: To separate multiple paths, use 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 are not 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 doesn't 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 the 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're documenting.

Oracle Solaris, Linux, and macOS:

- 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're 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 macOS:

```
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 the system modules used for modular releases.

The system modules are the compiled modules built in to the environment (compile time and runtime). In the case of a custom-linked image, you can also include library and application modules. At compile time, the system modules can be overridden using the `-system` option, which specifies a JDK image from which to load system modules.

The module definitions present on these paths (for example, `--module-source-path path`), together with the system modules, define the number of observable modules. A module path is a sequence, each element of which is either a module definition or a directory containing module definitions.

--upgrade-module-path *path*

Overrides the location of upgradable options. The `--upgrade-module-path` option contains compiled definitions of modules intended to be used in place of upgradable modules built in to the environment (compile time and runtime).

-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.

-Xmaxerrs *number*

Sets the maximum number of errors to print.

-Xmaxwarns *number*

Sets the maximum number of warnings to print.

-xold

Calls the legacy `javadoc` tool.

-javafx

Generates HTML documentation using the JavaFX extensions to the Standard Doclet. The generated documentation includes a Property Summary section in addition to the

other summary sections generated by the standard Java doclet. The listed properties are linked to the sections for the getter and setter methods of each property. If there are no documentation comments written explicitly for getter and setter methods, then the documentation comments from the property method are automatically copied to the generated documentation for these methods. This option also adds a new `@defaultValue` tag that allows documenting the default value for a property.

Example:

```
javadoc -javafx MyClass.java -d testdir
```

Using the link Option

You use `-link` option to classes referenced to by your code, but not documented in the current `javadoc` command run.

For links to go to valid pages, you must know where those HTML pages are located and specify that location with the `extdocURL` option. This allows third-party documentation to link to Java. Omit the `-link` option when you want the `javadoc` command to create links only to APIs within the documentation it's generating in the current run. Without the `-link` option, the `javadoc` command doesn't create links to documentation for external references because it doesn't know whether or where that documentation exists. The `-link` option can create links in several places in the generated documentation. See [Javadoc Doclets](#). Another use is for cross-links between sets of packages: Execute the `javadoc` command on one set of packages, then run the `javadoc` command again on another set of packages, creating links both ways between both sets.

Differences Between the `-link` and `-linkoffline` Options

Use the `-link` option in the following cases:

- When you use a relative path to the external API document.
- When you use an absolute URL to the external API document if your shell lets you open a connection to that URL for reading.

Use the `-linkoffline` option when you use an absolute URL to the external API document, if your shell doesn't allow a program to open a connection to that URL for reading. This can occur when you're behind a firewall and the document you want to link to is on the other side.

Example 3-1 Example of Using an Absolute Link to External Documents

Use the following command if you want to link to the `java.lang`, `java.io` and other Java platform packages.

```
javadoc -link http://docs.oracle.com/javase/8/docs/api/com.mypackage
```

The command generates documentation for the package `com.mypackage` with links to the Java SE packages. The generated documentation contains links to the `Object` class, for example, in the class `trees`. Other options, such as the `-sourcepath` and `-d` options, aren't shown.

Example 3-2 Example of Using a Relative Link to External Documents

- In this example, there are two packages with documents that are generated in different runs of the `javadoc` command, and those documents are separated by a relative path.

- The packages are `com.apipackage`, an API, and `com.spipackage`, a service provider Interface (SPI).
- You want the documentation to reside in `docs/api/com/apipackage` and `docs/spi/com/spipackage`.
- Assuming that the API package documentation is already generated, and that `docs` is the current directory, you document the SPI package with links to the API documentation by running: `javadoc -d ./spi -link ../api com.spipackage`.

 **Note:**

The `-link` option is relative to the destination directory (`docs/spi`).

How to Reference a Class

For a link to an externally referenced class to appear (and not just its text label), the class must be referenced in a particular way. It isn't sufficient for the class to be referenced in the body of a method. It must be referenced in either of the following: `import` statement or in a declaration.

- In any kind of import statement. By wildcard import, import explicitly by name, or automatically import for `java.lang.*`.
- In a declaration: `void mymethod(File f) {}`.

The reference can be in the return type or parameter type of a method, constructor, field, class, or interface, or in an `implements`, `extends`, or `throws` statement.

When you use the `-link` option, there can be many links that unintentionally don't appear. The text would appear without being a link. You can detect such text by the warnings they emit. The simplest way to properly reference a class and add the link is to import that class.

In a declaration: `void mymethod(File f) {}`

Package List

The `-link` option requires that a file named `package-list`, which is generated by the `javadoc` command, exists at the URL that you specify with the `-link` option. In JDK 8, the `package-list` file is a simple text file that lists the names of packages documented at that location.

When `javadoc` is run without the `-link` option and encounters a name that belongs to an externally referenced class, it prints the name with no link. However, when the `-link` option is used, the `javadoc` command searches the `package-list` file at the specified `extdocURL` location for that package name. When it finds the package name, it prefixes the name with `extdocURL`.

For there to be no broken links, all of the documentation for the external references must exist at the specified URLs. The `javadoc` command does not check that these pages exist, but only that the `package-list` exists.

Multiple Links

You can supply multiple `-link` options to link to any number of externally generated documents. Specify a different link option for each external document to link to javadoc `-link extdocURL1 -link extdocURL2 ... -link extdocURLn com.mypackage` where `extdocURL1, extdocURL2, ... extdocURLn` point respectively to the roots of external documents, each of which contains a file named `package-list`.

Cross Linking

Note:

Bootstrapping might be required when cross-linking two or more documents that were previously generated. If the `package-list` file doesn't exist for either document when you run the `javadoc` command on the first document, then the `package-list` doesn't yet exist for the second document. Therefore, to create the external links, you must regenerate the first document after you generate the second document.

In this case, the purpose of first generating a document is to create its `package-list` (or you can create it by hand if you are certain of the package names). Then, generate the second document with its external links. The `javadoc` command prints a warning when a needed external `package-list` file doesn't exist.

Using the linkoffline Option

You use `linkoffline` option to link to the `java.lang`, `java.io` and other Java SE packages

Absolute Links to External Documents

You might have a situation where you want to link to the `java.lang`, `java.io` and other Java SE packages. However, your shell doesn't have web access. In this case, do the following:

1. Open the `package-list` file in a browser at [API Specification](#).
2. Save the file to a local directory, and point to this local copy with the second argument, `packagelistLoc`. In this example, the package list file was saved to the current directory.

The following command generates documentation for the package `com.mypackage` with links to the Java SE packages. The generated documentation contains links to the `Object` class, for example, in the class `trees`. Other necessary options, such as `-sourcepath`, aren't shown.

```
javadoc -linkoffline http://docs.oracle.com/javase/8/docs/api/.com.mypackage
```

Relative Links to External Documents

It's not very common to use `-linkoffline` with relative paths, for the simple reason that the `-link` option is usually enough. When you use the `-linkoffline` option, the `package-list` file is usually local, and when you use relative links, the file you're linking to is also local, so it's usually unnecessary to give a different path for the two

arguments to the `-linkoffline` option. When the two arguments are identical, you can use the `-link` option.

Create a package-list File Manually

If a `package-list` file doesn't exist yet, but you know what package names your document will link to, then you can manually create your own copy of this file and specify its path with `packagelistLoc`. An example would be where the `package-list` file for `com.spipackage` didn't exist when `com.apipackage` package was first generated. This technique is useful when you need to generate documentation that links to new external documentation whose package names you know, but which isn't yet published. Similarly, two companies can share their unpublished `package-list` files so they can release their cross-linked documentation simultaneously.

Link to Multiple Documents

You can include the `-linkoffline` option once for each generated document that you want to refer to:

```
javadoc -linkoffline extdocURL1 packagelistLoc1 -linkoffline extdocURL2  
packagelistLoc2 ...
```

Update Documents

You can also use the `-linkoffline` option when your project has dozens or hundreds of packages. If you've already run the `javadoc` command on the entire source tree, then you can quickly make small changes to documentation comments and rerun the `javadoc` command on a portion of the source tree. Be aware that the second run works properly only when your changes are to documentation comments and not to declarations. If you were to add, remove, or change any declarations from the source code, then broken links could show up in the index, package tree, inherited member lists, Use page, and other places.

First, create a new destination directory, such as `update`, for this new small run. In this example, the original destination directory is named `html`. In the simplest example, change the directory to the parent of `html`. Set the first argument of the `-linkoffline` option to the current directory and set the second argument to the relative path to `html`, where it can find the `package-list` file and pass in only the package names of the packages that you want to update:

```
javadoc -d update -linkoffline . html com.mypackage
```

Oracle Solaris, Linux, and macOS: When the `javadoc` command completes, copy these generated class pages in `update/com/package` (not the overview or index) to the original files in the `html/com/package`.

Windows: When the `javadoc` command completes, copy these generated class pages in `update\com\package` (not the overview or index) to the original files in `html\com\package`.

Using the Tag Option

Use `Xaoptcmf` arguments to determine where in the source code the tag is allowed to be placed, and whether the tag can be disabled (using `x`).

Placement of Tags

You can supply either `a`, to allow the tag in all places, or any combination of the other letters:

- `x` (disable tag)
- `a` (all)
- `o` (overview)
- `p` (packages)
- `t` (types, that is classes and interfaces)
- `c` (constructors)
- `m` (methods)
- `f` (fields)
- `s` (modules)

Examples of Single Tags

An example of a tag option for a tag that can be used anywhere in the source code is:

```
-tag todo:a:"To Do:".
```

If you want the `@todo` tag to be used only with constructors, methods, and fields, then you use: `-tag todo:cmf:"To Do:"`.

Notice the last colon (`:`) isn't a parameter separator, but is part of the heading text. You can use either tag option for source code that contains the `@todo` tag, such as: `@todo`
The documentation for this method needs work.

Colons in Tag Names

Use a backslash to escape a colon that you want to use in a tag name. Use the `-tag ejb\:\:bean:a:"EJB Bean:"` option for the following documentation comment:

```
/**  
 * @ejb:bean  
 */
```

Spell-Checking Tag Names

Some developers put custom tags in the source code that they don't always want to produce as output. In these cases, it's important to list all tags that are in the source code, enabling the ones you want to output and disabling the ones you don't want to output. The presence of `x` disables the tag, while its absence enables the tag. This gives the `javadoc` command enough information to know whether a tag it encounters is unknown, which is probably the results of a typographical error or a misspelling. The `javadoc` command prints a warning in these cases. You can add `x` to the placement values already present, so that when you want to enable the tag, you can simply delete the `x`. For example, if the `@todo` tag is a tag that you want to suppress on output, then you would use: `-tag todo:Xcmf:"To Do:"`. If you would rather keep it simple, then

use this: `-tag todo:X`. The syntax `-tag todo:X` works even when the `@todo` tag is defined by a taglet.

Order of Tags

The order of the `-tag` and `-taglet` options determines the order that the tags are produced. You can mix the custom tags with the standard tags to intersperse them. The tag options for standard tags are placeholders only for determining the order. They take only the standard tag's name. Subheadings for standard tags can't be altered. For example, if the `-tag` option is missing, then the position of the `-taglet` option determines its order. If they're both present, then whichever appears last on the command line determines its order. This happens because the tags and taglets are processed in the order that they appear on the command line. For example, if the `-taglet` and `-tag` options have the name `todo` value, then the one that appears last on the command line determines the order.

Example of a Complete Set of Tags

This example inserts `To Do` after `Parameters` and before `Throws` in the output. By using `X`, it also specifies that the `@example` tag might be encountered in the source code that shouldn't be displayed during this run. If you use `@argfile` on the command line to specify a file containing options, then you can put the tags on separate lines in an argument file similar to this (no line continuation characters needed):

```
-tag param
-tag return
-tag todo:a:"To Do:"
-tag throws
-tag see
-tag example:X
```

When the `javadoc` command parses the documentation comments, any tag encountered that's neither a standard tag nor passed in with the `-tag` or `-taglet` options is considered unknown, and a warning is thrown.

The standard tags are initially stored internally in a list in their default order. Whenever the `-tag` options are used, those tags get appended to this list. Standard tags are moved from their default position. Therefore, if a `-tag` option is omitted for a standard tag, then it remains in its default position.

Avoiding Conflicts

If you want to create your own namespace, then you can use a dot-separated naming convention similar to that used for packages: `com.mycompany.todo`. Oracle continues to create standard tags whose names don't contain dots. Any tag that you create overrides the behavior of a tag by the same name defined by Oracle. If you create a `@todo` tag or taglet, then it always has the same behavior that you define, even when Oracle later creates a standard tag of the same name.

Annotations Versus Javadoc Tags

In general, if the markup that you want to add is intended to affect or produce documentation, then it should be a Javadoc tag. Otherwise, it should be an annotation. See [Custom Tags and Annotations](#) in *How to Write Doc Comments for the Javadoc Tool*.

You can also create more complex block tags or custom inline tags with the `-taglet` option.

javadoc Command-Line Argument Files

To shorten or simplify the `javadoc` command, you specify one or more files that contain arguments to the `javadoc` command (except `-J` options). This lets you to create `javadoc` commands of any length on any operating system.

When you run the `javadoc` command, pass the path and name of each argument file with the `@` leading character. When the `javadoc` command encounters an argument beginning with the `@` character, it expands the contents of that file into the argument list.

Examples

Single Argument File

You can use a single argument file named `argfile` to hold all `javadoc` command arguments: `javadoc @argfile`.

Two Argument Files

The argument file contains the contents of both files. You can create two argument files: One for the `javadoc` command options and the other for the package names or source file names. Notice the following lists have no line-continuation characters. Create a file named `options` that contains:

Oracle Solaris, Linux, and macOS:

```
-d docs-filelist
-use
-splitindex
-windowtitle 'Javadoc'
-doctitle 'Javadoc Guide'
-header '<b>Java™ SE </b>'
-bottom 'Copyright &copy; 1993-2011 Oracle and/or its affiliates. All rights
reserved.'
-group "Core Packages" "java.*"
-overview /java/jdk9/docs/api/overview-summary
-sourcepath /java/
```

Windows:

```
-d docs-filelist
-use
-splitindex
-windowtitle 'Javadoc'
-doctitle 'Javadoc Guide'
-header '<b>Java™ SE 7</b>'
-bottom 'Copyright &copy; 1993-2011 Oracle and/or its affiliates. All rights
reserved.'
-group "Core Packages" "java.*"
-overview \java\jdk9\docs\api\overview-summary.html
-sourcepath \java\
```

Create a file named `packages` that contains:

```
com.mypackage1
com.mypackage2
com.mypackage3
```

Run the `javadoc` command as follows:

```
javadoc @options @packages
```

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`):

Oracle Solaris, Linux, and macOS:

```
javadoc @path1/options @path2/packages
```

Windows:

```
javadoc @path1\options @path2\packages
```

Option Arguments

The following example saves an argument to a `javadoc` command option in an argument file. The `-bottom` option is used because it can have a lengthy argument. You can create a file named `bottom` to contain the text argument:

```
<font size="-1">  
  <a href="http://bugreport.java.com/bugreport/">Submit a bug or feature</a> </  
font>
```

Run the `javadoc` command as follows: `javadoc -bottom @bottom @packages`.

You can also include the `-bottom` option at the start of the argument file and run the `javadoc` command as follows: `javadoc @bottom @packages`.

The Standard Doclet

The Standard Doclet is the doclet provided by Oracle that produces Javadoc's default HTML-formatted API output.

This topic contains the following sections:

- [Javadoc Standard Doclet](#)
- [Generated Files](#)

Javadoc Standard Doclet

Javadoc uses the Standard Doclet if no other doclet is specified using the Javadoc's `-doclet` option on the command line. In JDK 9, the Doclet API has been updated to use newer, more powerful APIs, that can better represent all the recent new language features. The Standard Doclet is updated to use this Doclet API.

The Standard Doclet is the doclet provided by Oracle that produces Javadoc's default HTML-formatted API output. The [API Specification](#) for the Java platform in this JDK documentation is an example of the Standard Doclet's output.

Standard Doclet Options

`-author`

Includes the `@author` text in the generated documents.

`-bottom text`

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 Multipurpose Internet Mail Extensions (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 *title*

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 "Java Library
v8" com.mypackage`.

--excludedocfilessubdir *name*

Excludes any `doc-file` subdirectories with the given name. This option 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's 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 *name p1:p2*

Groups specified packages together in overview page.

The `-group groupheading packagepattern:packagepattern` separates packages on the overview page into whatever groups you specify, one group per table. You specify each group with a different `-group` option. The groups appear on the page in the order specified on the command line. Packages are alphabetized within a group. For a specified `-group` option, the packages matching the list of *packagepattern* expressions appear in a table with the heading *groupheading*.

- The *groupheading* value can be any text and can include white space. This text is placed in the table heading for the group.
- The *packagepattern* value can be any package name at the start of any package name followed by an asterisk (*). The asterisk is the only wildcard allowed and means match any characters. Multiple patterns can be included in a group by separating them with colons (:). If you use an asterisk in a pattern or pattern list, then the pattern list must be inside quotation marks, such as `"java.lang*:java.util"`.

When you don't supply a `-group` option, all packages are placed in one group with the heading *Packages* and appropriate subheadings. If the subheadings don't include all documented packages (all groups), then the remaining packages appear in a separate group with the subheading *Other Packages*.

For example, the following `javadoc` command separates the three documented packages into *Core*, *Extension*, and *Other Packages*. The trailing dot (.) doesn't appear in `java.lang*`. Including the dot, such as `java.lang.*` omits the `java.lang` package.

```
javadoc -group "Core Packages" "java.lang*:java.util"  
        -group "Extension Packages" "javax.*"  
        java.lang java.lang.reflect java.util javax.servlet java.new
```

Core Packages

```
java.lang  
java.lang.reflect  
java.util
```

Extension Packages

```
javax.servlet
```

Other Packages

```
java.new
```

-header *header*

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* option can contain HTML tags and white space, but when it does, the *header* text must be enclosed in quotation marks. Use escape characters for internal quotation marks within a header. For example, `javadoc -header "Java Platform
v8" com.mypackage`.

-helpfile *path\filename*

Specifies the path of an alternate help file *path\filename* that the HELP link in the top and bottom navigation bars link to. Without this option, the `javadoc` command creates a `help-doc.html` help file 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 `help-doc.html`. The `javadoc` command adjusts the links in the navigation bar accordingly, for example:

Oracle Solaris, Linux, and macOS:

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

Windows:

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

-html4

Generates HTML4.0.1 output.

-html5

Generates HTML5 output. HTML5 increases the semantic value of web pages and makes it easier to create accessible web pages.

 **Note:**

For both `-html4` and `-html5`, These options assume that the HTML in the document comments is of the same version (4 or 5). If you don't specify any of these options, then, by default HTML4 output is generated. The standard doclet doesn't convert the HTML in the user documentation comments to the specified output version.

-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 `extdocURL` argument is the absolute or relative URL of the directory that contains the external Javadoc-generated documentation you want to link to. 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 `extdocURL` to enable your documents to link to a document on any website, or you can use a relative link to link only to a relative location. If you use a relative link, then the value that 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>
```

See [Using the link Option](#).

-linkoffline url1 url2

Offers 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 the `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 workaround to update documents: After you've 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. 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 the `package-list` file:

The `url1` or `url2` value is the absolute or relative URL of the directory that contains the external Javadoc-generated documentation that 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.

See [Using the linkoffline Option](#).

-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 is 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 generates 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 at 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 effect on the `-bottom` option. The `-nonavbar` option is useful when you're interested only in the content and have no need for navigation, such as when you're converting the files to PostScript or PDF for printing only.

-noqualifier all |*packagename1:packagename2...*

Omits qualifying package names from class names in output. The argument to the `-noqualifier` option is either `all` (all package qualifiers are omitted) or a colon-separated list of packages, with wild cards, to be removed as qualifiers. 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 them by using `diff`, because it prevents time stamps from causing a different occurrence of `diff` (which would otherwise be a `diff` on every page). The time stamp includes the `javadoc` command release number, and currently appears similar to this:
<!-- Generated by javadoc (build 1.5.0_01) on Thu Apr 02 14:04:52 IST 2009 -->.

-notree

Omits the class/interface hierarchy pages from the generated documents. These are the pages that 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 that you want for the *filename* value and place it anywhere that you want for the path, it's 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 macOS: 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`

See [Examples of Running the javadoc Command](#).

For information about the file specified by *filename*, see [Overview Comment Files in Source Files](#).

The overview page is created only when you pass two or more package names to the `javadoc` command. For a further explanation, see [HTML Frames in Generated Files](#).

The title on the overview page is set by `-doctitle`.

-serialwarn

Generates compile-time warnings for missing `@serial` tags. 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.

-stylesheet *filename*

Specifies the path of an alternate HTML stylesheet file. Without this option, the `javadoc` command automatically creates a stylesheet file `stylesheet.css` that's 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 macOS:

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

Windows:

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

-tag *tagname* :Xaoptcmf:"taghead"

Enables the `javadoc` command to interpret a simple, one-argument `@tagname` custom block tag in documentation comments. For the `javadoc` command to spell-check tag names, it's important to include a `-tag` option for every custom tag that's present in the source code, disabling (with `x`) those that aren't being displayed in the current run.

The colon (`:`) is always the separator. The `-tag` option produces the tag heading *taghead* 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 value for *taghead* causes *tagname* to be the heading.

See [Using the Tag Option](#).

-taglet *class*

Specifies the class file that starts 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

Specifies the text to be placed at the top of each output file.

-use

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` includes 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 uses of only 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 documents. This text is omitted by default. To find out what version of the `javadoc` command you're 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 "Java Library" com.mypackage`.

-Xdoclint

Enables recommended checks for problems in `javadoc` comments.

-Xdoclint:(*all|none|[-] group*)

Reports warnings for bad references, lack of accessibility and 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. As always, 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`. Change what the `-Xdoclint` option reports with the following options:

-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 variable *group* 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 [HTML4 Specification](#) and [HTML5 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 the 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 subpackages of the given

package. Prefix the package specifier with `-` to disable checks for the specified packages.

`-Xdocrootparent url`

Replaces all `@docRoot` followed by `/..` in any doc comments with `url`.

Generated Files

You use the `javadoc` command as a Standard Doclet that generates HTML-formatted documentation.

The Standard Doclet generates the basic content, cross-reference, and support pages. Each HTML page corresponds to a separate file. The `javadoc` command generates two types of files. The first type is named after classes and interfaces. The second type contains hyphens (such as `package-summary.html`) to prevent conflicts with the first type of file.

Basic Content Pages

- One class or interface page (`classname.html`) for each class or interface being documented.
- One package page ([package-summary.html](#)) for each package being documented. The `javadoc` command includes any HTML text provided in a file with the name `package.html` or `package-info.java` in the `package` directory of the source tree.
- One overview page (`overview-summary.html`) for the entire set of packages. The overview page is the front page of the generated document. The `javadoc` command includes any HTML text provided in a file specified by the `-overview` option. The overview page is created only when you pass two or more package names into the `javadoc` command. See [HTML Frames](#) and [Javadoc Doclet Options](#).

Cross-Reference Pages

- One class hierarchy page for the entire set of packages (`overview-tree.html`). To view the hierarchy page, click **Overview** in the navigation bar and click **Tree**.
- One class hierarchy page for each package (`package-tree.html`). To view the hierarchy page, go to a particular package, class, or interface page, and click **Tree** to display the hierarchy for that package.
- One Use page for each package (`package-use.html`) and a separate Use page for each class and interface (`class-use/classname.html`). The Use page describes what packages, classes, methods, constructors and fields use any part of the specified class, interface, or package. For example, given a class or interface `A`, its Use page includes subclasses of `A`, fields declared as `A`, methods that return `A`, and methods and constructors with parameters of type `A`. To view the Use page, go to the package, class, or interface and click the **Use** link in the navigation bar.
- A deprecated API page (`deprecated-list.html`) that lists all deprecated APIs and their suggested replacements. Avoid deprecated APIs because they can be removed in future implementations.

A constant field values page (`constant-values.html`) for the values of static fields.

- A serialized form page (`serialized-form.html`) that provides information about serializable and externalizable classes with field and method descriptions. The information on this page is of interest to reimplementors, and not to developers who want to use the API. To access the serialized form page, go to any serialized class and click **Serialized Form** in the **See Also** section of the class comment. The Standard Doclet generates a serialized form page that lists any class (public or non-public) that implements `Serializable` with its `readObject` and `writeObject` methods, the fields that are serialized, and the documentation comments from the `@serial`, `@serialField`, and `@serialData` tags. Public `Serializable` classes can be excluded by marking them (or their package) with `@serial exclude`, and package-private `Serializable` classes can be included by marking them (or their package) with an `@serial include`. You can generate the complete serialized form for public and private classes by running the `javadoc` command without specifying the `-private` option. See [Javadoc Doclet Options](#).
- An index page (`index-*.html`) of all class, interface, constructor, field and method names, in alphabetical order. The index page is internationalized for Unicode and can be generated as a single file or as a separate file for each starting character (such as A–Z for English).

Support Pages

- A help page (`help-doc.html`) that describes the navigation bar and the previous pages. Use `-helpfile` to override the default help file with your own custom help file.
- One `index.html` file that creates the HTML frames for display. Load this file to display the front page with frames. The `index.html` file contains no text content.
- Several frame files (`*-frame.html`) that contains lists of packages, classes, and interfaces. The frame files display the HTML frames.
- A `package-list` file that is used by the `-link` and `-linkoffline` options. The package list file is a text file that is not reachable through links.
- A style sheet file (`stylesheet.css`) that controls a limited amount of color, font family, font size, font style, and positioning information on the generated pages.
- A `doc-files` directory that holds image, example, source code, or other files that you want copied to the destination directory. These files aren't processed by the `javadoc` command. This directory is not processed unless it exists in the source tree.

See [Javadoc Doclet Options](#).

HTML Frames

The `javadoc` command generates the minimum number of frames necessary (two or three) based on the values passed to the command. It omits the list of packages when you pass a single package name or source files that belong to a single package as an argument to the `javadoc` command. Instead, the `javadoc` command creates one frame in the left-hand column that displays the list of classes. When you pass two or more package names, the `javadoc` command creates a third frame that lists all packages and an overview page (`overview-summary.html`). The HTML frames are enabled by default, but can be disabled by the `--no-frames` option. To bypass frames, click the **No Frames** link or enter the page set from the `overview-summary.html` page. The Javadoc Search feature provides a better way to navigate and saves screen space.

Generated File Structure

The generated class and interface files are organized in the same directory hierarchy that Java source files and class files are organized. This structure is one directory per subpackage.

Oracle Solaris, Linux, and macOS: For example, the document generated for the `java.math.BigDecimal` class would be located at `java/math/BigDecimal.html`.

Windows: For example, the document generated for the `java.math.BigDecimal` class would be located at `java\math\BigDecimal.html`.

The file structure for the `java.math` package follows, assuming that the destination directory is named `apidocs`. All files that contain the word *frame* appear in the upper-left or lower-left frames, as noted. All other HTML files appear in the right-hand frame.

Directories are bold. The asterisks (*) indicate the files and directories that are omitted when the arguments to the `javadoc` command are source file names rather than package names. When arguments are source file names, an empty package list is created. The `doc-files` directory isn't created in the destination unless it exists in the source tree.

Generated API Declarations

The `javadoc` command generates a declaration at the start of each class, interface, field, constructor, and method description for that API item. For example, the declaration for the `Boolean` class is:

```
public final class Boolean
extends Object
implements Serializable
```

The declaration for the `Boolean.valueOf` method is:

```
public static Boolean valueOf(String s)
```

The `javadoc` command can include the modifiers `public`, `protected`, `private`, `abstract`, `final`, `static`, `transient`, and `volatile`, but not `synchronized` or `native`. The `synchronized` and `native` modifiers are considered implementation detail and not part of the API specification.

Rather than relying on the keyword `synchronized`, APIs should document their concurrency semantics in the main description of the comment. For example, a description might be:

```
A single enumeration cannot be used by multiple threads concurrently.
```

The document shouldn't describe how to achieve these semantics. As another example, while the `Hashtable` option should be thread-safe, there is no reason to specify that it's achieved by synchronizing all of its exported methods. It's better to reserve the right to synchronize internally for higher concurrency.

Examples of Running the javadoc Command

You can run the `javadoc` command on entire packages or individual source files. Use the public programmatic interface to call the `javadoc` command from within programs written in the Java language.

The release number of the `javadoc` command can be determined with the `javadoc -J-version` option. The release number of the Standard Doclet appears in the output stream. It can be turned off with the `-quiet` option.

Use the public programmatic interface in `com.sun.tools.javadoc.Main` (and the `javadoc` command is reentrant) to call the `javadoc` command from within programs written in the Java language.

The following instructions call the Standard HTML Doclet. To call a custom doclet, use the `-doclet` and `-docletpath` options.

Simple Examples

The following are simple examples of running the `javadoc` command on entire packages or individual source files. Each package name has a corresponding directory name.

Oracle Solaris, Linux, and macOS: In the following examples, the source files are located at `/home/src/java/awt/*.java`. The destination directory is `/home/html`.

Windows: In the following examples, the source files are located at `C:\home\src\java\awt*.java`. The destination directory is `C:\home\html`.

Document One or More Packages: To document a package, the source files for that package must be located in a directory that has the same name as the package.

Oracle Solaris, Linux, and macOS:

- If a package name has several identifiers (separated by dots, such as `java.awt.color`), then each subsequent identifier must correspond to a deeper subdirectory (such as `java/awt/color`).
- You can split the source files for a single package among two such directory trees located at different places, as long as the `-sourcepath` option points to them both. For example, `src1/java/awt/color` and `src2/java/awt/color`.

Windows:

- If a package name has several identifiers (separated by dots, such as `java.awt.color`), then each subsequent identifier must correspond to a deeper subdirectory (such as `java\awt\color`).
- You can split the source files for a single package among two such directory trees located at different places, as long as the `-sourcepath` option points to them both. For example, `src1\java\awt\color` and `src2\java\awt\color`.

You can run the `javadoc` command either by changing directories (with the `cd` command) or by using the `-sourcepath` option. The following examples illustrate both alternatives:

Example 1 Recursive Run from One or More Packages

This example uses `-sourcepath` so the `javadoc` command can be run from any directory for recursion. It traverses the subpackages of the Java directory excluding packages rooted at `java.net` and `java.lang`. Notice this excludes `java.lang.ref`, a subpackage of `java.lang`. To also traverse down other package trees, append their names to the `-subpackages` argument, such as `java:javax:org.xml.sax`.

```
javadoc -d /home/html -sourcepath /home/src -subpackages java -exclude
```

Example 2 Change to Root and Run Explicit Packages

1. Change to the parent directory of the fully qualified package.
2. Run the `javadoc` command with the names of one or more packages that you want to document:

Oracle Solaris, Linux, and macOS:

```
cd /home/src/  
javadoc -d /home/html java.awt java.awt.event
```

Windows:

```
cd C:\home\src\  
javadoc -d C:\home\html java.awt java.awt.event
```

To also traverse down other package trees, append their names to the `-subpackages` argument, such as `java:javax:org.xml.sax`.

Example 3 Run from Any Directory on Explicit Packages in One Tree

In this case, it doesn't matter what the current directory is. Run the `javadoc` command and use the `-sourcepath` option with the parent directory of the top-level package. Provide the names of one or more packages that you want to document:

Oracle Solaris, Linux, and macOS:

```
javadoc -d /home/html -sourcepath /home/src java.awt java.awt.event
```

Windows:

```
javadoc -d C:\home\html -sourcepath C:\home\src java.awt java.awt.event
```

Example 4 Run from Any Directory on Explicit Packages in Multiple Trees

Run the `javadoc` command and use the `-sourcepath` option with a colon-separated list of the paths to each tree's root. Provide the names of one or more packages that you want to document. All source files for a specified package don't need to be located under a single root directory, but they must be found somewhere along the source path.

Oracle Solaris, Linux, and macOS:

```
javadoc -d /home/html -sourcepath /home/src1:/home/src2 java.awt java.awt.event
```

Windows:

```
javadoc -d C:\home\html -sourcepath C:\home\src1;C:\home\src2 java.awt  
java.awt.event
```

The result is that all cases generate HTML-formatted documentation for the `public` and `protected` classes and interfaces in packages `java.awt` and `java.awt.event` and save the HTML files in the specified destination directory. Because two or more

packages are being generated, the document has three HTML frames: one for the list of packages, another for the list of classes, and the third for the main class pages.

Document One or More Classes

The second way to run the `javadoc` command is to pass one or more source files. You can run `javadoc` either of the following two ways: by changing directories (with the `cd` command) or by fully specifying the path to the source files. Relative paths are relative to the current directory. The `-sourcepath` option is ignored when passing source files. You can use command-line wildcards, such as an asterisk (*), to specify groups of classes.

Example 1 Change to the Source Directory

Change to the directory that holds the source files. Then run the `javadoc` command with the names of one or more source files, you want to document.

This example generates HTML-formatted documentation for the classes `Button`, `Canvas`, and classes that begin with `Graphics`. Because source files rather than package names were passed in as arguments to the `javadoc` command, the document has two frames: one for the list of classes and the other for the main page.

Oracle Solaris, Linux, and macOS:

```
cd /home/src/java/awt
javadoc -d /home/html Button.java Canvas.java Graphics*.java
```

Windows:

```
cd C:\home\src\java\awt
javadoc -d C:\home\html Button.java Canvas.java Graphics*.java
```

Example 2 Change to the Root Directory of the Package

This is useful for documenting individual source files from different subpackages off of the same root. Change to the package root directory, and specify the source files with paths from the root.

Oracle Solaris, Linux, and macOS:

```
cd /home/src/
javadoc -d /home/html java/awt/Button.java java/math/BigDecimal.java
```

Windows:

```
cd C:\home\src
javadoc -d \home\html java\awt\Button.java java\math\BigDecimal.java
```

Example 3 Document Files from Any Directory

In this case, it doesn't matter what the current directory is. Run the `javadoc` command with the absolute path (or path relative to the current directory) to the source files that you want to document.

Oracle Solaris, Linux, and macOS:

```
javadoc -d /home/html /home/src/java/awt/Button.java \
/home/src/java/awt/Graphics*.java
```

Windows:

```
javadoc -d C:\home\html C:\home\src\java\awt\Button.java ^
C:\home\src\java\awt\Graphics*.java
```

Document Packages and Classes

You can document entire packages and individual classes at the same time. The following is an example that mixes two of the previous examples. You can use the `-sourcepath` option for the path to the packages but not for the path to the individual classes.

Example 1
Oracle Solaris, Linux, and macOS:

```
avadoc -d /home/html -sourcepath /home/src java.awt \  
/home/src/java/math/BigDecimal.java
```

Example 2
Windows:

```
javadoc -d C:\home\html -sourcepath C:\home\src java.awt ^  
C:\home\src\java\math\BigDecimal.java
```

Notes

- If you omit the `-windowtitle` option, then the `javadoc` command copies the document title to the window title. The `-windowtitle` option text is similar to the `-doctitle` option, but without HTML tags to prevent those tags from appearing as just characters (plain text) in the window title.
- If you omit the `-footer` option, then the `javadoc` command copies the header text to the footer.
- Other important options you might want to use, but weren't needed in the previous example, are the `-classpath` and `-link` options.
- The `javadoc` command reads only files that contain valid class names. If the `javadoc` command isn't correctly reading the contents of a file, then verify that the class names are valid.