NAME

javac – Reads Java class and interface definitions and compiles them into bytecode and class files.

SYNOPSIS

javac [*options*] [*sourcefiles*] [*classes*] [*@argfiles*]

Arguments can be in any order:

options Command-line options. See Options.

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

classes One or more classes to be processed for annotations (such as **MyPackage.MyClass**).

@argfiles
One or more files that list options and source files. The **-J** options are not allowed in these files. See Command-Line Argument Files.

DESCRIPTION

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

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

- For a small number of source files, list the file names on the command line.
- For a large number of source files, list the file names in a file that is separated by blanks or line breaks. Use the list file name preceded by an at sign (@) with the **javac** command.

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

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

Arrange source files in a directory tree that reflects their package tree. For example, if all of your source files are in **/workspace**, then put the source code for **com.mysoft.mypack.MyClass** in **/workspace/com/mysoft/mypack/MyClass.java**.

By default, the compiler puts each class file in the same directory as its source file. You can specify a separate destination directory with the **-d** option.

OPTIONS

The compiler has a set of standard options that are supported on the current development environment. An additional set of nonstandard options are specific to the current virtual machine and compiler implementations and are subject to change in the future. Nonstandard options begin with the **-X** option.

- See also Cross-Compilation Options
- See also Nonstandard Options

STANDARD OPTIONS

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

-cp *path* or -classpath *path*
Specifies where to find user class files, and (optionally) annotation processors and source files. This class path overrides the user class path in the **CLASSPATH** environment variable. If neither **CLASSPATH**, **-cp** nor **-classpath** is specified, then the user *class path* is the current directory. See Setting the Class Path.

If the **-sourcepath** option is not specified, then the user class path is also searched for source files.

If the **-processorpath** option is not specified, then the class path is also searched for annotation processors.

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

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

-d *directory*
Sets the destination directory for class files. The directory must already exist because **javac** does not create it. If a class is part of a package, then **javac** puts the class file in a subdirectory that reflects the package name and creates directories as needed.

If you specify **-d/home/myclasses** and the class is called **com.mypackage.MyClass**, then the class file is **/home/myclasses/com/mypackage/MyClass.class**.

If the *-d* option is not specified, then **javac** puts each class file in the same directory as the source file from which it was generated.

Note: The directory specified by the *-d* option 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 the **-deprecation** option, **javac** shows a summary of the source files that use or override deprecated members or classes. The **-deprecation** option is shorthand for **-Xlint:deprecation**.

-encoding *encoding*
Sets the source file encoding name, such as EUC-JP and UTF-8. If the **-encoding** option is not specified, then the platform default converter is used.

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

-extdirs *directories*
Overrides the location of the **ext** directory. The directories variable is a colon-separated list of directories. Each JAR file in the specified directories is searched for class files. All JAR files found become part of the class path.

If you are cross-compiling (compiling classes against bootstrap and extension classes of a different Java platform implementation), then this option specifies the directories that contain the extension classes. See Cross-Compilation Options for more information.

-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.

-implicit:[*class, none*]
: Controls the generation of class files for implicitly loaded source files. To automatically generate class files, use **-implicit:class**. To suppress class file generation, use **-implicit:none**. If this option is not specified, then the default is to automatically generate class files. In this case, the compiler issues a warning if any such class files are generated when also doing annotation processing. The warning is not issued when the **-implicit** option is set explicitly. See Searching for Types.

-J*option*
: Passes **option** to the Java Virtual Machine (JVM), where option is one of the options described on the reference page for the Java launcher. For example, **-J-Xms48m** sets the startup memory to 48 MB. See java(1).

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

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

-parameters
: Stores formal parameter names of constructors and methods in the generated class file so that the method **java.lang.reflect.Executable.getParameters** from the Reflection API can retrieve them.

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

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

-processorpath *path*
: Specifies where to find annotation processors. If this option is not used, then the class path is searched for processors.

-s *dir*
: Specifies the directory where to place the generated source files. The directory must already exist because **javac** does not create it. If a class is part of a package, then the compiler puts the source file in a subdirectory that reflects the package name and creates directories as needed.

  If you specify **-s /home/mysrc** and the class is called **com.mypackage.MyClass**, then the source file is put in **/home/mysrc/com/mypackage/MyClass.java**.

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

  1.3 The compiler does not support assertions, generics, or other language features introduced after Java SE 1.3.

  1.4 The compiler accepts code containing assertions, which were introduced in Java SE 1.4.

  1.5 The compiler accepts code containing generics and other language features introduced in Java SE 5.

  5 Synonym for 1.5.

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

Standard Edition.

6 Synonym for 1.6.

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

7 Synonym for 1.7.

1.8 This is the default value. The compiler accepts code with features introduced in Java SE 8.

8 Synonym for 1.8.

-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 (:) on Oracle Solaris and semicolons on Windows and can be directories, JAR archives, or ZIP archives. If packages are used, then the local path name within the directory or archive must reflect the package name.

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

-verbose

Uses verbose output, which includes information about each class loaded and each source file compiled.

-version

Prints release information.

-werror

Terminates compilation when warnings occur.

-X

Displays information about nonstandard options and exits.

CROSS-COMPILATION OPTIONS

By default, classes are compiled against the bootstrap and extension classes of the platform 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 the **-bootclasspath** and **-extdirs** options when cross-compiling.

-target *version*

Generates class files that target a specified release of the virtual machine. Class files will run on the specified target and on later releases, but not on earlier releases of the JVM. Valid targets are 1.1, 1.2, 1.3, 1.4, 1.5 (also 5), 1.6 (also 6), 1.7 (also 7), and 1.8 (also 8).

The default for the **-target** option depends on the value of the **-source** option:

- If the **-source** option is not specified, then the value of the **-target** option is 1.8
- If the **-source** option is 1.2, then the value of the **-target** option is 1.4
- If the **-source** option is 1.3, then the value of the **-target** option is 1.4
- If the **-source** option is 1.5, then the value of the **-target** option is 1.8
- If the **-source** option is 1.6, then the value of the **-target** is option 1.8
- If the **-source** option is 1.7, then the value of the **-target** is option 1.8
- For all other values of the **-source** option, the value of the **-target** option is the value of the **-source** option.

-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.

COMPACT PROFILE OPTION

Beginning with JDK 8, the **javac** compiler supports compact profiles. With compact profiles, applications that do not require the entire Java platform can be deployed and run with a smaller footprint. The compact profiles feature could be used to shorten the download time for applications from app stores. This feature makes for more compact deployment of Java applications that bundle the JRE. This feature is also useful in small devices.

The supported profile values are **compact1**, **compact2**, and **compact3**. These are additive layers. Each higher-numbered compact profile contains all of the APIs in profiles with smaller number names.

-profile
When using compact profiles, this option specifies the profile name when compiling. For example:

javac –profile compact1 Hello.java

javac does not compile source code that uses any Java SE APIs that is not in the specified profile. Here is an example of the error message that results from attempting to compile such source code:

cd jdk1.8.0/bin
./javac –profile compact1 Paint.java
Paint.java:5: error: Applet is not available in profile 'compact1'
import java.applet.Applet;

In this example, you can correct the error by modifying the source to not use the **Applet** class. You could also correct the error by compiling without the -profile option. Then the compilation would be run against the full set of Java SE APIs. (None of the compact profiles include the **Applet** class.)

An alternative way to compile with compact profiles is to use the **-bootclasspath** option to specify a path to an **rt.jar** file that specifies a profile's image. Using the **-profile** option instead does not require a profile image to be present on the system at compile time. This is useful when cross-compiling.

NONSTANDARD OPTIONS

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

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

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

-Xdoclint:[-]*group* [*/access*]
Enables or disables specific groups of checks, where *group* is one of the following values: **accessibility**, **syntax**, **reference**, **html** or **missing**. For more information about these groups of checks see the **-Xdoclint** option of the **javadoc** command. The **-Xdoclint** option is disabled by default in the **javac** command.

The variable *access* specifies the minimum visibility level of classes and members that the **-Xdoclint** option checks. It can have one of the following values (in order of most to least visible)

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

–Xdoclint:all/protected

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

–Xdoclint:all,–html/package

-Xdoclint:none
Disables all groups of checks.

-Xdoclint:all[*/access*]
Enables all groups of checks.

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

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

-Xlint:none
Disables all warnings.

-Xlint:*name*
Disables warning name. See Enable or Disable Warnings with the -Xlint Option for a list of warnings you can disable with this option.

-Xlint:*-name*
Disables warning name. See Enable or Disable Warnings with the -Xlint Option with the **-Xlint** option to get a list of warnings that you can disable with this option.

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

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

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

-Xprefer:[*newer,source*]
Specifies which file to read when both a source file and class file are found for a type. (See Searching for Types). If the **-Xprefer:newer** option is used, then it reads the newer of the source or class file for a type (default). If the **-Xprefer:source** option is used, then it reads the source file. Use -**Xprefer:source** when you want to be sure that any annotation processors can access annotations declared with a retention policy of **SOURCE**.

-Xpkginfo:[*always*,*legacy*,*nonempty*]
Control whether javac generates **package-info.class** files from package-info.java files. Possible mode arguments for this option include the following.

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

legacy Generate a **package-info.class** file only if package-info.java contains annotations. Don't generate a **package-info.class** file if package-info.java only contains comments.

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

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

-Xprint
Prints a textual representation of specified types for debugging purposes. Perform neither annotation processing nor compilation. The format of the output could change.

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

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

ENABLE OR DISABLE WARNINGS WITH THE -XLINT OPTION

Enable warning *name* with the **-Xlint:name** option, where **name** is one of the following warning names. Note that you can disable a warning with the **-Xlint:-name:** option.

cast Warns about unnecessary and redundant casts, for example:

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

classfile
Warns about issues related to class file contents.

deprecation
Warns about the use of deprecated items, for example:

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

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

dep-ann
Warns about items that are documented with an **@deprecated** Javadoc comment, but do not have a **@Deprecated** annotation, for example:

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

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

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

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

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

fallthrough

Checks the switch blocks for fall-through cases and provides a warning message for any that are found. Fall-through cases are cases in a switch block, other than the last case in the block, whose code 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 end with a break statement:

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

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

finally Warns about **finally** clauses that cannot complete normally, for example:

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

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

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

overrides
Warns about issues regarding method overrides. For example, consider the following two classes:

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

The compiler generates a warning similar to the following:.

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

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

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

```
javac –Xlint:path –classpath /nonexistentpath Example.java
```

processing
Warn about issues regarding annotation processing. The compiler generates this warning when you have a class that has an annotation, and you use an annotation processor that cannot handle that type of exception. For example, the following is a simple annotation processor:

Source file AnnocProc.java:

```
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import.javaz.lang.model.element.*;
@SupportedAnnotationTypes("NotAnno")
public class AnnoProc extends AbstractProcessor {
 public boolean process(Set<? extends TypeElement> elems, RoundEnvironment renv){
   return true;
 }
 public SourceVersion getSupportedSourceVersion() {
   return SourceVersion.latest();
```

```
  }
}
```

Source file AnnosWithoutProcessors.java:

```
@interface Anno { }
@Anno
class AnnosWithoutProcessors { }
```

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

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

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

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

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

rawtypes

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

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

The following example does not generate a **rawtypes** warning

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

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

Serial Warns about missing **serialVersionUID** definitions on serializable classes, for example:

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

```
    time = Calendar.getInstance().getTime();
  }
  public Date getTime() {
    return time;
  }
}
```

The compiler generates the following warning:

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

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

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

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

The compiler generates the following warning:

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

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

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

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

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

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

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

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

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

The **ls** command has the parameterized type **List<String>**. When the **List** referenced by **l** is assigned to **ls**, the compiler generates an unchecked warning. At compile time, the compiler and JVM cannot determine whether **l** refers to a **List<String>** type. In this case, **l** does not refer to a **List<String>** type. As a result, heap pollution occurs.

A heap pollution situation occurs when the **List** object **l**, whose static type is **List<Number>**, is assigned to another **List** object, **ls**, that has a different static type, **List<String>**. However, the compiler still allows this assignment. It must allow this assignment to preserve backward compatibility with releases of Java SE that do not support generics. Because of type erasure, **List<Number>** and **List<String>** both become **List**. Consequently, the compiler allows the assignment of the object **l,** which has a raw type of **List**, to the object **ls**.

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

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

Note: A non-reifiable type is a type whose type information is not fully available at runtime.

The compiler generates the following warning for the definition of the method **ArrayBuilder.addToList**

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

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

COMMAND-LINE ARGUMENT FILES

To shorten or simplify the **javac** command, you can specify one or more files that contain arguments to the **javac** command (except **-J** options). This enables you to create **javac** commands of any length on any

operating system.

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

File Names within an argument file are relative to the current directory, not the location of the argument file. Wild cards (*) are not allowed in these lists (such as for specifying ***.java**). Use of the at sign (@) to recursively interpret files is not supported. The **-J** options are not supported because they are passed to the launcher, which does not support argument files.

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

Example 1 Single Argument File

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

javac @argfile

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

Example 2 Two Argument Files

You can create two argument files: one for the **javac** options and the other for the source file names. Note that the following lists have no line-continuation characters.

Create a file named options that contains the following:

–d classes
–g
–sourcepath /java/pubs/ws/1.3/src/share/classes

Create a file named classes that contains the following:

MyClass1.java
MyClass2.java
MyClass3.java

Then, run the **javac** command as follows:

javac @options @classes

Example 3 Argument Files with Paths

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

javac @path1/options @path2/classes

ANNOTATION PROCESSING

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

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

HOW ANNOTATION PROCESSING WORKS

Unless annotation processing is disabled with the **-proc:none** option, the compiler searches for any annotation processors that are available. The search path can be specified with the **-processorpath** option.

If no path is specified, then the user class path is used. Processors are located by means of service provider-configuration files named **META-INF/services/javax.annotation.processing**.Processor on the search path. Such files should contain the names of any annotation processors to be used, listed one per line. Alternatively, processors can be specified explicitly, using the **-processor** option.

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

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

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

IMPLICITLY LOADED SOURCE FILES

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

SEARCHING FOR TYPES

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

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

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

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

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

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

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

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

PROGRAMMATIC INTERFACE

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

EXAMPLE

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

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

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

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

OLD INTERFACE

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

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

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

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

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

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

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

EXAMPLES

Example 1 Compile a Simple Program

This example shows how to compile the **Hello.java** source file in the greetings directory. The class defined in **Hello.java** is called **greetings.Hello**. The greetings directory is the package directory both for the source file and the class file and is underneath the current directory. This makes it possible to use the default user class path. It also makes it unnecessary to specify a separate destination directory with the **-d** option.

The source code in **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]);
      }
   }
}
```

Compile greetings.Hello:

```
javac greetings/Hello.java
```

Run **greetings.Hello**:

```
java greetings.Hello World Universe Everyone
```

Hello World
Hello Universe
Hello Everyone

Example 2 Compile Multiple Source Files

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

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

Example 3 Specify a User Class Path

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

pwd
/examples
javac greetings/Hi.java

Because **greetings.Hi** refers to other classes in the **greetings** package, the compiler needs to find these other classes. The previous example works because the default user class path is the directory that contains the package directory. If you want to recompile this file without concern for which directory you are in, then add the examples directory to the user class path by setting **CLASSPATH**. This example uses the **-classpath** option.

javac –classpath /examples /examples/greetings/Hi.java

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

**javac –classpath /examples:/lib/Banners.jar **
/examples/greetings/Hi.java

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

java –classpath /examples:/lib/Banners.jar greetings.Hi

Example 4 Separate Source Files and Class Files

The following example uses **javac** to compile code that runs on JVM 1.7.

**javac –source 1.7 –target 1.7 –bootclasspath jdk1.7.0/lib/rt.jar **
–extdirs "" OldCode.java

The **-source 1.7** option specifies that release 1.7 (or 7) of the Java programming language be used to compile **OldCode.java**. The option **-target 1.7** option ensures that the generated class files are compatible with JVM 1.7. Note that in most cases, the value of the **-target** option is the value of the **-source** option; in this example, you can omit the **-target** option.

You must specify the **-bootclasspath** option to specify the correct version of the bootstrap classes (the **rt.jar** library). If not, then the compiler generates a warning:

javac –source 1.7 OldCode.java

warning: [options] bootstrap class path not set in conjunction with
−source 1.7

If you do not specify the correct version of bootstrap classes, then the compiler uses the old language rules (in this example, it uses version 1.7 of the Java programming language) combined with the new bootstrap classes, which can result in class files that do not work on the older platform (in this case, Java SE 7) because reference to nonexistent methods can get included.

Example 5 Cross Compile

This example uses **javac** to compile code that runs on JVM 1.7.

**javac −source 1.7 −target 1.7 −bootclasspath jdk1.7.0/lib/rt.jar **
−extdirs "" OldCode.java

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

You must specify the **-bootclasspath** option to specify the correct version of the bootstrap classes (the **rt.jar** library). If not, then the compiler generates a warning:

javac −source 1.7 OldCode.java
warning: [options] bootstrap class path not set in conjunction with −source 1.7

If you do not specify the correct version of bootstrap classes, then the compiler uses the old language rules combined with the new bootstrap classes. This combination can result in class files that do not work on the older platform (in this case, Java SE 7) because reference to nonexistent methods can get included. In this example, the compiler uses release 1.7 of the Java programming language.

SEE ALSO

- java(1)
- jdb(1)
- javadoc(1)
- jar(1)
- jdb(1)