

## Java SE 25 &amp; JDK 25

## Class File

`java.lang.Object``java.io.File`**All Implemented Interfaces:**`Serializable, Comparable<File>`

---

```
public class File
  extends Object
  implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

1. An optional system-dependent *prefix* string, such as a disk-drive specifier, "/" for the UNIX root directory, or "\\\\" for a Microsoft Windows UNC pathname, and
2. A sequence of zero or more string *names*.

The first name in an abstract pathname may be a directory name or, in the case of Microsoft Windows UNC pathnames, a hostname. Each subsequent name in an abstract pathname denotes a directory; the last name may denote either a directory or a file. The *empty* abstract pathname has no prefix and an empty name sequence. Accessing a file with the empty abstract pathname is equivalent to accessing the current user directory.

The conversion of a pathname string to or from an abstract pathname is inherently system-dependent. When an abstract pathname is converted into a pathname string, each name is separated from the next by a single copy of the default *separator character*. The default name-separator character is defined by the system property `file.separator`, and is made available in the public static fields `separator` and `separatorChar` of this class. When a pathname string is converted into an abstract pathname, the names within it may be separated by the default name-separator character or by any other name-separator character that is supported by the underlying system.

A pathname, whether abstract or in string form, may be either *absolute* or *relative*. An absolute pathname is complete in that no other information is required in order to locate the file that it denotes. A relative pathname, in contrast, must be interpreted in terms of information taken from some other pathname. By default the classes in the `java.io` package always resolve relative pathnames against the current user directory. This directory is named by the system property `user.dir`, and is typically the directory in which the Java virtual machine was invoked.

Unless otherwise noted, [symbolic links](#) are automatically redirected to the *target* of the link, whether they are provided by a pathname string or via a `File` object.

The *parent* of an abstract pathname may be obtained by invoking the `getParent()` method of this class and consists of the pathname's prefix and each name in the pathname's name sequence except for the last. Each directory's absolute pathname is an ancestor of any `File` object with an absolute abstract pathname which begins with the directory's absolute pathname. For example, the directory denoted by the abstract pathname `"/usr"` is an ancestor of the directory denoted by the pathname `"/usr/local/bin"`.

The prefix concept is used to handle root directories on UNIX platforms, and drive specifiers, root directories and UNC pathnames on Microsoft Windows platforms, as follows:

- For UNIX platforms, the prefix of an absolute pathname is always `"/"`. Relative pathnames have no prefix. The abstract pathname denoting the root directory has the prefix `"/"` and an empty name sequence.
- For Microsoft Windows platforms, the prefix of a pathname that contains a drive specifier consists of the drive letter followed by `":"` and possibly followed by `\"` if the pathname is absolute. The prefix of a UNC pathname is `\\\"`; the hostname and the share name are the first two names in the name sequence. A relative pathname that does not specify a drive has no prefix.

Instances of this class may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a *partition*. A partition is an operating system-specific portion of storage for a file system. A single storage device (e.g. a physical disk-drive, flash memory, CD-ROM) may contain multiple partitions. The object, if any, will reside on the partition named by some ancestor of the absolute form of this pathname.

A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as *access permissions*. The file system may have multiple sets of access permissions on a single object. For example, one set may apply to the object's *owner*, and another may apply to all other users. The access permissions on an object may cause some methods in this class to fail.

Instances of the `File` class are immutable; that is, once created, the abstract pathname represented by a `File` object will never change.

## Interoperability with `java.nio.file` package

The `java.nio.file` package defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems. This API may be used to overcome many of the limitations of the `java.io.File` class. The `toPath` method may be used to obtain a `Path` that uses the abstract path represented by a `File` object to locate a file. The resulting `Path` may be used with the `Files` class to provide more efficient and extensive access to additional file

operations, file attributes, and I/O exceptions to help diagnose errors when an operation on a file fails.

Since:

1.0

See Also:

[Serialized Form](#)

Field Summary

Fields		
Modifier and Type	Field	Description
static final String	<a href="#">pathSeparator</a>	The system-dependent path-separator character, represented as a string for convenience.
static final char	<a href="#">pathSeparatorChar</a>	The system-dependent path-separator character.
static final String	<a href="#">separator</a>	The system-dependent default name-separator character, represented as a string for convenience.
static final char	<a href="#">separatorChar</a>	The system-dependent default name-separator character.

Constructor Summary

Constructors	
Constructor	Description
<a href="#">File</a> (File parent, String child)	Creates a new File instance from a parent abstract pathname and a child pathname string.
<a href="#">File</a> (String pathname)	Creates a new File instance by converting the given pathname string into an abstract pathname.
<a href="#">File</a> (String parent, String child)	Creates a new File instance from a parent pathname string and a child pathname string.
<a href="#">File</a> (URI uri)	Creates a new File instance by converting the given file: URI into an abstract pathname.

Method Summary

**All Methods**

## Static Methods

## Instance Methods

## Concrete Methods

## Deprecated

Modifier and Type	Method	Description
boolean	<a href="#">canExecute()</a>	Tests whether the application can execute the file denoted by this abstract pathname.
boolean	<a href="#">canRead()</a>	Tests whether the application can read the file denoted by this abstract pathname.
boolean	<a href="#">canWrite()</a>	Tests whether the application can modify the file denoted by this abstract pathname.
int	<a href="#">compareTo(File pathname)</a>	Compares two abstract pathnames lexicographically.
boolean	<a href="#">createNewFile()</a>	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
static File	<a href="#">createTempFile(String prefix, String suffix)</a>	Creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
static File	<a href="#">createTempFile(String prefix, String suffix, File directory)</a>	Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.
boolean	<a href="#">delete()</a>	Deletes the file or directory denoted by this abstract pathname.
void	<a href="#">deleteOnExit()</a>	Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
boolean	<a href="#">equals(Object obj)</a>	Tests this abstract pathname for equality with the given object.

boolean	<b>exists()</b>	Tests whether the file or directory denoted by this abstract pathname exists.
File	<b>getAbsoluteFile()</b>	Returns the absolute form of this abstract pathname.
String	<b>getAbsolutePath()</b>	Returns the absolute pathname string of this abstract pathname.
File	<b>getCanonicalFile()</b>	Returns the canonical form of this abstract pathname.
String	<b>getCanonicalPath()</b>	Returns the canonical pathname string of this abstract pathname.
long	<b>getFreeSpace()</b>	Returns the number of unallocated bytes in the partition <b>named</b> by this abstract path name.
String	<b>getName()</b>	Returns the name of the file or directory denoted by this abstract pathname.
String	<b>getParent()</b>	Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File	<b>getParentFile()</b>	Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
String	<b>getPath()</b>	Converts this abstract pathname into a pathname string.
long	<b>getTotalSpace()</b>	Returns the size of the partition <b>named</b> by this abstract pathname.
long	<b>getUsableSpace()</b>	Returns the number of bytes available to this virtual machine on the partition <b>named</b> by this abstract pathname.
int	<b>hashCode()</b>	Computes a hash code for this abstract pathname.

boolean	<b>isAbsolute()</b>	Tests whether this abstract pathname is absolute.
boolean	<b>isDirectory()</b>	Tests whether the file denoted by this abstract pathname is a directory.
boolean	<b>isFile()</b>	Tests whether the file denoted by this abstract pathname is a normal file.
boolean	<b>isHidden()</b>	Tests whether the file named by this abstract pathname is a hidden file.
long	<b>lastModified()</b>	Returns the time that the file denoted by this abstract pathname was last modified.
long	<b>length()</b>	Returns the length of the file denoted by this abstract pathname.
String[]	<b>list()</b>	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
String[]	<b>list(FileNameFilter filter)</b>	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
File[]	<b>listFiles()</b>	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
File[]	<b>listFiles(FileFilter filter)</b>	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.

<code>File[]</code>	<b><code>listFiles</code></b> ( <code>FilenameFilter</code> filter)	Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
static <code>File[]</code>	<b><code>listRoots()</code></b>	List the available filesystem roots.
<code>boolean</code>	<b><code>mkdir()</code></b>	Creates the directory named by this abstract pathname.
<code>boolean</code>	<b><code>mkdirs()</code></b>	Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
<code>boolean</code>	<b><code>renameTo</code></b> ( <code>File</code> dest)	Renames the file denoted by this abstract pathname.
<code>boolean</code>	<b><code>setExecutable</code></b> ( <code>boolean</code> executable)	A convenience method to set the owner's execute permission for this abstract pathname.
<code>boolean</code>	<b><code>setExecutable</code></b> ( <code>boolean</code> executable, <code>boolean</code> ownerOnly)	Sets the owner's or everybody's execute permission for this abstract pathname.
<code>boolean</code>	<b><code>setLastModified</code></b> ( <code>long</code> time)	Sets the last-modified time of the file or directory named by this abstract pathname.
<code>boolean</code>	<b><code>setReadable</code></b> ( <code>boolean</code> readable)	A convenience method to set the owner's read permission for this abstract pathname.
<code>boolean</code>	<b><code>setReadable</code></b> ( <code>boolean</code> readable, <code>boolean</code> ownerOnly)	Sets the owner's or everybody's read permission for this abstract pathname.
<code>boolean</code>	<b><code>setReadOnly()</code></b>	Marks the file or directory named by this abstract pathname so that only read operations are allowed.
<code>boolean</code>	<b><code>setWritable</code></b> ( <code>boolean</code> writable)	A convenience method to set the owner's write permission for this abstract pathname.

boolean	<b>setWritable</b> (boolean writable, boolean ownerOnly)	Sets the owner's or everybody's write permission for this abstract pathname.
Path	<b>toPath</b> ()	Returns a <a href="#">java.nio.file.Path</a> object constructed from this abstract path.
String	<b>toString</b> ()	Returns the pathname string of this abstract pathname.
URI	<b>toURI</b> ()	Constructs a file: URI that represents this abstract pathname.
URL	<b>toURL</b> ()	<b>Deprecated.</b> This method does not automatically escape characters that are illegal in URLs.

### Methods declared in class **Object**

[clone](#), [finalize](#), [getClass](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

## Field Details

### **separatorChar**

```
public static final char separatorChar
```

The system-dependent default name-separator character. This field is initialized to contain the first character of the value of the system property `file.separator`. On UNIX systems the value of this field is `'/'`; on Microsoft Windows systems it is `'\\'`.

#### See Also:

[System.getProperty\(java.lang.String\)](#)

### **separator**

```
public static final String separator
```

The system-dependent default name-separator character, represented as a string for convenience. This string contains a single character, namely [separatorChar](#).

### **pathSeparatorChar**

```
public static final char pathSeparatorChar
```

The system-dependent path-separator character. This field is initialized to contain the first character of the value of the system property `path.separator`. This character is used to separate filenames in a sequence of files given as a *path list*. On UNIX systems, this character is `' : '`; on Microsoft Windows systems it is `' ; '`.

**See Also:**

[System.getProperty\(java.lang.String\)](#)

**pathSeparator**

```
public static final String pathSeparator
```

The system-dependent path-separator character, represented as a string for convenience. This string contains a single character, namely `pathSeparatorChar`.

**Constructor Details****File**

```
public File(String pathname)
```

Creates a new `File` instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.

**Parameters:**

pathname - A pathname string

**Throws:**

[NullPointerException](#) - If the pathname argument is null

**File**

```
public File(String parent,  
            String child)
```

Creates a new `File` instance from a parent pathname string and a child pathname string.

If parent is null then the new `File` instance is created as if by invoking the single-argument `File` constructor on the given child pathname string.

Otherwise the parent pathname string is taken to denote a directory, and the child pathname string is taken to denote either a directory or a file. If the child pathname string is absolute then it is converted into a relative pathname in a system-dependent way. If parent is the empty string then the new `File` instance is created by converting child into

an abstract pathname and resolving the result against a system-dependent default directory. Otherwise each pathname string is converted into an abstract pathname and the child abstract pathname is resolved against the parent.

**Parameters:**

parent - The parent pathname string

child - The child pathname string

**Throws:**

`NullPointerException` - If child is null

**File**

```
public File(File parent,  
            String child)
```

Creates a new File instance from a parent abstract pathname and a child pathname string.

If parent is null then the new File instance is created as if by invoking the single-argument File constructor on the given child pathname string.

Otherwise the parent abstract pathname is taken to denote a directory, and the child pathname string is taken to denote either a directory or a file. If the child pathname string is absolute then it is converted into a relative pathname in a system-dependent way. If parent is the empty abstract pathname then the new File instance is created by converting child into an abstract pathname and resolving the result against a system-dependent default directory. Otherwise each pathname string is converted into an abstract pathname and the child abstract pathname is resolved against the parent.

**Parameters:**

parent - The parent abstract pathname

child - The child pathname string

**Throws:**

`NullPointerException` - If child is null

**File**

```
public File(URI uri)
```

Creates a new File instance by converting the given file: URI into an abstract pathname.

The exact form of a file: URI is system-dependent, hence the transformation performed by this constructor is also system-dependent.

For a given abstract pathname *f* it is guaranteed that

```
new File( f.toURI()).equals( f.getAbsoluteFile())
```

so long as the original abstract pathname, the URI, and the new abstract pathname are all created in (possibly different invocations of) the same Java virtual machine. This relationship typically does not hold, however, when a `file:` URI that is created in a virtual machine on one operating system is converted into an abstract pathname in a virtual machine on a different operating system.

**Parameters:**

`uri` - An absolute, hierarchical URI with a scheme equal to `"file"`, a non-empty path component, and undefined authority, query, and fragment components

**Throws:**

`NullPointerException` - If `uri` is `null`

`IllegalArgumentException` - If the preconditions on the parameter do not hold

**Since:**

1.4

**See Also:**

`toURI()`, `URI`

## Method Details

### getName

```
public String getName()
```

Returns the name of the file or directory denoted by this abstract pathname. This is just the last name in the pathname's name sequence. If the pathname's name sequence is empty, then the empty string is returned.

**Returns:**

The name of the file or directory denoted by this abstract pathname, or the empty string if this pathname's name sequence is empty

### getParent

```
public String getParent()
```

Returns the pathname string of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.

The *parent* of an abstract pathname consists of the pathname's prefix, if any, and each name in the pathname's name sequence except for the last. If the name sequence is empty then the pathname does not name a parent directory.

**Returns:**

The pathname string of the parent directory named by this abstract pathname, or null if this pathname does not name a parent

## getParentFile

```
public File getParentFile()
```

Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.

The *parent* of an abstract pathname consists of the pathname's prefix, if any, and each name in the pathname's name sequence except for the last. If the name sequence is empty then the pathname does not name a parent directory.

### Returns:

The abstract pathname of the parent directory named by this abstract pathname, or null if this pathname does not name a parent

### Since:

1.2

## getPath

```
public String getPath()
```

Converts this abstract pathname into a pathname string. The resulting string uses the `default name-separator character` to separate the names in the name sequence.

### Returns:

The string form of this abstract pathname

## isAbsolute

```
public boolean isAbsolute()
```

Tests whether this abstract pathname is absolute. The definition of absolute pathname is system dependent. On UNIX systems, a pathname is absolute if its prefix is `"/"`. On Microsoft Windows systems, a pathname is absolute if its prefix is a drive specifier followed by `"\"`, or if its prefix is `"\\\\"`.

### Returns:

true if this abstract pathname is absolute, false otherwise

## getAbsolutePath

```
public String getAbsolutePath()
```

Returns the absolute pathname string of this abstract pathname.

If this abstract pathname is already absolute, then the pathname string is simply returned as if by the `getPath()` method. If this abstract pathname is the empty abstract pathname then the pathname string of the current user directory, which is named by the system property `user.dir`, is returned. Otherwise this pathname is resolved in a system-dependent way. On UNIX systems, a relative pathname is made absolute by resolving it against the current user directory. On Microsoft Windows systems, a relative pathname is made absolute by resolving it against the current directory of the drive named by the pathname, if any; if not, it is resolved against the current user directory.

**Returns:**

The absolute pathname string denoting the same file or directory as this abstract pathname

**See Also:**

`isAbsolute()`

## **getAbsolutePath**

```
public File getAbsolutePath()
```

Returns the absolute form of this abstract pathname. Equivalent to `new File(this.getAbsolutePath())`.

**Returns:**

The absolute abstract pathname denoting the same file or directory as this abstract pathname

**Since:**

1.2

## **getCanonicalPath**

```
public String getCanonicalPath()
                throws IOException
```

Returns the canonical pathname string of this abstract pathname.

A canonical pathname is both absolute and unique. The precise definition of canonical form is system-dependent. This method first converts this pathname to absolute form if necessary, as if by invoking the `getAbsolutePath()` method, and then maps it to its unique form in a system-dependent way. This typically involves removing redundant names such as `."` and `.."` from the pathname, resolving symbolic links, and converting drive letters to a standard case (on Microsoft Windows platforms).

Every pathname that denotes an existing file or directory has a unique canonical form. Every pathname that denotes a nonexistent file or directory also has a unique canonical form. The canonical form of the pathname of a nonexistent file or directory may be different from the canonical form of the same pathname after the file or directory is created. Similarly, the canonical form of the pathname of an existing file or directory may be different from the canonical form of the same pathname after the file or directory is deleted.

**Returns:**

The canonical pathname string denoting the same file or directory as this abstract pathname

**Throws:**

[IOException](#) - If an I/O error occurs, which is possible because the construction of the canonical pathname may require filesystem queries

**Since:**

1.1

**See Also:**

[Path.toRealPath\(LinkOption...\)](#)

## getCanonicalFile

```
public File getCanonicalFile()
           throws IOException
```

Returns the canonical form of this abstract pathname. Equivalent to `new File(this.getCanonicalPath())`.

**Returns:**

The canonical pathname string denoting the same file or directory as this abstract pathname

**Throws:**

[IOException](#) - If an I/O error occurs, which is possible because the construction of the canonical pathname may require filesystem queries

**Since:**

1.2

**See Also:**

[Path.toRealPath\(LinkOption...\)](#)

## toURL

[@Deprecated](#)

```
public URL toURL()
```

throws [MalformedURLException](#)

**Deprecated.**

*This method does not automatically escape characters that are illegal in URLs. It is recommended that new code convert an abstract pathname into a URL by first converting it into a URI, via the [toURI](#) method, and then converting the URI into a URL via the [URI.toURL](#) method.*

Converts this abstract pathname into a file: URL. The exact form of the URL is system-dependent. If it can be determined that the file denoted by this abstract pathname is a directory, then the resulting URL will end with a slash.

**Returns:**

A URL object representing the equivalent file URL

**Throws:**

[MalformedURLException](#) - If the path cannot be parsed as a URL

**Since:**

1.2

**See Also:**

[toURI\(\)](#), [URI](#), [URI.toURL\(\)](#), [URL](#)

**toURI**

```
public URI toURI()
```

Constructs a file: URI that represents this abstract pathname.

The exact form of the URI is system-dependent. If it can be determined that the file denoted by this abstract pathname is a directory, then the resulting URI will end with a slash.

For a given abstract pathname *f*, it is guaranteed that

```
new File( f.toURI()).equals( f.getAbsoluteFile())
```

so long as the original abstract pathname, the URI, and the new abstract pathname are all created in (possibly different invocations of) the same Java virtual machine. Due to the system-dependent nature of abstract pathnames, however, this relationship typically does not hold when a file: URI that is created in a virtual machine on one operating system is converted into an abstract pathname in a virtual machine on a different operating system.

Note that when this abstract pathname represents a UNC pathname then all components of the UNC (including the server name component) are encoded in the URI path. The authority component is undefined, meaning that it is represented as null. The [Path](#) class defines the [toUri](#) method to encode the server name in the authority component of the

resulting URI. The `toPath` method may be used to obtain a `Path` representing this abstract pathname.

**Returns:**

An absolute, hierarchical URI with a scheme equal to "file", a path representing this abstract pathname, and undefined authority, query, and fragment components

**Since:**

1.4

**See Also:**

`File(java.net.URI)`, `URI`, `URI.toURL()`

**canRead**

```
public boolean canRead()
```

Tests whether the application can read the file denoted by this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to read files that are marked as unreadable. Consequently, this method may return `true` even though the file does not have read permissions.

**Returns:**

`true` if and only if the file specified by this abstract pathname exists *and* can be read by the application; `false` otherwise

**canWrite**

```
public boolean canWrite()
```

Tests whether the application can modify the file denoted by this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to modify files that are marked read-only. Consequently, this method may return `true` even though the file is marked read-only.

**Returns:**

`true` if and only if the file system actually contains a file denoted by this abstract pathname *and* the application is allowed to write to the file; `false` otherwise.

**exists**

```
public boolean exists()
```

Tests whether the file or directory denoted by this abstract pathname exists.

**Returns:**

true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

## isDirectory

```
public boolean isDirectory()
```

Tests whether the file denoted by this abstract pathname is a directory.

Where it is required to distinguish an I/O exception from the case that the file is not a directory, or where several attributes of the same file are required at the same time, then the [Files.readAttributes](#) method may be used.

### Returns:

true if and only if the file denoted by this abstract pathname exists *and* is a directory; false otherwise

## isFile

```
public boolean isFile()
```

Tests whether the file denoted by this abstract pathname is a normal file. A file is *normal* if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file.

Where it is required to distinguish an I/O exception from the case that the file is not a normal file, or where several attributes of the same file are required at the same time, then the [Files.readAttributes](#) method may be used.

### Returns:

true if and only if the file denoted by this abstract pathname exists *and* is a normal file; false otherwise

## isHidden

```
public boolean isHidden()
```

Tests whether the file named by this abstract pathname is a hidden file. The exact definition of *hidden* is system-dependent. On UNIX systems, a file is considered to be hidden if its name begins with a period character ( '.' ). On Microsoft Windows systems, a file is considered to be hidden if it has been marked as such in the filesystem.

### Implementation Note:

If the file is a symbolic link, then on UNIX system it is considered to be hidden if the name of the link itself, not that of its target, begins with a period character. On Windows systems, a symbolic link is considered hidden if its target is so marked in the filesystem.

### Returns:

true if and only if the file denoted by this abstract pathname is hidden according to the conventions of the underlying platform

**Since:**

1.2

**lastModified**

```
public long lastModified()
```

Returns the time that the file denoted by this abstract pathname was last modified.

**API Note:**

While the unit of time of the return value is milliseconds, the granularity of the value depends on the underlying file system and may be larger. For example, some file systems use time stamps in units of seconds.

Where it is required to distinguish an I/O exception from the case where 0L is returned, or where several attributes of the same file are required at the same time, or where the time of last access or the creation time are required, then the [Files.readAttributes](#) method may be used. If however only the time of last modification is required, then the [Files.getLastModifiedTime](#) method may be used instead.

**Returns:**

A long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs. The value may be negative indicating the number of milliseconds before the epoch

**length**

```
public long length()
```

Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.

Where it is required to distinguish an I/O exception from the case that 0L is returned, or where several attributes of the same file are required at the same time, then the [Files.readAttributes](#) method may be used.

**Returns:**

The length, in bytes, of the file denoted by this abstract pathname, or 0L if the file does not exist. Some operating systems may return 0L for pathnames denoting system-dependent entities such as devices or pipes.

**createNewFile**

```
public boolean createNewFile()  
    throws IOException
```

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The [FileLock](#) facility should be used instead.

**Returns:**

true if the named file does not exist and was successfully created; false if the named file already exists

**Throws:**

[IOException](#) - If an I/O error occurred

**Since:**

1.2

**delete**

```
public boolean delete()
```

Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. If this pathname denotes a symbolic link, then the link itself, not its target, will be deleted.

Note that the [Files](#) class defines the [delete](#) method to throw an [IOException](#) when a file cannot be deleted. This is useful for error reporting and to diagnose why a file cannot be deleted.

**Returns:**

true if and only if the file or directory is successfully deleted; false otherwise

**deleteOnExit**

```
public void deleteOnExit()
```

Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates. If this pathname denotes a symbolic link, then the link itself, not its target, will be deleted. Files (or directories) are deleted in the reverse order that they are registered. Invoking this method to delete a file or directory that is already registered for deletion has no effect. Deletion will be attempted only for normal termination of the virtual machine, as defined by the Java Language Specification.

Once deletion has been requested, it is not possible to cancel the request. This method should therefore be used with care.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The [FileLock](#) facility should be used instead.

**Since:**

1.2

**See Also:**

[delete\(\)](#)

## list

```
public String[] list()
```

Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

If this abstract pathname does not denote a directory, then this method returns `null`. Otherwise an array of strings is returned, one for each file or directory in the directory. Names denoting the directory itself and the directory's parent directory are not included in the result. Each string is a file name rather than a complete path.

There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

Note that the [Files](#) class defines the [newDirectoryStream](#) method to open a directory and iterate over the names of the files in the directory. This may use less resources when working with very large directories, and may be more responsive when working with remote directories.

**Returns:**

An array of strings naming the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

## list

```
public String[] list(FilenameFilter filter)
```

Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. The behavior of this method is the same as that of the [list\(\)](#) method, except that the strings in the returned array must satisfy the filter. If the given filter is `null` then all names are accepted. Otherwise, a name satisfies the filter if and only if the value `true` results when the

`FilenameFilter.accept(File, String)` method of the filter is invoked on this abstract pathname and the name of a file or directory in the directory that it denotes.

**Parameters:**

`filter` - A filename filter

**Returns:**

An array of strings naming the files and directories in the directory denoted by this abstract pathname that were accepted by the given filter. The array will be empty if the directory is empty or if no names were accepted by the filter. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

**See Also:**

`Files.newDirectoryStream(Path, String)`

**listFiles**

```
public File[] listFiles()
```

Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

If this abstract pathname does not denote a directory, then this method returns `null`. Otherwise an array of `File` objects is returned, one for each file or directory in the directory. Pathnames denoting the directory itself and the directory's parent directory are not included in the result. Each resulting abstract pathname is constructed from this abstract pathname using the `File(File, String)` constructor. Therefore if this pathname is absolute then each resulting pathname is absolute; if this pathname is relative then each resulting pathname will be relative to the same directory.

There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

Note that the `Files` class defines the `newDirectoryStream` method to open a directory and iterate over the names of the files in the directory. This may use less resources when working with very large directories.

**Returns:**

An array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

**Since:**

1.2

**listFiles**

```
public File[] listFiles(FilenameFilter filter)
```

Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. The behavior of this method is the same as that of the `listFiles()` method, except that the pathnames in the returned array must satisfy the filter. If the given `filter` is `null` then all pathnames are accepted. Otherwise, a pathname satisfies the filter if and only if the value `true` results when the `FilenameFilter.accept(File, String)` method of the filter is invoked on this abstract pathname and the name of a file or directory in the directory that it denotes.

**Parameters:**

`filter` - A filename filter

**Returns:**

An array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

**Since:**

1.2

**See Also:**

`Files.newDirectoryStream(Path, String)`

**listFiles**

```
public File[] listFiles(FileFilter filter)
```

Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. The behavior of this method is the same as that of the `listFiles()` method, except that the pathnames in the returned array must satisfy the filter. If the given `filter` is `null` then all pathnames are accepted. Otherwise, a pathname satisfies the filter if and only if the value `true` results when the `FileFilter.accept(File)` method of the filter is invoked on the pathname.

**Parameters:**

`filter` - A file filter

**Returns:**

An array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

**Since:**

1.2

**See Also:**

`Files.newDirectoryStream(Path, java.nio.file.DirectoryStream.Filter)`

## **mkdir**

```
public boolean mkdir()
```

Creates the directory named by this abstract pathname.

**Returns:**

true if and only if the directory was created; false otherwise

## **mkdirs**

```
public boolean mkdirs()
```

Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

**Returns:**

true if and only if the directory was created, along with all necessary parent directories; false otherwise

## **renameTo**

```
public boolean renameTo(File dest)
```

Renames the file denoted by this abstract pathname. If this pathname denotes a symbolic link, then the link itself, not its target, will be renamed.

Many aspects of the behavior of this method are inherently platform-dependent: The rename operation might not be able to move a file from one filesystem to another, it might not be atomic, and it might not succeed if a file with the destination abstract pathname already exists. The return value should always be checked to make sure that the rename operation was successful. As instances of `File` are immutable, this `File` object is not changed to name the destination file or directory.

Note that the `Files` class defines the `move` method to move or rename a file in a platform independent manner.

**Parameters:**

dest - The new abstract pathname for the named file

**Returns:**

true if and only if the renaming succeeded; false otherwise

**Throws:**

`NullPointerException` - If parameter dest is null

## setLastModified

```
public boolean setLastModified(long time)
```

Sets the last-modified time of the file or directory named by this abstract pathname.

All platforms support file-modification times to the nearest second, but some provide more precision. The argument will be truncated to fit the supported precision. If the operation succeeds and no intervening operations on the file take place, then the next invocation of the `lastModified()` method will return the (possibly truncated) time argument that was passed to this method.

### Parameters:

`time` - The new last-modified time, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970)

### Returns:

true if and only if the operation succeeded; false otherwise

### Throws:

`IllegalArgumentException` - If the argument is negative

### Since:

1.2

## setReadOnly

```
public boolean setReadOnly()
```

Marks the file or directory named by this abstract pathname so that only read operations are allowed. After invoking this method the file or directory will not change until it is either deleted or marked to allow write access. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to modify files that are marked read-only. Whether or not a read-only file or directory may be deleted depends upon the underlying system.

### Returns:

true if and only if the operation succeeded; false otherwise

### Since:

1.2

## setWritable

```
public boolean setWritable(boolean writable,  
                           boolean ownerOnly)
```

Sets the owner's or everybody's write permission for this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to modify files that disallow write operations.

The `Files` class defines methods that operate on file attributes including file permissions. This may be used when finer manipulation of file permissions is required.

**Parameters:**

`writable` - If `true`, sets the access permission to allow write operations; if `false` to disallow write operations

`ownerOnly` - If `true`, the write permission applies only to the owner's write permission; otherwise, it applies to everybody. If the underlying file system can not distinguish the owner's write permission from that of others, then the permission will apply to everybody, regardless of this value.

**Returns:**

`true` if and only if the operation succeeded. The operation will fail if the user does not have permission to change the access permissions of this abstract pathname.

**Since:**

1.6

**setWritable**

```
public boolean setWritable(boolean writable)
```

A convenience method to set the owner's write permission for this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to modify files that disallow write operations.

An invocation of this method of the form `file.setWritable(arg)` behaves in exactly the same way as the invocation

```
file.setWritable(arg, true)
```

**Parameters:**

`writable` - If `true`, sets the access permission to allow write operations; if `false` to disallow write operations

**Returns:**

`true` if and only if the operation succeeded. The operation will fail if the user does not have permission to change the access permissions of this abstract pathname.

**Since:**

1.6

## setReadable

```
public boolean setReadable(boolean readable,  
                           boolean ownerOnly)
```

Sets the owner's or everybody's read permission for this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to read files that are marked as unreadable.

The [Files](#) class defines methods that operate on file attributes including file permissions. This may be used when finer manipulation of file permissions is required.

If the platform supports setting a file's read permission, but the user does not have permission to change the access permissions of this abstract pathname, then the operation will fail. If the platform does not support setting a file's read permission, this method does nothing and returns the value of the `readable` parameter.

### Parameters:

`readable` - If `true`, sets the access permission to allow read operations; if `false` to disallow read operations

`ownerOnly` - If `true`, the read permission applies only to the owner's read permission; otherwise, it applies to everybody. If the underlying file system can not distinguish the owner's read permission from that of others, then the permission will apply to everybody, regardless of this value.

### Returns:

`true` if the operation succeeds, `false` if it fails, or the value of the `readable` parameter if setting the read permission is not supported.

### Since:

1.6

## setReadable

```
public boolean setReadable(boolean readable)
```

A convenience method to set the owner's read permission for this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to read files that are marked as unreadable.

An invocation of this method of the form `file.setReadable(arg)` behaves in exactly the same way as the invocation

```
file.setReadable(arg, true)
```

If the platform supports setting a file's read permission, but the user does not have permission to change the access permissions of this abstract pathname, then the operation will fail. If the platform does not support setting a file's read permission, this method does nothing and returns the value of the `readable` parameter.

**Parameters:**

`readable` - If `true`, sets the access permission to allow read operations; if `false` to disallow read operations

**Returns:**

`true` if the operation succeeds, `false` if it fails, or the value of the `readable` parameter if setting the read permission is not supported.

**Since:**

1.6

**setExecutable**

```
public boolean setExecutable(boolean executable,  
                             boolean ownerOnly)
```

Sets the owner's or everybody's execute permission for this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to execute files that are not marked executable.

The [Files](#) class defines methods that operate on file attributes including file permissions. This may be used when finer manipulation of file permissions is required.

If the platform supports setting a file's execute permission, but the user does not have permission to change the access permissions of this abstract pathname, then the operation will fail. If the platform does not support setting a file's execute permission, this method does nothing and returns the value of the `executable` parameter.

**Parameters:**

`executable` - If `true`, sets the access permission to allow execute operations; if `false` to disallow execute operations

`ownerOnly` - If `true`, the execute permission applies only to the owner's execute permission; otherwise, it applies to everybody. If the underlying file system can not distinguish the owner's execute permission from that of others, then the permission will apply to everybody, regardless of this value.

**Returns:**

`true` if the operation succeeds, `false` if it fails, or the value of the `executable` parameter if setting the execute permission is not supported.

**Since:**

1.6

## setExecutable

```
public boolean setExecutable(boolean executable)
```

A convenience method to set the owner's execute permission for this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to execute files that are not marked executable.

An invocation of this method of the form `file.setExecutable(arg)` behaves in exactly the same way as the invocation

```
file.setExecutable(arg, true)
```

If the platform supports setting a file's execute permission, but the user does not have permission to change the access permissions of this abstract pathname, then the operation will fail. If the platform does not support setting a file's execute permission, this method does nothing and returns the value of the executable parameter.

### Parameters:

`executable` - If `true`, sets the access permission to allow execute operations; if `false` to disallow execute operations

### Returns:

`true` if the operation succeeds, `false` if it fails, or the value of the `executable` parameter if setting the execute permission is not supported.

### Since:

1.6

## canExecute

```
public boolean canExecute()
```

Tests whether the application can execute the file denoted by this abstract pathname. On some platforms it may be possible to start the Java virtual machine with special privileges that allow it to execute files that are not marked executable. Consequently, this method may return `true` even though the file does not have execute permissions.

### Returns:

`true` if and only if the abstract pathname exists *and* the application is allowed to execute the file

### Since:

1.6

## listRoots

```
public static File[] listRoots()
```

List the available filesystem roots.

A particular Java platform may support zero or more hierarchically-organized file systems. Each file system has a root directory from which all other files in that file system can be reached.

This method returns an array of `File` objects that denote the root directories of the available filesystem roots. It is guaranteed that the canonical pathname of any file physically present on the local machine will begin with one of the roots returned by this method. There is no guarantee that a root directory can be accessed.

**Implementation Note:**

Windows platforms, for example, have a root directory for each active drive; UNIX platforms have a single root directory, namely `"/"`. The set of filesystem roots is affected by various system-level operations such as the disconnecting or unmounting of physical or virtual disk drives.

The canonical pathname of a file that resides on some other machine and is accessed via a remote-filesystem protocol such as SMB or NFS may or may not begin with one of the roots returned by this method. If the pathname of a remote file is syntactically indistinguishable from the pathname of a local file then it will begin with one of the roots returned by this method. Thus, for example, `File` objects denoting the root directories of the mapped network drives of a Windows platform will be returned by this method, while `File` objects containing UNC pathnames will not be returned by this method.

**Returns:**

An array of `File` objects denoting the available filesystem roots, or `null` if the set of roots could not be determined. The array will be empty if there are no filesystem roots.

**Since:**

1.2

**See Also:**

[FileStore](#)

**getTotalSpace**

```
public long getTotalSpace()
```

Returns the size of the partition [named](#) by this abstract pathname. If the total number of bytes in the partition is greater than `Long.MAX_VALUE`, then `Long.MAX_VALUE` will be returned.

**Returns:**

The size, in bytes, of the partition or `0L` if this abstract pathname does not name a partition or if the size cannot be obtained

**Since:**

1.6

**See Also:**[FileStore.getTotalSpace\(\)](#)**getFreeSpace**

```
public long getFreeSpace()
```

Returns the number of unallocated bytes in the partition [named](#) by this abstract path name. If the number of unallocated bytes in the partition is greater than [Long.MAX\\_VALUE](#), then [Long.MAX\\_VALUE](#) will be returned.

The returned number of unallocated bytes is a hint, but not a guarantee, that it is possible to use most or any of these bytes. The number of unallocated bytes is most likely to be accurate immediately after this call. It is likely to be made inaccurate by any external I/O operations including those made on the system outside of this virtual machine. This method makes no guarantee that write operations to this file system will succeed.

**Returns:**

The number of unallocated bytes on the partition or 0L if the abstract pathname does not name a partition or if this number cannot be obtained. This value will be less than or equal to the total file system size returned by [getTotalSpace\(\)](#).

**Since:**

1.6

**See Also:**[FileStore.getUnallocatedSpace\(\)](#)**getUsableSpace**

```
public long getUsableSpace()
```

Returns the number of bytes available to this virtual machine on the partition [named](#) by this abstract pathname. If the number of available bytes in the partition is greater than [Long.MAX\\_VALUE](#), then [Long.MAX\\_VALUE](#) will be returned. When possible, this method checks for write permissions and other operating system restrictions and will therefore usually provide a more accurate estimate of how much new data can actually be written than [getFreeSpace\(\)](#).

The returned number of available bytes is a hint, but not a guarantee, that it is possible to use most or any of these bytes. The number of available bytes is most likely to be accurate immediately after this call. It is likely to be made inaccurate by any external I/O operations including those made on the system outside of this virtual machine. This method makes no guarantee that write operations to this file system will succeed.

**Returns:**

The number of available bytes on the partition or 0L if the abstract pathname does not name a partition or if this number cannot be obtained. On systems where this information is not available, this method will be equivalent to a call to `getFreeSpace()`.

**Since:**

1.6

**See Also:**

`FileStore.getUsableSpace()`

**createTempFile**

```
public static File createTempFile(String prefix,
                                String suffix,
                                File directory)
    throws IOException
```

Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. If this method returns successfully then it is guaranteed that:

1. The file denoted by the returned abstract pathname did not exist before this method was invoked, and
2. Neither this method nor any of its variants will return the same abstract pathname again in the current invocation of the virtual machine.

This method provides only part of a temporary-file facility. To arrange for a file created by this method to be deleted automatically, use the `deleteOnExit()` method.

The prefix argument must be at least three characters long. It is recommended that the prefix be a short, meaningful string such as "hjb" or "mail". The suffix argument may be null, in which case the suffix ".tmp" will be used.

To create the new file, the prefix and the suffix may first be adjusted to fit the limitations of the underlying platform. If the prefix is too long then it will be truncated, but its first three characters will always be preserved. If the suffix is too long then it too will be truncated, but if it begins with a period character ( '.' ) then the period and the first three characters following it will always be preserved. Once these adjustments have been made the name of the new file will be generated by concatenating the prefix, five or more internally-generated characters, and the suffix.

If a file with the generated name cannot be created by the underlying platform, then an `IOException` will be thrown. This could occur for example if the supplied prefix or suffix contains one or more characters not supported by the underlying file system.

If the directory argument is null then the system-dependent default temporary-file directory will be used. The default temporary-file directory is specified by the system property `java.io.tmpdir`. On UNIX systems the default value of this property is typically `"/tmp"` or `"/var/tmp"`; on Microsoft Windows systems it is typically `"C:\\\\WINNT\\\\TEMP"`. A

different value may be given to this system property when the Java virtual machine is invoked, but programmatic changes to this property are not guaranteed to have any effect upon the temporary directory used by this method.

If the `directory` argument is not `null` and its abstract pathname is valid and denotes an existing, writable directory, then the file will be created in that directory. Otherwise the file will not be created and an `IOException` will be thrown. Under no circumstances will a directory be created at the location specified by the `directory` argument.

**Parameters:**

`prefix` - The prefix string to be used in generating the file's name; must be at least three characters long

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

`directory` - The directory in which the file is to be created, or `null` if the default temporary-file directory is to be used

**Returns:**

An abstract pathname denoting a newly-created empty file

**Throws:**

[IllegalArgumentException](#) - If the `prefix` argument contains fewer than three characters

[IOException](#) - If a file could not be created

**Since:**

1.2

**createTempFile**

```
public static File createTempFile(String prefix,
                                  String suffix)
                               throws IOException
```

Creates an empty file in the default temporary-file directory, using the given `prefix` and `suffix` to generate its name. Invoking this method is equivalent to invoking `createTempFile(prefix, suffix, null)`.

The `Files.createTempFile` method provides an alternative method to create an empty file in the temporary-file directory. Files created by that method may have more restrictive access permissions to files created by this method and so may be more suited to security-sensitive applications.

**Parameters:**

`prefix` - The prefix string to be used in generating the file's name; must be at least three characters long

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

**Returns:**

An abstract pathname denoting a newly-created empty file

**Throws:**

[IllegalArgumentException](#) - If the prefix argument contains fewer than three characters

[IOException](#) - If a file could not be created

**Since:**

1.2

**See Also:**

[Files.createTempDirectory\(String,FileAttribute\[\]\)](#)

## compareTo

```
public int compareTo(File pathname)
```

Compares two abstract pathnames lexicographically. The ordering defined by this method depends upon the underlying system. On UNIX systems, alphabetic case is significant in comparing pathnames; on Microsoft Windows systems it is not.

**Specified by:**

[compareTo](#) in interface [Comparable<File>](#)

**Parameters:**

pathname - The abstract pathname to be compared to this abstract pathname

**Returns:**

Zero if the argument is equal to this abstract pathname, a value less than zero if this abstract pathname is lexicographically less than the argument, or a value greater than zero if this abstract pathname is lexicographically greater than the argument

**Since:**

1.2

## equals

```
public boolean equals(Object obj)
```

Tests this abstract pathname for equality with the given object. Returns true if and only if the argument is not null and is an abstract pathname that is the same as this abstract pathname. Whether or not two abstract pathnames are equal depends upon the underlying operating system. On UNIX systems, alphabetic case is significant in comparing pathnames; on Microsoft Windows systems it is not.

**Overrides:**

`equals` in class `Object`

**API Note:**

This method only tests whether the abstract pathnames are equal; it does not access the file system and the file is not required to exist.

**Parameters:**

`obj` - The object to be compared with this abstract pathname

**Returns:**

true if and only if the objects are the same; false otherwise

**See Also:**

`compareTo(File)`,  
`Files.isSameFile(Path,Path)`

**hashCode**

```
public int hashCode()
```

Computes a hash code for this abstract pathname. Because equality of abstract pathnames is inherently system-dependent, so is the computation of their hash codes. On UNIX systems, the hash code of an abstract pathname is equal to the exclusive *or* of the hash code of its pathname string and the decimal value 1234321. On Microsoft Windows systems, the hash code is equal to the exclusive *or* of the hash code of its pathname string converted to lower case and the decimal value 1234321. Locale is not taken into account on lowercasing the pathname string.

**Overrides:**

`hashCode` in class `Object`

**Returns:**

A hash code for this abstract pathname

**See Also:**

`Object.equals(java.lang.Object)`,  
`System.identityHashCode(Object)`

**toString**

```
public String toString()
```

Returns the pathname string of this abstract pathname. This is just the string returned by the `getPath()` method.

**Overrides:**

`toString` in class `Object`

**Returns:**

The string form of this abstract pathname

**toPath**

```
public Path toPath()
```

Returns a `java.nio.file.Path` object constructed from this abstract path. The resulting Path is associated with the `default-file-system`.

The first invocation of this method works as if invoking it were equivalent to evaluating the expression:

```
FileSystems.getDefault().getPath(this.getPath());
```

Subsequent invocations of this method return the same Path.

If this abstract pathname is the empty abstract pathname then this method returns a Path that may be used to access the current user directory.

**Returns:**

a Path constructed from this abstract path

**Throws:**

`InvalidPathException` - if a Path object cannot be constructed from the abstract path (see `FileSystem.getPath`)

**Since:**

1.7

**See Also:**

`Path.toFile()`

---

[Report a bug or suggest an enhancement](#)

For further API reference and developer documentation see the [Java SE Documentation](#) , which contains more detailed, developer-targeted descriptions with conceptual overviews, definitions of terms, workarounds, and working code examples. [Other versions](#).

Java is a trademark or registered trademark of Oracle and/or its affiliates in the US and other countries.

Copyright © 1993, 2025, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.

All rights reserved. Use is subject to [license terms](#) and the [documentation redistribution policy](#) . Modify [Cookie Preferences](#). Modify [Ad Choices](#) .