21.3 Working with `Path` Objects

The `java.nio.file.Path` interface provides a myriad of methods to query and manipulate `Path` objects. In this section, selected methods from the `Path` interface are presented for working on `Path` objects. A majority of these methods perform syntactic operations on the path string contained in a `Path` object. As `Path` objects are immutable, the methods return a new `Path` object, making it possible to use this distinctive style of method chaining. There is also no requirement that the path string in a `Path` object must refer to an existing resource. Very few methods enforce this requirement, and if they do, they will throw a checked `IOException` if that it is not the case.

Querying `Path` Objects

The following methods of the `java.nio.file.Path` interface can be used for querying a `Path` object for its various properties. The names of most methods reflect their operation. The description in the API of these methods and Example 21.1 below should aid in understanding their functionality.

`String toString()`

Returns the text representation of this path.

`boolean isAbsolute()`

Determines whether this path is an absolute path or not.

`FileSystem getFileSystem()`

Returns the file system that created this object. Each `Path` object is created with respect to a file system.

`Path getFileName()`

Returns the name of the directory entry denoted by this path as a `Path` object—that is, the *last* name element in this path which denotes either a file or a directory.

`Path getParent()`

Returns the parent path, or null if this path does not have a parent—that is, logically going up one level in the directory hierarchy from the current position given by this path. It returns `null` if this `Path` does not have a parent—for example, if the path denotes the root component or it is a relative path that comprises a single name element, as there is no parent in these cases.

`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 getNameCount()`

Returns the number of name elements in the path. It returns the value *`n`*, where *`n`*`-1` is the index of the last name element and the first name element has index `0`. Note that the root component is not included in the count of the name elements.

`Path getName(int index)`

Returns a name element of this path as a `Path` object, where the name element closest to the root component has index `0`.

`Path subpath(int beginIndex, int endIndex)`

Returns a relative `Path` that is a subsequence of the name elements of this path, where `beginIndex` is inclusive but `endIndex` is exclusive (i.e., name elements in the range `[beginIndex, endIndex)`). Illegal indices will result in an `IllegalArgumentException`.

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

Determine whether this path starts with either the given path or a `Path` object constructed from the given `other` string. The methods can be used for conditional handing of paths.

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

Determine whether this path ends with either the given path or a `Path` object constructed from the given `other` string. These methods can be used for conditional handing of paths.

Example 21.1 illustrates methods for querying a `Path` object. An absolute path is created at (1). The output from the program shows the result of executing methods for text representation at (2), determining whether a path is absolute at (3), which file system created the path at (4), accessing name elements at (5), and testing the prefix and the suffix of a path at (6).

Example 21.1 *Querying* Path *Objects*

```
import java.nio.file.FileSystem;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

public class QueryingPaths {

  public static void main(String[] args) {
    FileSystem dfs = FileSystems.getDefault();     // The default file system
    String nameSeparator = dfs.getSeparator();     // The name separator

    Path absPath = Path.of(nameSeparator, "a", "b", "c");               // (1)

    System.out.printf("toString(): %s%n",   absPath);                   // (2)
    System.out.printf("isAbsolute(): %s%n", absPath.isAbsolute());      // (3)
    System.out.printf("getFileSystem(): %s%n",
                       absPath.getFileSystem().getClass().getName());   // (4)

    System.out.println("\n***Accessing Name Elements:");                // (5)
    System.out.printf("getFileName(): %s%n",  absPath.getFileName());
    System.out.printf("getParent(): %s%n",    absPath.getParent());
    System.out.printf("getRoot(): %s%n",      absPath.getRoot());
    System.out.printf("getNameCount(): %d%n", absPath.getNameCount());

    List<Path> pl = new ArrayList<>();
    absPath.forEach(p -> pl.add(p));
    System.out.printf("List of name elements: %s%n",  pl);

    System.out.printf("getName(0): %s%n",     absPath.getName(0));
    System.out.printf("subpath(0,2): %s%n",   absPath.subpath(0,2));

    System.out.println("\n***Path Prefix and Suffix:");                 // (6)
    System.out.printf("startsWith(\"%s\"): %s%n",
                      nameSeparator + "a",
                      absPath.startsWith(nameSeparator + "a"));
    System.out.printf("endsWith(\"b/c\"): %s%n",
                      absPath.endsWith("b/c"));
  }
}
```

Possible output from the program:

```
toString(): /a/b/c
isAbsolute(): true
getFileSystem(): sun.nio.fs.MacOSXFileSystem

***Accessing Name Elements:
getFileName(): c
getParent(): /a/b
getRoot(): /
getNameCount(): 3
List of name elements: [a, b, c]
```

```
getName(0): a
subpath(0,2): a/b

***Path Prefix and Suffix:
startsWith("/a"): true
endsWith("b/c"): true
```

Converting `Path` Objects

The `Path` interface provides many methods for converting paths. A majority of these methods manipulate the path strings syntactically and do not assume that the path strings of the `Path` objects denote actual directory entries in the file system.

`Path toAbsolutePath()`

Returns a `Path` object representing the absolute path of this `Path` object. If this `Path` object represents a relative path, an absolute path is constructed by appending it to the absolute path of the current directory. If this `Path` is an absolute path, it just returns this `Path` object.

`Path normalize()`

Returns a `Path` object that is created from this `Path` object with redundant name elements eliminated. This typically involves eliminating the `"."` and `"`*`dir`*`/.."` strings in this `Path` object, as these do not change the hierarchy of a path. The method applies the elimination procedure repeatedly until all such redundancies are eliminated.

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

These method resolve the given `Path` object (either specified or created from the specified string) against this `Path` object.

If the `other Path` object represents an absolute path, the `other Path` object is returned as the result, regardless of whether this `Path` object represents an absolute or a relative path. Otherwise, the method creates a result `Path` object by *joining* the `other Path` object to this `Path` object.

`default Path resolveSibling(Path other)`
`default Path resolveSibling(String other)`

Resolve the given `Path` object (either specified or created from the specified string) against this `Path` object's *parent* path. That is, they resolve the given `Path` object by calling the `resolve(other)` method on this `Path` object's parent path.

`Path relativize(Path other)`

Constructs a relative `Path` object between this `Path` object and the given `other Path` object. The constructed relative `Path` object when resolved against this `Path` object should yield a path that represents the same directory entry as the given `other Path` object.

A relative Path object can only be constructed when this Path object and the given Path object *both* represent either absolute paths or relative paths.

If this path is /w/x and the given path is /w/x/y/z, the resulting relative path is y/z. That is, the given path /w/x/y/z and the resulting relative path y/z represent the same directory entry.

For two Path objects that are equal, the empty path is returned.

The relativize() method is the inverse of the resolve() method.

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

Returns a Path object that represents the *real* path of an existing directory entry. Note that this method throws an IOException if the path does not exist in the file system. It converts the path to an absolute path and removes any redundant elements, and does not follow symbolic links if the enum constant LinkOption.NOFOLLOW_LINKS is specified (p. 1301).

Converting a Path to an Absolute Path

The method toAbsolutePath() of the Path interface converts a path to an absolute path. Table 21.2 illustrates how this method works for Path objects declared in the second column. A print statement that calls the method on each path declaration, analogous to the one below, creates the text representation of the resulting absolute path shown in the rightmost column. The current directory has the absolute path /a/b.

```
System.out.println(absPath1.toAbsolutePath());      // (1) /a
```

In Table 21.2, (1) and (2) show that if the Path object already represents an absolute path, the same Path object is returned. If the Path object represents a relative path, as shown at (3), (4), and (5), the resulting absolute path is created by appending the relative path to the absolute path of the current directory. The path need not exist in the file system, and the method does not attempt to clean up the path string of the resulting Path object—in contrast to the normalize() method (p. 1299).

Table 21.2 *Converting to an Absolute Path*

	Absolute path of the current directory: /a/b **Path**	**Text representation of the absolute path returned by the toAbsolutePath() method**
(1)	`Path absPath1 = Path.of("/a");`	/a
(2)	`Path absPath2 = Path.of("/a/b/c");`	/a/b/c
(3)	`Path relPath1 = Path.of("d");`	/a/b/d
(4)	`Path relPath2 = Path.of("../f");`	/a/b/../f
(5)	`Path relPath3 = Path.of("./../g");`	/a/b/./../g

Normalizing a Path

The `normalize()` method of the `Path` interface removes redundancies in a `Path` object. For example, the current directory designator `"."` is redundant in a `Path` object, as it does not add any new level to the path hierarchy. Also the string `"dir/.."` in a path is redundant, as it implies going one level down in the path hierarchy and then one level up again—not changing the hierarchy represented by the path.

Table 21.3 illustrates how the `normalize()` method converts the `Path` objects declared in the second column. A print statement that calls the method on each path declaration, analogous to the one below, creates the text representation of the resulting path shown in the rightmost column.

```
System.out.println(path1.normalize());       // (1) a/b/c
```

The numbers below refer to the rows in Table 21.3.

(1) All occurrences of the current directory designator `"."` are redundant and are eliminated from the path.

(2) The occurrences of the redundant string `"a/.."` are eliminated from the path.

(3) The occurrences of the parent directory designator `".."` are significant in this case, as each occurrence implies traversing one level up the path hierarchy.

(4) All occurrences of the current directory designator `"."` are eliminated from the path, but occurrences of the parent directory designator `".."` are significant.

(5) The occurrence of the redundant current directory designator `"."` is eliminated from the path, resulting in an empty path whose text representation is the empty string.

Because of redundancies in a path, comparison on paths is best performed on normalized paths (p. 1303).

Table 21.3 *Normalizing Paths*

	Path	**Text representation of the path returned by the `normalize()` method**
(1)	`Path path1 = Path.of("./a/./b/c/.");`	`a/b/c`
(2)	`Path path2 = Path.of("a/../a/../b");`	`b`
(3)	`Path path3 = Path.of("../../d");`	`../../d`
(4)	`Path path4 = Path.of("./.././../.");`	`../..`
(5)	`Path path5 = Path.of(".");`	*empty string*

Resolving Two Paths

Table 21.4 illustrates how the `resolve()` method of the `Path` interface performs resolution between two paths. The four combinations of mixing absolute and relative paths when calling the `resolve()` method are represented by the rows (R1 and R2) and the columns (C1 and C2) in Table 21.4. The results shown are obtained by executing a print statement that calls the `resolve()` method, analogous to the one below, for each combination.

```
System.out.println(absPath1.resolve(absPath2));       // (R1, C1)
```

If the given path is an absolute path, it is returned as the result, as in column C1, regardless of whether the path on which the method is invoked is an absolute or a relative path. Otherwise, the method creates a result path by appending the given path to the path on which the method is invoked, as in column C2.

In the special case when the given path is an empty path, the method returns the path on which it was invoked:

```
Path anyPath = Path.of("/a/n/y");
Path emptyPath = Path.of("");
System.out.println(anyPath.resolve(emptyPath));       // /a/n/y
```

Note that the paths need not exist to use this method, and the resulting path after resolution is not normalized.

Table 21.4 *Resolving Paths*

***p1*.resolve(*p2*), where *p1* can be absPath1 or relPath1, and where *p2* can be absPath2 or relPath2**		**C1**	**C2**
		`Path absPath2 = Path.of("/c");`	`Path relPath2 = Path.of("../e/f");`
R1	`Path absPath1 = Path.of("/a/b");`	`/c`	`/a/b/../e/f`
R2	`Path relPath1 = Path.of("d");`	`/c`	`d/../e/f`

Constructing the Relative Path between Two Paths

Table 21.5 illustrates how the `relativize()` method of the `Path` interface constructs a relative path between this path and the given path, so that the resulting relative path denotes the same directory entry as the given path.

The four combinations of mixing absolute and relative paths when calling the `relativize()` method are represented by the rows (R1 and R2) and the columns (C1 and C2) in Table 21.5. The results shown are obtained by executing a print statement that calls the `relativize()` method, analogous to the one below, for each combination.

```
System.out.println(absPath1.revitalize(absPath2));       // (R1, C1)
```

For the case (R1, C1) where both paths are absolute paths in Table 21.5, the resulting relative path is constructed relative to the root of the directory hierarchy. The relative path between the absolute path `/a/b` and the given absolute path `/c` is `../../c`, indicating traversing two levels up from the path `/a/b` to the root and joining with the given path /c. Both the resulting relative path ../../c and the given path /c denote the same directory entry.

For the case (R2, C2) where both paths are relative paths in Table 21.5, the resulting relative path is constructed relative to the *current directory*. The relative path between the relative path d and the given relative path e/f is ../e/f, indicating traversing one level up from the path d to the current directory and joining with the given path e/f. Both the resulting relative path ../e/f and the given path e/f denote the same directory entry.

From Table 21.5, we see that an IllegalArgumentException is thrown when both paths are neither absolute paths nor relative paths, as it is not possible to create a relative path between paths that do not satisfy this criteria.

The code below shows the relationship between the relativize() and the resolve() methods:

```
Path p = Path.of("/a/b");
Path other = Path.of("/a/b/c/d");
Path q = p.relativize(other);                                // c/d
System.out.println(p.relativize(p.resolve(q)).equals(q));   // true
System.out.println(p.resolve(q).equals(other));             // true
```

Note that the paths need not exist to use this method, as it operates syntactically on the path strings.

Table 21.5 *Constructing a Relative Path between Two Paths*

***p1*.relativize(*p2*), where *p1* can be absPath1 or relPath1, and where *p2* can be absPath2 or relPath2**		**C1**	**C2**
		`Path absPath2 = Path.of("/c");`	`Path relPath2 = Path.of("e/f");`
R1	`Path absPath1 = Path.of("/a/b");`	../../c	IllegalArgumentException
R2	`Path relPath1 = Path.of("d");`	IllegalArgumentException	../e/f

Link Option

The enum type LinkOption defines how symbolic links should be handled by a file operation. This enum type defines only one constant, shown in Table 21.6. If the constant NOFOLLOW_LINKS is specified in a method call, symbolic links are not followed by the method.

The enum type LinkOption implements both the CopyOption interface (p. 1308) and the OpenOption interface (p. 1314). Many file operations declare a variable arity parameter of one of these interface types or just the enum type LinkOption, making it possible to configure their operation.

Table 21.6 *Link Option*

Enum `java.nio.file.LinkOption` implements the `java.nio.file.CopyOption` and the `java.nio.file.OpenOption` interfaces	**Description**
`NOFOLLOW_LINKS`	Do not follow symbolic links.

Converting a Path to a Real Path

The `toRealPath()` method of the `Path` interface converts this path to an absolute path that denotes the same directory entry as this path. The name elements in the path must represent actual directories in the file system or the method throws an `IOException`. It accepts a variable arity parameter of the enum type `java.nio.file.LinkOption` (Table 21.6). If the variable arity parameter is not specified, symbolic links are followed to their final target.

The `toRealPath()` method performs several operations to construct the real path:

- If this path is a relative path, it is first converted to an absolute path.
- Any redundant name elements are removed to create a new path. In other words, it normalizes the result path.
- If `LinkOption.NOFOLLOW_LINKS` is specified, any symbolic links are not followed.

Table 21.7 show the results of calling the `toRealPath()` method on selected paths. Since the method throws an `IOException`, it should typically be called in a `try-catch` construct.

```
try {
  Path somePath = Path.of("some/path");
  Path realPath = somePath.toRealPath(LinkOption.NOFOLLOW_LINKS);
  System.out.println(realPath);
} catch (NoSuchFileException nsfe) {
  nsfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Table 21.7 *Converting to a Real Path*

	Current directory: `/book/chap01` **The symbolic link `./alias_appendixA` has the target `/book/appendixA`.**	**Text representation of the `Path` returned by the `toRealPath()` method**
(1)	`Path currDir = Path.of(".");`	`/book/chap01`
(2)	`Path parentDir = Path.of("..");`	`/book`
(3)	`Path path3 = Path.of("./examples/.././examples/D.java");`	`/book/chap01/examples/D.java`

Table 21.7 *Converting to a Real Path (Continued)*

	Current directory: /book/chap01 The symbolic link ./alias_appendixA has the target /book/appendixA.	Text representation of the Path returned by the toRealPath() method
(4)	`Path path4 = Path.of("./alias_appendixA");`	`path4.toRealPath()` returns `/book/appendixA`. `path4.toRealPath(LinkOption.NOFOLLOW_LINKS)` returns `/book/chap01/alias_appendixA`.

Comparing Path Objects

The methods in the Path interface for comparing two paths only consult the path strings, and do not access the file system or require that the paths exist in the file system.

`boolean equals(Object other)`

Determines whether this path is equal to the given object by comparing their path strings. It does *not* eliminate any redundancies from the paths before testing for equality. See also the Files.isSameFile() method (p. 1306).

`int compareTo(Path other)`

A Path object implements the Comparable<Path> interface. This method compares two path strings lexicographically according to the established contract of this method. It means paths can be compared, searched, and sorted.

The code below illustrates sorting paths according to their natural order. Comparison of the paths is based purely on comparison of their path strings, as is the equality comparison.

```
Path p1 = Path.of("/", "a", "b", "c", "d");
Path p2 = Path.of("/", "a", "b");
Path p3 = Path.of("/", "a", "b", "c");
Path p4 = Path.of("a", "b");

// Sorting paths according to natural order:
List<Path> sortedPaths = Stream.of(p1, p2, p3, p4)
                               .sorted()
                               .toList();
System.out.println(sortedPaths);
// [/a/b, /a/b/c, /a/b/c/d, a/b]

// Comparing for lexicographical equality:
System.out.println(p2);                          // Absolute path: /a/b
System.out.println(p3.subpath(0, 2));            // Relative path: a/b
System.out.println(p2.equals(p3.subpath(0, 2))); // false
```