NAME

javac − Java compiler

SYNOPSIS

javac [**−bootclasspath** *bootclasspath*] [**−classpath** *classpath*] [**−d** *directory*]
[**−deprecation**] [**−encoding** *encoding*]
[**−extdirs** *directories*]
[**−g** | **−g:none** | **−g:***keyword-list*] [**−J***option*] [**−nowarn**] [**−O**]
[**−sourcepath** *sourcepath*] [**−target** *version*] [**−verbose**] [**−X**]
[**−Xstdout** *filename*] [*sourcefiles*] [*@files*]

PARAMETERS

Options may be in any order. For a discussion of parameters which apply to a specific option, see below.

sourcefiles	One or more source files to be compiled (such as **MyClass.java**).
@files	One or more files that list source files.

DESCRIPTION

The **javac** tool reads class and interface definitions, written in the Java programming language, and compiles them into bytecode class files.

There are two ways to pass source code file names to **javac**:

- For a small number of source files, simply list the file names on the command line.
- For a large number of source files, list the the file names in a file, separated by blanks or line breaks. Then use the list file name on the **javac** command line, preceded by an @ character.

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

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

You should arrange source files in a directory tree that reflects their package tree. For example, if you keep all your source files in **/workspace**, the source code for **com.mysoft.mypack.MyClass** should be in **/workspace/com/mysoft/mypack/MyClass.java**.

By default, the compiler puts each class file in the same directory as its source file. You can specify a separate destination directory with **−d** (see **OPTIONS**, below).

Searching for Types

When compiling a source file, the compiler often needs information about a type it does not yet recognize. The compiler needs type information for every class or interface used, extended, or implemented in the source file. This includes classes and interfaces not explicitly mentioned in the source file but which provide information through inheritance.

For example, when you subclass **java.applet.Applet**, you are also using Applet's ancestor classes: **java.awt.Panel**, **java.awt.Container**, **java.awt.Component**, and **java.awt.Object**.

When the compiler needs type information, it looks for a source file or class file which defines the type. The compiler searches first in the bootstrap and extension classes, then in the user class path. The user class path is defined by setting the **CLASSPATH** environment variable or by using the **−classpath** command line option. (For details, see **Setting the Class Path**.) If you use the **−sourcepath** option, the compiler searches the indicated path for source files; otherwise the compiler searches the user class path both for class files and source files. You can specify different bootstrap or extension classes with the **−bootclasspath** and **−extdirs** options; see **Cross-Compilation Options** below.

A successful type search may produce a class file, a source file, or both. Here is how **javac** handles each situation:

- Search produces a class file but no source file: **javac** uses the class file.

- Search produces a source file but no class file: **javac** compiles the source file and uses the resulting class file.
- Search produces both a source file and a class file: **javac** determines whether the class file is out of date. If the class file is out of date, **javac** recompiles the source file and uses the updated class file. Otherwise, **javac** just uses the class file.

  By default, **javac** considers a class file out of date only if it is older than the source file.

Note that **javac** can silently compile source files not mentioned on the command line. Use the **−verbose** option to trace automatic compilation.

OPTIONS

The compiler has a set of standard options that are supported on the current development environment and will be supported in future releases. An additional set of non-standard options are specific to the current virtual machine implementation and are subject to change in the future. Non-standard options begin with **−X**.

Standard Options

−classpath *classpath*

Sets the user class path, overriding the user class path in the **CLASSPATH** environment variable. If neither **CLASSPATH** or **−classpath** is specified, the user class path consists of the current directory. See **Setting the Class Path** for more details.

If the **−sourcepath** option is not specified, the user class path is searched for source files as well as class files.

−d *directory*

Sets the destination directory for class files. The destination directory must already exist; javac will not create the destination directory. If a class is part of a package, **javac** puts the class file in a subdirectory reflecting the package name, creating directories as needed. For example, if you specify **−d /home/myclasses** and the class is called **com.mypackage.MyClass**, then the class file is called **/home/myclasses/com/mypackage/MyClass.class**.

If **−d** is not specified, **javac** puts the class file in the same directory as the source file.

Note that the directory specified by **−d** is not automatically added to your user class path.

−deprecation

Shows a description of each use or override of a deprecated member or class. Without **−deprecation**, **javac** shows the names of source files that use or override deprecated members or classes.

−encoding *encoding*

Sets the source file encoding name, such as EUCJIS/SJIS. If **−encoding** is not specified, the platform default converter is used.

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

−g:none

Does not generate any debugging information.

−g:*keyword-list*

Generates only some kinds of debugging information, specified by a comma separated list of keywords. Valid keywords are:

source	Source file debugging information
lines	Line number debugging information
vars	Local variable debugging information

−help Prints a synopsis of standard options.

−nowarn
Disables warning messages.

−source *release*
Enables support for compiling source code containing assertions.

When release is set to 1.4, the compiler accepts code containing assertions. Assertions were introduced in J2SE 1.4.

When release is set to 1.3, the compiler does *not* support assertions. The compiler defaults to the 1.3-behavior if the **−source** flag is not used.

−sourcepath *sourcepath*
Specifies the source code path to search for class or interface definitions. As with the user class path, source path entries are separated by colons (**:**) and can be directories, JAR archives, or ZIP archives. If packages are used, the local path name within the directory or archive must reflect the package name.

Note that classes found through the classpath are subject to automatic recompilation if their sources are found.

−verbose
Verbose output. This includes information about each class loaded and each source file compiled.

Cross-Compilation Options

By default, classes are compiled against the bootstrap and extension classes of the JDK that **javac** shipped with. But **javac** also supports cross-compiling, where classes are compiled against a bootstrap and extension classes of a different Java platform implementation. It is important to use **−bootclasspath** and **−extdirs** when cross-compiling; see **Cross-Compilation Example** below.

−bootclasspath *bootclasspath*
Cross-compiles against the specified set of boot classes. As with the user class path, boot class path entries are separated by colons (**:**) and can be directories, JAR archives, or ZIP archives.

−extdirs *directories*
Cross-compiles against the specified extension directories. Directories are a colon-separated list of directories. Each JAR archive in the specified directories is searched for class files.

−target *version*
Generates class files that will work on VMs with the specified version. The default is to generate class files to be compatible with 1.2 VMs, with one exception. When the **−source** 1.4 option is used, the default target is 1.4. The versions supported are:

1.1	Ensures that generated class files will be compatible with 1.1 and 1.2 VMs.
1.2	Generates class files that will run on 1.2 VMs, but will not run on 1.1 VMs. This is the default.
1.3	Generates class files that run on VMs in the Java 2 SDK, v1.3 and later, but will not run on 1.1 or 1.2 VMs.
1.4	Generates class files that are compatible only with 1.4 VMs.

Non-Standard Options

−J*option*
Passes option to the java launcher called by **javac**. For example, **−J−Xms48m** sets the startup memory to 48 megabytes. Although it does not begin with **−X**, it is not a ‘standard option’ of **javac**. It is a common convention for **−J** to pass options to the underlying VM executing applications written in Java.

Note that **CLASSPATH**, **−classpath**, **−bootclasspath**, and **−extdirs** do not specify the classes used to run **javac**. Fiddling with the implementation of the compiler in this way is usually pointless and always risky. If you do need to do this, use the **−J** option to pass through options to the underlying java launcher.

−X Displays information about non-standard options and exit.

−Xstdout filename
Send compiler messages to the named file. By default, compiler messages go to **System.err**.

−Xswitchcheck
Checks **switch** blocks for fall-through cases and provides a warning message for any that are found. Fall-through cases are cases in a **switch** block, other than the last case in the block, whose code does not include a **break** statement, allowing code execution to "fall through" from that case to the next case. For example, the code following the **case** 1 label in this **switch** block does not contain a **break** statement:

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

If the **−Xswtichcheck** flag were used when compiling this code, the compiler would emit a warning about "possible fall-through into case," along with the line number of the case in question.

COMMAND LINE ARGUMENT FILES

To shorten or simplify the javac command line, you can specify one or more files that themselves contain arguments to the **javac** command. This enables you to create javac commands of any length on any operating system.

An argument file can include javac options and source filenames in any combination. The arguments within a file can be space-separated or newline-separated. Filenames within an argument file are relative to the current directory, not the location of the argument file. Wildcards (*) are not allowed in these lists (such as for specifying ***.java**). Use of the **@** character to recursively interpret files is not supported.

When executing javac, pass in the path and name of each argument file with the **@** leading character. When javac encounters an argument beginning with the character **@**, it expands the contents of that file into the argument list.

Example - Single Arg File

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

```
C:> javac @argfile
```

This argument file could contain the contents of both files shown in the next example.

Example - Two Arg Files

You can create two argument files -- one for the javac options and the other for the source filenames: (Notice the following lists have no line-continuation characters.)

Create a file named **options** containing:

```
−d classes
−g
−sourcepath \java\pubs\ws\1.3\src\share\classes
```

Create a file named
classes containing:

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

You would then run javac with:

```
C:> javac @options @classes
```

Example - Arg Files with Paths

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

```
C:> javac @path1\options @path2\classes
```

EXAMPLES

Compiling a Simple Program

One source file, **Hello.java**, defines a class called **greetings.Hello**. The greetings directory is the package directory both for the source file and the class file and is off the current directory. This allows us to use the default user class path. It also makes it unnecessary to specify a separate destination directory with **−d**.

```
example% ls
  greetings/
example% ls greetings
  Hello.java
example% cat greetings/Hello.java
  package greetings;

  public class Hello {
      public static void main(String[] args) {
        for (int i=0; i < args.length; i++) {
            System.out.println("Hello " + args[i]);
        }
      }
  }
example% javac greetings/Hello.java
example% ls greetings
  Hello.class   Hello.java
example% java greetings.Hello World Universe Everyone
  Hello World
  Hello Universe
  Hello Everyone
```

Compiling Multiple Source Files

This example compiles all the source files in the package greetings.

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

Specifying a User Class Path

Having changed one of the source files in the previous example, we recompile it:

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

Since **greetings.Hi** refers to other classes in the greetings package, the compiler needs to find these other classes. The example above works, because our default user class path happens to be the directory containing the package directory. But suppose we want to recompile this file and not worry about which directory we're in? Then we need to add **/examples** to the user class path. We can do this by setting **CLASSPATH**, but here we'll use the **−classpath** option.

```
example% javac −classpath \examples /examples/greetings/Hi.java
```

If we change **greetings.Hi** again, to use a banner utility, that utility also needs to be accessible through the user class path.

```
example% javac −classpath /examples:/lib/Banners.jar \
            /examples/greetings/Hi.java
```

To execute a class in greetings, we need access both to greetings and to the classes it uses.

```
example% java −classpath /examples:/lib/Banners.jar greetings.Hi
```

Separating Source Files and Class Files

It often makes sense to keep source files and class files in separate directories, especially on large projects. We use **−d** to indicate the separate class file destination. Since the source files are not in the user class path, we use **−sourcepath** to help the compiler find them.

```
example% ls
  classes/  lib/     src/
example% ls src
  farewells/
example% ls src/farewells
  Base.java     GoodBye.java
example% ls lib
  Banners.jar
example% ls classes
example% javac −sourcepath src −classpath classes:lib/Banners.jar \
  src/farewells/GoodBye.java −d classes
example% ls classes
  farewells/
example% ls classes/farewells
  Base.class     GoodBye.class
```

Note that the compiler compiled src/farewells/Base.java, even though we didn't specify it on the command line. To trace automatic compiles, use the **−verbose** option.

Cross-Compilation Example

Here we use the JDK 1.2 **javac** to compile code that will run on a 1.1 VM.

```
example% javac −target 1.1 −bootclasspath jdk1.1.7/lib/classes.zip \
    −extdirs "" OldCode.java
```

The **−target 1.1** option ensures that the generated class files will be compatible with 1.1 VMs. In JDK1.2, **javac** compiles for 1.1 by default, so this option is not strictly required. However, it is good form because other compilers may have other defaults.

The JDK 1.2 **javac** would also by default compile against its own 1.2 bootstrap classes, so we need to tell **javac** to compile against JDK 1.1 bootstrap classes instead. We do this with **−bootclasspath** and **−extdirs**. Failing to do this might allow compilation against a 1.2 API that would not be present on a 1.1 VM and fail at runtime.

SEE ALSO

jar(1), **java**(1), **javadoc**(1), **javah**(1), **javap**(1), **jdb**(1)

See or search the Java web site for the following:

The Java Extensions Mechanism @
http://java.sun.com/j2se/1.4/docs/guide/extensions/index.html