Java SE 25 & JDK 25

Class Files

java.lang.Object
    java.nio.file.Files

```
public final class Files
extends Object
```

This class consists exclusively of static methods that operate on files, directories, or other types of files.

In most cases, the methods defined here will delegate to the associated file system provider to perform the file operations.

Since:

1.7

Method Summary

All Methods Static Methods Concrete Methods

Modifier and Type	Method	Description
static long	**copy**(InputStream in, Path target, CopyOption... options)	Copies all bytes from an input stream to a file.
static long	**copy**(Path source, OutputStream out)	Copies all bytes from a file to an output stream.
static Path	**copy**(Path source, Path target, CopyOption... options)	Copy a file to a target file.
static Path	**createDirectories**(Path dir, FileAttribute<?>... attrs)	Creates a directory by creating all nonexistent parent directories first.
static Path	**createDirectory**(Path dir, FileAttribute<?>... attrs)	Creates a new directory.
static Path	**createFile**(Path path, FileAttribute<?>... attrs)	Creates a new and empty file, failing if the file already exists.
static Path	**createLink**(Path link, Path existing)	Creates a new link (directory entry) for an existing file

		(optional operation).
static Path	**createSymbolicLink**(Path link, Path target, FileAttribute<?>... attrs)	Creates a symbolic link to a target *(optional operation)*.
static Path	**createTempDirectory**(String prefix, FileAttribute<?>... attrs)	Creates a new directory in the default temporary-file directory, using the given prefix to generate its name.
static Path	**createTempDirectory**(Path dir, String prefix, FileAttribute<?>... attrs)	Creates a new directory in the specified directory, using the given prefix to generate its name.
static Path	**createTempFile**(String prefix, String suffix, FileAttribute<?>... attrs)	Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static Path	**createTempFile**(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)	Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
static void	**delete**(Path path)	Deletes a file.
static boolean	**deleteIfExists**(Path path)	Deletes a file if it exists.
static boolean	**exists**(Path path, LinkOption... options)	Tests whether a file exists.
static Stream<Path>	**find**(Path start, int maxDepth, BiPredicate<Path, BasicFileAttributes> matcher FileVisitOption... options)	Returns a `Stream` that is lazily populated with `Path` by searching for files in a file tree rooted at a given starting file.
static Object	**getAttribute**(Path path, String attribute, LinkOption... options)	Reads the value of a file attribute.
static <V extends FileAttributeView> V	**getFileAttributeView**(Path path, Class<V> type, LinkOption... options)	Returns a file attribute view of a given type.

static FileStore	**getFileStore**(Path path)	Returns the FileStore representing the file store where a file is located.
static FileTime	**getLastModifiedTime**(Path path, LinkOption... options)	Returns a file's last modified time.
static UserPrincipal	**getOwner**(Path path, LinkOption... options)	Returns the owner of a file.
static Set<PosixFilePermission>	**getPosixFilePermissions**(Path path, LinkOption... options)	Returns a file's POSIX file permissions.
static boolean	**isDirectory**(Path path, LinkOption... options)	Tests whether a file is a directory.
static boolean	**isExecutable**(Path path)	Tests whether a file is executable.
static boolean	**isHidden**(Path path)	Tells whether or not a file is considered *hidden*.
static boolean	**isReadable**(Path path)	Tests whether a file is readable.
static boolean	**isRegularFile**(Path path, LinkOption... options)	Tests whether a file is a regular file with opaque content.
static boolean	**isSameFile**(Path path, Path path2)	Tests if two paths locate the same file.
static boolean	**isSymbolicLink**(Path path)	Tests whether a file is a symbolic link.
static boolean	**isWritable**(Path path)	Tests whether a file is writable.
static Stream<String>	**lines**(Path path)	Read all lines from a file as a Stream.
static Stream<String>	**lines**(Path path, Charset cs)	Read all lines from a file as a Stream.
static Stream<Path>	**list**(Path dir)	Returns a lazily populated Stream, the elements of which

		are the entries in the directory.
static long	**mismatch**(Path path, Path path2)	Finds and returns the position of the first mismatched byte in the content of two files, or `-1L` if there is no mismatch.
static Path	**move**(Path source, Path target, CopyOption... options)	Move or rename a file to a target file.
static BufferedReader	**newBufferedReader**(Path path)	Opens a file for reading, returning a `BufferedReader` to read text from the file in an efficient manner.
static BufferedReader	**newBufferedReader**(Path path, Charset cs)	Opens a file for reading, returning a `BufferedReader` that may be used to read text from the file in an efficient manner.
static BufferedWriter	**newBufferedWriter**(Path path, Charset cs, OpenOption... options)	Opens or creates a file for writing, returning a `BufferedWriter` that may be used to write text to the file in an efficient manner.
static BufferedWriter	**newBufferedWriter**(Path path, OpenOption... options)	Opens or creates a file for writing, returning a `BufferedWriter` to write text to the file in an efficient manner.
static SeekableByteChannel	**newByteChannel**(Path path, OpenOption... options)	Opens or creates a file, returning a seekable byte channel to access the file.
static SeekableByteChannel	**newByteChannel**(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs)	Opens or creates a file, returning a seekable byte channel to access the file.
static DirectoryStream<Path>	**newDirectoryStream**(Path dir)	Opens a directory, returning a `DirectoryStream` to iterate

		over all entries in the directory.
static DirectoryStream<Path>	**newDirectoryStream**(Path dir, String glob)	Opens a directory, returning a DirectoryStream to iterate over the entries in the directory.
static DirectoryStream<Path>	**newDirectoryStream**(Path dir, DirectoryStream.Filter<? super Path> filter)	Opens a directory, returning a DirectoryStream to iterate over the entries in the directory.
static InputStream	**newInputStream**(Path path, OpenOption... options)	Opens a file, returning an input stream to read from the file.
static OutputStream	**newOutputStream**(Path path, OpenOption... options)	Opens or creates a file, returning an output stream that may be used to write bytes to the file.
static boolean	**notExists**(Path path, LinkOption... options)	Tests whether the file located by this path does not exist.
static String	**probeContentType**(Path path)	Probes the content type of a file.
static byte[]	**readAllBytes**(Path path)	Reads all the bytes from a file.
static List<String>	**readAllLines**(Path path)	Read all lines from a file.
static List<String>	**readAllLines**(Path path, Charset cs)	Read all lines from a file.
static <A extends BasicFileAttributes> A	**readAttributes**(Path path, Class<A> type, LinkOption... options)	Reads a file's attributes as a bulk operation.
static Map<String, Object>	**readAttributes**(Path path, String attributes, LinkOption... options)	Reads a set of file attributes as a bulk operation.
static String	**readString**(Path path)	Reads all content from a file into a string, decoding from bytes to characters using the UTF-8 charset.

static String	**readString**(Path path, Charset cs)	Reads all characters from a file into a string, decoding from bytes to characters using the specified charset.
static Path	**readSymbolicLink**(Path link)	Reads the target of a symbolic link *(optional operation)*.
static Path	**setAttribute**(Path path, String attribute, Object value, LinkOption... options)	Sets the value of a file attribute.
static Path	**setLastModifiedTime**(Path path, FileTime time)	Updates a file's last modified time attribute.
static Path	**setOwner**(Path path, UserPrincipal owner)	Updates the file owner.
static Path	**setPosixFilePermissions**(Path path, Set<PosixFilePermission> per	Sets a file's POSIX permissions.
static long	**size**(Path path)	Returns the size of a file (in bytes).
static Stream<Path>	**walk**(Path start, int maxDepth, FileVisitOption... options)	Returns a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
static Stream<Path>	**walk**(Path start, FileVisitOption... options)	Returns a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
static Path	**walkFileTree**(Path start, FileVisitor<? super Path> visitor)	Walks a file tree.
static Path	**walkFileTree**(Path start, Set<FileVisitOption> options int maxDepth, FileVisitor<? super Path> visitor)	Walks a file tree.

static Path	**write**(Path path, byte[] bytes, OpenOption... options)	Writes bytes to a file.
static Path	**write**(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options)	Write lines of text to a file.
static Path	**write**(Path path, Iterable<? extends CharSequence> lines, OpenOption... options)	Write lines of text to a file.
static Path	**writeString**(Path path, CharSequence csq, Charset cs, OpenOption... options)	Write a CharSequence to a file.
static Path	**writeString**(Path path, CharSequence csq, OpenOption... options)	Write a CharSequence to a file.

Methods declared in class Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Details

newInputStream

```
public static InputStream newInputStream(Path path,
                                         OpenOption... options)
                                  throws IOException
```

Opens a file, returning an input stream to read from the file. The stream will not be buffered, and is not required to support the mark or reset methods. The stream will be safe for access by multiple concurrent threads. Reading commences at the beginning of the file. Whether the returned stream is *asynchronously closeable* and/or *interruptible* is highly file system provider specific and therefore not specified.

The options parameter determines how the file is opened. If no options are present then it is equivalent to opening the file with the READ option. In addition to the READ option, an implementation may also support additional implementation specific options.

Parameters:

`path` - the path to the file to open

`options` - options specifying how the file is opened

Returns:

a new input stream

Throws:

`IllegalArgumentException` - if an invalid combination of options is specified

`UnsupportedOperationException` - if an unsupported option is specified

`IOException` - if an I/O error occurs

newOutputStream

```
public static OutputStream newOutputStream(Path path,
                                           OpenOption... options)
                                    throws IOException
```

Opens or creates a file, returning an output stream that may be used to write bytes to the file. The resulting stream will not be buffered. The stream will be safe for access by multiple concurrent threads. Whether the returned stream is *asynchronously closeable* and/or *interruptible* is highly file system provider specific and therefore not specified.

This method opens or creates a file in exactly the manner specified by the `newByteChannel` method with the exception that the `READ` option may not be present in the array of options. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of `0` if it exists.

Usage Examples:

```
Path path = ...

// truncate and overwrite an existing file, or create the file if
// it doesn't initially exist
OutputStream out = Files.newOutputStream(path);

// append to an existing file, fail if the file does not exist
out = Files.newOutputStream(path, APPEND);

// append to an existing file, create file if it doesn't initially exis
out = Files.newOutputStream(path, CREATE, APPEND);
```

```
// always create new file, failing if it already exists
out = Files.newOutputStream(path, CREATE_NEW);
```

Parameters:

`path` - the path to the file to open or create

`options` - options specifying how the file is opened

Returns:

a new output stream

Throws:

`IllegalArgumentException` - if `options` contains an invalid combination of options

`UnsupportedOperationException` - if an unsupported option is specified

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified *(optional specific exception)*

`IOException` - if an I/O error occurs

newByteChannel

```
public static SeekableByteChannel newByteChannel
(Path path,
 Set<? extends OpenOption> options,
 FileAttribute<?>... attrs)
                                        throws IOException
```

Opens or creates a file, returning a seekable byte channel to access the file.

The `options` parameter determines how the file is opened. The `READ` and `WRITE` options determine if the file should be opened for reading and/or writing. If neither option (or the `APPEND` option) is present then the file is opened for reading. By default reading or writing commence at the beginning of the file.

In the addition to `READ` and `WRITE`, the following options may be present:

Option	Description
`APPEND`	If this option is present then the file is opened for writing and each invocation of the channel's `write` method first advances the position to the end of the file and then writes the requested data. Whether the advancement of the position and the writing of the data are done in a single atomic operation is system-dependent and therefore unspecified. This option may not be used in conjunction with the `READ` or `TRUNCATE_EXISTING` options.
`TRUNCATE_EXISTING`	If this option is present then the existing file is truncated to a size of 0 bytes. This option is ignored when the file is opened only for

Option	Description
	reading.
CREATE_NEW	If this option is present then a new file is created, failing if the file already exists or is a symbolic link. When creating a file the check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other file system operations. This option is ignored when the file is opened only for reading.
CREATE	If this option is present then an existing file is opened if it exists, otherwise a new file is created. This option is ignored if the CREATE_NEW option is also present or the file is opened only for reading.
DELETE_ON_CLOSE	When this option is present then the implementation makes a *best effort* attempt to delete the file when closed by the close method. If the close method is not invoked then a *best effort* attempt is made to delete the file when the Java virtual machine terminates.
SPARSE	When creating a new file this option is a *hint* that the new file will be sparse. This option is ignored when not creating a new file.
SYNC	Requires that every update to the file's content or metadata be written synchronously to the underlying storage device. (see Synchronized I/O file integrity).
DSYNC	Requires that every update to the file's content be written synchronously to the underlying storage device. (see Synchronized I/O file integrity).

An implementation may also support additional implementation specific options.

The attrs parameter is optional file-attributes to set atomically when a new file is created.

In the case of the default provider, the returned seekable byte channel is a FileChannel.

Usage Examples:

```
Path path = ...

// open file for reading
ReadableByteChannel rbc = Files.newByteChannel(path, EnumSet.of(READ)))

// open file for writing to the end of an existing file, creating
// the file if it doesn't already exist
WritableByteChannel wbc = Files.newByteChannel(path, EnumSet.of(CREATE,

// create file with initial permissions, opening it for both reading an
FileAttribute<Set<PosixFilePermission>> perms = ...
```

```
SeekableByteChannel sbc =
    Files.newByteChannel(path, EnumSet.of(CREATE_NEW,READ,WRITE), perms
```

Parameters:

`path` - the path to the file to open or create

`options` - options specifying how the file is opened

`attrs` - an optional list of file attributes to set atomically when creating the file

Returns:

a new seekable byte channel

Throws:

`IllegalArgumentException` - if the set contains an invalid combination of options

`UnsupportedOperationException` - if an unsupported open option is specified or the array contains attributes that cannot be set atomically when creating the file

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified and the file is being opened for writing *(optional specific exception)*

`IOException` - if an I/O error occurs

See Also:

`FileChannel.open(Path,Set,FileAttribute[])`

newByteChannel

```
public static SeekableByteChannel newByteChannel(Path path,
                                                 OpenOption... options)
                                          throws IOException
```

Opens or creates a file, returning a seekable byte channel to access the file.

This method opens or creates a file in exactly the manner specified by the `newByteChannel` method.

Parameters:

`path` - the path to the file to open or create

`options` - options specifying how the file is opened

Returns:

a new seekable byte channel

Throws:

`IllegalArgumentException` - if the set contains an invalid combination of options

`UnsupportedOperationException` - if an unsupported open option is specified

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified and the file is being opened for writing *(optional specific exception)*

`IOException` - if an I/O error occurs

See Also:

`FileChannel.open(Path,OpenOption[])`

newDirectoryStream

```
public static DirectoryStream<Path> newDirectoryStream(Path dir)
                                                throws IOException
```

Opens a directory, returning a `DirectoryStream` to iterate over all entries in the directory. The elements returned by the directory stream's `iterator` are of type `Path`, each one representing an entry in the directory. The `Path` objects are obtained as if by `resolving` the name of the directory entry against `dir`.

When not using the try-with-resources construct, then directory stream's `close` method should be invoked after iteration is completed so as to free any resources held for the open directory.

When an implementation supports operations on entries in the directory that execute in a race-free manner then the returned directory stream is a `SecureDirectoryStream`.

Parameters:

`dir` - the path to the directory

Returns:

a new and open `DirectoryStream` object

Throws:

`NotDirectoryException` - if the file could not otherwise be opened because it is not a directory *(optional specific exception)*

`IOException` - if an I/O error occurs

newDirectoryStream

```
public static DirectoryStream<Path> newDirectoryStream(Path dir,
                                                       String glob)
                                                throws IOException
```

Opens a directory, returning a `DirectoryStream` to iterate over the entries in the directory. The elements returned by the directory stream's `iterator` are of type `Path`, each one representing an entry in the directory. The `Path` objects are obtained as if by `resolving` the name of the directory entry against `dir`. The entries returned by the iterator are filtered by matching the `String` representation of their file names against the given *globbing* pattern.

For example, suppose we want to iterate over the files ending with ".java" in a directory:

```
Path dir = ...
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "*.ja
    :
}
```

The globbing pattern is specified by the `getPathMatcher` method.

When not using the try-with-resources construct, then directory stream's `close` method should be invoked after iteration is completed so as to free any resources held for the open directory.

When an implementation supports operations on entries in the directory that execute in a race-free manner then the returned directory stream is a `SecureDirectoryStream`.

Parameters:

`dir` - the path to the directory

`glob` - the glob pattern

Returns:

a new and open `DirectoryStream` object

Throws:

`PatternSyntaxException` - if the pattern is invalid

`NotDirectoryException` - if the file could not otherwise be opened because it is not a directory *(optional specific exception)*

`IOException` - if an I/O error occurs

newDirectoryStream

```
public static DirectoryStream<Path> newDirectoryStream
(Path dir,
 DirectoryStream.Filter<? super Path> filter)
                                                throws IOException
```

Opens a directory, returning a `DirectoryStream` to iterate over the entries in the directory. The elements returned by the directory stream's `iterator` are of type `Path`, each one representing an entry in the directory. The `Path` objects are obtained as if by `resolving` the name of the directory entry against `dir`. The entries returned by the iterator are filtered by the given `filter`.

When not using the try-with-resources construct, then directory stream's `close` method should be invoked after iteration is completed so as to free any resources held for the open directory.

Where the filter terminates due to an uncaught error or runtime exception then it is propagated to the `hasNext` or `next` method. Where an `IOException` is thrown, it results in the `hasNext` or `next` method throwing a `DirectoryIteratorException` with the `IOException` as the cause.

When an implementation supports operations on entries in the directory that execute in a race-free manner then the returned directory stream is a `SecureDirectoryStream`.

Usage Example: Suppose we want to iterate over the files in a directory that are larger than 8K.

```
DirectoryStream.Filter<Path> filter = new DirectoryStream.Filter<Path>(
    public boolean accept(Path file) throws IOException {
        return (Files.size(file) > 8192L);
    }
};
Path dir = ...
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, filte
    :
}
```

Parameters:

`dir` - the path to the directory

`filter` - the directory stream filter

Returns:

a new and open `DirectoryStream` object

Throws:

`NotDirectoryException` - if the file could not otherwise be opened because it is not a directory *(optional specific exception)*

`IOException` - if an I/O error occurs

createFile

```
public static Path createFile(Path path,
                              FileAttribute<?>... attrs)
                       throws IOException
```

Creates a new and empty file, failing if the file already exists. The check for the existence of the file and the creation of the new file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the directory.

The `attrs` parameter is optional `file-attributes` to set atomically when creating the file. Each attribute is identified by its `name`. If more than one attribute of the same name is

included in the array then all but the last occurrence is ignored.

Parameters:

`path` - the path to the file to create

`attrs` - an optional list of file attributes to set atomically when creating the file

Returns:

the file

Throws:

`UnsupportedOperationException` - if the array contains an attribute that cannot be set atomically when creating the file

`FileAlreadyExistsException` - If a file of that name already exists *(optional specific exception)*

`IOException` - if an I/O error occurs or the parent directory does not exist

createDirectory

```
public static Path createDirectory(Path dir,
                                   FileAttribute<?>... attrs)
                            throws IOException
```

Creates a new directory. The check for the existence of the file and the creation of the directory if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the directory. The `createDirectories` method should be used where it is required to create all nonexistent parent directories first.

The `attrs` parameter is optional `file-attributes` to set atomically when creating the directory. Each attribute is identified by its `name`. If more than one attribute of the same name is included in the array then all but the last occurrence is ignored.

Parameters:

`dir` - the directory to create

`attrs` - an optional list of file attributes to set atomically when creating the directory

Returns:

the directory

Throws:

`UnsupportedOperationException` - if the array contains an attribute that cannot be set atomically when creating the directory

`FileAlreadyExistsException` - if a directory could not otherwise be created because a file of that name already exists *(optional specific exception)*

`IOException` - if an I/O error occurs or the parent directory does not exist

createDirectories

```
public static Path createDirectories(Path dir,
                                     FileAttribute<?>... attrs)
                              throws IOException
```

Creates a directory by creating all nonexistent parent directories first. Unlike the `createDirectory` method, an exception is not thrown if the directory could not be created because it already exists.

The `attrs` parameter is optional `file-attributes` to set atomically when creating the nonexistent directories. Each file attribute is identified by its `name`. If more than one attribute of the same name is included in the array then all but the last occurrence is ignored.

If this method fails, then it may do so after creating some, but not all, of the parent directories.

Parameters:

`dir` - the directory to create

`attrs` - an optional list of file attributes to set atomically when creating the directory

Returns:

the directory

Throws:

`UnsupportedOperationException` - if the array contains an attribute that cannot be set atomically when creating the directory

`FileAlreadyExistsException` - if `dir` exists but is not a directory *(optional specific exception)*

`IOException` - if an I/O error occurs

createTempFile

```
public static Path createTempFile(Path dir,
                                  String prefix,
                                  String suffix,
                                  FileAttribute<?>... attrs)
                           throws IOException
```

Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. The resulting `Path` is associated with the same `FileSystem` as the given directory.

The details as to how the name of the file is constructed is implementation dependent and therefore not specified. Where possible the `prefix` and `suffix` are used to construct

candidate names in the same manner as the File.createTempFile(String,String,File) method.

As with the File.createTempFile methods, this method is only part of a temporary-file facility. Where used as a *work file*, the resulting file may be opened using the DELETE_ON_CLOSE option so that the file is deleted when the appropriate close method is invoked. Alternatively, a shutdown-hook, or the File.deleteOnExit() mechanism may be used to delete the file automatically.

The attrs parameter is optional file-attributes to set atomically when creating the file. Each attribute is identified by its name. If more than one attribute of the same name is included in the array then all but the last occurrence is ignored. When no file attributes are specified, then the resulting file may have more restrictive access permissions than files created by the File.createTempFile(String,String,File) method.

Parameters:

dir - the path to directory in which to create the file

prefix - the prefix string to be used in generating the file's name; may be null

suffix - the suffix string to be used in generating the file's name; may be null, in which case ".tmp" is used

attrs - an optional list of file attributes to set atomically when creating the file

Returns:

the path to the newly created file that did not exist before this method was invoked

Throws:

IllegalArgumentException - if the prefix or suffix parameters cannot be used to generate a candidate file name

UnsupportedOperationException - if the array contains an attribute that cannot be set atomically when creating the directory

IOException - if an I/O error occurs or dir does not exist

createTempFile

```
public static Path createTempFile(String prefix,
                                  String suffix,
                                  FileAttribute<?>... attrs)
                           throws IOException
```

Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. The resulting Path is associated with the default FileSystem.

This method works in exactly the manner specified by the createTempFile(Path,String,String,FileAttribute[]) method for the case that the dir parameter is the temporary-file directory.

Parameters:

`prefix` - the prefix string to be used in generating the file's name; may be `null`

`suffix` - the suffix string to be used in generating the file's name; may be `null`, in which case "`.tmp`" is used

`attrs` - an optional list of file attributes to set atomically when creating the file

Returns:

the path to the newly created file that did not exist before this method was invoked

Throws:

`IllegalArgumentException` - if the prefix or suffix parameters cannot be used to generate a candidate file name

`UnsupportedOperationException` - if the array contains an attribute that cannot be set atomically when creating the directory

`IOException` - if an I/O error occurs or the temporary-file directory does not exist

createTempDirectory

```
public static Path createTempDirectory(Path dir,
                                       String prefix,
                                       FileAttribute<?>... attrs)
                                throws IOException
```

Creates a new directory in the specified directory, using the given prefix to generate its name. The resulting `Path` is associated with the same `FileSystem` as the given directory.

The details as to how the name of the directory is constructed is implementation dependent and therefore not specified. Where possible the `prefix` is used to construct candidate names.

As with the `createTempFile` methods, this method is only part of a temporary-file facility. A `shutdown-hook`, or the `File.deleteOnExit()` mechanism may be used to delete the directory automatically.

The `attrs` parameter is optional `file-attributes` to set atomically when creating the directory. Each attribute is identified by its `name`. If more than one attribute of the same name is included in the array then all but the last occurrence is ignored. When no file attributes are specified, then the resulting directory may have more restrictive access permissions than directories created by the createDirectory(Path, FileAttribute<?>...) method.

Parameters:

`dir` - the path to directory in which to create the directory

`prefix` - the prefix string to be used in generating the directory's name; may be `null`

`attrs` - an optional list of file attributes to set atomically when creating the directory

Returns:

the path to the newly created directory that did not exist before this method was invoked

Throws:

`IllegalArgumentException` - if the prefix cannot be used to generate a candidate directory name

`UnsupportedOperationException` - if the array contains an attribute that cannot be set atomically when creating the directory

`IOException` - if an I/O error occurs or `dir` does not exist

createTempDirectory

```
public static Path createTempDirectory(String prefix,
                                        FileAttribute<?>... attrs)
                                 throws IOException
```

Creates a new directory in the default temporary-file directory, using the given prefix to generate its name. The resulting `Path` is associated with the default `FileSystem`.

This method works in exactly the manner specified by `createTempDirectory(Path,String,FileAttribute[])` method for the case that the `dir` parameter is the temporary-file directory.

Parameters:

`prefix` - the prefix string to be used in generating the directory's name; may be `null`

`attrs` - an optional list of file attributes to set atomically when creating the directory

Returns:

the path to the newly created directory that did not exist before this method was invoked

Throws:

`IllegalArgumentException` - if the prefix cannot be used to generate a candidate directory name

`UnsupportedOperationException` - if the array contains an attribute that cannot be set atomically when creating the directory

`IOException` - if an I/O error occurs or the temporary-file directory does not exist

createSymbolicLink

```
public static Path createSymbolicLink(Path link,
                                       Path target,
                                       FileAttribute<?>... attrs)
                                throws IOException
```

Creates a symbolic link to a target *(optional operation)*.

The `target` parameter is the target of the link. It may be an `absolute` or relative path and may not exist. When the target is a relative path then file system operations on the resulting link are relative to the path of the link.

The `attrs` parameter is optional `attributes` to set atomically when creating the link. Each attribute is identified by its `name`. If more than one attribute of the same name is included in the array then all but the last occurrence is ignored.

Where symbolic links are supported, but the underlying `FileStore` does not support symbolic links, then this may fail with an `IOException`. Additionally, some operating systems may require that the Java virtual machine be started with implementation specific privileges to create symbolic links, in which case this method may throw `IOException`.

Parameters:

`link` - the path of the symbolic link to create

`target` - the target of the symbolic link

`attrs` - the array of attributes to set atomically when creating the symbolic link

Returns:

the path to the symbolic link

Throws:

`UnsupportedOperationException` - if the implementation does not support symbolic links or the array contains an attribute that cannot be set atomically when creating the symbolic link

`FileAlreadyExistsException` - if a file with the name already exists *(optional specific exception)*

`IOException` - if an I/O error occurs

createLink

```
public static Path createLink(Path link,
                              Path existing)
                       throws IOException
```

Creates a new link (directory entry) for an existing file *(optional operation)*.

The `link` parameter locates the directory entry to create. The `existing` parameter is the path to an existing file. This method creates a new directory entry for the file so that it can be accessed using `link` as the path. On some file systems this is known as creating a "hard link". If the `existing` parameter is the path to a symbolic link, then whether the new link is for the target of the symbolic link or for the symbolic link itself is platform dependent and therefore not specified. Whether the file attributes are maintained for the file or for each directory entry is file system specific and therefore not specified. Typically, a file system

requires that all links (directory entries) for a file be on the same file system. Furthermore, on some platforms, the Java virtual machine may require to be started with implementation specific privileges to create hard links or to create links to directories.

Parameters:

`link` - the link (directory entry) to create

`existing` - a path to an existing file

Returns:

the path to the link (directory entry)

Throws:

`UnsupportedOperationException` - if the implementation does not support adding an existing file to a directory

`FileAlreadyExistsException` - if the entry could not otherwise be created because a file of that name already exists *(optional specific exception)*

`IOException` - if an I/O error occurs

delete

```
public static void delete(Path path)
                  throws IOException
```

Deletes a file.

An implementation may require to examine the file to determine if the file is a directory. Consequently this method may not be atomic with respect to other file system operations. If the file is a symbolic link then the symbolic link itself, not the final target of the link, is deleted.

If the file is a directory then the directory must be empty. In some implementations a directory has entries for special files or links that are created when the directory is created. In such implementations a directory is considered empty when only the special entries exist. This method can be used with the `walkFileTree` method to delete a directory and all entries in the directory, or an entire *file-tree* where required.

On some operating systems it may not be possible to remove a file when it is open and in use by this Java virtual machine or other programs.

Parameters:

`path` - the path to the file to delete

Throws:

`NoSuchFileException` - if the file does not exist *(optional specific exception)*

`DirectoryNotEmptyException` - if the file is a directory and could not otherwise be deleted because the directory is not empty *(optional specific exception)*

IOException - if an I/O error occurs

deleteIfExists

```
public static boolean deleteIfExists(Path path)
                              throws IOException
```

Deletes a file if it exists.

As with the delete(Path) method, an implementation may need to examine the file to determine if the file is a directory. Consequently this method may not be atomic with respect to other file system operations. If the file is a symbolic link, then the symbolic link itself, not the final target of the link, is deleted.

If the file is a directory then the directory must be empty. In some implementations a directory has entries for special files or links that are created when the directory is created. In such implementations a directory is considered empty when only the special entries exist.

On some operating systems it may not be possible to remove a file when it is open and in use by this Java virtual machine or other programs.

Parameters:

path - the path to the file to delete

Returns:

true if the file was deleted by this method; false if the file could not be deleted because it did not exist

Throws:

DirectoryNotEmptyException - if the file is a directory and could not otherwise be deleted because the directory is not empty *(optional specific exception)*

IOException - if an I/O error occurs

copy

```
public static Path copy(Path source,
                        Path target,
                        CopyOption... options)
                 throws IOException
```

Copy a file to a target file.

This method copies a file to the target file with the options parameter specifying how the copy is performed. By default, the copy fails if the target file already exists or is a symbolic link, except if the source and target are the same file, in which case the method completes without copying the file. File attributes are not required to be copied to the target file. If symbolic links are supported, and the file is a symbolic link, then the final target of the link

is copied. If the file is a directory then an empty directory is created in the target location (entries in the directory are not copied). This method can be used with the `walkFileTree` method to copy a directory and all entries in the directory, or an entire *file-tree* where required.

The `options` parameter may include any of the following:

Option	Description
`REPLACE_EXISTING`	Replace an existing file. A non-empty directory cannot be replaced. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.
`COPY_ATTRIBUTES`	Attempts to copy the file attributes associated with this file to the target file. The exact file attributes that are copied is platform and file system dependent and therefore unspecified. Minimally, the `last-modified-time` is copied to the target file if supported by both the source and target file stores. Copying of file timestamps may result in precision loss.
`NOFOLLOW_LINKS`	Symbolic links are not followed. If the file is a symbolic link, then the symbolic link itself, not the target of the link, is copied. It is implementation specific if file attributes can be copied to the new link. In other words, the `COPY_ATTRIBUTES` option may be ignored when copying a symbolic link.

An implementation of this interface may support additional implementation specific options.

Copying a file is not an atomic operation. If an `IOException` is thrown, then it is possible that the target file is incomplete or some of its file attributes have not been copied from the source file. When the `REPLACE_EXISTING` option is specified and the target file exists, then the target file is replaced. The check for the existence of the file and the creation of the new file may not be atomic with respect to other file system activities.

Usage Example: Suppose we want to copy a file into a directory, giving it the same file name as the source file:

```
Path source = ...
Path newdir = ...
Files.copy(source, newdir.resolve(source.getFileName());
```

Parameters:

`source` - the path to the file to copy

`target` - the path to the target file (may be associated with a different provider to the source path)

`options` - options specifying how the copy should be done

Returns:

the path to the target file

Throws:

`UnsupportedOperationException` - if the array contains a copy option that is not supported

`FileAlreadyExistsException` - if the target file exists but cannot be replaced because the `REPLACE_EXISTING` option is not specified *(optional specific exception)*

`DirectoryNotEmptyException` - the `REPLACE_EXISTING` option is specified but the file cannot be replaced because it is a non-empty directory *(optional specific exception)*

`IOException` - if an I/O error occurs

move

```
public static Path move(Path source,
                        Path target,
                        CopyOption... options)
                 throws IOException
```

Move or rename a file to a target file.

By default, this method attempts to move the file to the target file, failing if the target file exists except if the source and target are the `same` file, in which case this method has no effect. If the file is a symbolic link then the symbolic link itself, not the target of the link, is moved. This method may be invoked to move an empty directory. In some implementations a directory has entries for special files or links that are created when the directory is created. In such implementations a directory is considered empty when only the special entries exist. When invoked to move a directory that is not empty then the directory is moved if it does not require moving the entries in the directory. For example, renaming a directory on the same `FileStore` will usually not require moving the entries in the directory. When moving a directory requires that its entries be moved then this method fails (by throwing an `IOException`). To move a *file tree* may involve copying rather than moving directories and this can be done using the `copy` method in conjunction with the `Files.walkFileTree` utility method.

The `options` parameter may include any of the following:

Option	Description
`REPLACE_EXISTING`	Replace an existing file. A non-empty directory cannot be replaced. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.
`ATOMIC_MOVE`	The move is performed as an atomic file system operation and all other options are ignored. If the target file exists then it is implementation specific if the existing file is replaced or this method fails by throwing an `IOException`. If the move cannot be performed as

Option	Description
	an atomic file system operation then `AtomicMoveNotSupportedException` is thrown. This can arise, for example, when the target location is on a different `FileStore` and would require that the file be copied, or target location is associated with a different provider to this object.

If the `ATOMIC_MOVE` option is not specified, then the check whether the target file exists and the actual move might not be atomic with respect to other filesystem activities.

An implementation of this interface may support additional implementation specific options.

Moving a file will copy the `last-modified-time` to the target file if supported by both source and target file stores. Copying of file timestamps may result in precision loss. An implementation may also attempt to copy other file attributes but is not required to fail if the file attributes cannot be copied. When the move is performed as a non-atomic operation, and an `IOException` is thrown, then the state of the files is not defined. The original file and the target file may both exist, the target file may be incomplete or some of its file attributes may not been copied from the original file.

Usage Examples: Suppose we want to rename a file to "newname", keeping the file in the same directory:

```
Path source = ...
Files.move(source, source.resolveSibling("newname"));
```

Alternatively, suppose we want to move a file to new directory, keeping the same file name, and replacing any existing file of that name in the directory:

```
Path source = ...
Path newdir = ...
Files.move(source, newdir.resolve(source.getFileName()), REPLACE_EXISTI
```

Parameters:

`source` - the path to the file to move

`target` - the path to the target file (may be associated with a different provider to the source path)

`options` - options specifying how the move should be done

Returns:

the path to the target file

Throws:

`UnsupportedOperationException` - if the array contains a copy option that is not supported

`FileAlreadyExistsException` - if the target file exists but cannot be replaced because the `REPLACE_EXISTING` option is *not* specified. It may also be thrown when the `REPLACE_EXISTING` option *is* specified, the move is not atomic, and the target file is created by some other entity at around the same time that this method is called

`DirectoryNotEmptyException` - the `REPLACE_EXISTING` option is specified but the file cannot be replaced because it is a non-empty directory, or the source is a non-empty directory containing entries that would be required to be moved *(optional specific exceptions)*

`AtomicMoveNotSupportedException` - if the options array contains the `ATOMIC_MOVE` option but the file cannot be moved as an atomic file system operation.

`IOException` - if an I/O error occurs

readSymbolicLink

```
public static Path readSymbolicLink(Path link)
                             throws IOException
```

Reads the target of a symbolic link *(optional operation)*.

If the file system supports symbolic links then this method is used to read the target of the link, failing if the file is not a symbolic link. The target of the link need not exist. The returned `Path` object will be associated with the same file system as `link`.

Parameters:

`link` - the path to the symbolic link

Returns:

a `Path` object representing the target of the link

Throws:

`UnsupportedOperationException` - if the implementation does not support symbolic links

`NotLinkException` - if the target could otherwise not be read because the file is not a symbolic link *(optional specific exception)*

`IOException` - if an I/O error occurs

getFileStore

```
public static FileStore getFileStore(Path path)
                              throws IOException
```

Returns the `FileStore` representing the file store where a file is located.

Once a reference to the `FileStore` is obtained it is implementation specific if operations on the returned `FileStore`, or `FileStoreAttributeView` objects obtained from it, continue to depend on the existence of the file. In particular the behavior is not defined for the case that the file is deleted or moved to a different file store.

Parameters:

`path` - the path to the file

Returns:

the file store where the file is stored

Throws:

`IOException` - if an I/O error occurs

isSameFile

```
public static boolean isSameFile(Path path,
                                 Path path2)
                          throws IOException
```

Tests if two paths locate the same file.

If both `Path` objects are `equal` then this method returns `true` without checking if the file exists. If the two `Path` objects are associated with different providers then this method returns `false`. Otherwise, this method checks if both `Path` objects locate the same file, and depending on the implementation, may require to open or access both files.

If the file system and files remain static, then this method implements an equivalence relation for non-null `Paths`.

- It is *reflexive*: for `Path f`, `isSameFile(f,f)` should return `true`.
- It is *symmetric*: for two `Paths f` and `g`, `isSameFile(f,g)` will equal `isSameFile(g,f)`.
- It is *transitive*: for three `Paths f`, `g`, and `h`, if `isSameFile(f,g)` returns `true` and `isSameFile(g,h)` returns `true`, then `isSameFile(f,h)` will return `true`.

Parameters:

`path` - one path to the file

`path2` - the other path

Returns:

`true` if, and only if, the two paths locate the same file

Throws:

`IOException` - if an I/O error occurs

See Also:

`BasicFileAttributes.fileKey()`

mismatch

```
public static long mismatch(Path path,
                            Path path2)
                     throws IOException
```

Finds and returns the position of the first mismatched byte in the content of two files, or `-1L` if there is no mismatch. The position will be in the inclusive range of `0L` up to the size (in bytes) of the smaller file.

Two files are considered to match if they satisfy one of the following conditions:

- The two paths locate the same file, even if two equal paths locate a file that does not exist, or
- The two files are the same size, and every byte in the first file is identical to the corresponding byte in the second file.

Otherwise there is a mismatch between the two files and the value returned by this method is:

- The position of the first mismatched byte, or
- The size of the smaller file (in bytes) when the files are of different sizes and every byte of the smaller file is identical to the corresponding byte of the larger file.

This method may not be atomic with respect to other file system operations. This method is always *reflexive* (for `Path f`, `mismatch(f,f)` returns `-1L`). If the file system and files remain static, then this method is *symmetric* (for two `Paths f` and `g`, `mismatch(f,g)` will return the same value as `mismatch(g,f)`).

If both `Path` objects are equal, then this method returns `true` without checking if the file exists.

Parameters:

`path` - the path to the first file

`path2` - the path to the second file

Returns:

the position of the first mismatch or `-1L` if no mismatch

Throws:

`IOException` - if an I/O error occurs

Since:

12

isHidden

```
public static boolean isHidden(Path path)
                        throws IOException
```

Tells whether or not a file is considered *hidden*.

API Note:

The exact definition of hidden is platform or provider dependent. On UNIX for example a file is considered to be hidden if its name begins with a period character ('.'). On Windows a file is considered hidden if the DOS `hidden` attribute is set.

Depending on the implementation this method may require to access the file system to determine if the file is considered hidden.

Parameters:

`path` - the path to the file to test

Returns:

`true` if the file is considered hidden

Throws:

`IOException` - if an I/O error occurs

probeContentType

```
public static String probeContentType(Path path)
                               throws IOException
```

Probes the content type of a file.

This method uses the installed `FileTypeDetector` implementations to probe the given file to determine its content type. Each file type detector's `probeContentType` is invoked, in turn, to probe the file type. If the file is recognized then the content type is returned. If the file is not recognized by any of the installed file type detectors then a system-default file type detector is invoked to guess the content type.

A given invocation of the Java virtual machine maintains a system-wide list of file type detectors. Installed file type detectors are loaded using the service-provider loading facility defined by the `ServiceLoader` class. Installed file type detectors are loaded using the system class loader. If the system class loader cannot be found then the platform class loader is used. File type detectors are typically installed by placing them in a JAR file on the application class path, the JAR file contains a provider-configuration file named `java.nio.file.spi.FileTypeDetector` in the resource directory `META-INF/services`, and the file lists one or more fully-qualified names of concrete subclass of `FileTypeDetector` that have a zero argument constructor. If the process of locating or instantiating the installed file type detectors fails then an unspecified error is thrown. The ordering that installed providers are located is implementation specific.

The return value of this method is the string form of the value of a Multipurpose Internet Mail Extension (MIME) content type as defined by *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* . The string is guaranteed to be parsable according to the grammar in the RFC.

Parameters:

`path` - the path to the file to probe

Returns:

The content type of the file, or `null` if the content type cannot be determined

Throws:

`IOException` - if an I/O error occurs

External Specifications

RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies

getFileAttributeView

```
public static <V extends FileAttributeView> V getFileAttributeView
(Path path,
 Class<V> type,
 LinkOption... options)
```

Returns a file attribute view of a given type.

A file attribute view provides a read-only or updatable view of a set of file attributes. This method is intended to be used where the file attribute view defines type-safe methods to read or update the file attributes. The `type` parameter is the type of the attribute view required and the method returns an instance of that type if supported. The `BasicFileAttributeView` type supports access to the basic attributes of a file. Invoking this method to select a file attribute view of that type will always return an instance of that class.

The `options` array may be used to indicate how symbolic links are handled by the resulting file attribute view for the case that the file is a symbolic link. By default, symbolic links are followed. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed. This option is ignored by implementations that do not support symbolic links.

Usage Example: Suppose we want read or set a file's ACL, if supported:

```
Path path = ...
AclFileAttributeView view = Files.getFileAttributeView(path, AclFileAtt
if (view != null) {
    List<AclEntry> acl = view.getAcl();
```

```
        :
    }
```

Type Parameters:

V - The `FileAttributeView` type

Parameters:

`path` - the path to the file

`type` - the `Class` object corresponding to the file attribute view

`options` - options indicating how symbolic links are handled

Returns:

a file attribute view of the specified type, or `null` if the attribute view type is not available

readAttributes

```
public static <A extends BasicFileAttributes> A readAttributes
(Path path,
 Class<A> type,
 LinkOption... options)
                                                    throws IOException
```

Reads a file's attributes as a bulk operation.

The `type` parameter is the type of the attributes required and this method returns an instance of that type if supported. All implementations support a basic set of file attributes and so invoking this method with a `type` parameter of `BasicFileAttributes.class` will not throw `UnsupportedOperationException`.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is read. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

It is implementation specific if all file attributes are read as an atomic operation with respect to other file system operations.

Usage Example: Suppose we want to read a file's attributes in bulk:

```
Path path = ...
BasicFileAttributes attrs = Files.readAttributes(path, BasicFileAttribu
```

Alternatively, suppose we want to read file's POSIX attributes without following symbolic links:

```
PosixFileAttributes attrs =
    Files.readAttributes(path, PosixFileAttributes.class, NOFOLLOW_LINK
```

Type Parameters:

`A` - The `BasicFileAttributes` type

Parameters:

`path` - the path to the file

`type` - the `Class` of the file attributes required to read

`options` - options indicating how symbolic links are handled

Returns:

the file attributes

Throws:

`UnsupportedOperationException` - if an attributes of the given type are not supported

`IOException` - if an I/O error occurs

setAttribute

```
public static Path setAttribute(Path path,
                                String attribute,
                                Object value,
                                LinkOption... options)
                         throws IOException
```

Sets the value of a file attribute.

The `attribute` parameter identifies the attribute to be set and takes the form:

[*view-name*:]*attribute-name*

where square brackets [...] delineate an optional component and the character `':'` stands for itself.

view-name is the `name` of a `FileAttributeView` that identifies a set of file attributes. If not specified then it defaults to `"basic"`, the name of the file attribute view that identifies the basic set of file attributes common to many file systems. *attribute-name* is the name of the attribute within the set.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of

the final target of the link is set. If the option NOFOLLOW_LINKS is present then symbolic links are not followed.

Usage Example: Suppose we want to set the DOS "hidden" attribute:

```
Path path = ...
Files.setAttribute(path, "dos:hidden", true);
```

Parameters:

path - the path to the file

attribute - the attribute to set

value - the attribute value

options - options indicating how symbolic links are handled

Returns:

the given path

Throws:

UnsupportedOperationException - if the attribute view is not available

IllegalArgumentException - if the attribute name is not specified, or is not recognized, or the attribute value is of the correct type but has an inappropriate value

ClassCastException - if the attribute value is not of the expected type or is a collection containing elements that are not of the expected type

IOException - if an I/O error occurs

getAttribute

```
public static Object getAttribute(Path path,
                                  String attribute,
                                  LinkOption... options)
                           throws IOException
```

Reads the value of a file attribute.

The attribute parameter identifies the attribute to be read and takes the form:

[*view-name*:]*attribute-name*

where square brackets [...] delineate an optional component and the character ':' stands for itself.

view-name is the name of a FileAttributeView that identifies a set of file attributes. If not specified then it defaults to "basic", the name of the file attribute view that identifies the

basic set of file attributes common to many file systems. *attribute-name* is the name of the attribute.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is read. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Usage Example: Suppose we require the user ID of the file owner on a system that supports a "`unix`" view:

```
Path path = ...
int uid = (Integer)Files.getAttribute(path, "unix:uid");
```

Parameters:

`path` - the path to the file

`attribute` - the attribute to read

`options` - options indicating how symbolic links are handled

Returns:

the attribute value

Throws:

`UnsupportedOperationException` - if the attribute view is not available

`IllegalArgumentException` - if the attribute name is not specified or is not recognized

`IOException` - if an I/O error occurs

readAttributes

```
public static Map<String,Object> readAttributes(Path path,
                                                 String attributes,
                                                 LinkOption... options)
                                          throws IOException
```

Reads a set of file attributes as a bulk operation.

The `attributes` parameter identifies the attributes to be read and takes the form:

[*view-name*:]*attribute-list*

where square brackets [...] delineate an optional component and the character `':'` stands for itself.

view-name is the `name` of a `FileAttributeView` that identifies a set of file attributes. If not specified then it defaults to `"basic"`, the name of the file attribute view that identifies the basic set of file attributes common to many file systems.

The *attribute-list* component is a comma separated list of one or more names of attributes to read. If the list contains the value "*" then all attributes are read. Attributes that are not supported are ignored and will not be present in the returned map. It is implementation specific if all attributes are read as an atomic operation with respect to other file system operations.

The following examples demonstrate possible values for the `attributes` parameter:

Example	Description
"*"	Read all basic-file-attributes.
"size,lastModifiedTime,lastAccessTime"	Reads the file size, last modified time, and last access time attributes.
"posix:*"	Read all POSIX-file-attributes.
"posix:permissions,owner,size"	Reads the POSIX file permissions, owner, and file size.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is read. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Parameters:

`path` - the path to the file

`attributes` - the attributes to read

`options` - options indicating how symbolic links are handled

Returns:

a map of the attributes returned; The map's keys are the attribute names, its values are the attribute values

Throws:

`UnsupportedOperationException` - if the attribute view is not available

`IllegalArgumentException` - if no attributes are specified or an unrecognized attribute is specified

`IOException` - if an I/O error occurs

getPosixFilePermissions

```
public static Set<PosixFilePermission> getPosixFilePermissions
(Path path,
 LinkOption... options)
                                                    throws IOException
```

Returns a file's POSIX file permissions.

The `path` parameter is associated with a `FileSystem` that supports the `PosixFileAttributeView`. This attribute view provides access to file attributes commonly associated with files on file systems used by operating systems that implement the Portable Operating System Interface (POSIX) family of standards.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is read. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Parameters:

`path` - the path to the file

`options` - options indicating how symbolic links are handled

Returns:

the file permissions

Throws:

`UnsupportedOperationException` - if the associated file system does not support the `PosixFileAttributeView`

`IOException` - if an I/O error occurs

setPosixFilePermissions

```
public static Path setPosixFilePermissions(Path path,
                                           Set<PosixFilePermission> perms)
                                    throws IOException
```

Sets a file's POSIX permissions.

The `path` parameter is associated with a `FileSystem` that supports the `PosixFileAttributeView`. This attribute view provides access to file attributes commonly associated with files on file systems used by operating systems that implement the Portable Operating System Interface (POSIX) family of standards.

Parameters:

`path` - The path to the file

`perms` - The new set of permissions

Returns:

The given path

Throws:

`UnsupportedOperationException` - if the associated file system does not support the `PosixFileAttributeView`

ClassCastException - if the sets contains elements that are not of type PosixFilePermission

IOException - if an I/O error occurs

getOwner

```
public static UserPrincipal getOwner(Path path,
                                      LinkOption... options)
                               throws IOException
```

Returns the owner of a file.

The path parameter is associated with a file system that supports FileOwnerAttributeView. This file attribute view provides access to a file attribute that is the owner of the file.

Parameters:

path - The path to the file

options - options indicating how symbolic links are handled

Returns:

A user principal representing the owner of the file

Throws:

UnsupportedOperationException - if the associated file system does not support the FileOwnerAttributeView

IOException - if an I/O error occurs

setOwner

```
public static Path setOwner(Path path,
                            UserPrincipal owner)
                     throws IOException
```

Updates the file owner.

The path parameter is associated with a file system that supports FileOwnerAttributeView. This file attribute view provides access to a file attribute that is the owner of the file.

Usage Example: Suppose we want to make "joe" the owner of a file:

```
Path path = ...
UserPrincipalLookupService lookupService =
    provider(path).getUserPrincipalLookupService();
```

```
UserPrincipal joe = lookupService.lookupPrincipalByName("joe");
Files.setOwner(path, joe);
```

Parameters:

`path` - The path to the file

`owner` - The new file owner

Returns:

The given path

Throws:

`UnsupportedOperationException` - if the associated file system does not support the `FileOwnerAttributeView`

`IOException` - if an I/O error occurs

See Also:

`FileSystem.getUserPrincipalLookupService()`, `UserPrincipalLookupService`

isSymbolicLink

```
public static boolean isSymbolicLink(Path path)
```

Tests whether a file is a symbolic link.

Where it is required to distinguish an I/O exception from the case that the file is not a symbolic link then the file attributes can be read with the `readAttributes` method and the file type tested with the `BasicFileAttributes.isSymbolicLink()` method.

Parameters:

`path` - The path to the file

Returns:

`true` if the file is a symbolic link; `false` if the file does not exist, is not a symbolic link, or it cannot be determined if the file is a symbolic link or not.

isDirectory

```
public static boolean isDirectory(Path path,
                                  LinkOption... options)
```

Tests whether a file is a directory.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is read. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Where it is required to distinguish an I/O exception from the case that the file is not a directory then the file attributes can be read with the `readAttributes` method and the file type tested with the `BasicFileAttributes.isDirectory()` method.

Parameters:

`path` - the path to the file to test

`options` - options indicating how symbolic links are handled

Returns:

`true` if the file is a directory; `false` if the file does not exist, is not a directory, or it cannot be determined if the file is a directory or not.

isRegularFile

```
public static boolean isRegularFile(Path path,
                                    LinkOption... options)
```

Tests whether a file is a regular file with opaque content.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of the final target of the link is read. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Where it is required to distinguish an I/O exception from the case that the file is not a regular file then the file attributes can be read with the `readAttributes` method and the file type tested with the `BasicFileAttributes.isRegularFile()` method.

Parameters:

`path` - the path to the file

`options` - options indicating how symbolic links are handled

Returns:

`true` if the file is a regular file; `false` if the file does not exist, is not a regular file, or it cannot be determined if the file is a regular file or not.

getLastModifiedTime

```
public static FileTime getLastModifiedTime(Path path,
                                           LinkOption... options)
                                    throws IOException
```

Returns a file's last modified time.

The `options` array may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed and the file attribute of

the final target of the link is read. If the option NOFOLLOW_LINKS is present then symbolic links are not followed.

Parameters:

path - the path to the file

options - options indicating how symbolic links are handled

Returns:

a FileTime representing the time the file was last modified, or an implementation specific default when a time stamp to indicate the time of last modification is not supported by the file system

Throws:

IOException - if an I/O error occurs

See Also:

BasicFileAttributes.lastModifiedTime()

setLastModifiedTime

```
public static Path setLastModifiedTime(Path path,
                                       FileTime time)
                                throws IOException
```

Updates a file's last modified time attribute. The file time is converted to the epoch and precision supported by the file system. Converting from finer to coarser granularities result in precision loss. The behavior of this method when attempting to set the last modified time when it is not supported by the file system or is outside the range supported by the underlying file store is not defined. It may or not fail by throwing an IOException.

Usage Example: Suppose we want to set the last modified time to the current time:

```
Path path = ...
FileTime now = FileTime.fromMillis(System.currentTimeMillis());
Files.setLastModifiedTime(path, now);
```

Parameters:

path - the path to the file

time - the new last modified time

Returns:

the given path

Throws:

IOException - if an I/O error occurs

See Also:

BasicFileAttributeView.setTimes(FileTime, FileTime, FileTime)

size

```
public static long size(Path path)
                throws IOException
```

Returns the size of a file (in bytes). The size may differ from the actual size on the file system due to compression, support for sparse files, or other reasons. The size of files that are not regular files is implementation specific and therefore unspecified.

Parameters:

path - the path to the file

Returns:

the file size, in bytes

Throws:

IOException - if an I/O error occurs

See Also:

BasicFileAttributes.size()

exists

```
public static boolean exists(Path path,
                             LinkOption... options)
```

Tests whether a file exists.

The options parameter may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed. If the option NOFOLLOW_LINKS is present then symbolic links are not followed.

Note that the result of this method is immediately outdated. If this method indicates the file exists then there is no guarantee that a subsequent access will succeed. Care should be taken when using this method in security sensitive applications.

Parameters:

path - the path to the file to test

options - options indicating how symbolic links are handled

Returns:

true if the file exists; false if the file does not exist or its existence cannot be determined.

See Also:

notExists(Path, LinkOption...),
FileSystemProvider.checkAccess(Path, AccessMode...)

notExists

```
public static boolean notExists(Path path,
                                LinkOption... options)
```

Tests whether the file located by this path does not exist. This method is intended for cases where it is required to take action when it can be confirmed that a file does not exist.

The `options` parameter may be used to indicate how symbolic links are handled for the case that the file is a symbolic link. By default, symbolic links are followed. If the option `NOFOLLOW_LINKS` is present then symbolic links are not followed.

Note that this method is not the complement of the `exists` method. Where it is not possible to determine if a file exists or not then both methods return `false`. As with the `exists` method, the result of this method is immediately outdated. If this method indicates the file does exist then there is no guarantee that a subsequent attempt to create the file will succeed. Care should be taken when using this method in security sensitive applications.

Parameters:

`path` - the path to the file to test

`options` - options indicating how symbolic links are handled

Returns:

`true` if the file does not exist; `false` if the file exists or its existence cannot be determined

isReadable

```
public static boolean isReadable(Path path)
```

Tests whether a file is readable. This method checks that a file exists and that this Java virtual machine has appropriate privileges that would allow it open the file for reading. Depending on the implementation, this method may require to read file permissions, access control lists, or other file attributes in order to check the effective access to the file. Consequently, this method may not be atomic with respect to other file system operations.

Note that the result of this method is immediately outdated, there is no guarantee that a subsequent attempt to open the file for reading will succeed (or even that it will access the same file). Care should be taken when using this method in security sensitive applications.

Parameters:

`path` - the path to the file to check

Returns:

`true` if the file exists and is readable; `false` if the file does not exist, read access would be denied because the Java virtual machine has insufficient privileges, or access cannot be determined

isWritable

```
public static boolean isWritable(Path path)
```

Tests whether a file is writable. This method checks that a file exists and that this Java virtual machine has appropriate privileges that would allow it open the file for writing. Depending on the implementation, this method may require to read file permissions, access control lists, or other file attributes in order to check the effective access to the file. Consequently, this method may not be atomic with respect to other file system operations.

Note that result of this method is immediately outdated, there is no guarantee that a subsequent attempt to open the file for writing will succeed (or even that it will access the same file). Care should be taken when using this method in security sensitive applications.

Parameters:

`path` - the path to the file to check

Returns:

`true` if the file exists and is writable; `false` if the file does not exist, write access would be denied because the Java virtual machine has insufficient privileges, or access cannot be determined

isExecutable

```
public static boolean isExecutable(Path path)
```

Tests whether a file is executable. This method checks that a file exists and that this Java virtual machine has appropriate privileges to `execute` the file. The semantics may differ when checking access to a directory. For example, on UNIX systems, checking for execute access checks that the Java virtual machine has permission to search the directory in order to access file or subdirectories.

Depending on the implementation, this method may require to read file permissions, access control lists, or other file attributes in order to check the effective access to the file. Consequently, this method may not be atomic with respect to other file system operations.

Note that the result of this method is immediately outdated, there is no guarantee that a subsequent attempt to execute the file will succeed (or even that it will access the same file). Care should be taken when using this method in security sensitive applications.

Parameters:

`path` - the path to the file to check

Returns:

`true` if the file exists and is executable; `false` if the file does not exist, execute access would be denied because the Java virtual machine has insufficient privileges, or access cannot be determined

walkFileTree

```
public static Path walkFileTree(Path start,
                                Set<FileVisitOption> options,
                                int maxDepth,
                                FileVisitor<? super Path> visitor)
                         throws IOException
```

Walks a file tree.

This method walks a file tree rooted at a given starting file. The file tree traversal is *depth-first* with the given `FileVisitor` invoked for each file encountered. File tree traversal completes when all accessible files in the tree have been visited, or a visit method returns a result of `TERMINATE`. Where a visit method terminates due an `IOException`, an uncaught error, or runtime exception, then the traversal is terminated and the error or exception is propagated to the caller of this method.

For each file encountered this method attempts to read its `BasicFileAttributes`. If the file is not a directory then the `visitFile` method is invoked with the file attributes. If the file attributes cannot be read, due to an I/O exception, then the `visitFileFailed` method is invoked with the I/O exception.

Where the file is a directory, and the directory could not be opened, then the `visitFileFailed` method is invoked with the I/O exception, after which, the file tree walk continues, by default, at the next *sibling* of the directory.

Where the directory is opened successfully, then the entries in the directory, and their *descendants* are visited. When all entries have been visited, or an I/O error occurs during iteration of the directory, then the directory is closed and the visitor's `postVisitDirectory` method is invoked. The file tree walk then continues, by default, at the next *sibling* of the directory.

By default, symbolic links are not automatically followed by this method. If the `options` parameter contains the `FOLLOW_LINKS` option then symbolic links are followed. When following links, and the attributes of the target cannot be read, then this method attempts to get the `BasicFileAttributes` of the link. If they can be read then the `visitFile` method is invoked with the attributes of the link (otherwise the `visitFileFailed` method is invoked as specified above).

If the `options` parameter contains the `FOLLOW_LINKS` option then this method keeps track of directories visited so that cycles can be detected. A cycle arises when there is an entry in a directory that is an ancestor of the directory. Cycle detection is done by recording the `file-key` of directories, or if file keys are not available, by invoking the `isSameFile` method to test if a directory is the same file as an ancestor. When a cycle is detected it is treated as an I/O error, and the `visitFileFailed` method is invoked with an instance of `FileSystemLoopException`.

The `maxDepth` parameter is the maximum number of levels of directories to visit. A value of `0` means that only the starting file is visited. A value of `MAX_VALUE` may be used to indicate that all levels should be visited. The `visitFile` method is invoked for all files, including directories, encountered at `maxDepth`, unless the basic file attributes cannot be read, in which case the `visitFileFailed` method is invoked.

If a visitor returns a result of `null` then `NullPointerException` is thrown.

Parameters:

`start` - the starting file

`options` - options to configure the traversal

`maxDepth` - the maximum number of directory levels to visit

`visitor` - the file visitor to invoke for each file

Returns:

the starting file

Throws:

`IllegalArgumentException` - if the `maxDepth` parameter is negative

`IOException` - if an I/O error is thrown by a visitor method

walkFileTree

```
public static Path walkFileTree(Path start,
                                FileVisitor<? super Path> visitor)
                         throws IOException
```

Walks a file tree.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.walkFileTree(start, EnumSet.noneOf(FileVisitOption.class),
Integer.MAX_VALUE, visitor)
```

In other words, it does not follow symbolic links, and visits all levels of the file tree.

Parameters:

`start` - the starting file

`visitor` - the file visitor to invoke for each file

Returns:

the starting file

Throws:

`IOException` - if an I/O error is thrown by a visitor method

newBufferedReader

```
public static BufferedReader newBufferedReader(Path path,
                                               Charset cs)
                                        throws IOException
```

Opens a file for reading, returning a `BufferedReader` that may be used to read text from the file in an efficient manner. Bytes from the file are decoded into characters using the specified charset. Reading commences at the beginning of the file.

The `Reader` methods that read from the file throw `IOException` if a malformed or unmappable byte sequence is read.

Parameters:

`path` - the path to the file

`cs` - the charset to use for decoding

Returns:

a new buffered reader, with default buffer size, to read text from the file

Throws:

`IOException` - if an I/O error occurs opening the file

See Also:

`readAllLines(Path, Charset)`

newBufferedReader

```
public static BufferedReader newBufferedReader(Path path)
                                        throws IOException
```

Opens a file for reading, returning a `BufferedReader` to read text from the file in an efficient manner. Bytes from the file are decoded into characters using the `UTF-8 charset`.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.newBufferedReader(path, StandardCharsets.UTF_8)
```

Parameters:

`path` - the path to the file

Returns:

a new buffered reader, with default buffer size, to read text from the file

Throws:

`IOException` - if an I/O error occurs opening the file

Since:

1.8

newBufferedWriter

```
public static BufferedWriter newBufferedWriter(Path path,
                                               Charset cs,
                                               OpenOption... options)
                                        throws IOException
```

Opens or creates a file for writing, returning a `BufferedWriter` that may be used to write text to the file in an efficient manner. The `options` parameter specifies how the file is created or opened. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of `0` if it exists.

The `Writer` methods to write text throw `IOException` if the text cannot be encoded using the specified charset. Due to buffering, an `IOException` caused by an encoding error (unmappable-character or malformed-input) may be thrown when writing, flushing, or closing the buffered writer.

Parameters:

`path` - the path to the file

`cs` - the charset to use for encoding

`options` - options specifying how the file is opened

Returns:

a new buffered writer, with default buffer size, to write text to the file

Throws:

`IllegalArgumentException` - if `options` contains an invalid combination of options

`IOException` - if an I/O error occurs opening or creating the file

`UnsupportedOperationException` - if an unsupported option is specified

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified *(optional specific exception)*

See Also:

`write(Path,Iterable,Charset,OpenOption[])`

newBufferedWriter

```
public static BufferedWriter newBufferedWriter(Path path,
                                               OpenOption... options)
                                        throws IOException
```

Opens or creates a file for writing, returning a `BufferedWriter` to write text to the file in an efficient manner. The text is encoded into bytes for writing using the `UTF-8 charset`.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.newBufferedWriter(path, StandardCharsets.UTF_8, options)
```

Parameters:

`path` - the path to the file

`options` - options specifying how the file is opened

Returns:

a new buffered writer, with default buffer size, to write text to the file

Throws:

`IllegalArgumentException` - if `options` contains an invalid combination of options

`IOException` - if an I/O error occurs opening or creating the file

`UnsupportedOperationException` - if an unsupported option is specified

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified *(optional specific exception)*

Since:

1.8

copy

```
public static long copy(InputStream in,
                        Path target,
                        CopyOption... options)
                 throws IOException
```

Copies all bytes from an input stream to a file. On return, the input stream will be at end of stream.

By default, the copy fails if the target file already exists or is a symbolic link. If the `REPLACE_EXISTING` option is specified, and the target file already exists, then it is replaced if it is not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link is replaced. In this release, the `REPLACE_EXISTING` option is the only option required to be supported by this method. Additional options may be supported in future releases.

If an I/O error occurs reading from the input stream or writing to the file, then it may do so after the target file has been created and after some bytes have been read or written. Consequently the input stream may not be at end of stream and may be in an inconsistent state. It is strongly recommended that the input stream be promptly closed if an I/O error occurs.

This method may block indefinitely reading from the input stream (or writing to the file). The behavior for the case that the input stream is *asynchronously closed* or the thread

interrupted during the copy is highly input stream and file system provider specific and therefore not specified.

Usage example: Suppose we want to capture a web page and save it to a file:

```
Path path = ...
URI u = URI.create("http://www.example.com/");
try (InputStream in = u.toURL().openStream()) {
    Files.copy(in, path);
}
```

Parameters:

`in` - the input stream to read from

`target` - the path to the file

`options` - options specifying how the copy should be done

Returns:

the number of bytes read or written

Throws:

`IOException` - if an I/O error occurs when reading or writing

`FileAlreadyExistsException` - if the target file exists but cannot be replaced because the `REPLACE_EXISTING` option is not specified *(optional specific exception)*

`DirectoryNotEmptyException` - the `REPLACE_EXISTING` option is specified but the file cannot be replaced because it is a non-empty directory *(optional specific exception)*

`UnsupportedOperationException` - if `options` contains a copy option that is not supported

copy

```
public static long copy(Path source,
                        OutputStream out)
                 throws IOException
```

Copies all bytes from a file to an output stream.

If an I/O error occurs reading from the file or writing to the output stream, then it may do so after some bytes have been read or written. Consequently the output stream may be in an inconsistent state. It is strongly recommended that the output stream be promptly closed if an I/O error occurs.

This method may block indefinitely writing to the output stream (or reading from the file). The behavior for the case that the output stream is *asynchronously closed* or the thread

interrupted during the copy is highly output stream and file system provider specific and therefore not specified.

Note that if the given output stream is `Flushable` then its `flush` method may need to invoked after this method completes so as to flush any buffered output.

Parameters:

`source` - the path to the file

`out` - the output stream to write to

Returns:

the number of bytes read or written

Throws:

`IOException` - if an I/O error occurs when reading or writing

readAllBytes

```
public static byte[] readAllBytes(Path path)
                           throws IOException
```

Reads all the bytes from a file. The method ensures that the file is closed when all bytes have been read or an I/O error, or other runtime exception, is thrown.

Note that this method is intended for simple cases where it is convenient to read all bytes into a byte array. It is not intended for reading in large files.

Parameters:

`path` - the path to the file

Returns:

a byte array containing the bytes read from the file

Throws:

`IOException` - if an I/O error occurs reading from the stream

`OutOfMemoryError` - if an array of the required size cannot be allocated, for example the file is larger that `2GB`

readString

```
public static String readString(Path path)
                         throws IOException
```

Reads all content from a file into a string, decoding from bytes to characters using the `UTF-8 charset`. The method ensures that the file is closed when all content have been read or an I/O error, or other runtime exception, is thrown.

This method is equivalent to: `readString(path, StandardCharsets.UTF_8)`.

Parameters:

`path` - the path to the file

Returns:

a String containing the content read from the file

Throws:

`IOException` - if an I/O error occurs reading from the file or a malformed or unmappable byte sequence is read

`OutOfMemoryError` - if the file is extremely large, for example larger than `2GB`

Since:

11

readString

```
public static String readString(Path path,
                                Charset cs)
                         throws IOException
```

Reads all characters from a file into a string, decoding from bytes to characters using the specified charset. The method ensures that the file is closed when all content have been read or an I/O error, or other runtime exception, is thrown.

This method reads all content including the line separators in the middle and/or at the end. The resulting string will contain line separators as they appear in the file.

API Note:

This method is intended for simple cases where it is appropriate and convenient to read the content of a file into a String. It is not intended for reading very large files.

Parameters:

`path` - the path to the file

`cs` - the charset to use for decoding

Returns:

a String containing the content read from the file

Throws:

`IOException` - if an I/O error occurs reading from the file or a malformed or unmappable byte sequence is read

`OutOfMemoryError` - if the file is extremely large, for example larger than `2GB`

Since:

11

readAllLines

```
public static List<String> readAllLines(Path path,
                                        Charset cs)
                                 throws IOException
```

Read all lines from a file. This method ensures that the file is closed when all bytes have been read or an I/O error, or other runtime exception, is thrown. Bytes from the file are decoded into characters using the specified charset.

This method recognizes the following as line terminators:

- \u000D followed by \u000A, CARRIAGE RETURN followed by LINE FEED
- \u000A, LINE FEED
- \u000D, CARRIAGE RETURN

Additional Unicode line terminators may be recognized in future releases.

Note that this method is intended for simple cases where it is convenient to read all lines in a single operation. It is not intended for reading in large files.

Parameters:

path - the path to the file

cs - the charset to use for decoding

Returns:

the lines from the file as a List; whether the List is modifiable or not is implementation dependent and therefore not specified

Throws:

IOException - if an I/O error occurs reading from the file or a malformed or unmappable byte sequence is read

See Also:

newBufferedReader(Path, Charset)

readAllLines

```
public static List<String> readAllLines(Path path)
                                 throws IOException
```

Read all lines from a file. Bytes from the file are decoded into characters using the UTF-8 charset.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.readAllLines(path, StandardCharsets.UTF_8)
```

Parameters:

path - the path to the file

Returns:

the lines from the file as a `List`; whether the `List` is modifiable or not is implementation dependent and therefore not specified

Throws:

`IOException` - if an I/O error occurs reading from the file or a malformed or unmappable byte sequence is read

Since:

1.8

write

```
public static Path write(Path path,
                         byte[] bytes,
                         OpenOption... options)
                  throws IOException
```

Writes bytes to a file. The `options` parameter specifies how the file is created or opened. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of `0`. All bytes in the byte array are written to the file. The method ensures that the file is closed when all bytes have been written (or an I/O error or other runtime exception is thrown). If an I/O error occurs then it may do so after the file has been created or truncated, or after some bytes have been written to the file.

Usage example: By default the method creates a new file or overwrites an existing file. Suppose you instead want to append bytes to an existing file:

```
Path path = ...
byte[] bytes = ...
Files.write(path, bytes, StandardOpenOption.APPEND);
```

Parameters:

`path` - the path to the file

`bytes` - the byte array with the bytes to write

`options` - options specifying how the file is opened

Returns:

the path

Throws:

`IllegalArgumentException` - if `options` contains an invalid combination of options

`IOException` - if an I/O error occurs writing to or creating the file

`UnsupportedOperationException` - if an unsupported option is specified

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified *(optional specific exception)*

write

```
public static Path write(Path path,
                         Iterable<? extends CharSequence> lines,
                         Charset cs,
                         OpenOption... options)
                  throws IOException
```

Write lines of text to a file. Each line is a char sequence and is written to the file in sequence with each line terminated by the platform's line separator, as defined by the system property `line.separator`. Characters are encoded into bytes using the specified charset.

The `options` parameter specifies how the file is created or opened. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of `0`. The method ensures that the file is closed when all lines have been written (or an I/O error or other runtime exception is thrown). If an I/O error occurs then it may do so after the file has been created or truncated, or after some bytes have been written to the file.

Parameters:

`path` - the path to the file

`lines` - an object to iterate over the char sequences

`cs` - the charset to use for encoding

`options` - options specifying how the file is opened

Returns:

the path

Throws:

`IllegalArgumentException` - if `options` contains an invalid combination of options

`IOException` - if an I/O error occurs writing to or creating the file, or the text cannot be encoded using the specified charset

`UnsupportedOperationException` - if an unsupported option is specified

`FileAlreadyExistsException` - If a file of that name already exists and the `CREATE_NEW` option is specified *(optional specific exception)*

write

```
public static Path write(Path path,
                         Iterable<? extends CharSequence> lines,
                         OpenOption... options)
                  throws IOException
```

Write lines of text to a file. Characters are encoded into bytes using the UTF-8 charset.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.write(path, lines, StandardCharsets.UTF_8, options)
```

Parameters:

path - the path to the file

lines - an object to iterate over the char sequences

options - options specifying how the file is opened

Returns:

the path

Throws:

IllegalArgumentException - if options contains an invalid combination of options

IOException - if an I/O error occurs writing to or creating the file, or the text cannot be encoded as UTF-8

UnsupportedOperationException - if an unsupported option is specified

Since:

1.8

writeString

```
public static Path writeString(Path path,
                               CharSequence csq,
                               OpenOption... options)
                        throws IOException
```

Write a CharSequence to a file. Characters are encoded into bytes using the UTF-8 charset.

This method is equivalent to: writeString(path, csq, StandardCharsets.UTF_8, options).

Parameters:

path - the path to the file

csq - the CharSequence to be written

options - options specifying how the file is opened

Returns:

the path

Throws:

IllegalArgumentException - if `options` contains an invalid combination of options

IOException - if an I/O error occurs writing to or creating the file, or the text cannot be encoded using UTF-8

UnsupportedOperationException - if an unsupported option is specified

Since:

11

writeString

```
public static Path writeString(Path path,
                               CharSequence csq,
                               Charset cs,
                               OpenOption... options)
                        throws IOException
```

Write a CharSequence to a file. Characters are encoded into bytes using the specified charset.

All characters are written as they are, including the line separators in the char sequence. No extra characters are added.

The `options` parameter specifies how the file is created or opened. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of `0`.

Parameters:

`path` - the path to the file

`csq` - the CharSequence to be written

`cs` - the charset to use for encoding

`options` - options specifying how the file is opened

Returns:

the path

Throws:

IllegalArgumentException - if `options` contains an invalid combination of options

IOException - if an I/O error occurs writing to or creating the file, or the text cannot be encoded using the specified charset

UnsupportedOperationException - if an unsupported option is specified

Since:

11

list

```
public static Stream<Path> list(Path dir)
                         throws IOException
```

Returns a lazily populated `Stream`, the elements of which are the entries in the directory. The listing is not recursive.

The elements of the stream are `Path` objects that are obtained as if by `resolving` the name of the directory entry against `dir`. Some file systems maintain special links to the directory itself and the directory's parent directory. Entries representing these links are not included.

The stream is *weakly consistent*. It is thread safe but does not freeze the directory while iterating, so it may (or may not) reflect updates to the directory that occur after returning from this method.

The returned stream contains a reference to an open directory. The directory is closed by closing the stream.

Operating on a closed stream behaves as if the end of stream has been reached. Due to read-ahead, one or more elements may be returned after the stream has been closed.

If an `IOException` is thrown when accessing the directory after this method has returned, it is wrapped in an `UncheckedIOException` which will be thrown from the method that caused the access to take place.

API Note:

This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open directory is closed promptly after the stream's operations have completed.

Parameters:

`dir` - The path to the directory

Returns:

The `Stream` describing the content of the directory

Throws:

`NotDirectoryException` - if the file could not otherwise be opened because it is not a directory *(optional specific exception)*

`IOException` - if an I/O error occurs when opening the directory

Since:

1.8

See Also:

newDirectoryStream(Path)

walk

```
public static Stream<Path> walk(Path start,
                                int maxDepth,
                                FileVisitOption... options)
                         throws IOException
```

Returns a `Stream` that is lazily populated with `Path` by walking the file tree rooted at a given starting file. The file tree is traversed *depth-first* with a directory visited before the entries in that directory. The elements in the stream are `Path` objects that are obtained as if by `resolving` the relative path against `start`.

The `stream` walks the file tree as elements are consumed. The `Stream` returned is guaranteed to have at least one element, the starting file itself. For each file visited, the stream attempts to read its `BasicFileAttributes`. If the file is a directory and can be opened successfully, entries in the directory, and their *descendants* will follow the directory in the stream as they are encountered. When all entries have been visited, then the directory is closed. The file tree walk then continues at the next *sibling* of the directory.

The stream is *weakly consistent*. It does not freeze the file tree while iterating, so it may (or may not) reflect updates to the file tree that occur after returned from this method.

By default, symbolic links are not automatically followed by this method. If the `options` parameter contains the `FOLLOW_LINKS` option then symbolic links are followed. When following links, and the attributes of the target cannot be read, then this method attempts to get the `BasicFileAttributes` of the link.

If the `options` parameter contains the `FOLLOW_LINKS` option then the stream keeps track of directories visited so that cycles can be detected. A cycle arises when there is an entry in a directory that is an ancestor of the directory. Cycle detection is done by recording the `file-key` of directories, or if file keys are not available, by invoking the `isSameFile` method to test if a directory is the same file as an ancestor. When a cycle is detected it is treated as an I/O error with an instance of `FileSystemLoopException`.

The `maxDepth` parameter is the maximum number of levels of directories to visit. A value of `0` means that only the starting file is visited. A value of `MAX_VALUE` may be used to indicate that all levels should be visited.

The returned stream contains references to one or more open directories. The directories are closed by closing the stream.

If an `IOException` is thrown when accessing the directory after this method has returned, it is wrapped in an `UncheckedIOException` which will be thrown from the method that caused the access to take place.

API Note:

This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open directories are closed promptly after the stream's operations have completed.

Parameters:

`start` - the starting file

`maxDepth` - the maximum number of directory levels to visit

`options` - options to configure the traversal

Returns:

the `Stream` of `Path`

Throws:

`IllegalArgumentException` - if the `maxDepth` parameter is negative

`IOException` - if an I/O error is thrown when accessing the starting file.

Since:

1.8

walk

```
public static Stream<Path> walk(Path start,
                                FileVisitOption... options)
                         throws IOException
```

Returns a `Stream` that is lazily populated with `Path` by walking the file tree rooted at a given starting file. The file tree is traversed *depth-first* with a directory visited before the entries in that directory. The elements in the stream are `Path` objects that are obtained as if by `resolving` the relative path against `start`.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.walk(start, Integer.MAX_VALUE, options)
```

In other words, it visits all levels of the file tree.

The returned stream contains references to one or more open directories. The directories are closed by closing the stream.

API Note:

This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open directories are closed promptly after the stream's operations have completed.

Parameters:

`start` - the starting file

`options` - options to configure the traversal

Returns:

the `Stream` of `Path`

Throws:

`IOException` - if an I/O error is thrown when accessing the starting file.

Since:

1.8

See Also:

`walk(Path, int, FileVisitOption...)`

find

```
public static Stream<Path> find(Path start,
                                int maxDepth,
                                BiPredicate<Path, BasicFileAttributes> matcher,
                                FileVisitOption... options)
                         throws IOException
```

Returns a `Stream` that is lazily populated with `Path` by searching for files in a file tree rooted at a given starting file.

This method walks the file tree in exactly the manner specified by the `walk` method. For each file encountered, the given `BiPredicate` is invoked with its `Path` and `BasicFileAttributes`. The `Path` object is obtained as if by `resolving` the relative path against `start` and is only included in the returned `Stream` if the `BiPredicate` returns true. Compare to calling `filter` on the `Stream` returned by `walk` method, this method may be more efficient by avoiding redundant retrieval of the `BasicFileAttributes`.

The returned stream contains references to one or more open directories. The directories are closed by closing the stream.

If an `IOException` is thrown when accessing the directory after returned from this method, it is wrapped in an `UncheckedIOException` which will be thrown from the method that caused the access to take place.

API Note:

This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open directories are closed promptly after the stream's operations have completed.

Parameters:

`start` - the starting file

`maxDepth` - the maximum number of directory levels to search

`matcher` - the function used to decide whether a file should be included in the returned stream

`options` - options to configure the traversal

Returns:

the `Stream` of `Path`

Throws:

`IllegalArgumentException` - if the `maxDepth` parameter is negative

`IOException` - if an I/O error is thrown when accessing the starting file.

Since:

1.8

See Also:

`walk(Path, int, FileVisitOption...)`

lines

```
public static Stream<String> lines(Path path,
                                   Charset cs)
                            throws IOException
```

Read all lines from a file as a `Stream`. Unlike `readAllLines`, this method does not read all lines into a `List`, but instead populates lazily as the stream is consumed.

Bytes from the file are decoded into characters using the specified charset and the same line terminators as specified by `readAllLines` are supported.

The returned stream contains a reference to an open file. The file is closed by closing the stream.

The file contents should not be modified during the execution of the terminal stream operation. Otherwise, the result of the terminal stream operation is undefined.

After this method returns, then any subsequent I/O exception that occurs while reading from the file or when a malformed or unmappable byte sequence is read, is wrapped in an `UncheckedIOException` that will be thrown from the `Stream` method that caused the read to take place. In case an `IOException` is thrown when closing the file, it is also wrapped as an `UncheckedIOException`.

API Note:

This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open file is closed promptly after the stream's operations have completed.

Implementation Note:

This implementation supports good parallel stream performance for the standard charsets `UTF-8`, `US-ASCII` and `ISO-8859-1`. Such *line-optimal* charsets have the property that the encoded bytes of a line feed ('\n') or a carriage return ('\r') are efficiently identifiable from other encoded characters when randomly accessing the bytes of the file.

For non-*line-optimal* charsets the stream source's spliterator has poor splitting properties, similar to that of a spliterator associated with an iterator or that associated with a stream returned from `BufferedReader.lines()`. Poor splitting properties can result in poor parallel stream performance.

For *line-optimal* charsets the stream source's spliterator has good splitting properties, assuming the file contains a regular sequence of lines. Good splitting properties can result in good parallel stream performance. The spliterator for a *line-optimal* charset takes advantage of the charset properties (a line feed or a carriage return being efficient identifiable) such that when splitting it can approximately divide the number of covered lines in half.

Parameters:

`path` - the path to the file

`cs` - the charset to use for decoding

Returns:

the lines from the file as a `Stream`

Throws:

`IOException` - if an I/O error occurs opening the file

Since:

1.8

See Also:

`readAllLines(Path, Charset)`,
`newBufferedReader(Path, Charset)`,
`BufferedReader.lines()`

lines

```
public static Stream<String> lines(Path path)
                           throws IOException
```

Read all lines from a file as a `Stream`. Bytes from the file are decoded into characters using the `UTF-8 charset`.

The returned stream contains a reference to an open file. The file is closed by closing the stream.

The file contents should not be modified during the execution of the terminal stream operation. Otherwise, the result of the terminal stream operation is undefined.

This method works as if invoking it were equivalent to evaluating the expression:

```
Files.lines(path, StandardCharsets.UTF_8)
```

API Note:

This method must be used within a try-with-resources statement or similar control structure to ensure that the stream's open file is closed promptly after the stream's operations have completed.

Parameters:

`path` - the path to the file

Returns:

the lines from the file as a `Stream`

Throws:

`IOException` - if an I/O error occurs opening the file

Since:

1.8
