Module java.base
Package java.nio.file

Interface Path

All Superinterfaces:

Comparable<Path>, Iterable<Path>, Watchable

```
public interface Path
extends Comparable<Path>, Iterable<Path>, Watchable
```

An object that may be used to locate a file in a file system. It will typically represent a system dependent file path.

A `Path` represents a path that is hierarchical and composed of a sequence of directory and file name elements separated by a special separator or delimiter. A *root component*, that identifies a file system hierarchy, may also be present. The name element that is *farthest* from the root of the directory hierarchy is the name of a file or directory. The other name elements are directory names. A `Path` can represent a root, a root and a sequence of names, or simply one or more name elements. A `Path` is considered to be an *empty path* if it consists solely of one name element that is empty. Accessing a file using an *empty path* is equivalent to accessing the default directory of the file system. `Path` defines the `getFileName`, `getParent`, `getRoot`, and `subpath` methods to access the path components or a subsequence of its name elements.

In addition to accessing the components of a path, a `Path` also defines the `resolve` and `resolveSibling` methods to combine paths. The `relativize` method that can be used to construct a relative path between two paths. Paths can be `compared`, and tested against each other using the `startsWith` and `endsWith` methods.

This interface extends `Watchable` interface so that a directory located by a path can be `registered` with a `WatchService` and entries in the directory watched.

WARNING: This interface is only intended to be implemented by those developing custom file system implementations. Methods may be added to this interface in future releases.

Accessing Files

Paths may be used with the `Files` class to operate on files, directories, and other types of files. For example, suppose we want a `BufferedReader` to read text from a file "`access.log`". The file is located in a directory "`logs`" relative to the current working directory and

is UTF-8 encoded.

```
Path path = FileSystems.getDefault().getPath("logs", "access.log");
BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8);
```

Interoperability

Paths associated with the default `provider` are generally interoperable with the `java.io.File` class. Paths created by other providers are unlikely to be interoperable with the abstract path names represented by `java.io.File`. The `toPath` method may be used to obtain a `Path` from the abstract path name represented by a `java.io.File` object. The resulting `Path` can be used to operate on the same file as the `java.io.File` object. In addition, the `toFile` method is useful to construct a `File` from the `String` representation of a `Path`.

Concurrency

Implementations of this interface are immutable and safe for use by multiple concurrent threads.

Since:

1.7

Method Summary

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
`int`	`compareTo(Path other)`	Compares two abstract paths lexicographically.
`default boolean`	`endsWith(String other)`	Tests if this path ends with a `Path`, constructed by converting the given path string, in exactly the manner specified by the `endsWith(Path)` method.
`boolean`	`endsWith(Path other)`	Tests if this path ends with the given path.
`boolean`	`equals(Object other)`	Tests this path for equality with the given object.

Path	**getFileName**()	Returns the name of the file or directory denoted by this path as a Path object.
FileSystem	**getFileSystem**()	Returns the file system that created this object.
Path	**getName**(int index)	Returns a name element of this path as a Path object.
int	**getNameCount**()	Returns the number of name elements in the path.
Path	**getParent**()	Returns the *parent path*, or null if this path does not have a parent.
Path	**getRoot**()	Returns the root component of this path as a Path object, or null if this path does not have a root component.
int	**hashCode**()	Computes a hash code for this path.
boolean	**isAbsolute**()	Tells whether or not this path is absolute.
default **Iterator**<**Path**>	**iterator**()	Returns an iterator over the name elements of this path.
Path	**normalize**()	Returns a path that is this path with redundant name elements eliminated.
static **Path**	**of**(**String** first, **String**... more)	Returns a Path by converting a path string, or a sequence of strings that when joined form a path string.
static **Path**	**of**(**URI** uri)	Returns a Path by converting a URI.
default **WatchKey**	**register**(**WatchService** watcher, **WatchEvent.Kind**<?>... events)	Registers the file located by this path with a watch service.
WatchKey	**register**(**WatchService** watcher, **WatchEvent.Kind**<?>[] events,	Registers the file located by this path with a watch service.

	WatchEvent.Modifier... modifiers)	
Path	**relativize**(**Path** other)	Constructs a relative path between this path and a given path.
default **Path**	**resolve**(**String** other)	Converts a given path string to a `Path` and resolves it against this `Path` in exactly the manner specified by the `resolve` method.
default **Path**	**resolve**(**String** first, **String**... more)	Converts a path string to a path, resolves that path against this path, and then iteratively performs the same procedure for any additional path strings.
Path	**resolve**(**Path** other)	Resolve the given path against this path.
default **Path**	**resolve**(**Path** first, **Path**... more)	Resolves a path against this path, and then iteratively resolves any additional paths.
default **Path**	**resolveSibling**(**String** other)	Converts a given path string to a `Path` and resolves it against this path's `parent` path in exactly the manner specified by the `resolveSibling` method.
default **Path**	**resolveSibling**(**Path** other)	Resolves the given path against this path's `parent` path.
default boolean	**startsWith**(**String** other)	Tests if this path starts with a `Path`, constructed by converting the given path string, in exactly the manner specified by the `startsWith(Path)` method.
boolean	**startsWith**(**Path** other)	Tests if this path starts with the given path.
Path	**subpath**(int beginIndex, int endIndex)	Returns a relative `Path` that is a subsequence of the name elements of this path.
Path	**toAbsolutePath**()	Returns a `Path` object representing the absolute path of this path.
default **File**	**toFile**()	Returns a `File` object representing this path.

Path	**toRealPath**(**LinkOption**... options)	Returns the *real* path of an existing file.
String	**toString**()	Returns the string representation of this path.
URI	**toUri**()	Returns a URI to represent this path.

Methods declared in interface java.lang.Iterable

forEach, spliterator

Method Details

of

```
static Path of(String first,
               String... more)
```

Returns a `Path` by converting a path string, or a sequence of strings that when joined form a path string. If `more` does not specify any elements then the value of the `first` parameter is the path string to convert. If `more` specifies one or more elements then each non-empty string, including `first`, is considered to be a sequence of name elements and is joined to form a path string. The details as to how the Strings are joined is provider specific but typically they will be joined using the `name-separator` as the separator. For example, if the name separator is `"/"` and `getPath("/foo","bar","gus")` is invoked, then the path string `"/foo/bar/gus"` is converted to a `Path`. A `Path` representing an empty path is returned if `first` is the empty string and `more` does not contain any non-empty strings.

The `Path` is obtained by invoking the `getPath` method of the `default FileSystem`.

Note that while this method is very convenient, using it will imply an assumed reference to the default `FileSystem` and limit the utility of the calling code. Hence it should not be used in library code intended for flexible reuse. A more flexible alternative is to use an existing `Path` instance as an anchor, such as:

```
Path dir = ...
Path path = dir.resolve("file");
```

Parameters:

`first` - the path string or initial part of the path string

`more` - additional strings to be joined to form the path string

Returns:

the resulting `Path`

Throws:

`InvalidPathException` - if the path string cannot be converted to a `Path`

Since:

11

See Also:

`FileSystem.getPath(java.lang.String, java.lang.String...)`

of

```
static Path of(URI uri)
```

Returns a `Path` by converting a URI.

This method iterates over the `installed` providers to locate the provider that is identified by the URI `scheme` of the given URI. URI schemes are compared without regard to case. If the provider is found then its `getPath` method is invoked to convert the URI.

In the case of the default provider, identified by the URI scheme "file", the given URI has a non-empty path component, and undefined query and fragment components. Whether the authority component may be present is platform specific. The returned `Path` is associated with the `default` file system.

The default provider provides a similar *round-trip* guarantee to the `File` class. For a given `Path` *p* it is guaranteed that

```
Path.of(p.toUri()).equals( p.toAbsolutePath())
```

so long as the original `Path`, the `URI`, and the new `Path` are all created in (possibly different invocations of) the same Java virtual machine. Whether other providers make any guarantees is provider specific and therefore unspecified.

Parameters:

`uri` - the URI to convert

Returns:

the resulting `Path`

Throws:

`IllegalArgumentException` - if preconditions on the `uri` parameter do not hold. The format of the URI is provider specific.

`FileSystemNotFoundException` - The file system, identified by the URI, does not exist and cannot be created automatically, or the provider identified by the URI's scheme component is not installed

`SecurityException` - if a security manager is installed and it denies an unspecified permission to access the file system

Since:

11

getFileSystem

`FileSystem getFileSystem()`

Returns the file system that created this object.

Returns:

the file system that created this object

isAbsolute

`boolean isAbsolute()`

Tells whether or not this path is absolute.

An absolute path is complete in that it doesn't need to be combined with other path information in order to locate a file.

Returns:

`true` if, and only if, this path is absolute

getRoot

`Path getRoot()`

Returns the root component of this path as a `Path` object, or `null` if this path does not have a root component.

Returns:

a path representing the root component of this path, or `null`

getFileName

`Path getFileName()`

Returns the name of the file or directory denoted by this path as a `Path` object. The file name is the *farthest* element from the root in the directory hierarchy.

Returns:

a path representing the name of the file or directory, or `null` if this path has zero elements

getParent

`Path getParent()`

Returns the *parent path*, or `null` if this path does not have a parent.

The parent of this path object consists of this path's root component, if any, and each element in the path except for the *farthest* from the root in the directory hierarchy. This method does not access the file system; the path or its parent may not exist. Furthermore, this method does not eliminate special names such as "." and ".." that may be used in some implementations. On UNIX for example, the parent of "`/a/b/c`" is "`/a/b`", and the parent of "`x/y/.`" is "`x/y`". This method may be used with the `normalize` method, to eliminate redundant names, for cases where *shell-like* navigation is required.

If this path has more than one element, and no root component, then this method is equivalent to evaluating the expression:

```
    subpath(0, getNameCount()-1);
```

Returns:

a path representing the path's parent

getNameCount

```
int getNameCount()
```

Returns the number of name elements in the path.

Returns:

the number of elements in the path, or `0` if this path only represents a root component

getName

```
Path getName(int index)
```

Returns a name element of this path as a `Path` object.

The `index` parameter is the index of the name element to return. The element that is *closest* to the root in the directory hierarchy has index `0`. The element that is *farthest* from the root has index `count-1`.

Parameters:

`index` - the index of the element

Returns:

the name element

Throws:

`IllegalArgumentException` - if `index` is negative, `index` is greater than or equal to the number of elements, or this path has zero name elements

subpath

```
Path subpath(int beginIndex,
             int endIndex)
```

Returns a relative `Path` that is a subsequence of the name elements of this path.

The `beginIndex` and `endIndex` parameters specify the subsequence of name elements. The name that is *closest* to the root in the directory hierarchy has index `0`. The name that is *farthest* from the root has index `count-1`. The returned `Path` object has the name elements that begin at `beginIndex` and extend to the element at index `endIndex-1`.

Parameters:

`beginIndex` - the index of the first element, inclusive

`endIndex` - the index of the last element, exclusive

Returns:

a new `Path` object that is a subsequence of the name elements in this `Path`

Throws:

`IllegalArgumentException` - if `beginIndex` is negative, or greater than or equal to the number of elements. If `endIndex` is less than or equal to `beginIndex`, or larger than the number of elements.

startsWith

```
boolean startsWith(Path other)
```

Tests if this path starts with the given path.

This path *starts* with the given path if this path's root component *starts* with the root component of the given path, and this path starts with the same name elements as the given path. If the given path has more name elements than this path then `false` is returned.

Whether or not the root component of this path starts with the root component of the given path is file system specific. If this path does not have a root component and the given path has a root component then this path does not start with the given path.

If the given path is associated with a different `FileSystem` to this path then `false` is returned.

Parameters:

`other` - the given path

Returns:

`true` if this path starts with the given path; otherwise `false`

startsWith

```
default boolean startsWith(String other)
```

Tests if this path starts with a `Path`, constructed by converting the given path string, in exactly the manner specified by the `startsWith(Path)` method. On UNIX for example, the path "`foo/bar`" starts with "`foo`" and "`foo/bar`". It does not start with "`f`" or "`fo`".

Implementation Requirements:

The default implementation is equivalent for this path to:

```
    startsWith(getFileSystem().getPath(other));
```

Parameters:

`other` - the given path string

Returns:

`true` if this path starts with the given path; otherwise `false`

Throws:

`InvalidPathException` - If the path string cannot be converted to a Path.

endsWith

```
boolean endsWith(Path other)
```

Tests if this path ends with the given path.

If the given path has *N* elements, and no root component, and this path has *N* or more elements, then this path ends with the given path if the last *N* elements of each path, starting at the element farthest from the root, are equal.

If the given path has a root component then this path ends with the given path if the root component of this path *ends with* the root component of the given path, and the corresponding elements of both paths are equal. Whether or not the root component of this path

ends with the root component of the given path is file system specific. If this path does not have a root component and the given path has a root component then this path does not end with the given path.

If the given path is associated with a different `FileSystem` to this path then `false` is returned.

Parameters:

`other` - the given path

Returns:

`true` if this path ends with the given path; otherwise `false`

endsWith

```
default boolean endsWith(String other)
```

Tests if this path ends with a `Path`, constructed by converting the given path string, in exactly the manner specified by the `endsWith(Path)` method. On UNIX for example, the path "`foo/bar`" ends with "`foo/bar`" and "`bar`". It does not end with "`r`" or "`/bar`". Note that trailing separators are not taken into account, and so invoking this method on the `Path` "`foo/bar`" with the `String` "`bar/`" returns `true`.

Implementation Requirements:

The default implementation is equivalent for this path to:

```
    endsWith(getFileSystem().getPath(other));
```

Parameters:

`other` - the given path string

Returns:

`true` if this path ends with the given path; otherwise `false`

Throws:

`InvalidPathException` - If the path string cannot be converted to a Path.

normalize

`Path` `normalize()`

Returns a path that is this path with redundant name elements eliminated.

The precise definition of this method is implementation dependent but in general it derives from this path, a path that does not contain *redundant* name elements. In many file systems, the `"."` and `".."` are special names used to indicate the current directory and parent directory. In such file systems all occurrences of `"."` are considered redundant. If a `".."` is preceded by a non-`".."` name then both names are considered redundant (the process to identify such names is repeated until it is no longer applicable).

This method does not access the file system; the path may not locate a file that exists. Eliminating `".."` and a preceding name from a path may result in the path that locates a different file than the original path. This can arise when the preceding name is a symbolic link.

Returns:

the resulting path or this path if it does not contain redundant name elements; an empty path is returned if this path does not have a root component and all name elements are redundant

See Also:

`getParent()`,
`toRealPath(java.nio.file.LinkOption...)`

resolve

`Path` `resolve(Path other)`

Resolve the given path against this path.

If the `other` parameter is an `absolute` path then this method trivially returns `other`. If `other` is an *empty path* then this method trivially returns this path. Otherwise this method considers this path to be a directory and resolves the given path against this path. In the simplest case, the given path does not have a `root` component, in which case this method *joins* the given path to this path and returns a resulting path that `ends` with the given path. Where the given path has a root component then resolution is highly implementation dependent and therefore unspecified.

Parameters:

`other` - the path to resolve against this path

Returns:

the resulting path

See Also:

`relativize(java.nio.file.Path)`

resolve

```
default Path resolve(String other)
```

Converts a given path string to a `Path` and resolves it against this `Path` in exactly the manner specified by the `resolve` method. For example, suppose that the name separator is `"/"` and a path represents `"foo/bar"`, then invoking this method with the path string `"gus"` will result in the `Path` `"foo/bar/gus"`.

Implementation Requirements:

The default implementation is equivalent for this path to:

```
    resolve(getFileSystem().getPath(other));
```

Parameters:

`other` - the path string to resolve against this path

Returns:

the resulting path

Throws:

`InvalidPathException` - if the path string cannot be converted to a Path.

See Also:

`FileSystem.getPath(java.lang.String, java.lang.String...)`

resolve

```
default Path resolve(Path first,
                     Path... more)
```

Resolves a path against this path, and then iteratively resolves any additional paths.

This method resolves `first` against this `Path` as if by calling `resolve(Path)`. If `more` has one or more elements then it resolves the first element against the result, then iteratively resolves all subsequent elements. This method returns the result from the final resolve.

Implementation Requirements:

The default implementation is equivalent to the result obtained with:

```
    Path result = resolve(first);
    for (Path p : more) {
        result = result.resolve(p);
    }
```

Parameters:

`first` - the first path to resolve against this path

`more` - additional paths to iteratively resolve

Returns:

the resulting path

Since:

22

See Also:

`resolve(Path)`

resolve

```
default Path resolve(String first,
                     String... more)
```

Converts a path string to a path, resolves that path against this path, and then iteratively performs the same procedure for any additional path strings.

This method converts `first` to a `Path` and resolves that `Path` against this `Path` as if by calling `resolve(String)`. If `more` has one or more elements then it converts the first element to a path, resolves that path against the result, then iteratively converts and resolves all subsequent elements. This method returns the result from the final resolve.

Implementation Requirements:

The default implementation is equivalent to the result obtained with:

```
Path result = resolve(first);
for (String s : more) {
    result = result.resolve(s);
}
```

Parameters:

`first` - the first path string to convert to a path and resolve against this path

`more` - additional path strings to be iteratively converted to paths and resolved

Returns:

the resulting path

Throws:

`InvalidPathException` - if a path string cannot be converted to a Path.

Since:

22

See Also:

`resolve(Path,Path...)`,
`resolve(String)`

resolveSibling

`default Path resolveSibling(Path other)`

Resolves the given path against this path's `parent` path. This is useful where a file name needs to be *replaced* with another file name. For example, suppose that the name separator is "`/`" and a path represents "`dir1/dir2/foo`", then invoking this method with the `Path` "`bar`" will result in the `Path` "`dir1/dir2/bar`". If this path does not have a parent path, or `other` is `absolute`, then this method

returns `other`. If `other` is an empty path then this method returns this path's parent, or where this path doesn't have a parent, the empty path.

Implementation Requirements:

The default implementation is equivalent for this path to:

```
    (getParent() == null) ? other : getParent().resolve(other);
```

unless `other == null`, in which case a `NullPointerException` is thrown.

Parameters:

`other` - the path to resolve against this path's parent

Returns:

the resulting path

See Also:

`resolve(Path)`

resolveSibling

`default Path resolveSibling(String other)`

Converts a given path string to a `Path` and resolves it against this path's `parent` path in exactly the manner specified by the `resolveSibling` method.

Implementation Requirements:

The default implementation is equivalent for this path to:

```
    resolveSibling(getFileSystem().getPath(other));
```

Parameters:

`other` - the path string to resolve against this path's parent

Returns:

the resulting path

Throws:

`InvalidPathException` - if the path string cannot be converted to a Path.

See Also:

`FileSystem.getPath(java.lang.String, java.lang.String...)`

relativize

```
Path relativize(Path other)
```

Constructs a relative path between this path and a given path.

Relativization is the inverse of `resolution`. This method attempts to construct a `relative` path that when `resolved` against this path, yields a path that locates the same file as the given path. For example, on UNIX, if this path is `"/a/b"` and the given path is `"/a/b/c/d"` then the resulting relative path would be `"c/d"`. Where this path and the given path do not have a `root` component, then a relative path can be constructed. A relative path cannot be constructed if only one of the paths have a root component. Where both paths have a root component then it is implementation dependent if a relative path can be constructed. If this path and the given path are `equal` then an *empty path* is returned.

For any two `normalized` paths *p* and *q*, where *q* does not have a root component,

> *p*`.relativize(`*p* `.resolve(`*q*`)).equals(`*q*`)`

When symbolic links are supported, then whether the resulting path, when resolved against this path, yields a path that can be used to locate the `same` file as `other` is implementation dependent. For example, if this path is `"/a/b"` and the given path is `"/a/x"` then the resulting relative path may be `"../x"`. If `"b"` is a symbolic link then is implementation dependent if `"a/b/../x"` would locate the same file as `"/a/x"`.

Parameters:

`other` - the path to relativize against this path

Returns:

the resulting relative path, or an empty path if both paths are equal

Throws:

`IllegalArgumentException` - if `other` is not a `Path` that can be relativized against this path

toUri

`URI toUri()`

Returns a URI to represent this path.

This method constructs an absolute `URI` with a `scheme` equal to the URI scheme that identifies the provider. The exact form of the scheme specific part is highly provider dependent.

In the case of the default provider, the URI is hierarchical with a `path` component that is absolute. The query and fragment components are undefined. Whether the authority component is defined or not is implementation dependent. There is no guarantee that the `URI` may be used to construct a `java.io.File`. In particular, if this path represents a Universal Naming Convention (UNC) path, then the UNC server name may be encoded in the authority component of the resulting URI. In the case of the default provider, and the file exists, and it can be determined that the file is a directory, then the resulting `URI` will end with a slash.

The default provider provides a similar *round-trip* guarantee to the `File` class. For a given `Path` *p* it is guaranteed that

```
Path.of(p.toUri()).equals(p .toAbsolutePath())
```

so long as the original `Path`, the `URI`, and the new `Path` are all created in (possibly different invocations of) the same Java virtual machine. Whether other providers make any guarantees is provider specific and therefore unspecified.

When a file system is constructed to access the contents of a file as a file system then it is highly implementation specific if the returned URI represents the given path in the file system or it represents a *compound* URI that encodes the URI of the enclosing file system. A format for compound URIs is not defined in this release; such a scheme may be added in a future release.

Returns:

the URI representing this path

Throws:

`IOError` - if an I/O error occurs obtaining the absolute path, or where a file system is constructed to access the contents of a file as a file system, and the URI of the enclosing file system cannot be obtained

`SecurityException` - In the case of the default provider, and a security manager is installed, the `toAbsolutePath` method throws a security exception.

toAbsolutePath

```
Path toAbsolutePath()
```

Returns a `Path` object representing the absolute path of this path.

If this path is already `absolute` then this method simply returns this path. Otherwise, this method resolves the path in an implementation dependent manner, typically by resolving the path against a file system default directory. Depending on the implementation, this method may throw an I/O error if the file system is not accessible.

Returns:

a `Path` object representing the absolute path

Throws:

`IOError` - if an I/O error occurs

`SecurityException` - In the case of the default provider, a security manager is installed, and this path is not absolute, then the security manager's `checkPropertyAccess` method is invoked to check access to the system property `user.dir`

toRealPath

```
Path toRealPath(LinkOption... options)
         throws IOException
```

Returns the *real* path of an existing file.

The precise definition of this method is implementation dependent but in general it derives from this path, an `absolute` path that locates the `same` file as this path, but with name elements that represent the actual name of the directories and the file. For example, where filename comparisons on a file system are case insensitive then the name elements represent the names in their actual case. Additionally, the resulting path has redundant name elements removed.

If this path is relative then its absolute path is first obtained, as if by invoking the `toAbsolutePath` method.

The `options` array may be used to indicate how symbolic links are handled. By default, symbolic links are resolved to their final target. If the option `NOFOLLOW_LINKS` is present then this method does not resolve symbolic links. Some implementations allow special names such as "`..`" to refer to the parent directory. When deriving the *real path*, and a "`..`" (or equivalent) is preceded by a non-"`..`" name then an implementation will typically cause both names to be removed. When not resolving symbolic links and the preceding name is a symbolic link then the names are only removed if it guaranteed that the resulting path will locate the same file as this path.

Parameters:

`options` - options indicating how symbolic links are handled

Returns:

an absolute path represent the *real* path of the file located by this object

Throws:

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

`SecurityException` - In the case of the default provider, and a security manager is installed, its `checkRead` method is invoked to check read access to the file, and where this path is not absolute, its `checkPropertyAccess` method is invoked to check access to the system property `user.dir`

toFile

```
default File toFile()
```

Returns a `File` object representing this path. Where this `Path` is associated with the default provider, then this method is equivalent to returning a `File` object constructed with the `String` representation of this path.

If this path was created by invoking the `File` `toPath` method then there is no guarantee that the `File` object returned by this method is `equal` to the original `File`.

Implementation Requirements:

The default implementation is equivalent for this path to:

```
    new File(toString());
```

if the `FileSystem` which created this `Path` is the default file system; otherwise an `UnsupportedOperationException` is thrown.

Returns:

a `File` object representing this path

Throws:

`UnsupportedOperationException` - if this `Path` is not associated with the default provider

register

```
WatchKey register(WatchService watcher,
                  WatchEvent.Kind<?>[] events,
                  WatchEvent.Modifier... modifiers)
           throws IOException
```

Registers the file located by this path with a watch service.

In this release, this path locates a directory that exists. The directory is registered with the watch service so that entries in the directory can be watched. The `events` parameter is the events to register and may contain the following events:

- `ENTRY_CREATE` - entry created or moved into the directory
- `ENTRY_DELETE` - entry deleted or moved out of the directory
- `ENTRY_MODIFY` - entry in directory was modified

The `context` for these events is the relative path between the directory located by this path, and the path that locates the directory entry that is created, deleted, or modified.

The set of events may include additional implementation specific event that are not defined by the enum `StandardWatchEventKinds`

The `modifiers` parameter specifies *modifiers* that qualify how the directory is registered. This release does not define any *standard* modifiers. It may contain implementation specific modifiers.

Where a file is registered with a watch service by means of a symbolic link then it is implementation specific if the watch continues to depend on the existence of the symbolic link after it is registered.

Specified by:

`register` in interface `Watchable`

Parameters:

`watcher` - the watch service to which this object is to be registered

`events` - the events for which this object should be registered

`modifiers` - the modifiers, if any, that modify how the object is registered

Returns:

a key representing the registration of this object with the given watch service

Throws:

`UnsupportedOperationException` - if unsupported events or modifiers are specified

`IllegalArgumentException` - if an invalid combination of events or modifiers is specified

`ClosedWatchServiceException` - if the watch service is closed

`NotDirectoryException` - if the file is registered to watch the entries in a directory and the file is not a directory *(optional specific exception)*

`IOException` - if an I/O error occurs

`SecurityException` - In the case of the default provider, and a security manager is installed, the `checkRead` method is invoked to check read access to the file.

register

```
default WatchKey register(WatchService watcher,
                          WatchEvent.Kind<?>... events)
                   throws IOException
```

Registers the file located by this path with a watch service.

An invocation of this method behaves in exactly the same way as the invocation

```
    register(watcher, events, new WatchEvent.Modifier[0]);
```

Usage Example: Suppose we wish to register a directory for entry create, delete, and modify events:

```
    Path dir = ...
    WatchService watcher = ...

    WatchKey key = dir.register(watcher, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
```

Specified by:

`register` in interface `Watchable`

Implementation Requirements:

The default implementation is equivalent for this path to:

```
register(watcher, events, new WatchEvent.Modifier[0]);
```

Parameters:

`watcher` - The watch service to which this object is to be registered

`events` - The events for which this object should be registered

Returns:

A key representing the registration of this object with the given watch service

Throws:

`UnsupportedOperationException` - If unsupported events are specified

`IllegalArgumentException` - If an invalid combination of events is specified

`ClosedWatchServiceException` - If the watch service is closed

`NotDirectoryException` - If the file is registered to watch the entries in a directory and the file is not a directory *(optional specific exception)*

`IOException` - If an I/O error occurs

`SecurityException` - In the case of the default provider, and a security manager is installed, the `checkRead` method is invoked to check read access to the file.

iterator

```
default Iterator<Path> iterator()
```

Returns an iterator over the name elements of this path.

The first element returned by the iterator represents the name element that is closest to the root in the directory hierarchy, the second element is the next closest, and so on. The last element returned is the name of the file or directory denoted by this path. The `root` component, if present, is not returned by the iterator.

Specified by:

`iterator` in interface `Iterable<Path>`

Implementation Requirements:

The default implementation returns an `Iterator<Path>` which, for this path, traverses the `Path`s returned by `getName(index)`, where `index` ranges from zero to `getNameCount() - 1`, inclusive.

Returns:

an iterator over the name elements of this path

compareTo

`int compareTo(Path other)`

Compares two abstract paths lexicographically. The ordering defined by this method is provider specific, and in the case of the default provider, platform specific. This method does not access the file system and neither file is required to exist.

This method may not be used to compare paths that are associated with different file system providers.

Specified by:

`compareTo` in interface `Comparable<Path>`

Parameters:

`other` - the path compared to this path.

Returns:

zero if the argument is `equal` to this path, a value less than zero if this path is lexicographically less than the argument, or a value greater than zero if this path is lexicographically greater than the argument

Throws:

`ClassCastException` - if the paths are associated with different providers

equals

`boolean equals(Object other)`

Tests this path for equality with the given object.

If the given object is not a Path, or is a Path associated with a different `FileSystem`, then this method returns `false`.

Whether or not two path are equal depends on the file system implementation. In some cases the paths are compared without regard to case, and others are case sensitive. This method does not access the file system and the file is not required to exist. Where required, the `isSameFile` method may be used to check if two paths locate the same file.

This method satisfies the general contract of the `Object.equals` method.

Overrides:

`equals` in class `Object`

Parameters:

`other` - the object to which this object is to be compared

Returns:

`true` if, and only if, the given object is a `Path` that is identical to this `Path`

See Also:

`Object.hashCode()`, `HashMap`

hashCode

```
int hashCode()
```

Computes a hash code for this path.

The hash code is based upon the components of the path, and satisfies the general contract of the `Object.hashCode` method.

Overrides:

`hashCode` in class `Object`

Returns:

the hash-code value for this path

See Also:

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

toString

`String toString()`

Returns the string representation of this path.

If this path was created by converting a path string using the `getPath` method then the path string returned by this method may differ from the original String used to create the path.

The returned path string uses the default name `separator` to separate names in the path.

Overrides:

`toString` in class `Object`

Returns:

the string representation of this path

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, 2024, 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.