Module java.compiler
Package javax.annotation.processing

Interface Filer

```
public interface Filer
```

This interface supports the creation of new files by an annotation processor. Files created in this way will be known to the annotation processing tool implementing this interface, better enabling the tool to manage them. Source and class files so created will be considered for processing by the tool in a subsequent round of processing after the `close` method has been called on the `Writer` or `OutputStream` used to write the contents of the file. Three kinds of files are distinguished: source files, class files, and auxiliary resource files.

There are two distinguished supported locations (subtrees within the logical file system) where newly created files are placed: one for new source files, and one for new class files. (These might be specified on a tool's command line, for example, using flags such as `-s` and `-d`.) The actual locations for new source files and new class files may or may not be distinct on a particular run of the tool. Resource files may be created in either location. The methods for reading and writing resources take a relative name argument. A relative name is a non-null, non-empty sequence of path segments separated by `'/'`; `'.'` and `'..'` are invalid path segments. A valid relative name must match the "path-rootless" rule of RFC 3986[], section 3.3.

The file creation methods take a variable number of arguments to allow the *originating elements* to be provided as hints to the tool infrastructure to better manage dependencies. The originating elements are the classes or interfaces or packages (representing `package-info` files) or modules (representing `module-info` files) which caused an annotation processor to attempt to create a new file. In other words, the originating elements are intended to have the granularity of *compilation units* (JLS section 7.3[]), essentially file-level granularity, rather than finer-scale granularity of, say, a method or field declaration.

For example, if an annotation processor tries to create a source file, `GeneratedFromUserSource`, in response to processing

```
@Generate
public class UserSource {}
```

the type element for `UserSource` should be passed as part of the creation method call as in:

```
filer.createSourceFile("GeneratedFromUserSource",
                       eltUtils.getTypeElement("UserSource"));
```

If there are no originating elements, none need to be passed. This information may be used in an incremental environment to determine the need to rerun processors or remove generated files. Non-incremental environments may ignore the originating element information.

During each run of an annotation processing tool, a file with a given pathname may be created only once. If that file already exists before the first attempt to create it, the old contents will be deleted. Any subsequent attempt to create the same file during a run will throw a `FilerException`, as will attempting to create both a class file and source file for the same type name or same package name. The initial inputs to the tool are considered to be created by the zeroth round; therefore, attempting to create a source or class file corresponding to one of those inputs will result in a `FilerException`.

In general, processors must not knowingly attempt to overwrite existing files that were not generated by some processor. A `Filer` may reject attempts to open a file corresponding to an existing class or interface, like `java.lang.Object`. Likewise, the invoker of the annotation processing tool must not knowingly configure the tool such that the discovered processors will attempt to overwrite existing files that were not generated.

Processors can indicate a source or class file is generated by including a `Generated` annotation if the environment is configured so that that class or interface is accessible.

API Note:
Some of the effect of overwriting a file can be achieved by using a *decorator*-style pattern. Instead of modifying a class directly, the class is designed so that either its superclass is generated by annotation processing or subclasses of the class are generated by annotation processing. If the subclasses are generated, the parent class may be designed to use factories instead of public constructors so that only subclass instances would be presented to clients of the parent class.

Since:
1.6

External Specifications
RFC 3986: Uniform Resource Identifier (URI): Generic Syntax

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method	Description
`JavaFileObject`	`createClassFile(CharSequence name, Element... originatingElements)`	Creates a new class file, and returns an object to allow writing to it.
`FileObject`	`createResource(JavaFileManager.Location location, CharSequence moduleAndPkg, CharSequence relativeName, Element... originatingElements)`	Creates a new auxiliary resource file for writing and returns a file object for it.
`JavaFileObject`	`createSourceFile(CharSequence name, Element... originatingElements)`	Creates a new source file and returns an object to allow writing to it.
`FileObject`	`getResource(JavaFileManager.Location location, CharSequence moduleAndPkg, CharSequence relativeName)`	Returns an object for reading an existing resource.

Method Details

createSourceFile

```
JavaFileObject createSourceFile(CharSequence name,
                                Element... originatingElements)
                         throws IOException
```

Creates a new source file and returns an object to allow writing to it. A source file for a class, interface, or a package can be created. The file's name and path (relative to the root output location for source files) are based on the name of the item to be declared in that file as well as the specified module for the item (if any). If more than one class or interface is being declared in a single file (that is, a single compilation unit), the name of the file should correspond to the name of the principal top-level class or interface (the public one, for example).

A source file can also be created to hold information about a package, including package annotations. To create a source file for a named package, have the `name` argument be the package's name followed by `".package-info"`; to create a source file for an unnamed package, use `"package-info"`.

The optional module name is prefixed to the type name or package name and separated using a `"/"` character. For example, to create a source file for class `a.B` in module `foo`, use a `name` argument of `"foo/a.B"`.

If no explicit module prefix is given and modules are supported in the environment, a suitable module is inferred. If a suitable module cannot be inferred `FilerException` is thrown. An implementation may use information about the configuration of the annotation processing tool as part of the inference.

Creating a source file in or for an *unnamed* package in a *named* module is *not* supported.

If the environment is configured to support unnamed classes[PREVIEW], the name argument is used to provide the leading component of the name used for the output file. For example `filer.createSourceFile("Foo")` to create an unnamed class hosted in `Foo.java`. All unnamed classes must be in an unnamed package.

API Note:
To use a particular charset to encode the contents of the file, an `OutputStreamWriter` with the chosen charset can be created from the `OutputStream` from the returned object. If the `Writer` from the returned object is directly used for writing, its charset is determined by the implementation. An annotation processing tool may have an `-encoding` flag or analogous option for specifying this; otherwise, it will typically be the platform's default encoding.

To avoid subsequent errors, the contents of the source file should be compatible with the source version being used for this run.

Implementation Note:
In the reference implementation, if the annotation processing tool is processing a single module *M*, then *M* is used as the module for files created without an explicit module prefix. If the tool is processing multiple modules, and `Elements.getPackageElement(package-of(name))` returns a package, the module that owns the returned package is used as the target module. A separate option may be used to provide the target module if it cannot be determined using the above rules.

Parameters:
`name` - canonical (fully qualified) name of the principal class or interface being declared in this file or a package name followed by `".package-info"` for a package information file

`originatingElements` - class, interface, package, or module elements causally associated with the creation of this file, may be elided or `null`

Returns:
a `JavaFileObject` to write the new source file

Throws:
`FilerException` - if the same pathname has already been created, the same class or interface has already been created, the name is otherwise not valid for the entity requested to being created, if the target module cannot be determined, if the target module is not writable, or a module is specified when the environment doesn't support modules.

`IOException` - if the file cannot be created

See *Java Language Specification*:
7.3 Compilation Units

createClassFile

```
JavaFileObject createClassFile(CharSequence name,
                               Element... originatingElements)
                        throws IOException
```

Creates a new class file, and returns an object to allow writing to it. A class file for a class, interface, or a package can be created. The file's name and path (relative to the root output location for class files) are based on the name of the item to be declared as well as the specified module for the item (if any).

A class file can also be created to hold information about a package, including package annotations. To create a class file for a named package, have the `name` argument be the package's name followed by `".package-info"`; creating a class file for an unnamed package is not supported.

The optional module name is prefixed to the type name or package name and separated using a `"/"` character. For example, to create a class file for class `a.B` in module `foo`, use a `name` argument of `"foo/a.B"`.

If no explicit module prefix is given and modules are supported in the environment, a suitable module is inferred. If a suitable module cannot be inferred `FilerException` is thrown. An implementation may use information about the configuration of the annotation processing tool as part of the inference.

Creating a class file in or for an *unnamed* package in a *named* module is *not* supported.

If the environment is configured to support unnamed classes[PREVIEW], the name argument is used to provide the leading component of the name used for the output file. For example `filer.createClassFile("Foo")` to create an unnamed class hosted in `Foo.class`. All unnamed classes must be in an unnamed package.

API Note:
To avoid subsequent errors, the contents of the class file should be compatible with the source version being used for this run.

Implementation Note:
In the reference implementation, if the annotation processing tool is processing a single module *M*, then *M* is used as the module for files created without an explicit module prefix. If the tool is processing multiple modules, and `Elements.getPackageElement(package-of(name))` returns a package, the module that owns the returned package is used as the target module. A separate option may be used to provide the target module if it cannot be determined using the above rules.

Parameters:
`name` - binary name of the class or interface being written or a package name followed by `".package-info"` for a package information file

`originatingElements` - class or interface or package or module elements causally associated with the creation of this file, may be elided or `null`

Returns:
a `JavaFileObject` to write the new class file

Throws:
`FilerException` - if the same pathname has already been created, the same class or interface has already been created, the name is not valid for a class or interface, if the target module cannot be determined, if the target module is not writable, or a module is specified when the environment doesn't support modules.

`IOException` - if the file cannot be created

createResource

```
FileObject createResource(JavaFileManager.Location location,
                          CharSequence moduleAndPkg,
                          CharSequence relativeName,
                          Element... originatingElements)
                   throws IOException
```

Creates a new auxiliary resource file for writing and returns a file object for it. The file may be located along with the newly created source files, newly created binary files, or other supported location. The locations `CLASS_OUTPUT` and `SOURCE_OUTPUT` must be supported. The resource may be named relative to some module and/or package (as are source and class files), and from there by a relative pathname. In a loose sense, the full pathname of the new file will be the concatenation of `location`, `moduleAndPkg`, and `relativeName`. If `moduleAndPkg` contains a `"/"` character, the prefix before the `"/"` character is the module name and the suffix after the `"/"` character is the package name. The package suffix may be empty. If `moduleAndPkg` does not contain a `"/"` character, the entire argument is interpreted as a package name.

If the given location is neither a module oriented location, nor an output location containing multiple modules, and the explicit module prefix is given, `FilerException` is thrown.

If the given location is either a module oriented location, or an output location containing multiple modules, and no explicit modules prefix is given, a suitable module is inferred. If a suitable module cannot be inferred `FilerException` is thrown. An implementation may use information about the configuration of the annotation processing tool as part of the inference.

Files created via this method are *not* registered for annotation processing, even if the full pathname of the file would correspond to the full pathname of a new source file or new class file.

Implementation Note:
In the reference implementation, if the annotation processing tool is processing a single module *M*, then *M* is used as the module for files created without an explicit module prefix. If the tool is processing multiple modules, and `Elements.getPackageElement(package-of(name))` returns a package, the module that owns the returned package is used as the target module. A separate option may be used to provide the target module if it cannot be determined using the above rules.

Parameters:
`location` - location of the new file

`moduleAndPkg` - module and/or package relative to which the file should be named, or the empty string if none

`relativeName` - final pathname components of the file

`originatingElements` - class or interface or package or module elements causally associated with the creation of this file, may be elided or `null`

Returns:
a `FileObject` to write the new resource

Throws:
`IOException` - if the file cannot be created

`FilerException` - if the same pathname has already been created, if the target module cannot be determined, or if the target module is not writable, or if an explicit target module is specified and the location does not support it.

`IllegalArgumentException` - for an unsupported location

`IllegalArgumentException` - if `moduleAndPkg` is ill-formed

`IllegalArgumentException` - if `relativeName` is not relative

getResource

```
FileObject getResource(JavaFileManager.Location location,
                       CharSequence moduleAndPkg,
                       CharSequence relativeName)
                throws IOException
```

Returns an object for reading an existing resource. The locations `CLASS_OUTPUT` and `SOURCE_OUTPUT` must be supported.

If `moduleAndPkg` contains a `"/"` character, the prefix before the `"/"` character is the module name and the suffix after the `"/"` character is the package name. The package suffix may be empty; however, if a module name is present, it must be nonempty. If `moduleAndPkg` does not contain a `"/"` character, the entire argument is interpreted as a package name.

If the given location is neither a module oriented location, nor an output location containing multiple modules, and the explicit module prefix is given, `FilerException` is thrown.

If the given location is either a module oriented location, or an output location containing multiple modules, and no explicit modules prefix is given, a suitable module is inferred. If a suitable module cannot be inferred `FilerException` is thrown. An implementation may use information about the configuration of the annotation processing tool as part of the inference.

Implementation Note:
In the reference implementation, if the annotation processing tool is processing a single module *M*, then *M* is used as the module for files read without an explicit module prefix. If the tool is processing multiple modules, and `Elements.getPackageElement(package-of(name))` returns a package, the module that owns the returned package is used as the source module. A separate option may be used to provide the target module if it cannot be determined using the above rules.

Parameters:
`location` - location of the file

`moduleAndPkg` - module and/or package relative to which the file should be searched for, or the empty string if none

`relativeName` - final pathname components of the file

Returns:
an object to read the file

Throws:
`FilerException` - if the same pathname has already been opened for writing, if the source module cannot be determined, or if the target module is not writable, or if an explicit target module is specified and the location does not support it.

`IOException` - if the file cannot be opened

`IllegalArgumentException` - for an unsupported location

`IllegalArgumentException` - if `moduleAndPkg` is ill-formed

`IllegalArgumentException` - if `relativeName` is not relative