

# JavaDoc Tool

Vishnu Kotrajaras

Extraction from Java Documentation

## What for?

- Making HTML documentation
  - allows you to generate documentation for source files whose code is incomplete or erroneous.
  - So we can generate documentation before all debugging and troubleshooting is done.
  - classes you create must either be loaded as an extension or in the Javadoc tool's class path.

## Default page creation

- **Basic Content Pages**

- One **class or interface page** (*classname.html*) for each class or interface it is documenting.
- One **package page** (*package-summary.html*) for each package it is documenting. The Javadoc tool will include any HTML text provided in a file named *package.html* in the package directory of the source tree.
- One **overview page** (*overview-summary.html*) for the entire set of packages. This is the front page of the generated document. The Javadoc tool will include any HTML text provided in a file specified with the `-o` overview option. Note that this file is created only if you pass into javadoc two or more package names.

- **Cross-Reference Pages**

- One **class hierarchy page for the entire set of packages** (*overview-tree.html*). To view this, click on "Overview" in the navigation bar, then click on "Tree".
- One **class hierarchy page for each package** (*package-tree.html*) To view this, go to a particular package, class or interface page; click "Tree" to display the hierarchy for that package.
- One **"use" page for each package** (*package-use.html*) **and a separate one for each class and interface** (*class-use/classname.html*). 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.
  - You can access this page by first going to the package, class or interface, then clicking on the "Use" link in the navigation bar.

- A **deprecated API page** (deprecated-list.html) listing all deprecated names.
  - (A deprecated name is not recommended for use, generally due to improvements, and a replacement name is usually given)
- A **constant field values page** (constant-values.html) for the values of static fields.
- A **serialized form page** (serialized-form.html) for information about serializable and externalizable classes.
  - This information is of interest to re-implementors, not to developers using the API.
  - While there is no link in the navigation bar, you can get to this information by going to any serialized class and clicking "Serialized Form" in the "See also" section of the class comment.
  - The standard doclet automatically generates a **serialized form page**: any class (public or non-public) that implements Serializable is included, along with readObject and writeObject methods, the fields that are serialized, and the doc 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 `@serial include`.
- As of 1.4, you can generate the complete serialized form for public and private classes by running javadoc *without* specifying the `-private` option.
- An **index** (index-\*.html) of all class, interface, constructor, field and method names, alphabetically arranged. This 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 Files

- A **help page** (help-doc.html) that describes the navigation bar and the above pages.
  - You can provide your own custom help file to override the default using **-helpfile**.
- One **index.html file** which creates the HTML frames for display.
  - This is the file you load to display the front page with frames. This file itself contains no text content.
- Several **frame files** (\*-frame.html) containing lists of packages, classes and interfaces, used when HTML frames are being displayed.

- A **package list file** (package-list), used by the **-link** and **-linkoffline** options.
  - This is a text file, not HTML, and is not reachable through any links.
- A **style sheet file** (stylesheet.css) that controls a limited amount of color, font family, font size, font style and positioning on the generated pages.
- A **doc-files directory** that holds any image, example, source code or other files that you want copied to the destination directory.
  - These files are not processed by the Javadoc tool in any manner -- that is, any javadoc tags in them will be ignored. This directory is not generated unless it exists in the source tree.

## Package Comment Files

- Each package can have its own documentation comment, contained in its own "source" file, that the Javadoc tool will merge into the package summary page that it generates.
  - You typically include in this comment any documentation that applies to the entire package.
- To create a package comment file, you must name it **package.html** and place it in the package directory in the source tree along with the .java files.
  - The Javadoc tool will automatically look for this filename in this location. Notice that the filename is identical for all packages.

## Package Comment Files (cont.)

- The content of the package comment file is one big documentation comment, written in HTML, with one exception:
  - The documentation comment should not include the comment separators `/**` and `*/` or leading asterisks.
- When writing the comment, you should make the first sentence a summary about the package, and not put a title or any other text between `<body>` and the first sentence. You can include [package tags](#); as with any documentation comment, all tags except `{ @link }` must appear after the main description. If you add a `@see` tag in a package comment file, it must have a fully-qualified name.

## Package Comment Files (cont2.)

- the Javadoc tool does the following:
  - Copies all content between <body> and </body> tags for processing.
  - Processes any **package tags** that are present.
  - Inserts the processed text at the bottom of the package summary page it generates.
  - Copies the first sentence of the package comment to the top of the package summary page.
  - It also adds the package name and this first sentence to the list of packages on the overview page. The end-of-sentence is determined by the same rules used for the end of the first sentence of class and member main descriptions.

## Overview Comment File

- Each application or set of packages that you are documenting can have its own overview documentation comment, kept in its own "source" file, that the Javadoc tool will merge into the overview page that it generates.
- You typically include in this comment any documentation that applies to the entire application or set of packages.

## Overview Comment File (cont.)

- Name the file anything you want, typically **overview.html** and place it anywhere, typically at the top level of the source tree.
- You can have multiple overview comment files for the same set of source files, in case you want to run javadoc multiple times on different sets of packages.
- For example, if the source files for the java.applet package are contained in C:\user\src\java\applet directory, you could create an overview comment file at C:\user\src\overview.html.

## Overview Comment File (cont2.)

- The content of the overview comment file is one big documentation comment, written in HTML, like the package comment file described previously.
- you specify the overview comment file name with the **-overview** option. The file is then processed similar to that of a package comment file.

## Miscellaneous Unprocessed Files

- You can also include in your source any miscellaneous files that you want the Javadoc tool to copy to the destination directory.
- These typically includes graphic files, example Java source (.java) and class (.class) files, and self-standing HTML files whose content would overwhelm the documentation comment of a normal Java source file.
- To include unprocessed files, put them in a directory called **doc-files** which can be a subdirectory of any package directory that contains source files.
  - You can have one such subdirectory for each package.

## Miscellaneous Unprocessed Files (cont.)

- You might include images, example code, source files, .class files, applets and HTML files.
- For example, if you want to include the image of a button `button.gif` in the `java.awt.Button` class documentation, you place that file in the `/home/user/src/java/awt/doc-files/` directory. Notice the `doc-files` directory should not be located at `/home/user/src/java/doc-files` because `java` is not a package -- that is, it does not directly contain any source files.



## Miscellaneous Unprocessed Files (cont2.)

- All links to these unprocessed files must be hard-coded, because the Javadoc tool does not look at the files -- it simply copies the directory and all its contents to the destination.
- For example, the link in the Button.java doc comment might look like:

```
/**  
 * This button looks like this:  
 *   
 */
```

## Test Files and Template Files

- If you are passing in package names or wildcards, you need to follow certain rules to ensure these test files and templates files are not processed.
- **Test files**
  - Often developers want to put compilable, runnable test files for a given package in the *same* directory as the source files for that package.
  - But they want the test files to belong to a package other than the source file package.

## Test Files and Template Files (cont.)

- You need to put such test files in a subdirectory. For example, if you want to add test files for source files in `com.package1`, put them in a directory:
  - `com/package1/test-files/`
  - The test files will be ignored by the Javadoc tool (though this warning might appear: "No source files for package `com.package1.test-files`").
- If your test files contain doc comments, you can set up a separate run of the Javadoc tool to produce documentation of the test files by passing in source filenames with wildcards, such as
  - `com/package1/test-files/*.java`.

## Test Files and Template Files (cont2.)

- **Templates for source files**
  - Template files have names that often end in ".java" and are not compilable.
  - If you have a template for a source file that you want to keep in the source directory, you can name it with a dash (such as `Buffer-Template.java`), or any other illegal Java character, to prevent it from being processed.
  - This relies on the fact that the Javadoc tool will only process source files whose name, when stripped of the ".java" suffix, is actually a legal class name

## Command line

- **javadoc** [ options ] [ packagenames ] [ sourcefilenames ] [ -subpackages *pkg1:pkg2:...* ] [ @argfiles ]  
Arguments can be in any order.

- options
  - Command-line options, as specified in this document.
- packagenames
  - A series of names of packages, separated by spaces, such as `java.lang java.lang.reflect java.awt`.
  - You must separately specify each package you want to document. Wildcards are not allowed; use `-subpackages` for recursion.
  - The Javadoc tool uses `-sourcepath` to look for these package names.
- sourcefilenames
  - A series of source file names, separated by spaces, each of which can begin with a path and contain a wildcard such as asterisk (\*).
  - The Javadoc tool will process every file whose name ends with ".java", and whose name, when stripped of that suffix, is actually a legal class name.

- Therefore, you can name files with dashes (such as X-Buffer), or other illegal characters, to prevent them from being documented. This is useful for [test files and template files](#)
- The path that precedes the source file name determines where javadoc will look for the file. (The Javadoc tool does *not* use -sourcepath to look for these source file names.) Relative paths are relative to the current directory, so passing in Button.java is identical to ./Button.java. A source file name with an absolute path and a wildcard, for example, is /home/src/java/awt/Graphics\*.java.
- -subpackages *pkg1:pkg2:...*
  - Generates documentation from source files in the specified packages and recursively in their subpackages. An alternative to supplying packagenames or sourcefilenames.

- @argfiles
  - One or more files that contain a list of Javadoc options, packagenames and sourcefilenames in any order. Wildcards (\*) and -J options are not allowed in these files.



## Options

- All option names are case-insensitive, though their arguments can be case-sensitive.
- **-public**
  - Shows only public classes and members.
- **-protected**
  - Shows only protected and public classes and members. This is the default.
- **-package**
  - Shows only package, protected, and public classes and members.
- **-private**
  - Shows all classes and members.

## Options(2)

- **-doclet** *class*
  - Specifies the class file that starts the doclet used in generating the documentation.
  - Use the fully-qualified name.
  - defines the content and formats the output.
  - If the **-doclet** option is not used, javadoc uses the standard doclet for generating the default HTML format.
  - This class must contain the start(Root) method. The path to this starting class is defined by the **-docletpath** option.
  - For example, to call the MIF doclet, use:
    - `-doclet com.sun.tools.doclets.mif.MIFDoclet`

## Options(3)

- **-docletpath** *classpathlist*
  - Specifies the path to the doclet starting class file (specified with the -doclet option) and any jar files it depends on.
  - If the starting class file is in a jar file, then this specifies the path to that jar file.
  - You can specify an absolute path or a path relative to the current directory.
  - If *classpathlist* contains multiple paths or jar files, they should be separated with a colon (:) on Solaris and a semi-colon (;) on Windows. This option is not necessary if the doclet starting class is already in the search path.
  - Example
    - -docletpath C:\user\mifdoclet\lib\mifdoclet.jar  specify .jar
    - -docletpath C:\user\mifdoclet\classes\com\sun\tools\doclets\mif\  Don't need .class

## Options(4)

- **-sourcepath** *sourcepathlist*
  - Specifies the search paths for finding source files (.java) when passing package names or -subpackages into the javadoc command.
  - The *sourcepathlist* can contain multiple paths by separating them with a semicolon (;). The Javadoc tool will search in all subdirectories of the specified paths.
  - This option is also used 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 -- it will not locate .java files passed into the javadoc command. (To locate .java files, cd to that directory or include the path ahead of each file.

## Options(5)

- If `-sourcepath` is omitted, javadoc uses the class path to find the source files. Therefore, the default `-sourcepath` is the value of class path. If `-classpath` is omitted and you are passing package names into javadoc, it looks in the current directory (and subdirectories) for the source files.
- Set *sourcepathlist* to the root directory of the source tree for the package you are documenting.
- For example, suppose you want to document a package called `com.mypackage` whose source files are located at `C:\user\src\com\mypackage\*.java`. In this case you would specify the sourcepath to `C:\user\src`, the directory that contains `com\mypackage`, and then supply the package name `com.mypackage`:
  - `C:> javadoc -sourcepath C:\user\src com.mypackage`
- To point to two source paths:
  - `C:> javadoc -sourcepath C:\user1\src;C:\user2\src com.mypackage`

## Options(6)

- **`-classpath`** *classpathlist*
  - Specifies the paths where javadoc will look for [referenced classes](#) (.class files) -- these are the documented classes plus any classes referenced by those classes.
  - The *classpathlist* can contain multiple paths by separating them with a semicolon (;). The Javadoc tool will search in all subdirectories of the specified paths.
  - If `-sourcepath` is omitted, the Javadoc tool uses `-classpath` to find the source files as well as class files (for backward compatibility). Therefore, if you want to search for source and class files in separate paths, use both `-sourcepath` and `-classpath`.
  - For example, if you want to document `com.mypackage`, whose source files reside in the directory `C:\user\src\com\mypackage`, and if this package relies on a library in `C:\user\lib`, you would specify:
    - `C:> javadoc -classpath \user\lib -sourcepath \user\src com.mypackage`

## Options(7)

- As with other tools, if you do not specify `-classpath`, the Javadoc tool uses the `CLASSPATH` environment variable, if it is set. If both are not set, the Javadoc tool searches for classes from the current directory.
- **-verbose**
  - Provides more detailed messages while javadoc is running. 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 specifying the number of milliseconds to parse each java source file.
- **-quiet**
  - Shuts off non-error and non-warning messages, leaving only the warnings and errors appear, making them easier to view. Also suppresses the version string.

## Options(8)

- **-d *directory***
  - Specifies the destination directory where javadoc saves the generated HTML files.
  - Omitting this option causes the files to be saved to the current directory.
  - The value *directory* can be absolute, or relative to the current working directory.
  - As of 1.4, the destination directory is automatically created when javadoc is run.
  - For example, the following generates the documentation for the package `com.mypackage` and saves the results in the `C:\user\doc\` directory:
    - `C:> javadoc -d \user\doc com.mypackage`



## Options(9)

- **-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 given class or package.
  - Given class C, things that use class C would include subclasses of C, fields declared as C, methods that return C, and methods and constructors with parameters of type C.
  - For example:
    - The getName() method in the java.awt.Font class returns type String. Therefore, getName() uses String, and you will find that method on the "Use" page for String.
  - If a method uses String in its implementation but does not take a string as an argument or return a string, that is not considered a "use" of String.

## Options(10)

- **-windowtitle** *title*
  - Specifies the title to be placed in the HTML <title> tag. This appears in the window title and in any browser bookmarks (favorite places) that someone creates for this page.
  - If -windowtitle is omitted, the Javadoc tool uses the value of -doctitle for this option.
    - C:> javadoc -windowtitle "Java 2 Platform" com.mypackage
- **-doctitle** *title*
  - Specifies the title to be placed near the top of the overview summary file. The title will be placed as a centered, level-one heading directly beneath the upper navigation bar. The *title* may contain html tags and white space, though if it does, it must be enclosed in quotes.
    - C:> javadoc -doctitle "Java<sup><font size=2\>TM</font></sup>" com.mypackage

## Options(11)

- **-header** *header*
  - Specifies the header text to be placed at the top of each output file.
  - The header will be placed to the right of the upper navigation bar.
  - *header* may contain HTML tags and white space, but it must be enclosed in quotes.
  - Any internal quotation marks within *header* may have to be escaped.
    - C:> javadoc -header "<b>Java 2 Platform </b><br>v1.4" com.mypackage
- **-footer** *footer*
- **-bottom** *text*

## Options(12)

- **-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 are not 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 will necessarily be accessible via links.

## Options(13)

- Each link appears on the name of the identifier in its declaration. For example, the link to the source code of the Button class would be on the word "Button":
  - public class Button extends Component implements Accessible
- and the link to the source code of the `getLabel()` method in the Button class would be on the word "getLabel":
  - public String getLabel()

## Options(14)

- **-nodeprecated**
  - Prevents the generation of any deprecated API at all in the documentation.
  - This does what `-nodeprecatedlist` does, plus it does not generate any deprecated API throughout the rest of the documentation.
- **-nodeprecatedlist**
  - Prevents the generation of the file containing the **list** of deprecated APIs (`deprecated-list.html`) and the link in the navigation bar to that page. (However, javadoc continues to generate the deprecated API throughout the rest of the document.)
  - This is useful if your source code contains no deprecated API, and you want to make the navigation bar cleaner.

## Options(15)

- **-noindex**
  - Omits the index from the generated docs. The index is produced by default.
- **-nohelp**
  - Omits the HELP link in the navigation bars at the top and bottom of each page of output.
- **-nonavbar**
  - Prevents the generation of the navigation bar, header and footer, otherwise found at the top and bottom of the generated pages.
  - Has no affect on the "bottom" option.
  - The -nonavbar option is useful when you are interested only in the content and have no need for navigation, such as converting the files to PostScript or PDF for print only.

## Options(16)

- **-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 tool automatically creates a help file help-doc.html that is hard-coded in the Javadoc tool. This option enables you to override this default.
  - The *filename* can be any name and is not restricted to help-doc.html -- the Javadoc tool will adjust the links in the navigation bar accordingly. For example:
    - **C:> javadoc -helpfile C:\user\myhelp.html java.awt**

## Options(17)

- **-stylesheetfile** *path\filename*
  - Specifies the path of an alternate HTML stylesheet file.
  - Without this option, the Javadoc tool automatically creates a stylesheet file `stylesheet.css` that is hard-coded in the Javadoc tool. This option enables you to override this default.
  - The *filename* can be any name and is not restricted to `stylesheet.css`. For example:
    - **C:> javadoc -stylesheetfile C:\user\mystylesheet.css com.mypackage**

## Option (18- define our own tag)

- **-tag** *tagname*:**Xaoptcmf:"taghead"**
  - Enables the Javadoc tool to interpret a simple, one-argument custom block tag `@tagname` in doc comments. So the Javadoc tool can "spell-check" tag names.
  - It is important to include a `-tag` option for every custom tag that is present in the source code, disabling (with X) those that are not being output in the current run.
  - The colon (:) can be replaced with a dash (-), which frees up the colon character to appear in the *tagname*.
  - The `-tag` option outputs the tag's heading *taghead* in bold, followed on the next line by the text from its single argument.
  - This argument's text can contain inline tags, which are also interpreted.
  - Omitting *taghead* causes *tagname* to appear as the heading.

- **Placement of tags** - The **Xaoptcmf** part of the argument determines where in the source code the tag is allowed to be placed, and whether the tag can be disabled (using X). 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)

- **Examples of single tags** - An example of a tag option for a tag that that can be used anywhere in the source code is

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

- If you wanted `@todo` to be used only with constructors, methods and fields, you would use:

```
-tag todo:cmf:"To Do:"
```

- Notice the last colon (:) above is not a parameter separator, but is part of the heading text (as shown below). You would use either tag option for source code that contains the tag `@todo`, such as:

```
@todo The documentation for this method needs work.
```

- This line would produce output something like:

**To Do:**

The documentation for this method needs work.

- **Use of Colon as Separator** - The colon separator can be replaced by the dash, making the following two equivalent:

- tag todo:a:"To Do:"

- tag todo-a-"To Do:"

- This frees up the colon to be used in the tagname:

- tag ejb:bean-a-"EJB Bean:"

- **Spell-checking tag names (Disabling tags)** –

- Some developers put custom tags in the source code that they don't always want to output. In these cases, it is important to list all tags that are present 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 tool enough information to know if a tag it encounters is unknown, probably the results of a typo or a misspelling. It 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 @todo is a tag that you want to suppress on output, you would use:

- -tag todo:Xcmf:"To Do:"

- or, if you'd rather keep it simple:

- -tag todo:X

- The syntax -tag todo:X works even if @todo is defined by a taglet.

- **Order of tags** - The order of the `-tag` (and `-taglet`) options determine the order the tags are output. 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 cannot be altered.) This is illustrated in the following example.
- If `-tag` is missing, then the position of `-taglet` determines its order. If they are 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 `-taglet` and `-tag` both have the name "todo", the one that appears last on the command line will determine its 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 `@example` is a tag that might be encountered in the source code that should not be output during this run. Notice that if you use `@argfile`, you can put the tags on separate lines in an argument file like this (no line continuation characters needed):

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

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



- The standard tags are initially stored internally in a list in their default order. Whenever -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, it remains in its default position.
- **Avoiding Conflicts** - If you want to slice out your own namespace, you can use a dot-separated naming convention similar to that used for packages: com.mycompany.todo. Sun will continue to create standard tags whose names do not contain dots. Any tag you create will override the behavior of a tag by the same name defined by Sun. In other words, if you create a tag or taglet @todo, it will always have the same behavior you define, even if Sun later creates a standard tag of the same name.

## Options(19)

- **-subpackages** *package1:package2:...*
  - 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, as they are automatically included.
  - Each *package* argument is any top-level subpackage (such as java) or fully qualified package (such as javax.swing) that does not need to contain source files.
  - Arguments are separated by colons (on all operating systems). Wildcards are not needed or allowed. Use [-sourcepath](#) to specify where to find the packages.
  - For example:
    - C:> **javadoc -d docs -sourcepath C:\user\src -subpackages java:javax.swing**
    - This command generates documentation for packages named "java" and "javax.swing" and all their subpackages.

## Options(20)

- **-exclude** *packagename1:packagename2:...*
  - Unconditionally excludes the specified packages and their subpackages from the list formed by **-subpackages**. It excludes those packages even if they would otherwise be included by some previous or later **-subpackages** option. For example:
    - C:> **javadoc -sourcepath C:\user\src -subpackages java -exclude java.net:java.lang**
    - would include java.io, java.util, and java.math (among others), but would exclude packages rooted at java.net and java.lang. Notice this excludes java.lang.ref, a subpackage of java.lang).

## COMMAND LINE ARGUMENT FILES

- This enables you to create javadoc commands of any length on any operating system.
- An argument file can include Javadoc options, source filenames and package names in any combination, or just arguments to Javadoc options.
- The arguments within a file can be space-separated or newline-separated.
- Filenames within an argument file are relative to the current directory, not the location of the argument file.

- Wildcards (\*) are not allowed in these lists (such as for specifying \*.java). Use of the '@' character to recursively interpret files is not supported. The -J options are not supported because they are passed to the launcher, which does not support argument files.
- When executing javadoc, pass in the path and name of each argument file with the '@' leading character. When javadoc encounters an argument beginning with the character '@', it expands the contents of that file into the argument list.

## **Example - Single Arg File**

- You could use a single argument file named "argfile" to hold all Javadoc arguments:

```
C:> javadoc @argfile
```

## Example - Two Arg Files

- Create a file named "options" containing:
  - -d docs-filelist
  - -use -splitindex
  - -windowtitle 'Java 2 Platform v1.3 API Specification'
  - -doctitle 'Java<sup><font size="-2">TM</font></sup>  
2 Platform v1.4 API Specification'
  - -header '<b>Java 2 Platform </b><br><font size="-1">v1.4</font>'
  - -bottom 'Copyright 1993-2000 Sun Microsystems, Inc.  
All Rights Reserved.'
  - -group "Core Packages" "java.\*"
  - -overview  
  \java\pubs\ws\1.3\src\share\classes\overview-core.html
  - -sourcepath \java\pubs\ws\1.3\src\share\classes

- Create a file named "packages" containing:
  - com.mypackage1
  - com.mypackage2
  - com.mypackage3
- You would then run javadoc with:
  - C:> **javadoc @options @packages**

## Example - Arg Files with Paths

- The argument files can have paths, but any filenames inside the files are relative to the current working directory (not path1 or path2):
  - C:> **javadoc @path1\options @path2\packages**

## Example - Option Arguments

- You could create a file named "bottom" containing long argument
- Then run the Javadoc tool with:
  - C:> **javadoc -bottom @bottom @packages**
- Or you could include the -bottom option at the start of the argument file, and then just run it as:
  - C:> **javadoc @bottom @packages**

## Other running examples (1)

- This example uses `-sourcepath` so javadoc can be run from any directory and `-subpackages` (a new 1.4 option) 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`.
  - % **javadoc -d \home\html -sourcepath \home\src -subpackages java -exclude java.net:java.lang**
- To also traverse down other package trees, append their names to the `-subpackages` argument, such as `java:javafx:org.xml.sax`.

## Other running examples (2)

- Change to the parent directory of the fully-qualified package. Then run javadoc, supplying names of one or more packages you want to document:
  - C:> **cd C:\home\src\**
  - C:> **javadoc -d C:\home\html java.awt java.awt.event**

## Other running examples (3)

- In this case, it doesn't matter what the current directory is. Run javadoc supplying `-sourcepath` with the parent directory of the top-level package, and supplying names of one or more packages you want to document:
  - `C:> javadoc -d C:\home\html -sourcepath C:\home\src java.awt java.awt.event`

## Other running examples (4)

- **Run from any directory on explicit packages in multiple directory trees** - This is the same as case 3, but for packages in separate directory trees. Run javadoc supplying `-sourcepath` with the path to each tree's root (colon-separated) and supply names of one or more packages you want to document. All source files for a given package do not need to be located under a single root directory -- they just need to be found somewhere along the sourcepath.
  - `C:> javadoc -d C:\home\html -sourcepath C:\home\src1;C:\home\src2 java.awt java.awt.event`

## Other running examples (5)

- Change to the directory holding the .java files. Then run javadoc, supplying names of one or more source files you want to document.
  - C:> **cd C:\home\src\java\awt**
  - C:> **javadoc -d C:\home\html Button.java Canvas.java Graphics\*.java**
- This example generates HTML-formatted documentation for the classes Button, Canvas and classes beginning with Graphics. Because source files rather than package names were passed in as arguments to javadoc, the document has two frames -- for the list of classes and the main page.

## Other running examples (6)

- This is useful for documenting individual source files from different subpackages off the same root. Change to the package root directory, and supply the source files with paths from the root.
  - C:> **cd C:\home\src**
  - C:> **javadoc -d \home\html java\awt\Button.java java\applet\Applet.java**
- This example generates HTML-formatted documentation for the classes Button and Applet.



## Other running examples (7)

- In this case, it doesn't matter what the current directory is. Run javadoc supplying the absolute path (or path relative to the current directory) to the .java files you want to document.

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