

## CHAPTER 8

# *Java Development Tools*

Sun's implementation of Java includes a number of tools for Java developers. Chief among these are the Java interpreter and the Java compiler, of course, but there are a number of others as well. This chapter documents all the tools shipped with the Java 2 SDK (formerly known as the JDK), except for the RMI and IDL tools that are specific to enterprise programming. Those tools are documented in *Java Enterprise in a Nutshell* (O'Reilly).

The tools documented here are part of Sun's development kit; they are implementation details and not part of the Java specification itself. If you are using a Java development environment other than Sun's SDK (or a port of it), you should consult your vendor's tool documentation.

Some examples in this chapter use Unix conventions for file and path separators. If Windows is your development platform, change forward slashes in filenames to backward slashes, and colons in path specifications to semicolons.

### **appletviewer**

**JDK 1.0 and later**

#### **The Java Applet Viewer**

##### *Synopsis*

```
appletviewer [ options ] url | file...
```

##### *Description*

*appletviewer* reads or downloads the one or more HTML documents specified by the filename or URL on the command line. Next, it downloads any applets specified in any of those files and runs each applet in a separate window. If the specified document or documents do not contain any applets, *appletviewer* does nothing.

*appletviewer* recognizes applets specified with the `<APPLET>` tag and, in Java 1.2 and later, the `<OBJECT>` and `<EMBED>` tags.

##### *Options*

*appletviewer* recognizes the following options:



**-debug**

If this option is specified, *appletviewer* is started within *jdb* (the Java debugger). This allows you to debug the applets referenced by the document or documents.

**-encoding *enc***

This option specifies the character encoding that *appletviewer* should use when reading the contents of the specified files or URLs. It is used in the conversion of applet parameter values to Unicode. Java 1.1 and later.

**-J*javaoption***

This option passes the specified *javaoption* as a command-line argument to the Java interpreter. *javaoption* should not contain spaces. If a multiword option must be passed to the Java interpreter, multiple *-J* options should be used. See *java* for a list of valid Java interpreter options. Java 1.1 and later.

*appletviewer* also recognizes the *-classic*, *-native*, and *-green* options that the Java interpreter recognizes. See *java* for details on these options.

**Commands**

Each window displayed by *appletviewer* contains a single Applet menu, with the following commands available:

***Restart***

Stops and destroys the current applet, then reinitializes and restarts it.

***Reload***

Stops, destroys, and unloads the applet, then reloads, reinitializes, and restarts it.

***Stop***

Stops the current applet. Java 1.1 and later.

***Save***

Serializes the applet and saves the serialized applet in the file *Applet.ser* in the user's home directory. The applet should be stopped before selecting this option. Java 1.1 and later.

***Start***

Restarts a stopped applet. Java 1.1 and later.

***Clone***

Creates a new copy of the applet in a new *appletviewer* window.

***Tag***

Pops up a dialog box that displays the *<APPLET>* tag and all associated *<PARAM>* tags that created the current applet.

***Info***

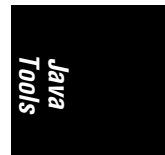
Pops up a dialog box that contains information about the applet. This information is provided by the *getAppletInfo()* and *getParameterInfo()* methods implemented by the applet.

***Edit***

This command is not implemented. The Edit menu item is disabled.

***Character Encoding***

Displays the current character encoding in the status line. Java 1.1 and later.





## *appletviewer*

### *Print*

Prints the applet. Java 1.1 and later.

### *Properties*

Displays a dialog that allows the user to set *appletviewer* preferences, including settings for firewall and caching proxy servers.

### *Close*

Closes the current *appletviewer* window.

### *Quit*

Quits *appletviewer*, closing all open windows.

### *Environment*

#### **CLASSPATH**

In Java 1.0 and Java 1.1, *appletviewer* uses the **CLASSPATH** environment variable in the same way the Java interpreter does. See *java* for details. In Java 1.2 and later, however, *appletviewer* ignores this environment variable to better simulate the action of a web browser.

### *Properties*

When it starts up, *appletviewer* reads property definitions from the file `~/hotjava/properties` (Unix) or `.hotjava\properties` relative to the **HOME** environment variable (Windows). These properties are stored in the system properties list and can specify the various error and status messages the applet viewer displays, as well as its security policies and use of proxy servers. The properties that affect security and proxies are described in the following sections. Most users of *appletviewer* do not need to use these properties.

### *Security properties*

The following properties specify the security restrictions *appletviewer* places on untrusted applets:

#### **acl.read**

A list of files and directories an untrusted applet is allowed to read. The elements of the list should be separated with colons on Unix systems and semicolons on Windows systems. On Unix systems, the `~` character is replaced with the home directory of the current user. If the plus sign appears as an element in the list, it is replaced by the value of the **acl.read.default** property. This provides an easy way to enable read access—by simply setting **acl.read** to `+`. By default, untrusted applets are not allowed to read any files or directories.

#### **acl.read.default**

A list of files and directories that are readable by untrusted applets if the **acl.read** property contains a plus sign.

#### **acl.write**

A list of files and directories an untrusted applet is allowed to write to. The elements of the list should be separated with colons on Unix systems and semicolons on Windows systems. On Unix systems, the `~` character is replaced with the home directory of the current user. If the plus sign appears as an element in the list, it is replaced by the value of the **acl.write.default** property. This provides an easy way to enable write access—by simply setting **acl.write** to `+`. By default, untrusted applets are not allowed to write to any files or directories.



**acl.write.default**

A list of files and directories that are writable by untrusted applets if the `acl.write` property contains a plus sign.

**appletviewer.security.mode**

Specifies the types of network access an untrusted applet is allowed to perform. If it is set to “none”, the applet can perform no networking at all. The value “host” is the default; it specifies that the applet can connect only to the host from which it was loaded. The value “unrestricted” specifies that an applet can connect to any host without restrictions.

**package.restrict.access.package-prefix**

Properties of this form can be set to `true` to prevent untrusted applets from using classes in any package that has the specified package name prefix as the first component of its name. For example, to prevent applets from using any of the Sun classes (such as the Java compiler and the applet viewer itself) that are shipped with the Java SDK, you can specify the following property:

```
package.restrict.access.sun=true
```

*appletviewer* sets this property to `true` by default for the `sun.*` and `netscape.*` packages.

**package.restrict.definition.package-prefix**

Properties of this form can be set to `true` to prevent untrusted applets from defining classes in a package that has the specified package name prefix as the first component of its name. For example, to prevent an applet from defining classes in any of the standard Java packages, you can specify the following property:

```
package.restrict.definition.java=true
```

*appletviewer* sets this property to `true` by default for the `java.*`, `sun.*`, and `netscape.*` packages.

**property.applet**

When a property of this form is set to `true` (as of Java 1.1), it specifies that an applet should be allowed to read the property named `property` from the system properties list. By default, applets are allowed to read only 10 standard system properties (as detailed in *Java Foundation Classes in a Nutshell* [O'Reilly]). For example, to allow an applet to read the `user.home` property, specify a property of the form:

```
user.home.applet=true
```

**Proxy properties**

*appletviewer* uses the following properties to configure its use of firewall and caching proxy servers:

**firewallHost**

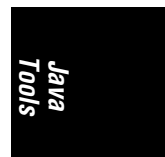
The firewall proxy host to connect to if `firewallSet` is `true`.

**firewallPort**

The port of the firewall proxy host to connect to if `firewallSet` is `true`.

**firewallSet**

Whether the applet viewer should use a firewall proxy. Values are `true` or `false`.





## *appletviewer*

### proxyHost

The caching proxy host to connect to if proxySet is true.

### proxyPort

The port of the caching proxy host to connect to if proxySet is true.

### proxySet

Whether the applet viewer should use a caching proxy. Values are true or false.

### *See also*

*java, javac, jdb*

## **extcheck**

**Java 2 SDK 1.2 and later**

### JAR Version Conflict Utility

#### *Synopsis*

```
extcheck [-verbose] jarfile
```

#### *Description*

*extcheck* checks to see if the extension contained in the specified *jarfile* (or a newer version of that extension) has already been installed on the system. It does this by reading the **Specification-Title** and **Specification-Version** manifest attributes from the specified *jarfile* and from all of the JAR files found in the system extensions directory.

*extcheck* is designed for use in automated installation scripts. Without the **-verbose** option, it does not print the results of its check. Instead, it sets its exit code to 0 if the specified extension does not conflict with any installed extensions and can be safely installed. It sets its exit code to a nonzero value if an extension with the same name is already installed and has a specification version number equal to or greater than the version of the specified file.

#### *Options*

##### **-verbose**

Lists the installed extensions as they are checked and displays the results of the check.

### *See also*

*jar*

## **jar**

**JDK 1.1 and later**

### Java Archive Tool

#### *Synopsis*

```
jar c|t|u|x|f|m|M|O|V [jar-file] [manifest] [-C directory] [input-files]
jar -i [jar-file]
```

#### *Description*

*jar* is a tool that can create and manipulate Java Archive (JAR) files. A JAR file is a ZIP file that contains Java class files, auxiliary resource files required by those classes, and optional meta-information. This meta-information includes a manifest file that lists the contents of the JAR archive and provides auxiliary information about each file.

The *jar* command can create JAR files, list the contents of JAR files, and extract files from a JAR archive. In Java 1.2 and later, it can also add files to an existing archive or update the manifest file of an archive. In Java 1.3 and later, *jar* can also add an index entry to a JAR file.



**Options**

The syntax of the *jar* command is reminiscent of the Unix *tar* (tape archive) command. Most options to *jar* are specified as a block of concatenated letters passed as a single argument, rather than as individual command-line arguments. The first letter of the first argument specifies what action *jar* is to perform; it is required. Other letters are optional. The various file arguments depend on which letters are specified.

**Command options**

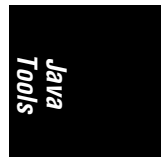
The first letter of the first option to *jar* specifies the basic operation *jar* is to perform. Here are the four possible options:

- c**  
Creates a new JAR archive. A list of input files and/or directories must be specified as the final arguments to *jar*. The newly created JAR file has a *META-INF/MANIFEST.MF* file as its first entry. This automatically created manifest lists the contents of the JAR file and contains a message digest for each file.
- t**  
Lists the contents of a JAR archive.
- u**  
Updates the contents of a JAR archive. Any files listed on the command line are added to the archive. When used with the *m* option, this adds the specified manifest information to the JAR file. Java 1.2 and later.
- x**  
Extracts the contents of a JAR archive. The files and directories specified on the command line are extracted and created in the current working directory. If no file or directory names are specified, all the files and directories in the JAR file are extracted.

**Modifier options**

Each of the four command specifier letters can be followed by additional letters that provide further detail about the operation to be performed:

- f**  
Indicates that *jar* is to operate on a JAR file whose name is specified on the command line. If this option is not present, *jar* reads a JAR file from standard input and/or writes a JAR file to standard output. If the *f* option is present, the command line must contain the name of the JAR file to operate on.
- m**  
When *jar* creates or updates a JAR file, it automatically creates (or updates) a manifest file named *META-INF/MANIFEST.MF* in the JAR archive. This default manifest simply lists the contents of the JAR file. Many JAR files require additional information to be specified in the manifest; the *m* option tells the *jar* command that a manifest template is specified on the command line. *jar* reads this manifest file and stores all the information it contains into the *META-INF/MANIFEST.MF* file it creates. This *m* option should be used only with the *c* or *u* commands, not with the *t* or *x* commands.
- M**  
Used with the *c* and *u* commands to tell *jar* not to create a default manifest file.





## *jar*

*v*

Tells *jar* to produce verbose output.

*0*

Used with the *c* and *u* commands to tell *jar* to store files in the JAR archive without compressing them. Note that this option is the digit zero, not the letter O.

### *Files*

The first option to *jar* consists of an initial command letter and various option letters. This first option is followed by a list of files:

*jar*

If the first option contains the letter *f*, that option must be followed by the name of the JAR file to create or manipulate.

*manifest*

If the first option contains the letter *m*, that option must be followed by the name of the file that contains manifest information. If the first option contains both the letters *f* and *m*, the JAR and manifest files should be listed in the same order the *f* and *m* options appear. In other words, if *f* comes before *m*, the JAR filename should come before the manifest filename. Otherwise, if *m* comes before *f*, the manifest filename should be specified before the JAR filename.

*files*

The list of one or more files and/or directories to be inserted into or extracted from the JAR archive.

### *Additional options*

In addition to all the options listed previously, *jar* also supports the following:

*-C dir*

Used within the list of files to process; it tells *jar* to change to the specified *dir* while processing the subsequent files and directories. The subsequent file and directory names are interpreted relative to *dir* and are inserted into the JAR archive without *dir* as a prefix. Any number of *-C* options can be used; each remains in effect until the next is encountered. The directory specified by a *-C* option is interpreted relative to the current working directory, not the directory specified by the previous *-C* option. Java 1.2 and later.

*-i jarfile*

The *-i* option is used instead of the *c*, *t*, *u*, and *x* commands. It tells *jar* to produce an index of all JAR files referenced by the specified *jarfile*. The index is stored in a file named *META-INF/INDEX.LIST*; a Java interpreter or applet viewer can use the information in this index to optimize its class and resource lookup algorithm and avoid downloading unnecessary JAR files. Java 1.3 and later.

### *Examples*

The *jar* command has a confusing array of options, but, in most cases, its use is quite simple. To create a simple JAR file that contains all the class files in the current directory and all files in a subdirectory called *images*, you can type:

```
% jar cf my.jar *.class images
```

To verbosely list the contents of a JAR archive:

```
% jar tvf your.jar
```



To extract the manifest file from a JAR file for examination or editing:

```
% jar xf the.jar META-INF/MANIFEST.MF
```

To update the manifest of a JAR file:

```
% jar ufm my.jar manifest.template
```

*See also*

*jarsigner*

## **jarsigner**

**Java 2 SDK 1.2 and later**

### **JAR Signing and Verification Tool**

#### *Synopsis*

```
jarsigner [options] jarfile signer
jarsigner -verify jarfile
```

#### *Description*

*jarsigner* adds a digital signature to the specified *jarfile*, or, if the *-verify* option is specified, it verifies the digital signature or signatures already attached to the JAR file. The specified *signer* is a case-insensitive nickname or alias for the entity whose signature is to be used. The specified *signer* name is used to look up the private key that generates the signature.

When you apply your digital signature to a JAR file, you are implicitly vouching for the contents of the archive. You are offering your personal word that the JAR file contains only nonmalicious code, files that do not violate copyright laws, and so forth. When you verify a digitally signed JAR file, you can determine who the signer or signers of the file are and (if the verification succeeds) that the contents of the JAR file have not been changed, corrupted, or tampered with since the signature or signatures were applied. Verifying a digital signature is entirely different from deciding whether or not you trust the person or organization whose signature you verified.

*jarsigner* and the related *keytool* program replace the *javakey* program of Java 1.1.

#### *Options*

*jarsigner* defines a number of options, many of which specify how a private key is to be found for the specified *signer*. Most of these options are unnecessary when using the *-verify* option to verify a signed JAR file:

##### **-certs**

If this option is specified along with either the *-verify* or *-verbose* option, it causes *jarsigner* to display details of the public-key certificates associated with the signed JAR file.

##### **-Jjavaoption**

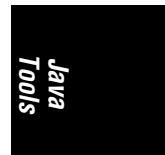
Passes the specified *javaoption* directly to the Java interpreter.

##### **-keypass password**

Specifies the password that encrypts the private key of the specified *signer*. If this option is not specified, *jarsigner* prompts you for the password.

##### **-keystore url**

A *keystore* is a file that contains keys and certificates. This option specifies the file-name or URL of the keystore in which the private- and public-key certificates of the specified *signer* are looked up. The default is the file named *.keystore* in the user's home directory (the value of the system property *user.home*). This is also the default location of the keystore managed by *keytool*.





## *jarsigner*

### **-sigfile *basename***

Specifies the base names of the *.SF* and *.DSA* files added to the *META-INF/* directory of the JAR file. If you leave this option unspecified, the base filename is chosen based on the *signer* name.

### **-signedjar *outputfile***

Specifies the name for the signed JAR file created by *jarsigner*. If this option is not specified, *jarsigner* overwrites the *jarfile* specified on the command line.

### **-storepass *password***

Specifies the password that verifies the integrity of the keystore (but does not encrypt the private key). If this option is omitted, *jarsigner* prompts you for the password.

### **-storetype *type***

Specifies the type of keystore specified by the **-keystore** option. The default is the system-default keystore type, which on most systems is the Java Keystore type, known as “JKS”. If you have the Java Cryptography Extension installed, you may want to use a “JCEKS” keystore instead.

### **-verbose**

Displays extra information about the signing or verification process.

### **-verify**

Specifies that *jarsigner* should verify the specified JAR file rather than sign it.

### ***See also***

*jar*, *keytool*, *javakey*

---

## **java**

**JDK 1.0 and later**

### **The Java Interpreter**

#### ***Synopsis***

```
java [ interpreter-options ] classname [ program-arguments ]  
java [ interpreter-options ] -jar jarfile [ program-arguments ]
```

#### ***Description***

*java* is the Java byte-code interpreter; it runs Java programs. The program to be run is the class specified by *classname*. This must be a fully qualified name: it must include the package name of the class, but not the *.class* file extension. For example:

```
% java david.games.Checkers  
% java Test
```

The specified class must define a *main()* method with exactly the following signature:

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

This method serves as the program entry point: the interpreter begins execution here.

In Java 1.2 and later, a program can be packaged in an executable JAR file. To run a program packaged in this fashion, use the **-jar** option to specify the JAR file. The manifest of an executable JAR file must contain a **Main-Class** attribute that specifies which class within the JAR file contains the *main()* method at which the interpreter is to begin execution.

Any command-line options that precede the name of the class or JAR file to execute are options to the Java interpreter itself. Any options that follow the class name or JAR file-name are options to the program; they are ignored by the Java interpreter and passed as an array of strings to the *main()* method of the program.



The Java interpreter runs until the `main()` method exits, and any threads (except for threads marked as daemon threads) created by the program have also exited.

#### *Interpreter versions*

The *java* program is the basic version of the Java interpreter. In addition to this program, however, there are several other versions of the Java interpreter. Each of these versions is similar to *java*, but has a specialized function. The various interpreter programs are the following:

##### *java*

This is the basic version of the Java interpreter; it is usually the correct one to use. The behavior and set of supported options changed significantly between Java 1.1 and Java 1.2, and there have been minor changes between other releases.

##### *oldjava*

This version of the interpreter is included in Java 1.2 and Java 1.3 for compatibility with the Java 1.1 interpreter. It loads classes using the Java 1.1 class-loading scheme. Very few Java applications need to use this version of the interpreter, and it has been removed from Java 1.4.

##### *javaw*

This version of the interpreter is included only on Windows platforms. Use *javaw* when you want to run a Java program (from a script, for example) without forcing a console window to appear. In Java 1.2 and Java 1.3, there is also an *oldjavaw* program that combines the features of *oldjava* and *javaw*.

##### *java\_g*

In Java 1.0 and Java 1.1, *java\_g* is a debugging version of the Java interpreter. It includes a few specialized command-line options, but is rarely used. Windows platforms also define a *javaw\_g* program. *java\_g* is not included in Java 1.2 or later versions.

#### *Client or Server VM*

Sun's "HotSpot" virtual machine comes in two versions: one is tuned for use with short-lived client applications and one for use with long-running server code. In Java 1.4, you can select the server version of the VM with the `-server` option. You can specify the "Client VM" (which is the default) with the `-client` option.

#### *Classic VM*

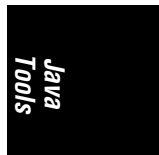
In Java 1.3, you can use the `-classic` option to specify that you want to use the "Classic VM" (essentially the same as the Java 1.2 VM) instead of the HotSpot VM (which uses incremental compilation). This option has been removed in Java 1.4.

#### *Just-in-time compiler*

In Java 1.2, and in Java 1.3 when you specify the `-classic` option, the Java interpreter uses a just-in-time compiler (if one is available for your platform). A JIT converts Java byte codes to native machine instructions at runtime and significantly speeds up the execution of a typical Java program. If you do not want to use the JIT, you can disable it by setting the `JAVA_COMPILER` environment variable to "NONE" or the `java.compiler` system property to "NONE" using the `-D` option:

```
% setenv JAVA_COMPILER NONE           // Unix csh syntax
% java -Djava.compiler=NONE MyProgram
```

If you want to use a different JIT compiler implementation, set the environment variable or system property to the name of the desired implementation. This environment variable and property are no longer used in Java 1.4, which uses the HotSpot VM, which includes efficient JIT technology.





## *java*

### *Threading systems*

On Solaris and related Unix platforms, you have a choice of the type of threads used by the Java 1.2 interpreter and the “Classic VM” of Java 1.3. To use native OS threads, specify `-native`. To use nonnative, or green, threads (the default), specify `-green`. In Java 1.3, the default “Client VM” uses native threads. Specifying `-green` or `-native` in Java 1.3 implicitly specifies `-classic` as well. These options are no longer supported (or necessary) in Java 1.4.

### *Options*

#### `-classic`

Runs the “Classic VM” instead of the default high-performance “Client VM.” Java 1.3 only.

#### `-classpath path`

Specifies the directories, JAR files, and ZIP files *java* searches when trying to load a class. In Java 1.0 and 1.1, and with the *oldjava* interpreter, this option specifies the location of system classes, extension classes, and application classes. In Java 1.2 and later, this option specifies only the location of application classes.

#### `-client`

Optimizes the incremental compilation of the HotSpot VM for typical client-side applications. Java 1.4 and later. See also the `-server` option.

#### `-cp`

A synonym for `-classpath`. Java 1.2 and later.

#### `-cs, -checksource`

Both options tell *java* to check the modification times on the specified class file and its corresponding source file. If the class file cannot be found or if it is out of date, it is automatically recompiled from the source. Java 1.0 and Java 1.1 only; these options are not available in Java 1.2 and later.

#### `-Dpropertyname=value`

Defines *propertyname* to equal *value* in the system properties list. Your Java program can then look up the specified value by its property name. You can specify any number of `-D` options. For example:

```
% java -Dawt.button.color=gray -Dmy.class.pointsize=14 my.class
```

#### `-d32`

Runs in 32-bit mode. This option is valid in Java 1.4 and later but is currently implemented only for Solaris platforms.

#### `-d64`

Runs in 64-bit mode. This option is valid in Java 1.4 and later but is currently implemented only for Solaris platforms.

#### `-da[:where]`

Disables assertions. See `-disableassertions`. Java 1.4 and later.

#### `-debug`

Causes *java* to start up in a way that allows the *jdb* debugger to attach itself to the interpreter session. In Java 1.2 and later, this option has been replaced with `-Xdebug`.



**-disableassertions[:where]**

Disables assertions. It is new in Java 1.4 and can be abbreviated **-da**. Used alone, it disables all assertions (except those in the system classes), which is the default. To disable assertions in a single class, follow the option with a colon and the fully qualified class name. To disable assertions in an entire package (and all of its sub-packages), follow this option with a colon, the name of the package, and three dots. See also **-enableassertions** and **-disablesystemassertions**.

**-disablesystemassertions**

Disables assertions in all system classes (which is the default). This option is new in Java 1.4. It can be abbreviated **-dsa** and takes no options.

**-dsa**

An abbreviation for **-disablesystemassertions**. Java 1.4 and later.

**-ea[:where]**

Enables assertions. An abbreviation for **-enableassertions**. Java 1.4 and later.

**-enableassertions[:where]**

Enables assertions. This option is new in Java 1.4 and can be abbreviated **-ea**. Used alone, it enables all assertions (except in system classes). To enable assertions in a single class, follow the option with a colon and the full class name. To enable assertions in an entire package (and all of its subpackages), follow the option with a colon, the name of the package, and three dots. See also **-disableassertions** and **-enablesystemassertions**.

**-enablesystemassertions**

Enables assertions in all system classes. May be abbreviated **-esa**. Java 1.4 and later.

**-esa**

An abbreviation for **-enablesystemassertions**. Java 1.4 and later.

**-green**

Selects nonnative, or green, threads on operating systems such as Solaris and Linux that support multiple styles of threading. This is the default in Java 1.2. In Java 1.3, using this option also selects the **-classic** option. See also **-native**. Java 1.2 and 1.3 only.

**-help, -?**

Prints a usage message and exits. See also **-X**.

**-jar jarfile**

Runs the specified executable *jarfile*. The manifest of the specified *jarfile* must contain a **Main-Class** attribute that identifies the class with the **main()** method at which program execution is to begin. Java 1.2 and later.

**-native**

Selects native threads, instead of the default green threads, on operating systems such as Solaris that support multiple styles of threading. Using native threads can be advantageous in some circumstances, such as when running on a multi-CPU computer. In Java 1.3, the default HotSpot virtual machine uses native threads. Java 1.2 and 1.3 only.

**-showversion**

Works like the **-version** option, except that the interpreter continues running after printing the version information. Java 1.3 and later.



## *java*

### `-verbose, -verbose:class`

Prints a message each time *java* loads a class. In Java 1.2 and later, you can use `-verbose:class` as a synonym.

### `-verbose:gc`

Prints a message when garbage collection occurs. Java 1.2 and later. Prior to Java 1.2, use `-verbosegc`.

### `-verbose:jni`

Prints a message when native methods are called. Java 1.2 and later.

### `-version`

Prints the version of the Java interpreter and exits.

### `-X`

Displays usage information for the nonstandard interpreter options (those beginning with `-X`) and exits. See also `-help`. Java 1.2 and later.

### `-Xbatch`

Tells the HotSpot VM to perform all just-in-time compilation in the foreground, regardless of the time required for compilation. Without this option, the VM compiles methods in the background while interpreting them in the foreground. Java 1.3 and later.

### `-Xbootclasspath:path`

Specifies a search path consisting of directories, ZIP files, and JAR files the *java* interpreter should use to look up system classes. Use of this option is very rare. Java 1.2 and later.

### `-Xbootclasspath/a:path`

Appends the specified *path* to the system classpath. Java 1.3 and later.

### `-Xbootclasspath/p:path`

Prepends the specified *path* to the system boot classpath. Java 1.3 and later.

### `-Xcheck:jni`

Performs additional checks when using Java Native Interface functions. Java 1.2 and later.

### `-Xdebug`

Starts the interpreter in a way that allows a debugger to communicate with it. Java 1.2 and later. Prior to Java 1.2, use `-debug`.

### `-Xfuture`

Strictly checks the format of all class files loaded. Without this option, *java* performs the same checks that were performed in Java 1.1. Java 1.2 and later.

### `-Xincgc`

Uses incremental garbage collection. In this mode the garbage collector runs continuously in the background, and a running program is rarely, if ever, subject to noticeable pauses while garbage collection occurs. Using this option typically results in a 10% decrease in overall performance, however. Java 1.3 and later.

### `-Xint`

Tells the HotSpot VM to operate in interpreted mode only, without performing any just-in-time compilation. Java 1.3 and later.



**-Xloggc:filename**

Logs garbage collection events with timestamps to the named file.

**-Xmixed**

Tells the HotSpot VM to perform just-in-time compilation on frequently used methods (“hotspots”) and execute other methods in interpreted mode. This is the default behavior. Contrast with **-Xbatch** and **-Xint**. Java 1.3 and later.

**-Xms initmem[k|m]**

Specifies how much memory is allocated for the heap when the interpreter starts up. By default, *initmem* is specified in bytes. You can specify it in kilobytes by appending the letter *k* or in megabytes by appending the letter *m*. The default is 1 MB. For large or memory-intensive applications (such as the Java compiler), you can improve runtime performance by starting the interpreter with a larger amount of memory. You must specify an initial heap size of at least 1,000 bytes. Java 1.2 and later. Prior to Java 1.2, use **-ms**.

**-Xmxmaxmem[k|m]**

Specifies the maximum heap size the interpreter uses for dynamically allocated objects and arrays. *maxmem* is specified in bytes by default. You can specify *maxmem* in kilobytes by appending the letter *k* and in megabytes by appending the letter *m*. The default is 16 MB. You cannot specify a heap size less than 1,000 bytes. Java 1.2 and later. Prior to Java 1.2, use **-mx**.

**-Xnoclassgc**

Does not garbage-collect classes. Java 1.2 and later. In Java 1.1, use **-noclassgc**.

**-Xprof**

Prints profiling output to standard output. Java 1.3 and later. In Java 1.2, or when using the **-classic** option, use **-Xrunhprof**. Prior to Java 1.2, use **-prof**.

**-Xrs**

Requests that the interpreter use fewer operating system signals. This option may improve performance on some systems. Java 1.2 and later.

**-Xrunhprof:suboptions**

Turns on CPU, heap, or monitor profiling. *suboptions* is a comma-separated list of *name=value* pairs. Use **-Xrunhprof:help** for a list of supported options and values. Java 1.2 and later. Prior to Java 1.2, rudimentary profiling support is available with the **-prof** option. In Java 1.3, this option is supported if **-classic** is used, but is not supported by the new HotSpot VM. See **-Xprof**.

**-Xsssize[k|m]**

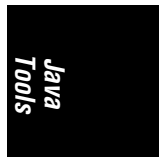
Sets the thread stack size in bytes, kilobytes, or megabytes. Java 1.3 and later.

**Loading classes**

The Java interpreter knows where to find the system classes that comprise the Java platform. In Java 1.2 and later, it also knows where to find the class files for all extensions installed in the system extensions directory. However, the interpreter must be told where to find the nonsystem classes that comprise the application to be run.

Class files are stored in directories that correspond to their package name. For example, the class `com.davidflanagan.util.Util` is stored in a file `com/davidflanagan/util/Util.class`. By default, the interpreter uses the current working directory as the root and looks for all classes in and beneath this directory.

The interpreter can also search for classes within ZIP and JAR files. To tell the interpreter where to look for classes, you specify a *classpath*: a list of directories and ZIP





## *java*

and JAR archives. When looking for a class, the interpreter searches each of the specified locations in the order in which they are specified.

The easiest way to specify a classpath is to set the `CLASSPATH` environment variable, which works much like the `PATH` variable used by a Unix shell or a Windows command-interpreter path. To specify a classpath in Unix, you might type a command like this:

```
% setenv CLASSPATH .:/myclasses:/usr/lib/javatools.jar:/usr/lib/javaapps
```

On a Windows system, you might use a command like the following:

```
C:\> set CLASSPATH=.;c:\myclasses;c:\javatools\classes.zip;d:\javaapps
```

Note that Unix and Windows use different characters to separate directory and path components.

You can also specify a classpath with the `-classpath` or `-cp` options to the Java interpreter. A path specified with one of these options overrides any path specified by the `CLASSPATH` environment variable. In Java 1.2 and later, the `-classpath` option specifies only the search path for application and user classes. Prior to Java 1.2, or when using the *oldjava* interpreter, this option specifies the search path for all classes, including system classes and extension classes.

*See also*

*javac, jdb*

## **javac**

**JDK 1.0 and later**

### **The Java Compiler**

#### *Synopsis*

```
javac [ options ] files  
oldjavac [ options ] files
```

#### *Description*

*javac* is the Java compiler; it compiles Java source code (in *.java* files) into Java byte codes (in *.class* files). The Java compiler is itself written in Java. The Java compiler has been completely rewritten in Java 1.3, and its performance has been substantially improved. Although the new *javac* is substantially compatible with previous versions of the compiler, the old version of the compiler is provided (in Java 1.3 only) as *oldjavac*.

*javac* can be passed any number of Java source files, whose names must all end with the *.java* extension. *javac* produces a separate *.class* class file for each class defined in the source files. Each source file can contain any number of classes, although only one can be a **public** top-level class. The name of the source file (minus the *.java* extension) must match the name of the **public** class it contains.

In Java 1.2 and later, if a filename specified on the command line begins with the character `@`, that file is taken not as a Java source file, but as a list of compiler options and Java source files. Thus, if you keep a list of Java source files for a particular project in a file named *project.list*, you can compile all those files at once with the command:

```
% javac @project.list
```

To compile a source file, *javac* must be able to find definitions of all classes used in the source file. It looks for definitions in both source-file and class-file form, automatically compiling any source files that have no corresponding class files or that have been modified since they were most recently compiled.



**Options****-bootclasspath *path***

Specifies the search *path* *javac* uses to look up system classes. This option is handy when you are using *javac* as a cross-compiler to compile classes against different versions of the Java API. For example, you might use the Java 1.3 compiler to compile classes against the Java 1.2 runtime environment. This option does not specify the system classes used to run the compiler itself, only the system classes read by the compiler. See also **-extdirs** and **-target**. Java 1.2 and later.

**-classpath *path***

Specifies the path *javac* uses to look up classes referenced in the specified source code. This option overrides any path specified by the **CLASSPATH** environment variable. The *path* specified is an ordered list of directories, ZIP files, and JAR archives, separated by colons on Unix systems or semicolons on Windows systems. If the **-sourcepath** option is not set, this option also specifies the search path for source files.

Prior to Java 1.2, this option specifies the path to system and extension classes, as well as user and application classes, and must be used carefully. In Java 1.2 and later, it specifies only the search path for application classes. See the discussion of “Loading classes” in the documentation for the *java* command for further information.

**-d *directory***

Specifies the directory in which (or beneath which) class files should be stored. By default, *javac* stores the *.class* files it generates in the same directory as the *.java* files those classes were defined in. If the **-d** option is specified, however, the specified *directory* is treated as the root of the class hierarchy, and *.class* files are placed in this directory or the appropriate subdirectory below it, depending on the package name of the class. Thus, the following command:

```
% javac -d /java/classes Checkers.java
```

places the file *Checkers.class* in the directory */java/classes* if the *Checkers.java* file has no **package** statement. On the other hand, if the source file specifies that it is in a package:

```
package com.davidflanagan.games;
```

the *.class* file is stored in */java/classes/com/davidflanagan/games*. When the **-d** option is specified, *javac* automatically creates any directories it needs to store its class files in the appropriate place.

**-depend**

Tells *javac* to recursively search for out-of-date class files in need of recompilation. This option forces a thorough compilation, but can slow the process down significantly. In Java 1.2 and later, this option has been renamed **-Xdepend**.

**-deprecation**

Tells *javac* to issue a warning for every use of a deprecated API. By default, *javac* issues only a single warning for each source file that uses deprecated APIs. Java 1.1 and later.

**-encoding *encoding-name***

Specifies the name of the character encoding used by the source files if it differs from the default platform encoding.



## *javac*

- extdirs *path***  
Specifies a list of directories to search for extension JAR files. It is used along with **-bootclasspath** when doing cross-compilation for different versions of the Java runtime environment. Java 1.2 and later.
- g**  
Tells *javac* to add line number, source file, and local variable information to the output class files, for use by debuggers. By default, *javac* generates only the line numbers.
- g:none**  
Tells *javac* to include no debugging information in the output class files. Java 1.2 and later.
- g:keyword-list**  
Tells *javac* to output the types of debugging information specified by the comma-separated *keyword-list*. The valid keywords are: **source**, which specifies source-file information; **lines**, which specifies line number information; and **vars**, which specifies local variable debugging information. Java 1.2 and later.
- help**  
Prints a list of options.
- Jjavaoption**  
Passes the argument *javaoption* directly through to the Java interpreter. For example: **-J-Xmx32m**. *javaoption* should not contain spaces; if multiple arguments must be passed to the interpreter, use multiple **-J** options. Java 1.1 and later.
- nowarn**  
Tells *javac* not to print warning messages. Errors are still reported as usual.
- nowrite**  
Tells *javac* not to create any class files. Source files are parsed as usual, but no output is written. This option is useful when you want to check that a file will compile without actually compiling it. Java 1.0 and Java 1.1 only; this option is not available in Java 1.2 and later.
- O**  
Enables optimization of class files to improve their execution speed. Using this option can result in larger class files that are difficult to debug and cause longer compilation times. Prior to Java 1.2, this option is incompatible with **-g**; turning on **-O** implicitly turns off **-g** and turns on **-depend**.
- source *release-number***  
Specifies the version of Java the code is written in. Use **-source 1.4** to compile code that uses the **assert** statement. This option also sets the **-target** option. Java 1.4 and later.
- sourcepath *path***  
Specifies the list of directories, ZIP files, and JAR archives that *javac* searches when looking for source files. The files found in this source path are compiled if no corresponding class files are found or if the source files are newer than the class files. By default, source files are searched for in the same places class files are searched for. Java 1.2 and later.



**-target *version***

Specifies the class-file-format version to use for the generated class files. The default *version* is 1.1, which generates class files that can be read and executed by Java 1.0 and later virtual machines. If you specify *version* as 1.2, *javac* increments the class file version number, producing a class file that does not run with a Java 1.0 or Java 1.1 interpreter. There have not been any actual changes to the Java class-file format; the new version number is simply a convenient way to prevent classes that depend on the many new features of Java 1.2 from being run on out-of-date interpreters.

**-verbose**

Tells the compiler to display messages about what it is doing. In particular, it causes *javac* to list all the source files it compiles, including files that did not appear on the command line.

**-X**

Tells the *javac* compiler to display usage information for its nonstandard options (all of which begin with -X). Java 1.2 and Java 1.4 and later.

**-Xdepend**

Tells *javac* to recursively search for source files that need recompilation. This causes a very thorough but time-consuming compilation process. Java 1.2 only; this option was removed in Java 1.3.

**-Xstdout**

Tells *javac* to send warning and error messages to the standard output stream instead of the standard error stream. Java 1.2 only.

**-Xstdout *filename***

Tells *javac* to send warning and error messages the specified file instead of writing them to the console. Java 1.4 and later.

**-Xswitchcheck**

Warns about **case** clauses in **switch** statements that “fall through.”

**-Xverbosepath**

Displays verbose output explaining where various class files and source files were found. Java 1.2 only.

**Environment****CLASSPATH**

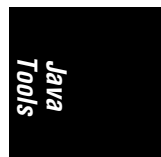
Specifies an ordered list (colon-separated on Unix, semicolon-separated on Windows systems) of directories, ZIP files, and JAR archives in which *javac* should look for user class files and source files. This variable is overridden by the **-classpath** option.

**See also**

*java*, *jdb*

**javadoc****JDK 1.0 and later****The Java Documentation Generator****Synopsis**

```
javadoc [ options ] @list package... sourcefiles...
```





## *javadoc*

### **Description**

*javadoc* generates API documentation, in HTML format (by default), for any number of packages and classes you specify. The *javadoc* command line can list any number of package names and any number of Java source files. For convenience, when working with a large number of command-line options, or a large number of package or class names, you can place them all in an auxiliary file and specify the name of that file on the command line, preceded by an @ character.

*javadoc* uses the *javac* compiler to process all the specified Java source files and all the Java source files in all the specified packages. It uses the information it gleans from this processing to generate detailed API documentation. Most importantly, the generated documentation includes the contents of all documentation comments included in the source files. See Chapter 7, for information about writing doc comments in your own Java code.

When you specify a Java source file for *javadoc* to process, you must specify the name of the file that contains the source, including a complete path to the file. It is more common, however, to use *javadoc* to create documentation for entire packages of classes. When you specify a package for *javadoc* to process, you specify the package name, not the directory that contains the source code for the package. In this case, you may need to specify the `-sourcepath` option so that *javadoc* can find your package source code correctly if it is not stored in a location already listed in your default classpath.

*javadoc* creates HTML documentation by default, but you can customize its behavior by defining a doclet class that generates documentation in whatever format you desire. You can write your own doclets using the doclet API defined by the `com.sun.javadoc` package. Documentation for this package is included in the standard documentation bundle for Java 1.2 and later.

*javadoc* has significant new functionality as of Java 1.2. This reference page documents the Java 1.2 and later versions of the program, but makes no attempt to distinguish new features of the Java 1.2 version from the features that existed in previous versions.

### **Options**

*javadoc* defines a large number of options. Some are standard options that are always recognized by *javadoc*. Other options are defined by the doclet that produces the documentation. The options for the standard HTML doclet are included in the following list:

#### **-1.1**

Simulates the output style and directory structure of the Java 1.1 version of *javadoc*. This option exists in Java 1.2 and 1.3 only, and has been removed in Java 1.4

#### **-author**

Includes authorship information specified with `@author` in the generated documentation. Default doclet only.

#### **-bootclasspath**

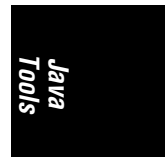
Specifies the location of an alternate set of system classes. This can be useful when cross-compiling. See *javac* for more information on this option.

#### **-bottom text**

Displays *text* at the bottom of each generated HTML file. *text* can contain HTML tags. See also `-footer`. Default doclet only.



- breakiterator**  
Uses the `java.text.BreakIterator` algorithm for determining the end of the summary sentence in doc comments. Default doclet only.
- charset *encoding***  
Specifies the character encoding for the output. This depends on the encoding used in the documentation comments of your source code, of course. The *encoding* value is used in a `<META>` tag in the HTML output. Default doclet only.
- classpath *path***  
Specifies a path *javadoc* uses to look up both class files and, if you do not specify the **-sourcepath** option, source files. Because *javadoc* uses the *javac* compiler, it needs to be able to locate class files for all classes referenced by the packages being documented. See *java* and *javac* for more information about this option and the default value provided by the `CLASSPATH` environment variable.
- d *directory***  
Specifies the directory in and beneath which *javadoc* should store the HTML files it generates. If this option is omitted, the current directory is used. Default doclet only.
- docencoding *encoding***  
Specifies the encoding to be used for output HTML documents. The name of the encoding specified here may not exactly match the name of the charset specified with the **-charset** option. Default doclet only.
- docfilessubdirs**  
Recursively copies any subdirectories of a *doc-files* directory instead of simply copying the files contained directly within *doc-files*. Default doclet only.
- doclet *classname***  
Specifies the name of the doclet class to use to generate the documentation. If this option is not specified, *javadoc* generates documentation using the default HTML doclet.
- docletpath *classpath***  
Specifies a path from which the class specified by the **-doclet** tag can be loaded if it is not available from the default classpath.
- doctitle *text***  
Provides a title to display at the top of the documentation overview file. This file is often the first thing readers see when they browse the generated documentation. The title can contain HTML tags. Default doclet only.
- encoding *encoding-name***  
Specifies the character encoding of the input source files and the documentation comments they contain. This can be different from the desired output encoding specified by **-docencoding**. The default is the platform default encoding.
- exclude *packages***  
Excludes the named packages from the set of packages defined by a **-subpackages** option. *packages* is a colon-separated list of package names. Default doclet only.
- excludedocfilessubdir *dirs***  
Excludes the specified subdirectories of a *doc-files* directory when **-docfilessubdirs** is specified. This is useful for excluding version control directories, for example. *dirs* is a colon-separated list of directory names relative to the *doc-files* directory. Default doclet only.





## *javadoc*

### **-extdirs *dirlist***

Specifies a list of directories to search for standard extensions. Only necessary when cross-compiling with **-bootclasspath**. See *javac* for details.

### **-footer *text***

Specifies text to be displayed near the bottom of each file, to the right of the navigation bar. *text* can contain HTML tags. See also **-bottom** and **-header**. Default doclet only.

### **-group *title packagelist***

*javadoc* generates a top-level overview page that lists all packages in the generated document. By default, these packages are listed in alphabetical order in a single table. You can break them into groups of related packages with this option, however. The *title* specifies the title of the package group, such as "Core Packages." The *packagelist* is a colon-separated list of package names, each of which can include a trailing \* character as a wildcard. The *javadoc* command line can contain any number of **-group** options. For example:

```
javadoc -group "AWT Packages" java.awt*
        -group "Swing Packages" javax.accessibility:javax.swing*
```

### **-header *text***

Specifies text to be displayed near the top of each file, to the right of the upper navigation bar. *text* can contain HTML tags. See also **-footer**, **-doctitle**, and **-windowtitle**. Default doclet only.

### **-help**

Displays a usage message for *javadoc*.

### **-helpfile *file***

Specifies the name of an HTML file that contains help for using the generated documentation. *javadoc* includes links to this help file in all files it generates. If this option is not specified, *javadoc* creates a default help file. Default doclet only.

### **-J*javaoption***

Passes the argument *javaoption* directly through to the Java interpreter. When processing a large number of packages, you may need to use this option to increase the amount of memory *javadoc* is allowed to use. For example:

```
% javadoc -J-Xmx64m
```

Note that because **-J** options are passed directly to the Java interpreter before *javadoc* starts up, they cannot be included in an external file specified on the command line with the **@list** syntax.

### **-link *url***

Specifies an absolute or relative URL of the top-level directory of another *javadoc*-generated document. *javadoc* uses this URL as the base URL for links from the current document to packages, classes, methods, and fields that are not documented in the current document. For example, when using *javadoc* to produce documentation for your own packages, you can use this option to link your documentation to the *javadoc* documentation for the core Java APIs. Default doclet only.

The directory specified by *url* must contain a file named *package-list*, and *javadoc* must be able to read this file at runtime. This file is automatically generated by a previous run of *javadoc*; it contains a list of all packages documented at the *url*.

More than one **-link** option can be specified, although this does not work properly in early releases of Java 1.2. If no **-link** option is specified, references in the



generated documentation to classes and members that are external to the documentation are not hyperlinked.

**-linkoffline *url packagelist***

Similar to the **-link** option, except that the *packagelist* file is explicitly specified on the command line. This is useful when the directory specified by *url* does not have a *package-list* file or when that file is not available when *javadoc* is run. Default doclet only.

**-linksource**

Creates an HTML version of each source file read and includes links to it from the documentation pages. Default doclet only.

**-locale *language\_country\_variant***

Specifies the locale to use for generated documentation. This is used to look up a resource file that contains localized messages and text for the output files.

**-nocomment**

Ignores all doc comments and generates documentation that includes only raw API information without any accompanying prose. Default doclet only.

**-nodeprecated**

Tells *javadoc* to omit documentation for deprecated features. This option implies **-nodeprecatedlist**. Default doclet only.

**-nodeprecatedlist**

Tells *javadoc* not to generate the *deprecated-list.html* file and not to output a link to it on the navigation bar. Default doclet only.

**-nohelp**

Tells *javadoc* not to generate a help file or a link to it in the navigation bar. Default doclet only.

**-noindex**

Tells *javadoc* not to generate index files. Default doclet only.

**-nonavbar**

Tells *javadoc* to omit the navigation bars from the top and bottom of every file. Also omits the text specified by **-header** and **-footer**. This is useful when generating documentation to be printed. Default doclet only.

**-noqualifier *packages* | all**

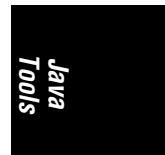
*javadoc* omits package names in its generated documentation for classes in the same package being documented. This option tells it to additionally omit package names for classes in the specified packages, or, if the **all** keyword is used, in all packages. *packages* is a colon-separated list of package names, which may include the **\*** wildcard to indicate subpackages. For example, **-noqualifier java.io:java.nio.\*** would exclude package names for all classes in the *java.io* package and in *java.nio* and its subpackages. Default doclet only.

**-nosince**

Ignores **@since** tags in doc comments. Default doclet only.

**-notree**

Tells *javadoc* not to generate the *tree.html* class hierarchy diagram or a link to it in the navigation bar. Default doclet only.





## *javadoc*

### **-overview *filename***

Reads an overview doc comment from *filename* and uses that comment in the overview page. This file does not contain Java source code, so the doc comment should not actually appear between **/\*\*** and **\*/** delimiters.

### **-package**

Includes package-visible classes and members in the output, as well as **public** and **protected** classes and members.

### **-private**

Includes all classes and members, including **private** and package-visible classes and members, in the generated documentation.

### **-protected**

Includes **public** and **protected** classes and members in the generated output. This is the default.

### **-public**

Includes only **public** classes and members in the generated output. Omits **protected**, **private**, and package-visible classes and members.

### **-quiet**

Suppresses output except warnings and error messages.

### **-serialwarn**

Issues warnings about serializable classes that do not adequately document their serialization format with **@serial** and related doc-comment tags. Default doclet only.

### **-source 1.4**

Specifies this option when running *javadoc* over Java 1.4 code that uses the new **assert** statement.

### **-sourcepath *path***

Specifies a search path for source files, typically set to a single root directory. *javadoc* uses this path when looking for the Java source files that implement a specified package.

### **-splitindex**

Generates multiple index files, one for each letter of the alphabet. Use this option when documenting large amounts of code. Otherwise, the single index file generated by *javadoc* will be too large to be useful. Default doclet only.

### **-stylesheetfile *file***

Specifies a file to use as a CSS stylesheet for the generated HTML. *javadoc* inserts appropriate links to this file in the generated documentation. Default doclet only.

### **-subpackages *packages***

Specifies that *javadoc* should process the specified packages and all of their sub-packages. *packages* is a colon-separated list of package names or package name prefixes. Using this option is often easier than explicitly listing all desired package names. For example:

**-subpackages java:javax**

See also **-exclude**. Default doclet only.



**-tag** *tagname:where:header-text*

Specifies that *javah* should handle a doc-comment tag named *tagname* by outputting the text *header-text* followed by whatever text follows the tag. This enables the use of simple custom tags (with the same syntax as `@return` and `@author`) in doc comments. *where* is a string of characters that specifies the types of doc comments in which this custom tag is allowed. The characters and their meanings are **a** (all: valid everywhere), **p** (packages), **t** (types: classes and interfaces), **c** (constructors), **m** (methods), and **f** (fields).

A secondary purpose of the **-tag** option is to specify the order in which tags are processed and in which their output appears. You can include the names of standard tags after the **-tag** option to specify this ordering. Custom tags and taglets can be included within this list of standard **-tag** options. Default doclet only.

**-taglet** *classname*

Specifies the classname of a “taglet” class to process a custom tag. Writing taglets is not covered here. **-taglet** tags may be interspersed with **-tag** tags to specify the order in which tags should be processed and output. Default doclet only.

**-tagletpath** *classpath*

Specifies a colon-separated list of JAR files or directories that form the classpath to be searched for taglet classes. Default doclet only.

**-use**

Generates and inserts links to an additional file for each class and package that lists the uses of the class or package.

**-verbose**

Displays additional messages while processing source files.

**-version**

Includes information from `@version` tags in the generated output. This option does not tell *javah* to print its own version number. Default doclet only.

**-windowtitle** *text*

Specifies *text* to be output in the `<TITLE>` tag of each generated file. This title typically appears in the web-browser titlebar and its history and bookmarks lists. *text* should not contain HTML tags. See also **-doctitle** and **-header**. Default doclet only.

**Environment****CLASSPATH**

This environment variable specifies the default classpath *javah* uses to find the class files and source files. It is overridden by the **-classpath** and **-sourcepath** options. See *java* and *javac* for further discussion of the classpath.

**See also**

*java*, *javac*

**javah**

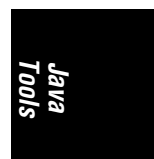
JDK 1.0 and later

**Native Method C Stub Generator****Synopsis**

```
javah [ options ] classnames
```

**Description**

*javah* generates C header and source files (*.h* and *.c* files) that are used when implementing Java native methods in C. The preferred native method interface has





## *javab*

changed between Java 1.0 and Java 1.1. In Java 1.1 and earlier, *javab* generates files for old-style native methods. In Java 1.1, the `-jni` option specifies that *javab* should generate new-style files. In Java 1.2 and later, this option becomes the default.

This reference page describes only how to use *javab*. A full description of how to implement Java native methods in C is beyond the scope of this book.

### *Options*

#### `-bootclasspath`

Specifies the path to search for system classes. See *javac* for further discussion. Java 1.2 and later.

#### `-classpath path`

Specifies the path *javab* uses to look up the classes named on the command line. This option overrides any path specified by the `CLASSPATH` environment variable. Prior to Java 1.2, this option can specify the location of the system classes and extensions. In Java 1.2 and later, it specifies only the location of application classes. See `-bootclasspath`. See also *java* for further discussion of the classpath.

#### `-d directory`

Specifies the directory into which *javab* stores the files it generates. By default, it stores them in the current directory. This option cannot be used with `-o`.

#### `-force`

Causes *javab* to always write output files, even if they contain no useful content.

#### `-help`

Causes *javab* to display a simple usage message and exit.

#### `-jni`

Specifies that *javab* should output header files for use with the new Java Native Interface (JNI), rather than using the old JDK 1.0 native interface. This option is the default in Java 1.2 and later. See also `-old`. Java 1.1 and later.

#### `-o outputfile`

Combines all output into a single file, *outputfile*, instead of creating separate files for each specified class.

#### `-old`

Outputs files for Java 1.0-style native methods. Prior to Java 1.2, this was the default. See also `-jni`. Java 1.2 and later.

#### `-stubs`

Generates `.c` stub files for the class or classes, instead of header files. This option is only for the Java 1.0 native methods interface. See `-old`.

#### `-td directory`

Specifies the directory where *javab* should store temporary files. On Unix systems, the default is */tmp*.

#### `-trace`

Specifies that *javab* should include tracing output commands in the stub files it generates. In Java 1.2 and later, this option is obsolete and has been removed. In its place, you can use the `-verbose:jni` option of the Java interpreter.



**-v, -verbose**

Specifies verbose mode. Causes *javap* to print messages about what it is doing. In Java 1.2 and later, **-verbose** is a synonym.

**-version**

Causes *javap* to display its version number.

### **Environment**

**CLASSPATH**

Specifies the default classpath *javap* searches to find the specified classes. See *java* for a further discussion of the classpath.

**See also**

*java*, *javac*

## **javap**

**JDK 1.0 and later**

### **The Java Class Disassembler**

#### **Synopsis**

*javap* [ *options* ] *classnames*

#### **Description**

*javap* reads the class files specified by the class names on the command line and prints a human-readable version of the API defined by those classes. *javap* can also disassemble the specified classes, displaying the Java VM byte codes for the methods they contain.

#### **Options**

**-b**

Enables backward compatibility with the output of the Java 1.1 version of *javap*. This option exists for programs that depend on the precise output format of *javap*. Java 1.2 and later.

**-bootclasspath *path***

Specifies the search path for the system classes. See *javac* for information about this rarely used option. Java 1.2 and later.

**-c**

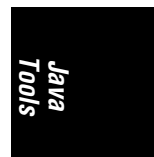
Displays the code (i.e., Java VM byte codes) for each method of each specified class. This option always disassembles all methods, regardless of their visibility level.

**-classpath *path***

Specifies the path *javap* uses to look up the classes named on the command line. This option overrides the path specified by the **CLASSPATH** environment variable. Prior to Java 1.2, this argument specifies the path for all system classes, extensions, and application classes. In Java 1.2 and later, it specifies only the application classpath. See also **-bootclasspath** and **-extdirs**. See *java* and *javac* for more information on the classpath.

**-extdirs *dirs***

Specifies one or more directories that should be searched for extension classes. See *javac* for information about this rarely used option. Java 1.2 and later.





## *javap*

- l  
Displays tables of line numbers and local variables, if available in the class files. This option is typically useful when used only with *-c*. The *javac* compiler does not include local variable information in its class files by default. See *-g* and related options to *javac*.
- help  
Prints a usage message and exits.
- Jjavaoption  
Passes the specified *javaoption* directly to the Java interpreter.
- package  
Displays package-visible, **protected**, and **public** class members, but not **private** members. This is the default.
- private  
Displays all class members, including **private** members.
- protected  
Displays only **protected** and **public** members.
- public  
Displays only **public** members of the specified classes.
- s  
Outputs the class member declarations using the internal VM type and method signature format, instead of the more readable source-code format.
- verbose  
Specifies verbose mode. Outputs additional information (in the form of Java comments) about each member of each specified class.
- verify  
Causes *javap* to perform partial class verification on the specified class or classes and display the results. Java 1.0 and 1.1. only; this option has been removed in Java 1.2 and later because it does not perform a sufficiently thorough verification.
- version  
Causes *javap* to display its version number.

### *Environment*

#### CLASSPATH

Specifies the default search path for application classes. The *-classpath* option overrides this environment variable. See *java* for a discussion of the classpath.

#### *See also*

*java, javac*

## **jdb**

**JDK 1.0 and later**

### **The Java Debugger**

#### *Synopsis*

```
jdb [ options ] class [ program options ]  
jdb connect options
```



**Description**

*jdb* is a debugger for Java classes. It is text-based, command-line-oriented, and has a command syntax like that of the Unix *dbx* or *gdb* debuggers used with C and C++ programs.

*jdb* is written in Java, so it runs within a Java interpreter. When *jdb* is invoked with the name of a Java class, it starts another copy of the *java* interpreter, using any interpreter options specified on the command line. The new interpreter is started with special options that enable it to communicate with *jdb*. The new interpreter loads the specified class file and then stops and waits for debugging commands before executing the first byte code.

*jdb* can also debug a program that is already running in another Java interpreter. Doing so requires special options be passed to both the *java* interpreter and to *jdb*. The Java debugging architecture has changed dramatically with the introduction of Java 1.3, and so have the *java* and *jdb* options used to allow *jdb* to connect to a running interpreter.

***jdb* expression syntax**

*jdb* debugging commands such as *print*, *dump*, and *suspend* allow you to refer to classes, objects, methods, fields, and threads in the program being debugged. You can refer to classes by name, with or without their package names. You can also refer to *static* class members by name. You can refer to individual objects by object ID, which is an eight-digit hexadecimal integer. Or, when the classes you are debugging contain local variable information, you can often use local variable names to refer to objects. You can use normal Java syntax to refer to the fields of an object and the elements of an array; you can also use this syntax to write quite complex expressions. In Java 1.3, *jdb* even supports method invocation using standard Java syntax.

A number of *jdb* commands require you to specify a thread. Each thread is given an integer identifier and is named using the syntax *t@n*, where *n* is the thread ID.

**Options**

When invoking *jdb* with a specified class file, any of the *java* interpreter options can be specified. See the *java* reference page for an explanation of these options. In addition, *jdb* supports the following options:

**-attach *[host:]port***

Specifies that *jdb* should connect to the Java “Client VM” that is already running on the specified host (or the local host, if unspecified) and listening for debugging connections on the specified port. Java 1.3 and later.

In order to use *jdb* to connect to a running VM in this way, the VM must have been started with a command line something like this:

```
% java -Xdebug -Xrunjwp:transport=dt_socket,address=8000,server=y,suspend=n
```

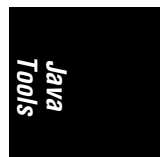
The Java 1.3 *jdb* architecture allows a complex set of interpreter-to-debugger connection options, and *java* and *jdb* provide a complex set of options and suboptions to enable it. A detailed description of those options is beyond the scope of this book.

**-help**

Displays a usage message listing supported options.

**-host *hostname***

In Java 1.2 and earlier, this option is used to connect to an already running interpreter. It specifies the name of the host upon which the desired interpreter session is running. If omitted, the default is the local host. This option must be used with *-password*. In Java 1.3, this option has been replaced by the *-attach* option.





## *jdb*

### **-launch**

Starts the specified application when *jdb* starts. This avoids the need to explicitly use the `run` command to start it. Java 1.3 and later.

### **-password *password***

In Java 1.2 and earlier, this option specifies a password that uniquely identifies a Java VM on a particular host. When used in conjunction with `-hostname`, this option enables *jdb* to connect to a running interpreter. The interpreter must have been started with the `-debug` or `-Xdebug` option, which causes it to display an appropriate *password* for use with this option. In Java 1.3, this option has been replaced by the `-attach` option.

### **-sourcepath *path***

Specifies the locations *jdb* searches when attempting to find source files that correspond to the class files being debugged. If unspecified, *jdb* uses the classpath by default. Java 1.3 and later.

### **-tclassic**

Tells *jdb* to invoke the “Classic VM” instead of the “Client VM” (HotSpot), which is the default VM in Java 1.3. Java 1.3 and later.

### **-version**

Displays the *jdb* version number and exits.

## **Commands**

*jdb* understands the following debugging commands:

### **? or help**

Lists all supported commands, with a short explanation of each.

### **!!**

A shorthand command that is replaced with the text of the last command entered. It can be followed with additional text that is appended to that previous command.

### **catch [ *exception-class* ]**

Causes a breakpoint whenever the specified exception is thrown. If no exception is specified, the command lists the exceptions currently being caught. Use `ignore` to stop these breakpoints from occurring.

### **classes**

Lists all classes that have been loaded.

### **clear**

Lists all currently set breakpoints.

### **clear *class.method* [(*param-type* . . . )]**

Clears the breakpoint set in the specified method of the specified class.

### **clear [ *class:line* ]**

Removes the breakpoint set at the specified line of the specified class.

### **cont**

Resumes execution. This command should be used when the current thread is stopped at a breakpoint.



**down** [ *n* ]

Moves down *n* frames in the call stack of the current thread. If *n* is not specified, moves down one frame.

**dump** *id* . . .

Prints the value of all fields of the specified object or objects. If you specify the name of a class, **dump** displays all class (static) methods and variables of the class, and also displays the superclass and list of implemented interfaces. Objects and classes can be specified by name or by their eight-digit hexadecimal ID numbers. Threads can also be specified with the shorthand **t@thread-number**.

**exit** or **quit**

Quits *jdb*.

**gc**

Runs the garbage collector to force unused objects to be reclaimed.

**ignore** *exception-class*

Does not treat the specified exception as a breakpoint. This command turns off a **catch** command. This command does not cause the Java interpreter to ignore exceptions; it merely tells *jdb* to ignore them.

**list** [ *line-number* ]

Lists the specified line of source code as well as several lines that appear before and after it. If no line number is specified, uses the line number of the current stack frame of the current thread. The lines listed are from the source file of the current stack frame of the current thread. Use the **use** command to tell *jdb* where to find source files.

**list** *method*

Displays the source code of the specified method.

**load** *classname*

Loads the specified class into *jdb*.

**locals**

Displays a list of local variables for the current stack frame. Java code must be compiled with the **-g** option in order to contain local variable information.

**memory**

Displays a summary of memory usage for the Java program being debugged.

**methods** *class*

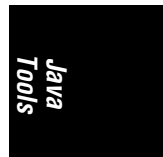
Lists all methods of the specified class. Use **dump** to list the instance variables of an object or the class (static) variables of a class.

**print** *id* . . .

Prints the value of the specified item or items. Each item can be a class, object, field, or local variable, and can be specified by name or by eight-digit hexadecimal ID number. You can also refer to threads with the special syntax **t@thread-number**. The **print** command displays an object's value by invoking its **toString()** method.

**next**

Executes the current line of source code, including any method calls it makes. See also **step**.





## *jdb*

**resume** [ *thread-id* . . . ]

Resumes execution of the specified thread or threads. If no threads are specified, all suspended threads are resumed. See also **suspend**.

**run** [ *class* ] [ *args* ]

Runs the **main()** method of the specified class, passing the specified arguments to it. If no class or arguments are specified, uses the class and arguments specified on the *jdb* command line.

**step**

Runs the current line of the current thread and stops again. If the line invokes a method, steps into that method and stops. See also **next**.

**stepi**

Executes a single Java VM instruction.

**step up**

Runs until the current method returns to its caller and stops again.

**stop**

Lists current breakpoints.

**stop at** *class:line*

Sets a breakpoint at the specified line of the specified class. Program execution stops when it reaches this line. Use **clear** to remove a breakpoint.

**stop in** *class.method* [(*param-type* . . . )]

Sets a breakpoint at the beginning of the specified method of the specified class. Program execution stops when it enters the method. Use **clear** to remove a breakpoint.

**suspend** [ *thread-id* . . . ]

Suspends the specified thread or threads. If no threads are specified, suspends all running threads. Use **resume** to restart them.

**thread** *thread-id*

Sets the current thread to the specified thread. This thread is used implicitly by a number of other *jdb* commands. The thread can be specified by name or number.

**threadgroup** *name*

Sets the current thread group.

**threadgroups**

Lists all thread groups running in the Java interpreter session being debugged.

**threads** [ *threadgroup* ]

Lists all threads in the named thread group. If no thread group is specified, lists all threads in the current thread group (specified by **threadgroup**).

**up** [ *n* ]

Moves up *n* frames in the call stack of the current thread. If *n* is not specified, moves up one frame.

**use** [ *source-file-path* ]

Sets the path used by *jdb* to look up source files for the classes being debugged. If no path is specified, displays the current source path.



where *[thread-id]* *[all]*

Displays a stack trace for the specified thread. If no thread is specified, displays a stack trace for the current thread. If *all* is specified, displays a stack trace for all threads.

wherei *[thread-id x]*

Displays a stack trace for the specified or current thread, including detailed program counter information.

### Environment

#### CLASSPATH

Specifies an ordered list (colon-separated on Unix, semicolon-separated on Windows systems) of directories, ZIP files, and JAR archives in which *jdb* should look for class definitions. When a path is specified with this environment variable, *jdb* always implicitly appends the location of the system classes to the end of the path. If this environment variable is not specified, the default path is the current directory and the system classes. This variable is overridden by the *-classpath* option.

*See also*

*java*

## keytool

Java 2 SDK 1.2 and later

### Key and Certificate Management Tool

#### Synopsis

*keytool command options*

#### Description

*keytool* manages and manipulates a *keystore*: a repository for public and private keys and public-key certificates. *keytool* defines various commands for generating keys, importing data into the keystore, and exporting and displaying keystore data. Keys and certificates are stored in a keystore using a case-insensitive name, or *alias*. *keytool* uses this alias to refer to a key or certificate.

The first option to *keytool* always specifies the basic command to be performed. Subsequent options provide details about how the command is to be performed. Only the command must be specified. If a command requires an option that does not have a default value, *keytool* prompts you interactively for the value.

#### Commands

##### -certreq

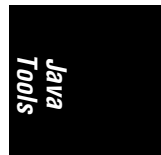
Generates a certificate signing request in PKCS#10 format for the specified alias. The request is written to the specified file or to the standard output stream. The request should be sent to a certificate authority (CA), which authenticates the requestor and sends back a signed certificate authenticating the requestor's public key. This signed certificate can then be imported into the keystore with the *-import* command. This command uses the following options: *-alias*, *-file*, *-keypass*, *-keystore*, *-sigalg*, *-storepass*, *-storetype*, and *-v*.

##### -delete

Deletes a specified alias from a specified keystore. This command uses the following options: *-alias*, *-keystore*, *-storepass*, *-storetype*, and *-v*.

##### -export

Writes the certificate associated with the specified alias to the specified file or to standard output. This command uses the following options: *-alias*, *-file*, *-keystore*, *-rfc*, *-storepass*, *-storetype*, and *-v*.





## *keytool*

### **-genkey**

Generates a public/private key pair and a self-signed X.509 certificate for the public key. Self-signed certificates are not often useful by themselves, so this command is often followed by *-certreq*. This command uses the following options: *-alias*, *-dname*, *-keyalg*, *-keypass*, *-keysize*, *-keystore*, *-sigalg*, *-storepass*, *-storetype*, *-v*, and *-validity*.

### **-help**

Lists all available *keytool* commands and their options. This command is not used with any other options.

### **-identitydb**

Reads keys and certificates from a Java 1.1 identity database managed with *javakey* and stores them into a keystore so they can be manipulated by *keytool*. The identity database is read from the specified file or from standard input if no file is specified. The keys and certificates are written into the specified keystore file, which is automatically created if it does not exist yet. This command uses the following options: *-file*, *-keystore*, *-storepass*, *-storetype*, and *-v*.

### **-import**

Reads a certificate or PKCS#7-formatted certificate chain from a specified file or from standard input and stores it as a trusted certificate in the keystore with the specified alias. This command uses the following options: *-alias*, *-file*, *-keypass*, *-keystore*, *-noprompt*, *-storepass*, *-storetype*, *-trustcacerts*, and *-v*.

### **-keyclone**

Duplicates the keystore entry of a specified alias and stores it in the keystore under a new alias. This command uses the following options: *-alias*, *-dest*, *-keypass*, *-keystore*, *-new*, *-storepass*, *-storetype*, and *-v*.

### **-keypasswd**

Changes the password that encrypts the private key associated with a specified alias. This command uses the following options: *-alias*, *-keypass*, *-new*, *-storetype*, and *-v*.

### **-list**

Displays (on standard output) the fingerprint of the certificate associated with the specified alias. With the *-v* option, prints certificate details in human-readable format. With *-rfc*, prints certificate contents in a machine-readable, printable-encoding format. This command uses the following options: *-alias*, *-keystore*, *-rfc*, *-storepass*, *-storetype*, and *-v*.

### **-printcert**

Displays the contents of a certificate read from the specified file or from standard input. Unlike most *keytool* commands, this one does not use a keystore. This command uses the following options: *-file* and *-v*.

### **-selfcert**

Creates a self-signed certificate for the public key associated with the specified alias and uses it to replace any certificate or certificate chain already associated with that alias. This command uses the following options: *-alias*, *-dname*, *-keypass*, *-keystore*, *-sigalg*, *-storepass*, *-storetype*, *-v*, and *-validity*.

### **-storepasswd**

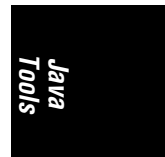
Changes the password that protects the integrity of the keystore as a whole. The new password must be at least six characters long. This command uses the following options: *-keystore*, *-new*, *-storepass*, *-storetype*, and *-v*.



**Options**

The various *keytool* commands can be passed various options from the following list. Many of these options have reasonable default values. *keytool* interactively prompts for any unspecified options that do not have defaults:

- alias *name***  
Specifies the alias to be manipulated in the keystore. The default is “mykey”.
- dest *newalias***  
Specifies the new alias name (the destination alias) for the **-keyclone** command. If not specified, *keytool* prompts for a value.
- dname *X.500-distinguished-name***  
Specifies the X.500 distinguished name to appear on the certificate generated by **-selfcert** or **-genkey**. A distinguished name is a highly qualified name intended to be globally unique. For example:  
  
CN=David Flanagan, OU=Editorial, O=OReilly, L=Cambridge, S=Massachusetts, C=US  
  
The **-genkey** command of *keytool* prompts for a distinguished name if none is specified. The **-selfcert** command uses the distinguished name of the current certificate if no replacement name is specified.
- file *file***  
Specifies the input or output file for many of the *keytool* commands. If left unspecified, *keytool* reads from the standard input or writes to the standard output.
- keyalg *algorithm-name***  
Used with **-genkey** to specify what type of cryptographic keys to generate. In the default Java implementation shipped from Sun, the only supported algorithm is “DSA”; this is the default if this option is omitted.
- keypass *password***  
Specifies the password that encrypts a private key in the keystore. If this option is unspecified, *keytool* first tries the **-storepass** password. If that does not work, it prompts for the appropriate password.
- keysize *size***  
Used with the **-genkey** command to specify the length in bits of the generated keys. If unspecified, the default is 1024.
- keystore *filename***  
Specifies the location of the keystore file. If unspecified, a file named *.keystore* in the user’s home directory is used.
- new *new-password-or-alias***  
Used with the **-keyclone** command to specify the new alias name and with **-keypasswd** and **-storepasswd** to specify the new password. If unspecified, *keytool* prompts for the value of this option.
- noprompt**  
Used with the **-import** command to disable interactive prompting of the user when a chain of trust cannot be established for an imported certificate. If this option is not specified, the **-import** command prompts the user.
- rfc**  
Used with the **-list** and **-export** commands to specify that certificate output should be in the printable encoding format specified by RFC-1421. If this option is not specified, **-export** outputs the certificate in binary format, and **-list** lists only the





## *keytool*

certificate fingerprint. This option cannot be combined with `-v` in the `-list` command.

### `-sigalg algorithm-name`

Specifies a digital signature algorithm that signs a certificate. If omitted, the default for this option depends on the type of underlying public key. If it is a DSA key, the default algorithm is "SHA1withDSA". If the key is an RSA key, the default signature algorithm is "MD5withRSA".

### `-storepass password`

Specifies a password that protects the integrity of the entire keystore file. This password also serves as a default password for any private keys that do not have their own `-keypass` specified. If `-storepass` is not specified, *keytool* prompts for it. The password must be at least six characters long.

### `-storetype type`

Specifies the type of the keystore to be used. If this option is not specified, the default is taken from the system security properties file. Often, the default is Sun's "JKS" Java Keystore type.

### `-trustcacerts`

Used with the `-import` command to specify that the self-signed certificate authority certificates contained in the keystore in the `jre/lib/security/cacerts` file should be considered trusted. If this option is omitted, *keytool* ignores that file.

### `-v`

Specifies verbose mode, if present, and makes many *keytool* commands produce additional output.

### `-validity time`

Used with the `-genkey` and `-selfcert` commands to specify the period of validity (in days) of the generated certificate. If unspecified, the default is 90 days.

### *See also*

*jarsigner*, *javakey*, *policytool*

## **native2ascii**

**JDK 1.1 and later**

### **Converts Java Source Code to ASCII**

#### *Synopsis*

```
native2ascii [ options ] [ inputfile [ outputfile ] ]
```

#### *Description*

*javac* can only process files encoded in the eight-bit Latin-1 encoding, with any other characters encoded using the `\uxxxx` Unicode notation. *native2ascii* is a simple program that reads a Java source file encoded using a local encoding and converts it to the Latin-1-plus-ASCII-encoded-Unicode form required by *javac*.

The *inputfile* and *outputfile* are optional. If unspecified, standard input and standard output are used, making *native2ascii* suitable for use in pipes.

#### *Options*

##### `-encoding encoding-name`

Specifies the encoding used by source files. If this option is not specified, the encoding is taken from the `file.encoding` system property.



-reverse

Specifies that the conversion should be done in reverse—from encoded \uxxxx characters to characters in the native encoding.

*See also*

java.io.InputStreamReader, java.io.OutputStreamWriter

## policytool

Java 2 SDK 1.2 and later

### Policy File Creation and Management Tool

#### Synopsis

policytool

#### Description

*policytool* displays a Swing user interface that makes it easy to edit security policy configuration files. The Java security architecture is based on policy files, which specify sets of permissions to be granted to code from various sources. By default, the Java security policy is defined by a system policy file stored in the *jre/lib/security/java.policy* file and a user policy file stored in the *.java.policy* file in the user's home directory. System administrators and users can edit these files with a text editor, but the syntax of the file is somewhat complex, so it is usually easier to use *policytool* to define and edit security policies.

#### Selecting the policy file to edit

When *policytool* starts up, it opens the *.java.policy* file in the user's home directory by default. Use the New, Open, and Save commands in the File menu to create a new policy file, open an existing file, and save an edited file, respectively.

#### Editing the policy file

The main *policytool* window displays a list of the entries contained in the policy file. Each entry specifies a code source and the permissions that are to be granted to code from that source. The window also contains buttons that allow you to add a new entry, edit an existing entry, or delete an entry from the policy file. If you add or edit an entry, *policytool* opens a new window that displays the details of that policy entry.

With the addition of the JAAS API to the core Java platform in Java 1.4, *policytool* has been updated to allow the specification of a **Principal** to whom a set of permissions will be granted.

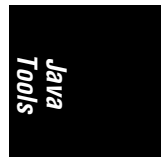
Every policy file has an associated keystore, from which it obtains the certificates it needs when verifying the digital signatures of Java code. You can usually rely on the default keystore, but if you need to specify the keystore explicitly for a policy file, use the Change Keystore command in the Edit menu of the main *policytool* window.

#### Adding or editing a policy entry

The policy entry editor window displays the code source for the policy entry and a list of permissions associated with that code source. It also contains buttons that allow you to add a new permission, delete a permission, or edit an existing permission.

When defining a new policy entry, the first step is to specify the code source. A code source is defined by a URL from which the code is downloaded and/or a list of digital signatures that must appear on the code. Specify one or both of these values by typing in a URL and/or a comma-separated list of aliases. These aliases identify trusted certificates in the keystore associated with the policy file.

After you have defined the code source for a policy entry, you must define the permissions to be granted to code from that source. Use the Add Permission and Edit Permission buttons to add and edit permissions. These buttons bring up yet another *policytool* window.





## *policytool*

### *Defining a permission*

To define a permission in the permission editor window, first select the desired permission type from the Permission drop-down menu. Then, select an appropriate target value from the Target Name menu. The choices in this menu are customized depending on the permission type you selected. For some types of permission, such as `FilePermission`, there is not a fixed set of possible targets, and you usually have to type in the specific target you want. For example, you might type `"/tmp"` to specify the directory `/tmp`, `"/tmp/*"` to specify all the files in that directory, or `"/tmp/*"` to specify all the files in the directory, and, recursively, any subdirectories. See the documentation of the individual `Permission` classes for a description of the targets they support.

Depending on the type of permission you select, you may also have to select one or more action values from the Actions menu. When you have selected a permission and appropriate target and action values, click the Okay button to dismiss the window.

### *See also*

*jarsigner*, *keytool*

## **serialver**

**JDK 1.1 and later**

### **Class Version Number Generator**

#### *Synopsis*

```
serialver [-show] classname...
```

#### *Description*

*serialver* displays the version number of a class or classes. This version number is used for the purposes of serialization: the version number must change each time the serialization format of the class changes.

If the specified class declares a long `serialVersionUID` constant, the value of that field is displayed. Otherwise, a unique version number is computed by applying the Secure Hash Algorithm (SHA) to the API defined by the class. This program is primarily useful for computing an initial unique version number for a class, which is then declared as a constant in the class. The output of *serialver* is a line of legal Java code, suitable for pasting into a class definition.

#### *Options*

##### **-show**

When the **-show** option is specified, *serialver* displays a simple graphical interface that allows the user to type in a single classname at a time and obtain its serialization UID. When using **-show**, no class names can be specified on the command line.

#### *Environment*

##### **CLASSPATH**

*serialver* is written in Java, so it is sensitive to the **CLASSPATH** environment variable in the same way the *java* interpreter is. The specified classes are looked up relative to this classpath.

### *See also*

`java.io.ObjectStreamClass`