NAME

javac - read Java declarations and compile them into class files

SYNOPSIS

`javac` [*options*] [*sourcefiles-or-classnames*]

options Command-line options.

sourcefiles-or-classnames

Source files to be compiled (for example, `Shape.java`) or the names of previously compiled classes to be processed for annotations (for example, `geometry.MyShape`).

DESCRIPTION

The `javac` command reads *source files* that contain module, package and type declarations written in the Java programming language, and compiles them into *class files* that run on the Java Virtual Machine.

The `javac` command can also **process annotations** in Java source files and classes.

Source files must have a file name extension of `.java`. Class files have a file name extension of `.class`. Both source and class files normally have file names that identify the contents. For example, a class called `Shape` would be declared in a source file called `Shape.java`, and compiled into a class file called `Shape.class`.

There are two ways to specify source files to `javac`:

- For a small number of source files, you can list their file names on the command line.
- For a large number of source files, you can use the **`@`*filename*** option on the command line to specify an *argument file* that lists their file names. See **Standard Options** for a description of the option and **Command-Line Argument Files** for a description of `javac` argument files.

The order of source files specified on the command line or in an argument file is not important. `javac` will compile the files together, as a group, and will automatically resolve any dependencies between the declarations in the various source files.

`javac` expects that source files are arranged in one or more directory hierarchies on the file system, described in **Arrangement of Source Code**.

To compile a source file, `javac` needs to find the declaration of every class or interface that is used, extended, or implemented by the code in the source file. This lets `javac` check that the code has the right to access those classes and interfaces. Rather than specifying the source files of those classes and interfaces explicitly, you can use command-line options to tell `javac` where to search for their source files. If you have compiled those source files previously, you can use options to tell `javac` where to search for the corresponding class files. The options, which all have names ending in "path", are described in **Standard Options**, and further described in **Configuring a Compilation** and **Searching for Module, Package and Type Declarations**.

By default, `javac` compiles each source file to a class file in the same directory as the source file. However, it is recommended to specify a separate destination directory with the **`-d`** option.

Command-line **options** and **environment variables** also control how `javac` performs various tasks:

- Compiling code to run on earlier releases of the JDK.
- Compiling code to run under a debugger.
- Checking for stylistic issues in Java source code.
- Checking for problems in `javadoc` comments (`/** ... */`).
- Processing annotations in source files and class files.
- Upgrading and patching modules in the compile-time environment.

`javac` supports **Compiling for Earlier Releases Of The Platform** and can also be invoked from Java code using one of a number of **APIs**

OPTIONS

`javac` provides **standard options**, and **extra options** that are either non-standard or are for advanced use.

Some options take one or more arguments. If an argument contains spaces or other whitespace characters, the value should be quoted according to the conventions of the environment being used to invoke javac. If the option begins with a single dash (`-`) the argument should either directly follow the option name, or should be separated with a colon (`:`) or whitespace, depending on the option. If the option begins with a double dash (`--`), the argument may be separated either by whitespace or by an equals (=) character with no additional whitespace. For example,

```
-Aname="J. Duke"
-proc:only
-d myDirectory
--module-version 3
--module-version=3
```

In the following lists of options, an argument of *path* represents a search path, composed of a list of file system locations separated by the platform path separator character, (semicolon `;` on Windows, or colon `:` on other systems.) Depending on the option, the file system locations may be directories, JAR files or JMOD files.

Standard Options

`@`*filename*

Reads options and file names from a file. 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 lets you to create `javac` commands of any length on any operating system. See **Command-Line Argument Files**.

`-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 (`.`).

`--add-modules` *module*`,` *module*

Specifies root modules to resolve in addition to the initial modules, or all modules on the module path if *module* is `ALL-MODULE-PATH`.

`--boot-class-path` *path* or `-bootclasspath` *path*

Overrides the location of the bootstrap class files.

Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details. For JDK 9 or later, see **`--system`**.

`--class-path` *path*, `-classpath` *path*, or `-cp` *path*

Specifies where to find user class files and annotation processors. This class path overrides the user class path in the `CLASSPATH` environment variable.

- If `--class-path`, `-classpath`, or `-cp` are not specified, then the user class path is the value of the `CLASSPATH` environment variable, if that is set, or else the current directory.
- If not compiling code for modules, if the **`--source-path`** or -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.

`-d` *directory*

Sets the destination directory (or *class output directory*) for class files. If a class is part of a package, then `javac` puts the class file in a subdirectory that reflects the module name (if appropriate) and package name. The directory, and any necessary subdirectories, will be created if they do not already exist.

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.

Except when compiling code for multiple modules, the contents of the class output directory will be organized in a package hierarchy. When compiling code for multiple modules, the contents of the output directory will be organized in a module hierarchy, with the contents of each module in a separate subdirectory, each organized as a package hierarchy.

Note: When compiling code for one or more modules, the class output directory will automatically be checked when searching for previously compiled classes. When not compiling for modules, for backwards compatibility, the directory is *not* automatically checked for previously compiled classes, and so it is recommended to specify the class output directory as one of the locations on the user class path, using the `--class-path` option or one of its alternate forms.

`-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`.

`--enable-preview`
Enables preview language features. Used in conjunction with either **`-source`** or **`--release`**.

`-encoding` *encoding*
Specifies character encoding used by source files, 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.

Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

`-extdirs` *directories*
Overrides the location of the installed extensions. `directories` is a list of directories, separated by the platform path separator (`;` on Windows, and `:` otherwise). 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 compiling for a release of the platform that supports the Extension Mechanism, then this option specifies the directories that contain the extension classes. See [Compiling for Other Releases of the Platform].

Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

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

`-g:[lines, vars, source]`
Generates only the kinds of debugging information specified by the comma-separated list of keywords. Valid keywords are:

`lines` Line number debugging information.

`vars` Local variable debugging information.

`source`
Source file debugging information.

`-g:none`
Does not generate debugging information.

`-h` *directory*
Specifies where to place generated native header files.

When you specify this option, a native header file is generated for each class that contains native

methods or that has one or more constants annotated with the **java.lang.annotation.Native** annotation. If the class is part of a package, then the compiler puts the native header file in a subdirectory that reflects the module name (if appropriate) and package name. The directory, and any necessary subdirectories, will be created if they do not already exist.

`--help`, `-help` or `-?`
Prints a synopsis of the standard options.

`--help-extra` or `-X`
Prints a synopsis of the set of extra options.

`--help-lint`
Prints the supported keys for the `-Xlint` option.

`-implicit:`[`none`, `class`]
Specifies whether or not to generate class files for implicitly referenced files:

- `-implicit:class` --- Automatically generates class files.
- `-implicit:none` --- Suppresses class file generation.

If this option is not specified, then the default automatically generates class files. In this case, the compiler issues a warning if any class files are generated when also doing annotation processing. The warning is not issued when the `-implicit` option is explicitly set. See **Searching for Module, Package and Type Declarations**.

`-J`*option*
Passes *option* to the runtime system, where *option* is one of the Java options described on **java** command. For example, `-J-Xms48m` sets the startup memory to 48 MB.

Note: The `CLASSPATH` environment variable, `-classpath` option, `-bootclasspath` option, and `-extdirs` option 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 compiler implementation, then use the `-J` option to pass options through to the underlying Java launcher.

`--limit-modules` *module*`,`*module*`*`
Limits the universe of observable modules.

`--module` *module-name* (`,` *module-name*)* or `-m` *module-name* (`,` *module-name*)*
Compiles those source files in the named modules that are newer than the corresponding files in the output directory.

`--module-path` *path* or `-p` *path*
Specifies where to find application modules.

`--module-source-path` *module-source-path*
Specifies where to find source files when compiling code in multiple modules. See [Compilation Modes] and **The Module Source Path Option**.

`--module-version` *version*
Specifies the version of modules that are being compiled.

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

`-parameters`
Generates metadata for reflection on method 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`, `full`]
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. If this option is not used, or

`-proc:full` is specified, annotation processing and compilation are done.

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

`--processor-module-path` *path*
Specifies the module path used for finding annotation processors.

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

`-profile` *profile*
Checks that the API used is available in the specified profile. This option is deprecated and may be removed in a future release.

Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

`--release` *release*
Compiles source code according to the rules of the Java programming language for the specified Java SE release, generating class files which target that release. Source code is compiled against the combined Java SE and JDK API for the specified release.

The supported values of *release* are the current Java SE release and a limited number of previous releases, detailed in the command-line help.

For the current release, the Java SE API consists of the `java.*`, `javax.*`, and `org.*` packages that are exported by the Java SE modules in the release; the JDK API consists of the `com.*` and `jdk.*` packages that are exported by the JDK modules in the release, plus the `javax.*` packages that are exported by standard, but non-Java SE, modules in the release.

For previous releases, the Java SE API and the JDK API are as defined in that release.

Note: When using `--release`, you cannot also use the **`--source`**/`-source` or **`--target`**/`-target` options.

Note: When using `--release` to specify a release that supports the Java Platform Module System, the `--add-exports` option cannot be used to enlarge the set of packages exported by the Java SE, JDK, and standard modules in the specified release.

`-s` *directory*
Specifies the directory used to place the generated source files. If a class is part of a package, then the compiler puts the source file in a subdirectory that reflects the module name (if appropriate) and package name. The directory, and any necessary subdirectories, will be created if they do not already exist.

Except when compiling code for multiple modules, the contents of the source output directory will be organized in a package hierarchy. When compiling code for multiple modules, the contents of the source output directory will be organized in a module hierarchy, with the contents of each module in a separate subdirectory, each organized as a package hierarchy.

`--source` *release* or `-source` *release*
Compiles source code according to the rules of the Java programming language for the specified Java SE release. The supported values of *release* are the current Java SE release and a limited number of previous releases, detailed in the command-line help.

If the option is not specified, the default is to compile source code according to the rules of the Java programming language for the current Java SE release.

`--source-path` *path* or `-sourcepath` *path*
Specifies where to find source files. Except when compiling multiple modules together, this is the source code path used to search for class or interface definitions.

Note: Classes found through the class path might be recompiled when their source files are also

found. See **Searching for Module, Package and Type Declarations**.

`--system` *jdk* | `none`
: Overrides the location of system modules.

`--target` *release* or `-target` *release*
: Generates `class` files suitable for the specified Java SE release. The supported values of *release* are the current Java SE release and a limited number of previous releases, detailed in the command-line help.

  Note: The target release must be equal to or higher than the source release. (See **`--source`**.)

`--upgrade-module-path` *path*
: Overrides the location of upgradeable modules.

`-verbose`
: Outputs messages about what the compiler is doing. Messages include information about each class loaded and each source file compiled.

`--version` or `-version`
: Prints version information.

`-Werror`
: Terminates compilation when warnings occur.

Extra Options

`--add-exports` *module*`/`*package*`=`*other-module*(`,`*other-module*)*
: Specifies a package to be considered as exported from its defining module to additional modules or to all unnamed modules when the value of *other-module* is `ALL-UNNAMED`.

`--add-reads` *module*`=`*other-module*(`,`*other-module*)*
: Specifies additional modules to be considered as required by a given module.

`--default-module-for-created-files` *module-name*
: Specifies the fallback target module for files created by annotation processors, if none is specified or inferred.

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

  Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

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

  Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

`--patch-module` *module*`=`*path*
: Overrides or augments a module with classes and resources in JAR files or directories.

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

  Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

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

  Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

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

  Note: This can only be used when compiling for releases prior to JDK 9. As applicable, see the descriptions in **`--release`**, **`-source`**, or **`-target`** for details.

`-Xdiags:`[`compact`, `verbose`]
: Selects a diagnostic mode.

`-Xdoclint`
: Enables recommended checks for problems in documentation comments.

`-Xdoclint:`(`all`|`none`|[`-`]*group*)[`/`*access*]
: Enables or disables specific groups of checks in documentation comments.

  group can have one of the following values: `accessibility`, `html`, `missing`, `reference`, `syntax`.

  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`, `private`.

  The default *access* level is `private`.

  When prefixed by `doclint:`, the *group* names and `all` can be used with `@SuppressWarnings` to suppress warnings about documentation comments in parts of the code being compiled.

  For more information about these groups of checks, see the **DocLint** section of the `javadoc` command documentation. The `-Xdoclint` option is disabled by default in the `javac` command.

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

  ```
  -Xdoclint:all,-html/package
  ```

`-Xdoclint/package:`[`-`]*packages*(`,`[`-`]*package*)*
: Enables or disables checks in specific packages. Each *package* is either the qualified name of a package or a package name prefix followed by `.*`, which expands to all sub-packages of the given package. Each *package* can be prefixed with a hyphen (`-`) to disable checks for a specified package or packages.

  For more information, see the **DocLint** section of the `javadoc` command documentation.

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

`-Xlint:`[`-`]*key*(`,`[`-`]*key*)*
: Supplies warnings to enable or disable, separated by comma. Precede a key by a hyphen (`-`) to disable the specified warning.

  Supported values for *key* are:

  - `all`: Enables all warnings.
  - `auxiliaryclass`: Warns about an auxiliary class that is hidden in a source file, and is used from other files.
  - `cast`: Warns about the use of unnecessary casts.

- `classfile`: Warns about the issues related to classfile contents.
- `deprecation`: Warns about the use of deprecated items.
- `dep-ann`: Warns about the items marked as deprecated in `javadoc` but without the `@Deprecated` annotation.
- `divzero`: Warns about the division by the constant integer 0.
- `empty`: Warns about an empty statement after `if`.
- `exports`: Warns about the issues regarding module exports.
- `fallthrough`: Warns about the falling through from one case of a switch statement to the next.
- `finally`: Warns about `finally` clauses that do not terminate normally.
- `lossy-conversions`: Warns about possible lossy conversions in compound assignment.
- `missing-explicit-ctor`: Warns about missing explicit constructors in public and protected classes in exported packages.
- `module`: Warns about the module system-related issues.
- `opens`: Warns about the issues related to module opens.
- `options`: Warns about the issues relating to use of command line options.
- `output-file-clash`: Warns if any output file is overwritten during compilation. This can occur, for example, on case-insensitive filesystems.
- `overloads`: Warns about the issues related to method overloads.
- `overrides`: Warns about the issues related to method overrides.
- `path`: Warns about the invalid path elements on the command line.
- `preview`: Warns about the use of preview language features.
- `processing`: Warns about the issues related to annotation processing.
- `rawtypes`: Warns about the use of raw types.
- `removal`: Warns about the use of an API that has been marked for removal.
- `requires-automatic`: Warns developers about the use of automatic modules in requires clauses.
- `requires-transitive-automatic`: Warns about automatic modules in requires transitive.
- `serial`: Warns about the serializable classes that do not provide a serial version ID. Also warns about access to non-public members from a serializable element.
- `static`: Warns about the accessing a static member using an instance.
- `strictfp`: Warns about unnecessary use of the `strictfp` modifier.
- `synchronization`: Warns about synchronization attempts on instances of value-based classes.
- `text-blocks`: Warns about inconsistent white space characters in text block indentation.
- `this-escape`: Warns about constructors leaking `this` prior to subclass initialization.
- `try`: Warns about the issues relating to the use of try blocks (that is, try-with-resources).
- `unchecked`: Warns about the unchecked operations.
- `varargs`: Warns about the potentially unsafe `vararg` methods.
- `none`: Disables all warnings.

With the exception of `all` and `none`, the keys can be used with the `@SuppressWarnings` annotation to suppress warnings in a part of the source code being compiled.

See **Examples of Using -Xlint keys**.

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

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

`-Xpkginfo:`[`always`, `legacy`, `nonempty`]
Specifies when and how the `javac` command generates `package-info.class` files from `package-info.java` files using one of the following options:

`always`
Generates 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`
Generates a `package-info.class` file only if `package-info.java` contains annotations. This option does not generate a `package-info.class` file if `package-info.java` contains only 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`
Generates a `package-info.class` file only if `package-info.java` contains annotations with `RetentionPolicy.CLASS` or `RetentionPolicy.RUNTIME`.

`-Xplugin:`*name args*
Specifies the name and optional arguments for a plug-in to be run. If *args* are provided, *name* and *args* should be quoted or otherwise escape the whitespace characters between the name and all the arguments. For details on the API for a plugin, see the API documentation for **jdk.compiler/com.sun.source.util.Plugin**.

`-Xprefer:`[`source`, `newer`]
Specifies which file to read when both a source file and class file are found for an implicitly compiled class using one of the following options. See **Searching for Module, Package and Type Declarations**.

- `-Xprefer:newer`: Reads the newer of the source or class files for a type (default).
- `-Xprefer:source` : 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`.

`-Xprint`
Prints a textual representation of specified types for debugging purposes. This does not perform annotation processing or 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.

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

ENVIRONMENT VARIABLES

CLASSPATH

If the **`--class-path`** option or any of its alternate forms are not specified, the class path will default to the value of the `CLASSPATH` environment variable if it is set. However, it is recommended that this environment variable should *not* be set, and that the `--class-path` option should be used to provide an explicit value for the class path when one is required.

JDK_JAVAC_OPTIONS

The content of the `JDK_JAVAC_OPTIONS` environment variable, separated by white-spaces ( ) or white-space characters (`\n`, `\t`, `\r`, or `\f`) is prepended to the command line arguments passed to `javac` as a list of arguments.

The encoding requirement for the environment variable is the same as the `javac` command line on the system. `JDK_JAVAC_OPTIONS` environment variable content is treated in the same manner as that specified in the command line.

Single quotes (`'`) or double quotes (`"`) can be used to enclose arguments that contain whitespace characters. All content between the open quote and the first matching close quote are preserved by simply removing the pair of quotes. In case a matching quote is not found, the launcher will abort with an error message. `@`*files* are supported as they are specified in the command line. However, as in `@`*files*, use of a wildcard is not supported.

Examples of quoting arguments containing white spaces:

```
export JDK_JAVAC_OPTIONS='@"C:\white spaces\argfile"'
export JDK_JAVAC_OPTIONS='"@C:\white spaces\argfile"'
export JDK_JAVAC_OPTIONS='@C:\"white spaces"\argfile'
```

COMMAND-LINE ARGUMENT FILES

An argument file can include command-line 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 to the location of the argument file. Wildcards (`*`) 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're 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.

Examples of Using javac @filename

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 the following **Two Argument Files** example.

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:

Linux and macOS:

```
-d classes
-g
-sourcepath /java/pubs/ws/1.3/src/share/classes
```

Windows:

```
-d classes
-g
-sourcepath C:\java\pubs\ws\1.3\src\share\classes
```

Create a file named `sources` that contains the following:

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

Then, run the `javac` command as follows:

```
javac @options @sources
```

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/sources
```

ARRANGEMENT OF SOURCE CODE

In the Java language, classes and interfaces can be organized into packages, and packages can be organized into modules. `javac` expects that the physical arrangement of source files in directories of the file system will mirror the organization of classes into packages, and packages into modules.

It is a widely adopted convention that module names and package names begin with a lower-case letter, and that class names begin with an upper-case letter.

Arrangement of Source Code for a Package

When classes and interfaces are organized into a package, the package is represented as a directory, and any subpackages are represented as subdirectories.

For example:

- The package `p` is represented as a directory called `p`.
- The package `p.q` -- that is, the subpackage `q` of package `p` -- is represented as the subdirectory `q` of directory `p`. The directory tree representing package `p.q` is therefore `p\q` on Windows, and `p/q` on other systems.
- The package `p.q.r` is represented as the directory tree `p\q\r` (on Windows) or `p/q/r` (on other systems).

Within a directory or subdirectory, `.java` files represent classes and interfaces in the corresponding package or subpackage.

For example:

- The class `X` declared in package `p` is represented by the file `X.java` in the `p` directory.
- The class `Y` declared in package `p.q` is represented by the file `Y.java` in the `q` subdirectory of directory `p`.
- The class `Z` declared in package `p.q.r` is represented by the file `Z.java` in the `r` subdirectory of `p\q` (on Windows) or `p/q` (on other systems).

In some situations, it is convenient to split the code into separate directories, each structured as described above, and the aggregate list of directories specified to `javac`.

Arrangement of Source Code for a Module

In the Java language, a module is a set of packages designed for reuse. In addition to `.java` files for classes and interfaces, each module has a source file called `module-info.java` which:

1. declares the module's name;
2. lists the packages exported by the module (to allow reuse by other modules);

3. lists other modules required by the module (to reuse their exported packages).

When packages are organized into a module, the module is represented by one or more directories representing the packages in the module, one of which contains the `module-info.java` file. It may be convenient, but it is not required, to use a single directory, named after the module, to contain the `module-info.java` file alongside the directory tree which represents the packages in the module (i.e., the *package hierarchy* described above). The exact arrangement of source code for a module is typically dictated by the conventions adopted by a development environment (IDE) or build system.

For example:

- The module `a.b.c` may be represented by the directory `a.b.c`, on all systems.
- The module's declaration is represented by the file `module-info.java` in the `a.b.c` directory.
- If the module contains package `p.q.r`, then the `a.b.c` directory contains the directory tree `p\q\r` (on Windows) or `p/q/r` (on other systems).

The development environment may prescribe some directory hierarchy between the directory named for the module and the source files to be read by `javac`.

For example:

- The module `a.b.c` may be represented by the directory `a.b.c`
- The module's declaration and the module's packages may be in some subdirectory of `a.b.c`, such as `src\main\java` (on Windows) or `src/main/java` (on other systems).

CONFIGURING A COMPILATION

This section describes how to configure `javac` to perform a basic compilation.

See **Configuring the Module System** for additional details for use when compiling for a release of the platform that supports modules.

Source Files

- Specify the source files to be compiled on the command line.

If there are no compilation errors, the corresponding class files will be placed in the **output directory**.

Some systems may limit the amount you can put on a command line; to work around those limits, you can use **argument files**.

When compiling code for modules, you can also specify source files indirectly, by using the **`--module`** or `-m` option.

Output Directory

- Use the **`-d`** option to specify an output directory in which to put the compiled class files.

This will normally be organized in a **package hierarchy**, unless you are compiling source code from multiple modules, in which case it will be organized as a **module hierarchy**.

When the compilation has been completed, if you are compiling one or more modules, you can place the output directory on the module path for the Java **launcher**; otherwise, you can place the place the output directory on the class path for the Java launcher.

Precompiled Code

The code to be compiled may refer to libraries beyond what is provided by the platform. If so, you must place these libraries on the class path or module path. If the library code is not in a module, place it on the class path; if it is in a module, place it on the module path.

- Use the **`--class-path`** option to specify libraries to be placed on the class path. Locations on the class path should be organized in a **package hierarchy**. You can also use alternate forms of the option: `-classpath` or `-cp`.
- Use the **`--module-path`** option to specify libraries to be placed on the module path. Locations on the module path should either be modules or directories of modules. You can also use an alternate form of the option: `-p`.

See **Configuring the Module System** for details on how to modify the default configuration of library modules.

Note: the options for the class path and module path are not mutually exclusive, although it is not common to specify the class path when compiling code for one or more modules.

Additional Source Files

The code to be compiled may refer to types in additional source files that are not specified on the command line. If so, you must put those source files on either the source path or module path. You can only specify one of these options: if you are not compiling code for a module, or if you are only compiling code for a single module, use the source path; if you are compiling code for multiple modules, use the module source path.

- Use the **`--source-path`** option to specify the locations of additional source files that may be read by javac. Locations on the source path should be organized in a **package hierarchy**. You can also use an alternate form of the option: `-sourcepath`.
- Use the **`--module-source-path`** option one or more times to specify the location of additional source files in different modules that may be read by javac, or when compiling source files in multiple modules. You can either specify the locations for each module **individually**, or you can organize the source files so that you can specify the locations all **together**. For more details, see **The Module Source Path Option**.

If you want to be able to refer to types in additional source files but do not want them to be compiled, use the **`-implicit`** option.

Note: if you are compiling code for multiple modules, you must always specify a module source path, and all source files specified on the command line must be in one of the directories on the module source path, or in a subdirectory thereof.

Example of Compiling Multiple Source Files

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

Linux and macOS:

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

Windows:

```
C:\>javac greetings\*.java
C:\>dir greetings
Aloha.class         GutenTag.class      Hello.class         Hi.class
Aloha.java          GutenTag.java       Hello.java          Hi.java
```

Example of Specifying a User Class Path

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

Linux and macOS:

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

Windows:

```
C:\>cd
\examples
C:\>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.

Linux and macOS:

```
javac -classpath /examples /examples/greetings/Hi.java
```

Windows:

```
C:\>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.

Linux and macOS:

```
javac -classpath /examples:/lib/Banners.jar \
            /examples/greetings/Hi.java
```

Windows:

```
C:\>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.

Linux and macOS:

```
java -classpath /examples:/lib/Banners.jar greetings.Hi
```

Windows:

```
C:\>java -classpath \examples;\lib\Banners.jar greetings.Hi
```

CONFIGURING THE MODULE SYSTEM

If you want to include additional modules in your compilation, use the **`--add-modules`** option. This may be necessary when you are compiling code that is not in a module, or which is in an automatic module, and the code refers to API in the additional modules.

If you want to restrict the set of modules in your compilation, use the **`--limit-modules`** option. This may be useful if you want to ensure that the code you are compiling is capable of running on a system with a limited set of modules installed.

If you want to break encapsulation and specify that additional packages should be considered as exported from a module, use the **`--add-exports`** option. This may be useful when performing white-box testing; relying on access to internal API in production code is strongly discouraged.

If you want to specify that additional packages should be considered as required by a module, use the **`--add-reads`** option. This may be useful when performing white-box testing; relying on access to internal API in production code is strongly discouraged.

You can patch additional content into any module using the **`--patch-module`** option. See [Patching a Module] for more details.

SEARCHING FOR MODULE, PACKAGE AND TYPE DECLARATIONS

To compile a source file, the compiler often needs information about a module or type, but the declaration is not in the source files specified on the command line.

`javac` 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 of `java.awt.Window`, you are also using the ancestor classes of `Window`: `java.awt.Container`, `java.awt.Component`, and `java.lang.Object`.

When compiling code for a module, the compiler also needs to have available the declaration of that mod-

ule.

A successful 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 a search finds and uses a source file, then by default `javac` compiles that source file. This behavior can be altered with **-implicit**.

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.

The way that `javac` locates the declarations of those types depends on whether the reference exists within code for a module or not.

Searching Package Oriented Paths

When searching for a source or class file on a path composed of package oriented locations, `javac` will check each location on the path in turn for the possible presence of the file. The first occurrence of a particular file shadows (hides) any subsequent occurrences of like-named files. This shadowing does not affect any search for any files with a different name. This can be convenient when searching for source files, which may be grouped in different locations, such as shared code, platform-specific code and generated code. It can also be useful when injecting alternate versions of a class file into a package, to debugging or other instrumentation reasons. But, it can also be dangerous, such as when putting incompatible different versions of a library on the class path.

Searching Module Oriented Paths

Prior to scanning any module paths for any package or type declarations, `javac` will lazily scan the following paths and locations to determine the modules that will be used in the compilation.

- The module source path (see the **--module-source-path** option)
- The path for upgradeable modules (see the **--upgrade-module-path** option)
- The system modules (see the **--system** option)
- The user module path (see the **--module-path** option)

For any module, the first occurrence of the module during the scan completely shadows (hides) any subsequent appearance of a like-named module. While locating the modules, `javac` is able to determine the packages exported by the module and to associate with each module a package oriented path for the contents of the module. For any previously compiled module, this path will typically be a single entry for either a directory or a file that provides an internal directory-like hierarchy, such as a JAR file. Thus, when searching for a type that is in a package that is known to be exported by a module, `javac` can locate the declaration directly and efficiently.

Searching for the Declaration of a Module

If the module has been previously compiled, the module declaration is located in a file named `module-info.class` in the root of the package hierarchy for the content of the module.

If the module is one of those currently being compiled, the module declaration will be either the file named `module-info.class` in the root of the package hierarchy for the module in the class output directory, or the file named `module-info.java` in one of the locations on the source path or one the module source path for the module.

Searching for the Declaration of a Type When the Reference is not in a Module

When searching for a type that is referenced in code that is not in a module, `javac` will look in the following places:

- The platform classes (or the types in exported packages of the platform modules) (This is for compiled class files only.)

- Types in exported packages of any modules on the module path, if applicable. (This is for compiled class files only.)
- Types in packages on the class path and/or source path:
  - If both are specified, `javac` looks for compiled class files on the class path and for source files on the source path.
  - If the class path is specified, but not source path, `javac` looks for both compiled class files and source files on the class path.
  - If the class path is not specified, it defaults to the current directory.

When looking for a type on the class path and/or source path, if both a compiled class file and a source file are found, the most recently modified file will be used by default. If the source file is newer, it will be compiled and will may override any previously compiled version of the file. You can use the **`-Xprefer`** option to override the default behavior.

Searching for the Declaration of a Type When the Reference is in a Module

When searching for a type that is referenced in code in a module, `javac` will examine the declaration of the enclosing module to determine if the type is in a package that is exported from another module that is readable by the enclosing module. If so, `javac` will simply and directly go to the definition of that module to find the definition of the required type. Unless the module is another of the modules being compiled, `javac` will only look for compiled class files files. In other words, `javac` will not look for source files in platform modules or modules on the module path.

If the type being referenced is not in some other readable module, `javac` will examine the module being compiled to try and find the declaration of the type. `javac` will look for the declaration of the type as follows:

- Source files specified on the command line or on the source path or module source path.
- Previously compiled files in the output directory.

DIRECTORY HIERARCHIES

`javac` generally assumes that source files and compiled class files will be organized in a file system directory hierarchy or in a type of file that supports in an internal directory hierarchy, such as a JAR file. Three different kinds of hierarchy are supported: a *package hierarchy*, a *module hierarchy*, and a *module source hierarchy*.

While `javac` is fairly relaxed about the organization of source code, beyond the expectation that source will be organized in one or package hierarchies, and can generally accommodate organizations prescribed by development environments and build tools, Java tools in general, and `javac` and the Java launcher in particular, are more stringent regarding the organization of compiled class files, and will be organized in package hierarchies or module hierarchies, as appropriate.

The location of these hierarchies are specified to `javac` with command-line options, whose names typically end in "path", like **`--source-path`** or **`--class-path`**. Also as a general rule, path options whose name includes the word `module`, like **`--module-path`**, are used to specify module hierarchies, although some module-related path options allow a package hierarchy to be specified on a per-module basis. All other path options are used to specify package hierarchies.

Package Hierarchy

In a package hierarchy, directories and subdirectories are used to represent the component parts of the package name, with the source file or compiled class file for a type being stored as a file with an extension of `.java` or `.class` in the most nested directory.

For example, in a package hierarchy, the source file for a class `com.example.MyClass` will be stored in the file *com/example/MyClass.java*

Module Hierarchy

In a module hierarchy, the first level of directories are named for the modules in the hierarchy; within each of those directories the contents of the module are organized in package hierarchies.

For example, in a module hierarchy, the compiled class file for a type called `com.example.MyClass` in a module called `my.library` will be stored in *my.library/com/example/MyClass.class*.

The various output directories used by `javac` (the class output directory, the source output directory, and native header output directory) will all be organized in a module hierarchy when multiple modules are being compiled.

Module Source Hierarchy

Although the source for each individual module should always be organized in a package hierarchy, it may be convenient to group those hierarchies into a module source hierarchy. This is similar to a module hierarchy, except that there may be intervening directories between the directory for the module and the directory that is the root of the package hierarchy for the source code of the module.

For example, in a module source hierarchy, the source file for a type called `com.example.MyClass` in a module called `my.library` may be stored in a file such as *my.library/src/main/java/com/example/MyClass.java*.

THE MODULE SOURCE PATH OPTION

The **`--module-source-path`** option has two forms: a *module-specific form*, in which a package path is given for each module containing code to be compiled, and a *module-pattern* form, in which the source path for each module is specified by a pattern. The module-specific form is generally simpler to use when only a small number of modules are involved; the module-pattern form may be more convenient when the number of modules is large and the modules are organized in a regular manner that can be described by a pattern.

Multiple instances of the `--module-source-path` option may be given, each one using either the module-pattern form or the module-specific form, subject to the following limitations:

- the module-pattern form may be used at most once
- the module-specific form may be used at most once for any given module

If the module-specific form is used for any module, the associated search path overrides any path that might otherwise have been inferred from the module-pattern form.

Module-specific form

The module-specific form allows an explicit search path to be given for any specific module. This form is:

- `--module-source-path` *module-name=file-path* (*path-separator file-path*)*

The path separator character is **`;`** on Windows, and **`:`** otherwise.

Note: this is similar to the form used for the **`--patch-module`** option.

Module-pattern form

The module-pattern form allows a concise specification of the module source path for any number of modules organized in regular manner.

- `--module-source-path` *pattern*

The pattern is defined by the following rules, which are applied in order:

- The argument is considered to be a series of segments separated by the path separator character (**`;`** on Windows, and **`:`** otherwise).
- Each segment containing curly braces of the form

  ```
  string1{alt1 ( ,alt2 )* } string2
  ```

  is considered to be replaced by a series of segments formed by "expanding" the braces:

  ```
  string1 alt1 string2
  string1 alt2 string2
  and so on...
  ```

  The braces may be nested.

  This rule is applied for all such usages of braces.

- Each segment must have at most one asterisk (`*`). If a segment does not contain an asterisk, it is considered to be as though the file separator character and an asterisk are appended.

  For any module *M*, the source path for that module is formed from the series of segments obtained by substituting the module name *M* for the asterisk in each segment.

  Note: in this context, the asterisk is just used as a special marker, to denote the position in the path of the module name. It should not be confused with the use of `*` as a file name wildcard character, as found on most operating systems.

PATCHING MODULES

javac allows any content, whether in source or compiled form, to be patched into any module using the **`--patch-module`** option. You may want to do this to compile alternative implementations of a class to be patched at runtime into a JVM, or to inject additional classes into the module, such as when testing.

The form of the option is:

- `--patch-module` *module-name=file-path* (*path-separator file-path*)*

The path separator character is **`;`** on Windows, and **`:`** otherwise. The paths given for the module must specify the root of a package hierarchy for the contents of the module

The option may be given at most once for any given module. Any content on the path will hide any like-named content later in the path and in the patched module.

When patching source code into more than one module, the **`--module-source-path`** must also be used, so that the output directory is organized in a module hierarchy, and capable of holding the compiled class files for the modules being compiled.

ANNOTATION PROCESSING

The `javac` command provides direct support for annotation processing.

The API for annotation processors is defined in the `javax.annotation.processing` and `javax.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.

If you use an annotation processor that generates additional source files to be included in the compilation, you can specify a default module to be used for the newly generated files, for use when a module declaration is not also generated. In this case, use the **`--default-module-for-created-files`** option.

Compilation Environment and Runtime Environment.

The declarations in source files and previously compiled class files are analyzed by `javac` in a *compilation environment* that is distinct from the *runtime environment* used to execute `javac` itself. Although there is a deliberate similarity between many `javac` options and like-named options for the Java **launcher**, such as `--class-path`, `--module-path` and so on, it is important to understand that in general the `javac` options just affect the environment in which the source files are compiled, and do not affect the operation of `javac` itself.

The distinction between the compilation environment and runtime environment is significant when it comes to using annotation processors. Although annotations processors process elements (declarations) that exist in the compilation environment, the annotation processor itself is executed in the runtime environment. If an annotation processor has dependencies on libraries that are not in modules, the libraries can be placed, along with the annotation processor itself, on the processor path. (See the **`--processor-path`** option.) If the annotation processor and its dependencies are in modules, you should use the processor module path instead. (See the **`--processor-module-path`** option.) When those are insufficient, it may be necessary to provide further configuration of the runtime environment. This can be done in two ways:

1. If `javac` is invoked from the command line, options can be passed to the underlying runtime by prefixing the option with **`-J`**.
2. You can start an instance of a Java Virtual Machine directly and use command line options and API to configure an environment in which `javac` can be invoked via one of its **APIs**.

COMPILING FOR EARLIER RELEASES OF THE PLATFORM

`javac` can compile code that is to be used on other releases of the platform, using either the **`--release`** option, or the **`--source`**/`-source` and **`--target`**/`-target` options, together with additional options to specify the platform classes.

Depending on the desired platform release, there are some restrictions on some of the options that can be used.

- When compiling for JDK 8 and earlier releases, you cannot use any option that is intended for use with the module system. This includes all of the following options:
  - **`--module-source-path`**, **`--upgrade-module-path`**, **`--system`**, **`--module-path`**, **`--add-modules`**, **`--add-exports`**, `--add-opens`, **`--add-reads`**, **`--limit-modules`**, **`--patch-module`**

  If you use the `--source`/`-source` or `--target`/`-target` options, you should also set the appropriate platform classes using the boot class path family of options.
- When compiling for JDK 9 and later releases, you cannot use any option that is intended to configure the boot class path. This includes all of the following options:
  - **`-Xbootclasspath/p:`**, **`-Xbootclasspath`**, **`-Xbootclasspath/a:`**, **`-endorseddirs`**, **`-Djava.endorsed.dirs`**, **`-extdirs`**, **`-Djava.ext.dirs`**, **`-profile`**

  If you use the `--source`/`-source` or `--target`/`-target` options, you should also set the appropriate platform classes using the `--system` option to give the location of an appropriate installed release of JDK.

When using the `--release` option, only the supported documented API for that release may be used; you cannot use any options to break encapsulation to access any internal classes.

APIS

The `javac` compiler can be invoked using an API in three different ways:

The **Java Compiler API**

This provides the most flexible way to invoke the compiler, including the ability to compile source files provided in memory buffers or other non-standard file systems.

The **ToolProvider API**

A `ToolProvider` for `javac` can be obtained by calling `ToolProvider.findFirst("javac")`. This returns an object with the equivalent functionality of the command-line tool.

Note: This API should not be confused with the like-named API in the **`javax.tools`** package.

The `javac` **Legacy API**

This API is retained for backward compatibility only. All new code should use either the Java Compiler API or the ToolProvider API.

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 OF USING -XLINT KEYS

`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 the `@deprecated` Javadoc comment, but do not have the `@Deprecated` annotation, for example:

```
/**
  * @deprecated As of Java SE 7, replaced by {@link #newMethod()}
  */
public static void deprecatedMethod() { }
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 be completed 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's executed, even though control was transferred outside the method.

`options`
Warns about issues that related to the use of command-line options. See **Compiling for Earlier Releases of the Platform**.

`overrides`
Warns about issues related to 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 that matches the formal parameter of the method `ClassWithOverriding-Method.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:

- **Linux and macOS:** `javac -Xlint:path -classpath /nonexistentpath Example.java`

- **Windows:** `javac -Xlint:path -classpath C:\nonexistentpath Example.java`

`processing`

Warns about issues related to 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 annotation. For example, the following is a simple annotation processor:

Source file AnnoProc.java:

```
import java.util.*;
import javax.annotation.processing.*;
import javax.lang.model.*;
import javax.lang.model.element.*;

@SupportedAnnotationTypes("NotAnno")
public class AnnoProc extends AbstractProcessor {
  public boolean process(Set<? extends TypeElement> elems, RoundEnvironm
     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 AnnosWithou
```

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: Ann
```

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's strongly recommended that all serializable classes explicitly declare `serialVersionUID` values because the default process of computing `serialVersionUID` values is highly sensitive to class details that can vary depending on compiler implementations. As a result, this 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 static variables, 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`.

`this-escape`

Warns about constructors leaking `this` prior to subclass initialization. For example, this class:

```
public class MyClass {
  public MyClass() {
    System.out.println(this.hashCode());
  }
}
```

generates the following warning:

```
MyClass.java:3: warning: [this-escape] possible 'this' escape
                         before subclass is fully initialized
```

```
System.out.println(this.hashCode());
                                   ^
```

A 'this' escape warning is generated when a constructor does something that might result in a subclass method being invoked before the constructor returns. In such cases the subclass method would be operating on an incompletely initialized instance. In the above example, a subclass of `MyClass` that overrides `hashCode()` to incorporate its own fields would likely produce an incorrect result when invoked as shown.

Warnings are only generated if a subclass could exist that is outside of the current module (or package, if no module) being compiled. So, for example, constructors in final and non-public classes do not generate warnings.

`try` Warns about issues relating to the 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 use 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);
    }
  }
}
```

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 typ
```

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's a possibility of heap pollution.