Java I/O: Part II 21

Chapter Topics

- Understanding the characteristics of a hierarchical file system
- Creating `Path` objects with the `of()` method of the `Path` interface, the `getPath()` method of the default file system, and the `get()` method of the `Paths` class
- Interoperability of `Path` objects with the `java.io.File` legacy class and the `java.net.URI` class
- Querying `Path` objects: converting, normalizing, resolving, relativizing, and comparing
- Using methods of the `Path` interface to perform operations on directory entries: Existence, uniqueness, copying, moving, and renaming directory entries
- Reading and writing byte and character data to files using `Path` objects and buffered I/O streams
- Understanding the purpose of specifying a variable arity parameter to customize the behavior of file operations with constants defined by the `LinkOption`, `StandardCopyOption`, `OpenOption`, and `FileVisitOption` enum types in the `java.nio.file` package
- Using methods of the `Files` class for accessing specific file attributes: size, kind of entry, file accessibility, last modified timestamp, owner, file permissions, and access by attribute name
- Retrieving read-only file attributes in a bulk operation provided by the `Files.readAttributes()` method, and accessing them using the interfaces `BasicFileAttributes`, `PosixFileAttributes`, and `DosFileAttributes`
- Retrieving readable and updatable file attributes in a bulk operation provided by the `Files.getFileAttributeView()` method that creates *views* represented by the interfaces `BasicFileAttributeView`, `PosixFileAttributeView`, and `DosFileAttributeView`
- Creating directory entries with methods provided by the `Files` class: regular and temporary files, regular and temporary directories, and symbolic links
- Using a stream to read text files (`Files.lines()`)
- Using streams for listing entries in a directory (`Files.list()`), for walking the directory hierarchy (`Files.walk()`), and for finding directory entries (`Files.find()`)

Java SE 17 Developer Exam Objectives	
[9.3] Create, traverse, read, and write Path objects and their properties using java.nio.file API	§21.1, p. 1287 *to* §21.8, p. 1345

Java SE 11 Developer Exam Objectives	
[9.3] Handle file system objects using java.nio.file API	§21.1, p. 1287 *to* §21.8, p. 1345

The standard I/O API in the `java.io` package provides the capabilities for reading and writing various kinds of data using the concept of I/O streams (Chapter 20, p. 1231). However, its support for *managing* files and directories in a file system has been superceded by the newer and enhanced I/O API called *NIO.2* (*Non-blocking I/O*, version 2) in the `java.nio` package.

The primary focus in this chapter is on the support provided by the NIO.2 API for programmatically interacting with a file system for accessing files, managing file attributes, and traversing the file system. Primary support for file I/O and accessing files and directories in a file system is provided by the `java.nio.file` package.

The `java.nio.file.Path` interface allows paths in the file system to be represented programmatically. It provides extensive support for constructing and querying paths, and for how paths can be used to manipulate files and directories in the file system.

The `java.nio.file.Files` utility class provides comprehensive support for file operations, such as querying, creating, deleting, copying, and moving files. In addition, it provides methods for creating streams on files and directories, making it possible to write powerful file operations.

Support for interoperability between the two I/O APIs is also covered, and other auxiliary classes are introduced where they are needed in the chapter.

21.1 Characteristics of a Hierarchical File System

We first look at some characteristics of a hierarchical file system, and introduce the terminology used in this chapter.

Hierarchical File Systems

A *file system* allows persistent storage and organization of data as a *hierarchical* (or *tree*) structure on some external media. A tree structure has a *root component* (also called *root node*) at the top, under which other tree nodes represent files and directories. Each directory can have files and nested directories (often called *subdirectories*). An operative system can have multiple file systems, each of which is identifiable by its root component.

Below is an example of a file system, where the root component is platform dependent. On Unix-based platforms, the root component of the file system is denoted by the slash character (`/`). The root component for Windows-based platforms is a combination of a *volume* name (i.e., file system name), a colon (`:`), and a backslash character (`\`). For example, `C:\` designates the root component of the ubiquitous volume named `C` on a Windows-based platform.

Click here to view code image

```
root_component
|
a
├── b
|   ├── c
|   |   └── d
|   |   |   |
... ... ... ...
```

When it is not necessary to distinguish between a file and a directory in the file system, we will use the term *directory entry* to mean both. Each directory entry in a hierarchical file system is uniquely identifiable by a *path* from the root component to the node representing the directory entry. The path of a directory entry in a file system is specified using the naming conventions of the host system, where *name elements* that comprise the path are separated by a platform-specific *name separator character* or *delimiter*. The name separator character for name elements on Unix-based platforms is the slash character (`/`), whereas on Windows-based platforms, it is the backslash character (`\`). For the most part, we will use conventions for Unix-based platforms.

Two examples of paths are given below, where each path has three name elements.

Click here to view code image

```
/a/b/c          on Unix-based platforms
C:\a\b\c        on Windows-based platforms
```

Absolute and Relative Paths

Directory entries can be referenced using both *absolute* and *relative* paths, but the path naming must follow the conventions of the host platform.

An absolute path starts with the platform-dependent root component of the file system, as in the examples above. All information is contained in an absolute path to reference the directory entry—that is, an absolute path uniquely identifies a directory entry in the file system.

A relative path is without designation of the root component and therefore requires additional path information to locate its directory entry in the file system. Typically, a relative path is interpreted in relation to the *current directory* (see the next subsection). Some examples of relative paths are given below, but they alone are not enough to uniquely identify a directory entry.

Click here to view code image

```
c/d                   on Unix-based platforms
c\d                   on Windows-based platforms
```

Note that in a path, the name elements are all parent directory names, except for the last name element, which can be either a file name or a directory name. The name of a directory entry does not distinguish whether it is a file or a directory in the file system. Although *file extensions* can be used in a file name for readability, it is immaterial for the file system. Care must also be exercised when choosing name elements, as characters allowed are usually platform dependent.

Java programs should not rely on system-specific path conventions. In the next section we will construct paths in a platform-independent way.

Current and Parent Directory Designators

A file system has a notion of a current directory which changes while traversing in the file system. The *current directory* is designated by the period character (`.`) and its *parent directory* by two periods (`..`). These designators can be used in constructing paths. Given that the current directory is `/a/b` in the directory tree shown earlier, Table 21.1 illustrates relative paths constructed using the current and the parent directory designators, and their corresponding absolute paths.

Table 21.1 *Using Current and Parent Directory Designators*

Relative path (Current directory: `/a/b`)	Absolute path
`./c/d`	`/a/b/c/d`
`.`	`/a/b`
`./..`	`/a`
`./../..`	`/` (i.e., the root component)

Symbolic Links

Apart from regular files, a file system may allow symbolic links to be created. A *symbolic link* (also called *a hard link*, *a shortcut*, or *an alias*) is a special file that acts as a reference to a directory entry, called the *target* of the symbolic link. Creating an *alias* of a directory entry in the file system is similar to creating a symbolic link. Using symbolic links is transparent to the user, as the file system takes care of using the target when a symbolic link is used in a file operation, with one caveat: Deleting the symbolic link does *not* delete the target. In many file operations, it is possible to indicate whether symbolic links should be followed or not.

21.2 Creating `Path` Objects

File operations that a Java program invokes are performed on a file system. Unless the program utilizes a specific file system constructed by the factory methods of the `java.nio.file.FileSystems` utility class, file operations are performed on a file system called the *default file system* that is accessible to the JVM. We can think of the default file system as the *local* file system. The default file system is platform specific and can be obtained as an instance of the `java.nio.file.FileSystem` abstract class by invoking the `getDefault()` factory method of the `FileSystems` utility class.

Click here to view code image

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

File operations can access the default file system directly to perform their operations. The default file system can be queried for various file system properties.

Click here to view code image

```
static FileSystem getDefault()          Declared in java.nio.file.FileSystems
```

Returns the platform-specific default file system. If the method is called several times, it returns the same `FileSystem` instance for the default file system.

Click here to view code image

```
abstract String getSeparator()          Declared in java.nio.file.FileSystem
```

Returns the platform-specific name separator for name elements in a path, represented as a string.

Paths in the file system are programmatically represented by objects that implement the `Path` interface. The `Path` interface and various other classes provide factory methods that create `Path` objects (see below). A `Path` object is also platform specific.

A `Path` object implements the `Iterable<Path>` interface, meaning it is possible to traverse over its name elements from the first name element to the last. It is also immutable and therefore thread-safe.

In this section, we look at how to create `Path` objects. A `Path` object can be queried and manipulated in various ways, and may not represent an existing directory entry in the file system (p. 1294). A `Path` object can be used in file operations to access and manipulate the directory entry it denotes in the file system (p. 1304).

The methods below can be used to create `Path` objects. The methods are also shown in **Figure 21.1**, together with the relationship between the different classes of these methods. Note in particular the interoperability between the `java.nio.file.Path` interface that represents a path and the two classes `java.io.File` and `java.net.URI`. The class `java.io.File` represents a pathname in the standard I/O API and the class `java.net.URI` represents a *Uniform Resource Identifier* (*URI*) that identifies a resource (e.g., a file or a website).

Click here to view code image

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

These methods are declared in the `java.nio.file.Path` interface.

Click here to view code image

```
abstract Path getPath(String first, String... more)
```

This method is declared in the `java.nio.file.FileSystem` class.

Click here to view code image

```
static Path get(String first, String... more)
static Path get(URI uri)
```

These static methods are declared in the `java.nio.file.Paths` class.

```
Path toPath()
```

This method is declared in the `java.io.File` class.

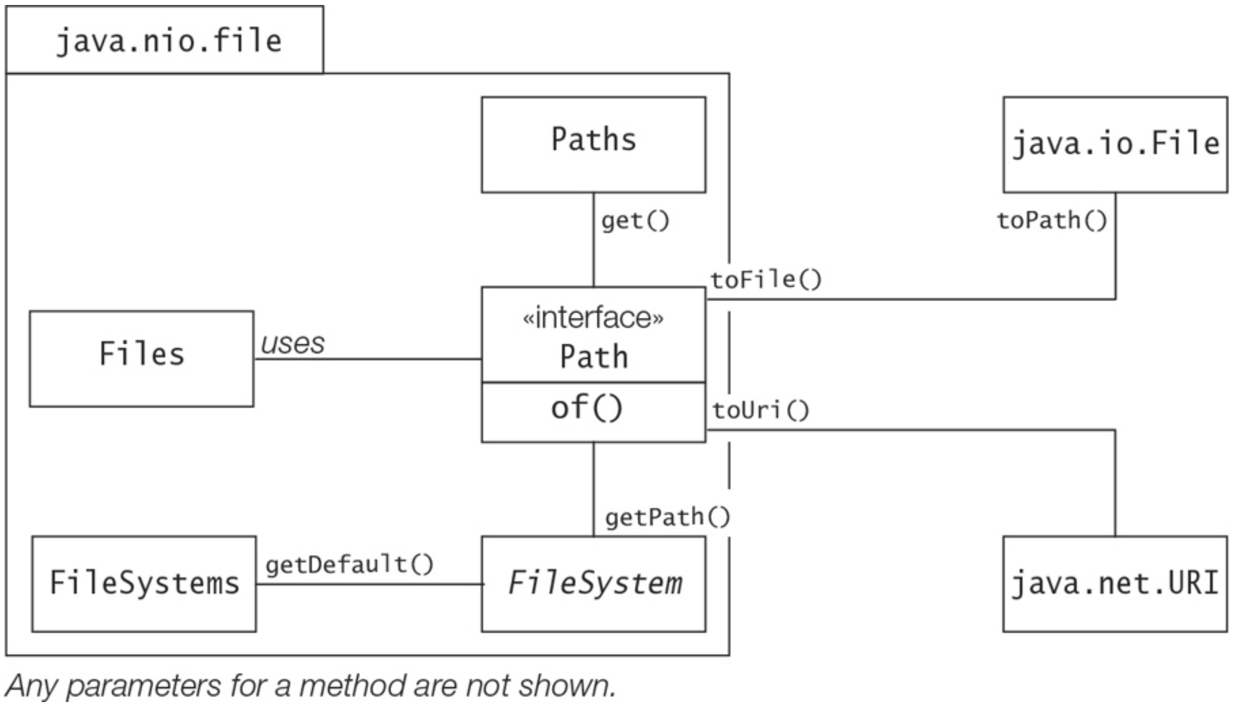


Figure 21.1 *Creating* `Path` *Objects*

Creating `Path` Objects with the `Path.of()` Method

The simplest way to create a `Path` is to use the static factory method `Path.of(String first, String... more)`. It joins the `first` string with any strings in the variable arity parameter `more` to create a path string that is the basis of constructing a `Path` object. This method creates a `Path` object in accordance with the default file system.

For instance, the default file system can be queried for the platform-specific name separator for name elements in a path.

The three absolute paths below are equivalent on a Unix platform, as they create a `Path` object based on the same path string. The `nameSeparator` string is the platform-specific name separator obtained from the default file system. At (1) and (2) below, only the `first` string parameter is specified in the `Path.of()` method, and it is the basis for constructing a `Path` object. However, specifying the variable arity parameter where possible is recommended when a `Path` object is constructed as shown at (3), as joining of the name elements is implicitly done using the platform-specific name separator. The `equals()` methods simply checks for equality on the path string, not on any directory entry denoted by the `Path` objects.

Click here to view code image

```
FileSystem dfs = FileSystems.getDefault();     // Obtain the default file system.
String nameSeparator = dfs.getSeparator();     // The name separator for a path.

Path absPath1 = Path.of("/a/b/c");                          // (1) /a/b/c
Path absPath2 = Path.of(nameSeparator + "a" +               // (2) /a/b/c
                        nameSeparator + "b" +
                        nameSeparator + "c");
Path absPath3 = Path.of(nameSeparator, "a", "b", "c");     // (3) /a/b/c
```

```
System.out.println(absPath1.equals(absPath2) &&
                   absPath2.equals(absPath3));             // true
```

The two absolute paths below are equivalent on a Windows platform, as they create `Path` objects based on the same path string. Note that the backslash character must be escaped with a second backslash in a string. Otherwise, it will be interpreted as starting an escape sequence (§2.1, p. 38).

Click here to view code image

```
Path absPath4 = Path.of("C:\\a\\b\\c");                    // (4) C:\a\b\c
Path absPath5 = Path.of("C:", "a", "b", "c");              // (5) C:\a\b\c
```

The `Path` created below is a relative path, as no root component is specified in the arguments.

Click here to view code image

```
Path relPath1 = Path.of("c", "d");                         //  c/d
```

Often we need to create a `Path` object to denote the current directory. This can be done via a *system property* named `"user.dir"` that can be looked up, as shown at (1) below, and its value used to construct a `Path` object, as shown at (2). The path string of the current directory can be used to create paths relative to the current directory, as shown at (3).

Click here to view code image

```
String pathOfCurrDir = System.getProperty("user.dir");  // (1)
Path currDir = Path.of(pathOfCurrDir);                  // (2)
Path relPath = Path.of(pathOfCurrDir, "d");             // (3) <curr-dir-path>/d
```

Creating `Path` Objects with the `Paths.get()` Method

The `Paths` utility class provides the `get(String first, String... more)` static factory method to construct `Path` objects. In fact, this method invokes the `Path.of(String first, String... more)` convenience method to obtain a `Path`.

Click here to view code image

```
Path absPath7 = Paths.get(nameSeparator, "a", "b", "c");
Path relPath3 = Paths.get("c", "d");
```

Creating `Path` Objects Using the Default File System

We have seen how to obtain the default file system that is accessible to the JVM:

Click here to view code image

```
FileSystem dfs = FileSystems.getDefault();     // The default file system
```

The default file system provides the `getPath(String first, String... more)` method to construct `Path` objects. In fact, the `Path.of(String first, String... more)` method is a convenience method that invokes the `FileSystem.getPath()` method to obtain a `Path` object.

Click here to view code image

```
Path absPath6 = dfs.getPath(nameSeparator, "a", "b", "c");
Path relPath2 = dfs.getPath("c", "d");
```

Interoperability with the `java.io.File` Legacy Class

An object of the legacy class `java.io.File` can be used to query the file system for information about a file or a directory. The class also provides methods to create, rename, and delete directory entries in the file system. Although there is an overlap of functionality between the `Path` interface and the `File` class, the `Path` interface is recommended over the `File` class for new code. The interoperability between a `File` object and a `Path` object allows the limitations of the legacy class `java.io.File` to be addressed.

Click here to view code image

```
File(String pathname)
Path toPath()
```

This constructor and this method of the `java.io.File` class can be used to create a `File` object from a pathname and to convert a `File` object to a `Path` object, respectively.

```
default File toFile()
```

This method of the `java.nio.file.Path` interface can be used to convert a `Path` object to a `File` object.

The code below illustrates the round trip between a `File` object and a `Path` object:

Click here to view code image

```
File file = new File(File.separator + "a" +
                     File.separator + "b" +
                     File.separator + "c");         // /a/b/c
```

```
// File --> Path, using the java.io.File.toPath() instance method
Path fileToPath = file.toPath();                    // /a/b/c

// Path --> File, using the java.nio.file.Path.toFile() default method.
File pathToFile = fileToPath.toFile();              // /a/b/c
```

Interoperability with the `java.net.URI` Class

A URI consists of a string that identifies a resource, which can be a local resource or a remote resource. Among other pertinent information, a URI specifies a *scheme* (e.g., `file`, `ftp`, `http`, and `https`) that indicates what protocol to use to handle the resource. The examples of URIs below show two *schema*: `file` and `http`. We will not elaborate on the syntax of URIs, or different schema, as it is beyond the scope of this book.

Click here to view code image

```
file:///a/b/c/d                      // Scheme: file, to access a local file.
http://www.whatever.com              // Scheme: http, to access a remote website.
```

Click here to view code image

```
URI(String str) throws URISyntaxException
static URI create(String str)
```

This constructor and this static method in the `java.net.URI` class create a `URL` object based on the specified string. The second method is preferred when it is known that the URI string is well formed.

Click here to view code image

```
// Create a URI object, using the URL.create(String str) static factory method.
URI uri1 = URI.create("file:///a/b/c/d");   // Local file.
```

The following methods can be used to convert a `URI` object to a `Path` object. The `Paths.get(URI uri)` static factory method actually invokes the `Path.of(URI uri)` static factory method.

Click here to view code image

```
// URI --> Path, using the Path.of(URI uri) static factory method.
Path uriToPath1 = Path.of(uri1);        // /a/b/c/d
```

```
// URI --> Path, using the Paths.get(URI uri) static factory method.
Path uriToPath2 = Paths.get(uri1);     // /a/b/c/d
```

The following method in the `Path` interface can be used to convert a `Path` object to a `URI` object:

Click here to view code image

```
// Path --> URI, using the Path.toUri() instance method.
URI pathToUri = uriToPath1.toUri();    // file:///a/b/c/d
```

Interoperability between a `Path` object and a `URI` object allows an application to leverage both the NIO.2 API and the network API.

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`.

Click here to view code image

```
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 `Illegal-ArgumentException`.

Click here to view code image

```
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.

Click here to view code image

```
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*

Click here to view code image

```
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:

Click here to view code image

```
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.

Click here to view code image

```
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.

Click here to view code image

```
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.

Click here to view code image

```
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 `Link-Option.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`.

Click here to view code image

```
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 =`	`/a`

	`Path.of("/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.

Click here to view code image

```
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.

Click here to view code image

```
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:

Click here to view code image

```
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.

Click here to view code image

```
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:

Click here to view code image

```
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.

Click here to view code image

```
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`
(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.

Click here to view code image

```
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.

Click here to view code image

```
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
```

21.4 Operations on Directory Entries

The static methods of the `Files` class interact with the file system to access directory entries in the file system. The methods make heavy use of `Path` objects that denote directory entries.

Characteristics of Methods in the `Files` Class

It is worth noting certain aspects of the static methods in the `Files` class, as this will aid in using and understanding the operations they perform.

Handling System Resources

The NIO.2 API uses many resources, such as files and streams, that should be closed after use to avoid any degradation of performance due to lack of system resources. As these resources implement the `java.io.Closeable` interface, they are best handled in a `try`-with-resources statement that guarantees to close them after its execution.

Handling Exceptions

Errors are bound to occur when interacting with the file system. Numerous errors can occur, and among the most common errors are the following:

- A required file or directory does not exist in the file system.
- Permissions to access a file are incorrect.

A majority of the static methods in the `Files` class throw a `java.io.IOException` that acts as the catchall for various I/O errors. Its subclass, `java.nio.file.NoSuchFile-Exception`, unequivocally makes clear the cause of the exception. As these exceptions are checked exceptions, code using these methods should diligently handle these exceptions with either a `try`-`catch`-`finally` construct or a `throws` clause.

For brevity, the exception handling may be omitted in some code presented in this chapter.

Handling Symbolic Links

The methods of the `Files` class are savvy with regard to symbolic links. Following of symbolic links is typically indicated by specifying or omitting the constant `LinkOption.NOFOLLOW_LINKS` for the variable arity parameter of the method (see below).

Specifying Variable Arity Parameters

Many static methods in the `Files` class have a variable arity parameter. Such a parameter allows zero or more options to customize the operation implemented by the method. For example, the following method takes into consideration symbolic links depending on whether or not the variable arity parameter `options` is specified:

Click here to view code image

```
// Method header:
static boolean exists(Path path, LinkOption... options)  // Variable arity param.

// Method calls:
Path path = Path.of("alias");
boolean result1 = Files.exists(path);            // Follow symbolic links.
boolean result2 = Files.exists(path,
                     LinkOption.NOFOLLOW_LINKS);  // Do not follow symbolic links.
```

Executing Atomic Operations

Certain file operations can be performed as *atomic operations*. Such an operation guarantees the integrity of any resource it uses. It runs independently and cannot be interrupted by other operations that might be running concurrently. See the *atomic move* operation (p. 1305).

Determining the Existence of a Directory Entry

The following static methods of the `Files` class test the existence or nonexistence of a directory entry denoted by a path.

Click here to view code image

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

The first method tests whether a directory entry exists. The second method tests whether the directory entry denoted by this path does not exist.

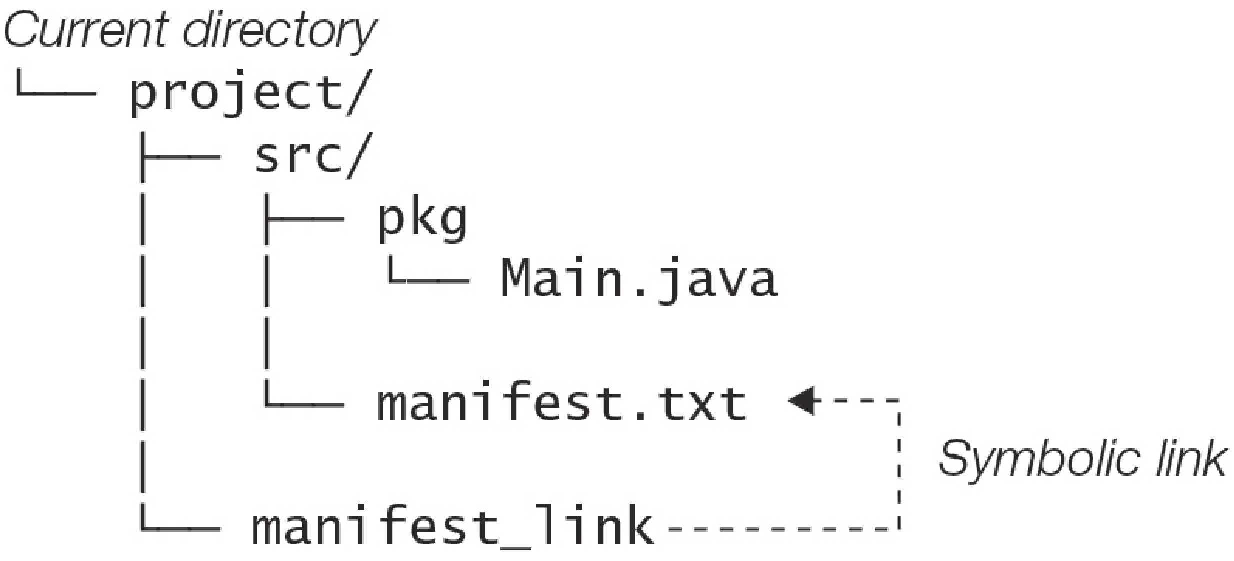
These methods normalize the path, and do not follow symbolic links if the enum constant `LinkOption.NOFOLLOW_LINKS` is specified for the `options` variable arity parameter.

These methods are *not* complements of each other. Both return `false` if they are not able to determine whether a directory entry exists or not. This can occur due to lack of access permissions.

Note that the result returned by these methods is immediately outdated. The outcome of subsequent access of the directory entry is unpredictable, as concurrently running threads might change the conditions after the method returns.

Although a `Path` object can represent a hypothetical path, ultimately the existence of the directory entry it denotes must be verified by interacting with the file system.

Given the following directory hierarchy, the code below demonstrates what result is returned by the `exists()` and `notExists()` methods of the `Files` class.



Click here to view code image

```
Path path1 = Path.of("project", "src", "pkg", "Main.java");
System.out.println(Files.exists(path1));                           // true
System.out.println(Files.notExists(path1));                        // false

Path path2 = Path.of("project", "..", "project", ".", "src", "pkg", "Main.java");
System.out.println(Files.exists(path2));                           // true
System.out.println(Files.notExists(path2));                        // false
```

```
Path path3 = Path.of("project", "readme.txt");
System.out.println(Files.exists(path3));                               // false
System.out.println(Files.notExists(path3));                            // true
```

Given that the path `./project/manifest_link` is a symbolic link to the path `./project/src/manifest.txt`, the code below demonstrates following symbolic links in the `exists()` method.

Click here to view code image

```
Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");

boolean result4 = Files.exists(target);                                // (1)
boolean result5 = Files.exists(symbLink);                              // (2)
boolean result6 = Files.exists(symbLink, LinkOption.NOFOLLOW_LINKS); // (3)

System.out.println("target: " + result4);                              // (1a) true
System.out.println("symbLink->target: " + result5);                    // (2a) true
System.out.println("symbLink_NOFOLLOW_LINKS: " + result6);             // (3a) true
```

Note that (1) and (2) above are equivalent. The existence of the target is tested at (2) as the symbolic link is followed by default. Whereas at (3), the existence of the symbolic link itself is tested, since the enum constant `LinkOption.NOFOLLOW_LINKS` is specified.

Uniqueness of a Directory Entry

The method `isSameFile()` in the `Files` class can be used to check whether two paths denote the *same* directory entry. It does not take into consideration other aspects of the directory entry, like its file name or contents.

Click here to view code image

```
static boolean isSameFile(Path path1, Path path2) throws IOException
```

Determines whether the two paths denote the same directory entry. If the paths are equal, it returns `true` and does not check whether the paths exist.

This method normalizes the paths and follows symbolic links.

It implements an *equivalence relation* (which is *reflexive*, *symmetric*, and *transitive*) for non-`null` paths, if the file system and the directory entries do not change.

The numbers below refer to corresponding lines in the code to illustrate the workings of the `isSameFile()` method:

(1) Paths are always normalized, as in the case of `path2`.
(2) Symbolic links are always followed, as in the case of `symbLink`.
(3) Only paths are compared. Paths passed to the method are not equal.
(4) Equal paths return `true`, and their existence is not checked. The path `./Main.java` does not exist.
(5) The paths must exist in the file system, if they are not equal. The path `./Main.java` does not exist, resulting in a `java.nio.file.NoSuchFileException.`

Click here to view code image

```
Path path1 = Path.of("project", "src", "pkg", "Main.java");
Path path2 = Path.of("project", "..", "project", ".", "src", "pkg", "Main.java");

Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");

System.out.println(Files.isSameFile(path1, path2));            // (1) true
System.out.println(Files.isSameFile(symbLink, target));        // (2) true
System.out.println(Files.isSameFile(path1, target));           // (3) false
System.out.println(Files.isSameFile(Path.of("Main.java"),
                                    Path.of("Main.java")));    // (4) true
System.out.println(Files.isSameFile(path1,
                           Path.of("Main.java")));  // (5) NoSuchFileException
```

Deleting Directory Entries

The methods `delete()` and `deleteIfExists()` in the `Files` class can be used for deleting directory entries.

Click here to view code image

```
static void delete(Path path) throws IOException
static boolean deleteIfExists(Path path) throws IOException
```

Delete a directory entry denoted by the `path`.

If the path does not exist, the first method throws a `NoSuchFileException`, but the second method does not.

Deleting a symbolic link only deletes the link, and not the target of the link.

To delete a directory, it must be empty or a `java.nio.file.DirectoryNotEmpty-Exception` is thrown.

Consider the following paths that exist:

Click here to view code image

```
Path projDir  = Path.of("project");
Path target   = Path.of("project", "src", "manifest.txt");
Path symbLink = Path.of("project", "manifest_link");
```

The `delete()` method throws a `NoSuchFileException`, if the path does not exist.

Click here to view code image

```
Files.delete(symbLink);                  // Exists. Link deleted, not target.
Files.delete(Path.of("Main.java"));     // Does not exist: NoSuchFileException
```

The `deleteIfExists()` method does not throw a `NoSuchFileException`. It indicates the result by the `boolean` return value.

Click here to view code image

```
System.out.println(Files.deleteIfExists(target));               // Exists.
                                                                 //  Deleted: true
System.out.println(Files.deleteIfExists(Path.of("Main.java"))); // Does not
                                                                 //   exist: false
```

In order to delete a directory, it must be empty. The directory `./project` is not empty. Both methods throw a `DirectoryNotEmptyException`.

Click here to view code image

```
Files.delete(projDir);                           // DirectoryNotEmptyException
System.out.println(Files.deleteIfExists(projDir)); // DirectoryNotEmptyException
```

Also keep in mind that deleting a directory entry might not be possible, if another program is using it.

Copy Options

The enum types `LinkOption` (Table 21.6) and `StandardCopyOption` (Table 21.8) implement the `CopyOption` interface. These options can be used to configure copying and moving of directory entries. A variable arity parameter of type `CopyOption...` is declared by the `copy()` and `move()` methods of the `Files` class that support specific constants of the `LinkOption` and `StandardCopyOption` enum types.

Table 21.8 *Standard Copy Options*

Enum `java.nio.file.StandardCopyOption` implements the `java.nio.file.CopyOption` interface	**Description**
`REPLACE_EXISTING`	Replace a file if it exists.
`COPY_ATTRIBUTES`	Copy file attributes to the new file (p. 1321).
`ATOMIC_MOVE`	Move the file as an atomic file system operation—that is, an operation that is either performed uninterrupted in its entirety, or it fails.

Copying Directory Entries

The overloaded `copy()` methods of the `Files` class implement copying contents of files. Two of the `copy()` methods can be configured by specifying *copy options*.

Click here to view code image

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

Copies a `source` directory entry to the `destination` directory entry. It returns the path to the `destination` directory entry. The default behavior is outlined below, but can be configured by copy options:

- If `destination` already exists or is a symbolic link, copying *fails*.
- • If `source` and `destination` are the same, the method completes without any copying.
- If `source` is a symbolic link, the target of the link is copied.
- If `source` is a directory, just an *empty* `destination` directory is created.

The following copy options can be specified to configure the default copying behavior:

- `StandardCopyOption.REPLACE_EXISTING`: If the `destination` exists, this option indicates to replace the `destination` if it is a file or an empty directory. If the `destination` exists and is a symbolic link, it indicates to replace the symbolic link and not its target.

- `StandardCopyOption.COPY_ATTRIBUTES`: This option indicates to copy the file attributes of the `source` to the `destination`. However, copying of attributes is platform dependent.
- `LinkOption.NOFOLLOW_LINKS`: This option indicates not to follow symbolic links. If the `source` is a symbolic link, then the symbolic link is copied and not its target.

Click here to view code image

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

Copies all bytes from an input stream to a `destination` path, and returns the number of bytes copied. The input stream will be at the end of the stream after copying, but may not be because of I/O errors.

By default, if the `destination` path already exists or is a symbolic link, copying *fails*.

It can be configured by the following copy option:

- `StandardCopyOption.REPLACE_EXISTING`: If the `destination` path exists and is a file or an empty directory, this option indicates to replace the `destination`. If the `destination` exists and is a symbolic link, this option indicates to replace the symbolic link and not its target.

Click here to view code image

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

Copies all bytes from the `source` to the `output` stream, and returns the number of bytes copied. It may be necessary to flush the `output` stream.

Note that this `copy()` method cannot be configured.

The output stream may be in an inconsistent state because of I/O errors.

The first thing to keep in mind is that the copy methods do *not* create the intermediate directories that are in the destination path. Given the destination path `./project/archive/destFile`, the parent path `./project/archive` must exist. Otherwise, a `NoSuchFileException` is thrown.

Note also that copying fails if the destination path already exists—a `FileAlreadyExistsException` is thrown, unless the method is qualified by the enum constant `StandardCopyOption.REPLACE_EXISTING`.

Copy and Replace Directory Entries

Consider copying the source file:

```
./project/src/pkg/Main.java
```

to the destination file:

Click here to view code image

```
./project/archive/src/pkg/Main.java
```

The numbers below refer to corresponding lines in the code below to illustrate copying of files:

(1) Creates the `Path` object that denotes the source file.
(2) Creates the `Path` object for the *parent* path of the destination file. This parent path must exist.
(3) Resolves the parent path with the source pathname, so that if the destination file is to have the same name as the source file, its `Path` object can be constructed from the parent path and the file name of the source file.
Items (4) through (6) illustrate the scenario when calling the `copy()` method successively.
(4) Creates the destination file and copies the contents of the source file to the destination file.
(5) Overwrites the contents of the destination file with the contents of the source file.
(6) Fails, as the destination file already exists.

Click here to view code image

```
Path source = Path.of("project", "src", "pkg", "Main.java");              // (1)
Path parentDestinationPath = Path.of("project", "archive", "src", "pkg");  // (2)
Path destination = parentDestinationPath.resolve(source.getFileName());    // (3)

Files.copy(source, destination);      // (4) OK. Destination file does not exist.
Files.copy(source, destination,       // (5) OK. Destination file replaced.
           StandardCopyOption.REPLACE_EXISTING);
Files.copy(source, destination);      // (6) FileAlreadyExistsException
```

The `copy()` method does *not* copy the entries in a source directory to a destination directory. The code below attempts to copy the source directory:

```
./project/src
```

to the destination directory:

```
./project/backup
```

Possible outcomes of the following copying operation can be any of the bulleted options listed below:

Click here to view code image

```
Path srcDir  = Path.of("project", "src");
Path destDir = Path.of("project", "backup");
Files.copy(srcDir, destDir, StandardCopyOption.REPLACE_EXISTING);
```

- If an entry named `backup` does not exist in the `project` directory, an empty directory named `backup` is created.
- If an entry named `backup` exists in the `project` directory and it is an empty directory, a new empty directory named `backup` is created.
- If an entry named `backup` exists in the `project` directory and it is a file, the file is deleted and an empty directory named `backup` is created.
- If an entry named `backup` exists in the `project` directory and it is a non-empty directory, the copying operation fails with a `DirectoryNotEmptyException`.

Another special case to consider is copying a file to a directory. Consider the scenario when copying the source file:

```
./project/src/pkg/Main.java
```

to the destination directory:

```
./project/classes
```

The code and possible outcomes are outlined below.

Click here to view code image

```
Path srcFile = Path.of("project", "src", "pkg", "Main.java");
Path destDir = Path.of("project", "classes");
Files.copy(srcFile, destDir, StandardCopyOption.REPLACE_EXISTING);
```

- If an entry named `classes` does not exist in the `project` directory, a file named `classes` is created and the contents of the source file are copied to the file `classes`.
- If an entry named `classes` exists in the `project` directory and it is a file, the file is deleted, a new file named `classes` is created, and the contents of the source file are copied to the file `classes`.
- If an entry named `classes` exists in the `project` directory and it is an empty directory, the directory is deleted, a file named `classes` is created, and the contents of the source

file are copied to the file `classes`.

- If an entry named `classes` exists in the `project` directory and it is a non-empty directory, the copying operation fails with a `DirectoryNotEmptyException`.

Copy Files Using I/O Streams

The `Files` class provides methods to copy files using byte I/O streams from the `java.io` package (§20.2, p. 1234). Bytes can be copied from a source file to an `Input-Stream` and from an `OutputStream` to a destination file. We demonstrate copying where input streams and output streams are assigned to files. See also reading and writing files using paths (p. 1314).

The code below reads bytes from the input file `project/src/pkg/Util.java` using a `BufferedInputStream` and writes the bytes to the output file path `project/backup/Util.java`.

Click here to view code image

```
String inputFileName  = "project/src/pkg/Util.java";
Path outputFilePath = Path.of("project", "backup", "Util.java");
try (var fis = new FileInputStream(inputFileName);
     var bis = new BufferedInputStream(fis)) {
  long bytesCopied = Files.copy(bis, outputFilePath,
                                StandardCopyOption.REPLACE_EXISTING);
  System.out.println("Bytes copied: " + bytesCopied);      // Bytes copied: 103
}
```

The code below copies bytes from the input file path `project/backup/Util.java` to the output file `project/archive/src/pkg/Util.java` using a `BufferedOutputStream`.

Click here to view code image

```
Path inputFilePath  = Path.of("project", "backup", "Util.java");
String outputFileName = "project/archive/src/pkg/Util.java";
try (var fos = new FileOutputStream(outputFileName);
     var bos = new BufferedOutputStream(fos)) {
  long bytesCopied = Files.copy(inputFilePath, bos);
  System.out.println("Bytes copied: " + bytesCopied);      // Bytes copied: 103
}
```

The following statement can be used to print the contents of a file to the standard output:

Click here to view code image

```
Files.copy(inputFilePath, System.out);     // Prints file content to standard out.
```

In general, any `InputStream` or `OutputStream` can be used in the respective `copy()` methods.

Moving and Renaming Directory Entries

The `move()` method of the `Files` class implements moving and renaming directory entries. The method can be configured by specifying *copy options*.

The `move()` method emulates the copying behavior of the `copy()` method. But in contrast to the `copy()` method, the `move()` method deletes the source if the operation succeeds. Both methods allow the destination to be overwritten, if the constant `StandardCopyOption.REPLACE_EXISTING` is specified.

Click here to view code image

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

Moves or renames the `source` to the `destination`. After moving, the `source` is deleted. The method returns the path to the `destination`. The default behavior is outlined below, but can be configured by copy options:

- If `destination` already exists, the move *fails*.
- If `source` and `destination` are the same, the method has no effect.
- If `source` is a symbolic link, the target of the link is moved.
- If `source` is an empty directory, the empty directory is moved to the `destination`.

The following copy options can be specified to configure the default moving behavior:

- `StandardCopyOption.REPLACE_EXISTING`: If the `destination` exists, this option indicates to replace the `destination` if it is a file or an empty directory. If the `destination` exists and is a symbolic link, this option indicates to replace the symbolic link and not its target.
- `StandardCopyOption.ATOMIC_MOVE`: The move is performed as an *atomic file system operation*. It is implementation specific whether the move is performed if the destination exists or whether an `IOException` is thrown. If the move cannot be performed, the method throws an `AtomicMoveNotSupportedException`.

Moving Directory Entries

The code below illustrates moving a file (`./project/src/manifest.txt`) to a new location (`./project/bkup/manifest.txt`). If the file exists at the new location, it is replaced by the source file.

Click here to view code image

```
Path srcFile  = Path.of("project", "src",  "manifest.txt");
Path destFile = Path.of("project", "bkup", "manifest.txt");
Files.move(srcFile, destFile, StandardCopyOption.REPLACE_EXISTING);
```

We can move a directory *and* its hierarchy (`./project/bkup`) to a new location (`./project/archive/backup`). The directory (and its contents) are moved to the new location and renamed (`backup`).

Click here to view code image

```
Path srcDir = Path.of("project", "bkup");
Path destDir = Path.of("project", "archive", "backup");  // Parent path exists.
Files.move(srcDir, destDir);
```

Renaming Directory Entries

The `move()` method can be used to rename a directory entry. The code below illustrates renaming an existing file (`Util.java`). Its name is changed (`UX.java`), but not its contents.

Click here to view code image

```
Path oldFileName  = Path.of("project", "backup", "Util.java");
Path newFileName  = Path.of("project", "backup", "UX.java");
Files.move(oldFileName, newFileName);
```

Analogously, the code below illustrates renaming an existing directory (`backup`). Its name is changed (`bkup`), but not its hierarchy.

Click here to view code image

```
Path oldDirName  = Path.of("project", "backup");
Path newDirName  = Path.of("project", "bkup");
Files.move(oldDirName, newDirName);
```

Atomic Move

The enum constant `StandardCopyOption.ATOMIC_MOVE` can be specified in the `move()` method to indicate an *atomic move*—that is, an operation that is indivisible. It either completes in its entirety or it fails. The upshot of an atomic operation is that other threads will never see incomplete or partial results. An `AtomicMoveNotSupported-Exception` will be thrown if the file system does not support this feature.

In the following code, the file `Util.java` in the directory `./project/src/pkg` is moved in an atomic operation to its new location `./project/archive/src/pkg`.

Click here to view code image

```
Path srcFile = Path.of("project", "src", "pkg", "Util.java");
Path destFile = Path.of("project", "archive", "src", "pkg", "Util.java");
Files.move(srcFile, destFile, StandardCopyOption.REPLACE_EXISTING,
                              StandardCopyOption.ATOMIC_MOVE);
```

21.5 Reading and Writing Files Using Paths

The `Files` class provides methods for reading and writing bytes and characters using I/O streams and `Path` objects.

Methods provided for reading and writing files typically close the file after use.

Open Options

The `java.nio.file.OpenOption` interface is implemented by objects that can be specified as options to configure how a file operation should open or create a file. Methods for writing to files can be configured for this purpose by specifying constants defined by the enum type `java.nio.file.StandardOpenOption` that implements the `OpenOption` interface.

Table 21.9 shows the options defined by constants of the `StandardOpenOption` enum type. Such options are specified as values for the variable arity parameter of type `OpenOption` in methods such as `newBufferedWriter()`, `write()`, `writeString()`, `newOutputStream()`, and `newInputStream()` of the `Files` class.

For write operations, if no options are supplied, it implies that the following options for opening and creating a file are present: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`—meaning the file is opened for writing, created if it does not exist, and truncated to size 0.

Table 21.9 *Selected Standard Open Options*

Enum `java.nio.file.StandardOpenOption` implements the `java.nio.file.OpenOption` interface	**Description**
`READ`	Open the file for read access.
`WRITE`	Open the file for write access.
`APPEND`	If the file is opened for `WRITE` access, write bytes to the end of the file. That is, its previous content is not overwritten.

`TRUNCATE_EXISTING`	If the file already exists and it is opened for `WRITE` access, truncate its length to 0 so that bytes are written from the beginning of the file.
`CREATE`	Open the file if it exists; otherwise, create a new file.
`CREATE_NEW`	Fail if the file already exists; otherwise, create a new file.
`DELETE_ON_CLOSE`	Delete the file when the stream is closed. Typically used for temporary files.

Reading and Writing Character Data

The `Files` class provides methods for reading and writing *character data* to files. These methods can be categorized as follows:

- Methods that create character I/O streams (`BufferedReader`, `BufferedWriter`) chained to a `Path` object that denotes a file. The methods of the buffered reader and writer can then be used to read and write characters to the file, respectively.
- Methods that directly use a `Path` object, and read and write characters to the file denoted by the `Path` object.

Reading and Writing Character Data Using Buffered I/O Streams

The `newBufferedReader()` and `newBufferedWriter()` methods of the `Files` class create buffered readers and writers, respectively, that are chained to a `Path` object denoting a file. Interoperability between character I/O streams in the `java.io` package can then be leveraged to chain appropriate I/O streams for reading and writing character data to a file (§20.3, p. 1241).

Previously we have used constructors of the `BufferedReader` class (§20.3, p. 1251) and the `BufferedWriter` class (§20.3, p. 1250) in the `java.io` package to instantiate buffered readers and writers that are chained to a `Reader` or a `Writer`, respectively. Using the methods of the `Files` class is the recommended practice for creating buffered readers and writers when dealing with text files.

Click here to view code image

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

```
                                            throws IOException
```

Opens the file denoted by the specified `path` for reading, and returns a `BufferedReader` of a default size to read text efficiently from the file, using either the UTF-8 charset or the specified charset to decode the bytes, respectively. Contrast these methods with the constructors of the `java.io.BufferReader` class (§20.3, p. 1251).

Click here to view code image

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

Opens or creates a file denoted by the specified `path` for writing, returning a `BufferedWriter` of a default size that can be used to write text efficiently to the file, using either the UTF-8 charset or the specified charset to encode the characters, respectively. See also constructors of the `java.io.BufferWriter` class (§20.3, p. 1250).

The code at (1) and at (3) in Example 21.2 illustrates writing lines to a text file using a buffered writer and reading lines from a text file using a buffered reader, respectively. The methods `newBufferedWriter()` and `newBufferedReader()` create the necessary buffered writer and reader at (2) and (4), respectively, whose methods are used to write and read the lines from the file.

Example 21.2 *Reading and Writing Text Files*

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class ReadingWritingTextFiles {

  public static void main(String[] args) throws IOException {
    // List of strings:
    List<String> lines = List.of("Guess who got caught?", "Who?",
                                 "NullPointerException.",
                                 "Seriously?", "No. Finally.");
    // Text file:
    String filename = "project/linesOnly.txt";
    Path path = Path.of(filename);

    // Writing lines using buffered writer:                             (1)
    try (BufferedWriter writer =  Files.newBufferedWriter(path)) {   // (2)
```

```
      for(String str : lines) {
        writer.write(str);                   // Write a string.
        writer.newLine();                    // Terminate with a newline.
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    // Read lines using buffered reader:                                (3)
    lines = new ArrayList<>();
    try(BufferedReader reader= Files.newBufferedReader(path)) {      // (4)
      String line = null;
      while ((line = reader.readLine()) != null) {  // EOF when null is returned.
        lines.add(line);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    System.out.printf("Lines read from file \"%s\":%n%s%n", path, lines);


    // Write the list of strings in one operation:
    Files.write(path, lines);                                        // (5)

    // Write the joined lines in one operation:
    String joinedLines = String.join(System.lineSeparator(), lines);
    Files.writeString(path, joinedLines);                            // (6)

    // Read all contents into a String, including line separators:
    String allContent = Files.readString(path);                      // (7)
    System.out.printf("All lines read from file \"%s\":%n%s%n", path, allContent);

    // Read all lines into a list of String:
    lines = Files.readAllLines(path);                                // (8)
    System.out.printf("List of lines read from file \"%s\":%n%s%n", path, lines);
  }
}
```

Output from the program:

Click here to view code image

```
Lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
All lines read from file "project/linesOnly.txt":
Guess who got caught?
Who?
NullPointerException.
Seriously?
No. Finally.
```

```
List of lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
```

Reading and Writing Character Data Using *Path Objects*

The `Files` class provides methods that directly use a `Path` object, and read and write characters to the file denoted by the `Path` object, without the need to specify an I/O stream. These methods also close the file when done.

Click here to view code image

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

Open or create a file denoted by the specified `path`, and writes text `lines` to the file, using either the UTF-8 charset or the specified charset, respectively.

No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Click here to view code image

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

Open or create a file denoted by the specified `path`, and writes characters in the `CharSequence csq` verbatim to the file, using either the UTF-8 charset or the specified charset, respectively.

No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Click here to view code image

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

Read all content from a file denoted by the specified `path` into a string, decoding the bytes to characters using the UTF-8 charset or the specified charset, respectively. The string returned will contain all characters, including line separators. These methods are not recommended for reading large files.

Click here to view code image

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

Read all *lines* from the file denoted by the specified `path`, decoding the bytes to characters using the UTF-8 charset or the specified charset, respectively. These methods are not recommended for reading large files, as these can result in a lethal `java.lang.OutOfMemoryError`.

The code at (5) to (8) in **Example 21.2** illustrates methods of the `Files` class that directly write and read character data to a file denoted by a `Path` object.

The `write()` method at (5) writes an `Iterable` (in this case, the `List` of `String`) to the file in one operation. It automatically terminates each string written with a newline.

Click here to view code image

```
Files.write(path, lines);                                 // (5)
```

The `writeString()` method at (6) writes the contents of a single `CharSequence` (in this case, the string `joinedLines`) to the file. The strings in the `lines` list are joined with an appropriate line separator by the `String.join()` method. The end result written to the file is again lines of text.

Click here to view code image

```
String joinedLines = String.join(System.lineSeparator(), lines);
Files.writeString(path, joinedLines);                      // (6)
```

The `readString()` method at (7) reads the *whole* file in one operation. It returns all characters read in a string, *including any line separators*.

Click here to view code image

```
String allContent = Files.readString(path);                // (7)
```

The `readAllLines()` method at (8) reads all *text lines* in the file in one operation, returning the lines read in a `List` of `String`.

Click here to view code image

```
lines = Files.readAllLines(path);                          // (8)
```

The methods in the `Files` class for writing and reading directly from a file denoted by a `Path` object should be used with care, as they might not scale up when handling large files.

This is especially the case regarding the `readString()` and `readAll-Lines()` methods that read the whole file in one fell swoop. A better solution for reading text files using streams is provided later in the chapter (p. 1345).

Reading and Writing Bytes

The `Files` class also provides methods for reading and writing *bytes* to files. These methods that can be categorized as follows:

- Methods that create low-level *byte* I/O streams (`InputStream`, `OutputStream`) chained to a `Path` object that denotes a file. The methods of the I/O streams can then be used to read and write bytes to the file.
- Methods that directly use a `Path` object, and read and write bytes to the file denoted by the `Path` object.

Reading and Writing Bytes Using I/O Streams

The `newInputStream()` and `newOutputStream()` methods of the `Files` class create an input stream and an output stream, respectively, that are chained to a `Path` object denoting a file. Interoperability between I/O streams in the `java.io` package can then be leveraged to chain appropriate I/O streams for reading and writing data to a file (§20.2, p. 1234).

Click here to view code image

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

Opens a file denoted by the specified `path`, and returns an input stream to read from the file. No options implies the `READ` option.

Click here to view code image

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

Opens or creates a file denoted by the specified `path`, and returns an output stream that can be used to write bytes to the file. No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Previously we have seen how the `copy()` methods of the `Files` class use byte I/O streams for reading and writing bytes to files (p. 1311).

The code at (1) in Example 21.3 is a reworking of Example 20.1, p. 1237, to copy bytes from a source file to a destination file using an explicit byte buffer. The main difference is that the input stream and the output stream on the respective files are created by the `newInput-Stream()` and `newOutputStream()` methods of the `Files` class, based on `Path` objects that denote the files, rather than on file I/O streams. As before, the methods `read()` and `write()` of the `InputStream` and `OutputStream` classes, respectively, are used to read and write the bytes from the source file to the destination file using a byte buffer.

Example 21.3 *Reading and Writing Bytes*

Click here to view code image

```
import java.io.*;
import java.nio.file.*;

public class ReadingWritingBytes {
  public static void main(String[] args) {
    // Source and destination files:
    Path srcPath  = Path.of("project", "source.dat");
    Path destPath = Path.of("project", "destination.dat");

    try (InputStream is = Files.newInputStream(srcPath);                   // (1)
         OutputStream os = Files.newOutputStream(destPath))  {
      byte[] buffer = new byte[1024];
      int length = 0;

      while((length = is.read(buffer, 0, buffer.length)) != -1) {
        os.write(buffer, 0, length);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    try {
      // Reads the file contents into an array of bytes:
      byte[] allBytes = Files.readAllBytes(srcPath);                       // (2)

      // Writes an array of bytes to a file:
      Files.write(destPath, allBytes);                                     // (3)
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Reading and Writing Bytes Using *Path Objects*

The `Files` class provides methods that directly use a `Path` object, and read and write bytes to the file denoted by the `Path` object, without the need to specify a file I/O stream. The method `readAllBytes()` reads all bytes from a file into a `byte` array in one operation, and the method `write()` writes the bytes in a `byte` array to a file. These methods also close the file when done.

Click here to view code image

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

Reads all the bytes from the file denoted by the specified `path`. The bytes are returned in a `byte` array.

Click here to view code image

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

Writes bytes in a `byte` array to the file denoted by the specified `path`. No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

The code at (2) and (3) in **Example 21.3** shows yet another example of copying the contents of a source file to a destination file. The `readAllBytes()` and `write()` methods accomplish the task in a single call to each method.

Click here to view code image

```
byte[] allBytes = Files.readAllBytes(srcPath);                        // (2)
...
Files.write(destPath, allBytes);                                      // (3)
```

Note that these methods are meant for simple cases, and not for handling large files, as data is handled using an array of bytes.

21.6 Managing File Attributes

Useful metadata is associated with directory entries in a file system—for example, the file permissions that indicate whether the entry is readable or writable, or whether it is a symbolic link, and its size. Such metadata in the file system is often referred to as *file attributes*. Managing file attributes is a separate concern from the data that is stored in files.

There are basically two approaches provided by the NIO.2 API for managing file attributes:

- Accessing *individual file attributes* associated with a directory entry in the file system (p. 1321)

- Accessing a *set of file attributes* associated with a directory entry in the file system as a *bulk operation* (p. 1328)

Accessing Individual File Attributes

The `Files` class provides a myriad of static methods to access individual file attributes of a directory entry. It is a good idea to consult the code in Example 21.4 as we take a closer look at the relevant methods in this subsection. Since methods in the `Files` class can throw an `IOException`, the `main()` method specifies a `throws` clause with this exception.

Example 21.4 *Accessing Individual Attributes*

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

import static java.lang.System.out;
import static java.nio.file.attribute.PosixFilePermission.*;

public class IndividualFileAttributes {

  public static void main(String[] args) throws IOException {

    Path fPath = Path.of("project", "src", "pkg", "Main.java");
    out.println("File: " + fPath);

    out.println("Accessing Individual File Attributes:");
    out.println("size file (bytes): " + Files.size(fPath));            // (1)
    out.println("isDirectory:       " + Files.isDirectory(fPath));       // (2)
    out.println("isRegularFile:     " + Files.isRegularFile(fPath));     // (3)
    out.println("isSymbolicLink:    " + Files.isSymbolicLink(fPath));    // (4)
    out.println();

    out.println("isReadable:        " + Files.isReadable(fPath));        // (5)
    out.println("isWritable:        " + Files.isWritable(fPath));        // (6)

    out.println("isExecutable:      " + Files.isExecutable(fPath));      // (7)
    out.println("isHidden:          " + Files.isHidden(fPath));          // (8)
    out.println();

    out.println("getLastModifiedTime: " + Files.getLastModifiedTime(fPath));// (9)
    out.println("getOwner:            " + Files.getOwner(fPath));         // (10)
    out.println();

    // Get the POSIX file permissions for the directory entry:
    Set<PosixFilePermission> filePermissions
        = Files.getPosixFilePermissions(fPath);                          // (11)
```

```
      out.println("getPosixFilePermissions (set): " + filePermissions);      // (12)
      out.println("getPosixFilePermissions (string): "
                + PosixFilePermissions.toString(filePermissions));           // (13)

      // Get the group of the directory entry:
      out.println("getAttribute-group:  " + Files.getAttribute(fPath,        // (14)
                                                           "posix:group"));
      out.println();

      // Update last modified time for the directory entry.                    (15)
      long currentTime = System.currentTimeMillis();
      FileTime timestamp = FileTime.fromMillis(currentTime);
      Files.setLastModifiedTime(fPath, timestamp);

      // Set new owner for the directory entry.                                (16)
      FileSystem fs = fPath.getFileSystem();   // File system that created the path.
      UserPrincipalLookupService upls
          = fs.getUserPrincipalLookupService();// Obtain service to look up user.
      UserPrincipal user = upls.lookupPrincipalByName("khalid"); // User lookup.
      Files.setOwner(fPath, user);                               // Set user.

      // Set POSIX file permissions for the directory entry:                   (17)
      Set<PosixFilePermission> newfilePermissions
          = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, GROUP_WRITE);   // (18a)
      //Set<PosixFilePermission> newfilePermissions
      //   = PosixFilePermissions.fromString("rw-rw----");                  // (18b)
      Files.setPosixFilePermissions(fPath, newfilePermissions);             // (19)
      filePermissions = Files.getPosixFilePermissions(fPath);
      out.println("getPosixFilePermissions (set): " + filePermissions);
      out.println("getPosixFilePermissions (string): "
            + PosixFilePermissions.toString(filePermissions));

      // Setting the value of a file attribute by its attribute name.
      Files.setAttribute(fPath, "lastAccessTime", timestamp);               // (20)
    }
  }
```

Possible output from the program:

Click here to view code image

```
File: project/src/pkg/Main.java
Accessing Individual File Attributes:
size file (bytes): 13

isDirectory:       false
isRegularFile:     true
isSymbolicLink:    false

isReadable:        true
isWritable:        true
```

```
isExecutable:      false
isHidden:          false

getLastModifiedTime: 2021-08-06T10:28:47.416033Z
getOwner:            khalid

getPosixFilePermissions (set): [OTHERS_READ, OWNER_WRITE, OWNER_READ, GROUP_READ]
getPosixFilePermissions (string): rw-r--r--
getAttribute-group:  admin

getPosixFilePermissions (set): [GROUP_WRITE, OWNER_WRITE, OWNER_READ, GROUP_READ]
getPosixFilePermissions (string): rw-rw----
```

Determining the File Size

The method `size()` in the `Files` class is called at (1) in **Example 21.4** to determine the size of the file denoted by the `Path` object. If the `Path` object denotes a directory, the method returns the size of the directory file and *not* the size of entries in the directory.

Click here to view code image

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

Returns the size of a file (in bytes). The size of files that are not regular files is unspecified, as it is implementation specific.

Determining the Kind of Directory Entry

The following methods in the `Files` class are called at (2), (3), and (4) in **Example 21.4** to determine what kind of directory entry is denoted by the `Path` object.

Click here to view code image

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

Return `true` if the directory entry is a directory, a regular file, or a symbolic link, respectively. They return `false` if the directory entry does not exist, or is not of the expected kind, or it is not possible to determine what kind of directory entry it is. In the first two methods, symbolic links are followed by default, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

Determining File Accessibility

The methods in the `Files` class shown below are called at (5), (6), (7), and (8) in **Example 21.4**, respectively, to determine accessibility of the directory entry denoted by the `Path` object.

Click here to view code image

```
static boolean isReadable(Path path)
static boolean isWritable(Path path)
static boolean isExecutable(Path path)
```

Test whether a file is readable, writable, or executable, respectively. The file must exist and the JVM must have the appropriate privileges to access the file.

Note that the result returned by these method is immediately outdated. The outcome of subsequent attempts to access the file is not guaranteed, as concurrently running threads might change the conditions after the method returns.

Click here to view code image

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

Determines whether or not a file is considered *hidden*. The exact definition of a hidden file is platform specific. On Unix platforms, files whose name begins with a period character (`.`) are considered to be hidden.

Timestamp for Last Modification Time

Three different timestamps are associated with a directory entry, whose purpose is evident from their names: *last modified time*, *last access time*, and *creation time*. The timestamps are represented by the `java.nio.file.attribute.FileTime` class that provides the following methods for interoperability with `Instant` objects and with `long` values in milliseconds.

Click here to view code image

```
class java.nio.file.attribute.FileTime

static FileTime from(Instant instant)
static FileTime fromMillis(long value)
```

These static methods create a `FileTime` object representing the same point of time value on the timeline as the specified `Instant` object, or a `FileTime` object from the `long value` that specifies the number of milliseconds since the epoch (`1970-01-01T00:00:00Z`), respectively.

```
Instant toInstant()
long toMillis()
```

These instance methods convert this `FileTime` object to an `Instant` or to a `long` value in milliseconds from the epoch, respectively.

The `Files` class only provides static methods to read and update the last modified time of a directory entry. In **Example 21.4**, the statement at (9) prints the last modified time of the directory entry. The code at (15) sets the last modified time of the directory entry to the current time.

Click here to view code image

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

Return or update the timestamp for the last modified time attribute of a directory entry, respectively. The timestamp is represented by the class `java.nio .file.attribute.FileTime`.

The first method follows symbolic links by default, unless the constant `Link-Option.NOFOLLOW_LINKS` is specified.

Accessing the Owner

The `Files` class only provides static methods to get and set the *owner* of a directory entry (i.e., one with a user account and appropriate access permissions). In **Example 21.4**, the statement at (10) prints the name of the owner of the directory entry. The code at (16) executes the necessary steps to obtain a user that can be set as the owner of the directory entry. This involves querying the file system to obtain the user look service and using the service to look up the user by name. We leave it to the reader to discover the exciting details from the API of the classes involved.

Click here to view code image

```
static UserPrincipal getOwner(Path path, LinkOption... options)
static Path setOwner(Path path, UserPrincipal owner)
```

Return or update the owner of a file, respectively.

Handling File Permissions

For a directory entry, POSIX-based file systems (*Portable Operating System Interface*) typically define *read*, *write*, and *execute* permissions for the *owner*, the *group* that the owner belongs to, and for *others*. In Java, these nine permissions are represented by the enum type `PosixFilePermission` (**Table 21.10**).

A human-readable form of file permissions affords interoperability with the enum type `PosixFilePermission`. This form is specified as a string of nine characters, where characters are interpreted as three permission groups of three characters. From the start of the string, the first permission group, the second permission group, and the third permission group specify the permissions for the owner, the group, and others, respectively. Each permission group is defined by the following pattern:

```
(r|-)(w|-)(x|-)
```

that is comprised of three groupings, where each grouping `(a|b)` is interpreted as either `a` or `b`. For example, `rwx` and `---` are valid permissions groups, but `w_w` and `xwr` are not. The characters `r`, `w`, and `x` stand for read, write, and execute permissions, respectively, and the character `-` indicates that the permission corresponding to the position of the character `-` is not set.

The set of file permissions created by the following statement:

Click here to view code image

```
Set<PosixFilePermission> permSet1
      = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);
```

is equivalent to the permissions in the string `"rw-r--r--"`.

The utility class `PosixFilePermissions` provides methods for converting between the two forms of specifying file permissions.

Table 21.10 *POSIX File Permissions*

Enum type java.nio.file.attribute.PosixFilePermission	**Description**
`OWNER_EXECUTE`	Execute/search permission, owner

Enum type java.nio.file.attribute.PosixFilePermission	Description
`OWNER_READ`	Read permission, owner
`OWNER_WRITE`	Write permission, owner
`GROUP_EXECUTE`	Execute/search permission, group
`GROUP_READ`	Read permission, group
`GROUP_WRITE`	Write permission, group
`OTHERS_EXECUTE`	Execute/search permission, others
`OTHERS_READ`	Read permission, others
`OTHERS_WRITE`	Write permission, others

Following are methods from the utility class `java.nio.file.attribute.PosixFilePermissions`:

Click here to view code image

```
static Set<PosixFilePermission> fromString(String permStr)
```

Returns the set of permissions corresponding to a given `String` representation. The `permStr` parameter is a `String` representing the permissions, as explained earlier.

Click here to view code image

```
static String toString(Set<PosixFilePermission> perms)
```

Returns the `String` representation of a set of permissions.

Click here to view code image

```
static FileAttribute<Set<PosixFilePermission>>
       asFileAttribute(Set<PosixFilePermission> perms)
```

Creates a `FileAttribute`, encapsulating a copy of the given file permissions, suitable for passing to methods that create files and directories (p. 1339).

The `getPosixFilePermissions()` and `setPosixFilePermissions`() methods of the `Files` class can be used to retrieve and update file permissions of a directory entry, as shown at (11) and (19), respectively. The methods `toString()` and `fromString()` of the `PosixFilePermissions` class at (13) and (18b) convert between a set of `PosixFilePermission` and a string representation of file permissions, respectively. Note that (18a) and (18b) define the same set of file permissions.

Click here to view code image

```
// Get the POSIX file permissions for the directory entry:
Set<PosixFilePermission> filePermissions
    = Files.getPosixFilePermissions(fPath);                             // (11)
out.println("getPosixFilePermissions (set):    " + filePermissions);    // (12)
out.println("getPosixFilePermissions (string): "
           + PosixFilePermissions.toString(filePermissions));          // (13)
...
// Set POSIX file permissions for the directory entry:                    (17)
Set<PosixFilePermission> newfilePermissions
         = EnumSet.of(OWNER_READ, OWNER_WRITE, GROUP_READ, OTHERS_READ);// (18a)
//Set<PosixFilePermission> newfilePermissions
//       = PosixFilePermissions.fromString("rw-r--r--");                // (18b)
Files.setPosixFilePermissions(fPath, newfilePermissions);               // (19)
```

The following methods from the utility class `java.nio.file.Files` can be used for retrieving and updating the POSIX-specific file permissions of a directory entry:

Click here to view code image

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

Returns POSIX permissions of a directory entry as a set of enum type `PosixFilePermission`. By default, symbolic links are followed, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

Click here to view code image

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

Sets the POSIX permissions of a directory entry, given by the parameter `perms`.

Accessing File Attributes through View and Attribute Names

The `getAttribute()` and `setAttribute()` methods of the `Files` class are general methods that can be used to read and update any file attribute by its name. These methods are useful when the `Files` class does not provide a specialized method for a particular file attribute.

The statement at (14) in **Example 21.4** prints the group of the directory entry. The full attribute name of the group attribute is specified as `"posix:group"`, as the group attribute can be accessed via the POSIX file attribute view (**p. 1336**).

The statement at (20) in **Example 21.4** sets the last access time of the directory entry to the current time. The attribute named `"lastAccessTime"` can be accessed via the basic file attribute view that is implied by default (**p. 1334**). The value of this file attribute is the `FileTime` object denoted by the `timestamp` reference.

Click here to view code image

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

Read and set, respectively, the value of a file `attribute` of a directory entry denoted by the given `Path` object. By default, symbolic links are followed, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

The `attribute` parameter has the general format:

```
view-name:attribute-name
```

where the *view-name* can be omitted. Typical view names are `"basic"`, `"posix"`, and `"dos"`. Omitting the *view-name* defaults to `"basic"`. The *attribute-name* is the name of the file attribute—for example, `"lastModifiedTime"` or `"lastAccess-Time"`. Attribute views are discussed in detail later (**p. 1328**).

The second method sets the file `attribute` to the `value` parameter.

Bulk Operations to Retrieve File Attributes

Accessing file attributes individually raises two concerns:

- Accessing an individual file attribute incurs a cost, and frequent such accesses can adversely impact performance.
- File attributes are very much file-system-specific, and therefore not conducive to generalizing over different file systems.

To address these concerns, the NIO.2 API provides *bulk operations* to retrieve file attributes—thus avoiding retrieval of individual file attributes and only handling file-system-specific attributes.

The following three approaches can be used to access file attributes of a directory entry in a file system:

- The methods of the `Files` class can be used to access individual file attributes of a directory entry denoted by a `Path`, as discussed earlier (p. 1321).
- A *bulk operation* using the `readAttributes()` method of the `Files` class can be used to retrieve a *set of file attributes* into a *read-only object* that implements *a file attributes interface*. This read-only object acts as a repository for file attribute values pertaining to the directory entry whose `Path` object was specified in the call to the `readAttributes()` method. The interface `BasicFileAttributes`, and its subinterfaces `PosixFileAttributes` and `DosFileAttributes`, define methods to read the retrieved file attribute values (Table 21.11, p. 1330).
  In the code below at (1), the set of file attributes to read for the directory entry denoted by the `path` parameter is determined by the runtime object `BasicFileAttributes.class`.
  Click here to view code image

  ```
  BasicFileAttributes bfa = Files.readAttributes(path,                    // (1)
                                                 BasicFileAttributes.class);
  ```

  The retrieved values of the file attributes are accessible by querying the `BasicFileAttributes` object that is returned by the `readAttributes()` method (p. 1330).
- A *bulk operation* using the `getFileAttributeView()` method of the `Files` class can be used to retrieve *a set of file attributes* into an *updatable file attribute view* that acts as a repository object. This object can be used to read or update selected file attributes pertaining to the directory entry whose `Path` object was specified in the call to the `getFileAttributeView()` method. The interface `BasicFileAttributeView`, and its subinterfaces `PosixFileAttributeView` and `DosFileAttributeView`, define methods to read and update the retrieved file attributes (Table 21.12, p. 1334).
  In the code below at (6), the set of file attributes to retrieve for the directory entry denoted by the `path` parameter is determined by the runtime object `BasicFileAttributeView.class`.
  Click here to view code image

  ```
  BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                                  BasicFileAttributeView.class);
  ```

The retrieved file attributes can be read and updated by querying the `Basic-FileAttributeView` object that is returned by the `getFileAttributeView()` method (p. 1334).

Details of the API for the `readAttributes()` and the `getFileAttributeView()` methods in the `Files` class are given below. By default, these methods follow symbolic links, unless the constant `LinkOption.NOFOLLOW_LINKS` is specified.

Click here to view code image

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

Reads a *set of read-only file attributes* as a *bulk operation* for the directory entry denoted by the `path` parameter. The file attributes to retrieve are determined by the type parameter `A`. The parameter `type` is the `Class<A>` of the file attributes required to retrieve. The type parameter `A` is typically the interface `BasicFileAttributes`, or one of its subinterfaces `PosixFileAttributes` or `DosFileAttributes` (Table 21.11, p. 1330).

Click here to view code image

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

Reads a *set of file attributes* of a file as a *bulk operation*. It returns a *file attribute view* of a given type, which can be used to *read or update* the retrieved file attribute values. The file attributes to retrieve are determined by the type parameter `V`. The parameter `type` is the `Class<V>` of the file attributes required to retrieve. The type parameter `V` is typically the interface `BasicFileAttributeView`, or one of its subinterfaces `PosixFileAttributeView` or `DosFileAttributeView` — that are all subinterfaces of the `FileAttributeView` interface (Table 21.12, p. 1334).

Click here to view code image

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

Reads a *set of file attributes* as a *bulk operation* for the directory entry denoted by the `path` parameter. It returns a map of file attribute names and their values.

The `attributes` parameter of type `String` has the general format:

```
view-name:attribute-list
```

where the *view-name* can be omitted, and *attribute-list* is a comma-separated list of attribute names. Omitting the *view-name* defaults to `"basic"`. For example, `"*"` will read all `BasicFileAttributes`, and `"lastModifiedTime,lastAccessTime"` will read only last modified time and last access time attributes.

Table 21.11 summarizes the interfaces that provide read-only access to file attribute values. The `BasicFileAttributes` interface defines the basic set of file attributes that are common to many file systems. Its two subinterfaces `PosixFileAttributes` and `DosFileAttributes` define *additional* file attributes associated with POSIX-based and DOS-based file systems, respectively. An object of the appropriate file attributes interface pertaining to a specific directory entry is returned by the `Files.read-Attributes()` method.

Table 21.11 *Interfaces for Read-Only Access to File Attributes*

Read-only file attributes interfaces in the java.nio.file.attribute package.	**Note that when an object of the read-only file attributes interface is created, it is opened on a specific directory entry in the file system. The object provides information about file attributes associated with this directory entry. The methods in these interfaces do not throw a checked exception.**
`BasicFileAttributes`	Provides *read-only* access to a basic set of file attributes that are common to many file systems (p. 1330).
`PosixFileAttributes extends BasicFileAttributes`	In addition to the basic set of file attributes, provides *read-only* access to file attributes associated with POSIX-based file systems (p. 1332).
`DosFileAttributes extends BasicFileAttributes`	In addition to the basic set of file attributes, provides *read-only* access to file attributes associated with DOS/ Windows-based file systems (p. 1333).

The *BasicFileAttributes Interface*

The methods of the `java.nio.file.attribute.BasicFileAttributes` interface reflect the basic set of file attributes that are common to most file systems.

Click here to view code image

```
interface java.nio.file.attribute.BasicFileAttributes

long size()
```

Returns the size of the directory entry (in bytes).

```
boolean isDirectory()
boolean isRegularFile()
boolean isSymbolicLink()
boolean isOther()
```

Determine whether the directory entry is of a specific kind.

Click here to view code image

```
FileTime lastModifiedTime()
FileTime lastAccessTime()
FileTime creationTime()
```

Return the appropriate timestamp for the directory entry.

The method `printBasicFileAttributes()` of the utility class `FileUtils`, shown below, prints the values of the basic file attributes by calling the relevant methods on the `BasicFileAttributes` object that is passed as a parameter.

Click here to view code image

```
// Declared in utility class FileUtils.
public static void printBasicFileAttributes(BasicFileAttributes bfa) {
  out.println("Printing basic file attributes:");
  out.println("lastModifiedTime: " + bfa.lastModifiedTime());
  out.println("lastAccessTime:   " + bfa.lastAccessTime());
  out.println("creationTime:     " + bfa.creationTime());

  out.println("size:             " + bfa.size());
  out.println("isDirectory:      " + bfa.isDirectory());
  out.println("isRegularFile:    " + bfa.isRegularFile());
  out.println("isSymbolicLink:   " + bfa.isSymbolicLink());
  out.println("isOther:          " + bfa.isOther());
  out.println();
}
```

The code below obtains a `BasicFileAttributes` object at (1) that pertains to the file denoted by the `path` reference. The `printBasicFileAttributes()` method is called at (2) with this `BasicFileAttributes` object as the parameter.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
BasicFileAttributes bfa = Files.readAttributes(path,                   // (1)
                                               BasicFileAttributes.class);
FileUtils.printBasicFileAttributes(bfa);                               // (2)
```

Possible output from the code:

Click here to view code image

```
File: project/src/pkg/Main.java
Printing basic file attributes:
lastModifiedTime: 2021-07-23T10:15:34.854Z
lastAccessTime:   2021-07-23T10:16:33.166281Z
creationTime:     2021-07-20T23:03:58Z
size:             116
isDirectory:      false

isRegularFile:    true
isSymbolicLink:   false
isOther:          false
```

Note that the basic file attributes in the `BasicFileAttributes` object are read-only, as the `BasicFileAttributes` interface does not provide any set methods. However, values of updatable file attributes can be changed by appropriate set methods of the `Files` class (p. 1321).

The *PosixFileAttributes Interface*

As the `PosixFileAttributes` interface is a subinterface of the `BasicFileAttributes` interface, a `PosixFileAttributes` object has both the basic set of file attributes and the POSIX-specific file attributes.

Click here to view code image

```
interface java.nio.file.attribute.PosixFileAttributes
          extends BasicFileAttributes
```

```
UserPrincipal owner()
GroupPrincipal group()
```

Return the owner or the group of the directory entry, represented by the interface `java.nio.file.attribute.UserPrincipal` and its subinterface `java.nio.file.attribute.GroupPrincipal`, respectively.

Click here to view code image

```
Set<PosixFilePermission> permissions()
```

Returns a set with a copy of the POSIX permissions for the directory entry. Permissions are defined by the enum type `PosixFilePermission` discussed earlier in this chapter (Table 21.10, p. 1326).

The methods of the subinterface `PosixFileAttributes` augment the basic set of file attributes with the following POSIX-specific attributes: owner, group, and file permissions. The method `printPosixFileAttributes()` in the utility class `FileUtils`, shown below, prints the values of the POSIX-specific file attributes by calling the relevant methods on the `PosixFileAttributes` object that is passed as a parameter.

Click here to view code image

```
// Declared in the utility class FileUtils.
public static void printPosixFileAttributes(PosixFileAttributes pfa) {
  out.println("Printing POSIX-specific file attributes:");
  UserPrincipal user = pfa.owner();
  GroupPrincipal group = pfa.group();
  Set<PosixFilePermission> permissions = pfa.permissions();
  String perms = PosixFilePermissions.toString(permissions);

  out.println("owner:            " + user);
  out.println("group:            " + group);
  out.println("permissions:      " + perms);
  out.println();
}
```

The code below obtains a `PosixFileAttributes` object at (3) that pertains to the file denoted by the `path` reference. Both the `printBasicFileAttributes()` method and the `print-PosixFileAttributes()` method are called at (4) and (5), respectively, with this `PosixFileAttributes` object as the parameter. The call to the `printBasicFile-Attributes()` method at (4) will print the basic file attributes in the `PosixFile-Attributes` object.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
PosixFileAttributes pfa = Files.readAttributes(path,                   // (3)
                                               PosixFileAttributes.class);
FileUtils.printBasicFileAttributes(pfa);                               // (4)
FileUtils.printPosixFileAttributes(pfa);                               // (5)
```

Possible output from the code:

Click here to view code image

```
File: project/src/pkg/Main.java
Printing basic file attributes:
...
Printing POSIX-specific file attributes:
owner:            javadude
group:            admin
permissions:      rw-r--r--
```

Note that both the basic and the POSIX-specific file attributes in the `PosixFileAttributes` object are readonly, as the `PosixFileAttributes` interface does not provide any set methods. However, values of updatable file attributes can be changed by appropriate set methods of the `Files` class (p. 1321).

The `DosFileAttributes` Interface

As the `DosFileAttributes` interface is a subinterface of the `BasicFileAttributes` interface, a `DosFileAttributes` object has both the basic set of file attributes and the DOS-specific file attributes. Its usage is analogous to the `PosixFileAttributes` interface discussed earlier (p. 1332).

Click here to view code image

```
interface java.nio.file.attribute.DosFileAttributes
          extends BasicFileAttributes

boolean isReadOnly()
boolean isSystem()
boolean isArchive()
boolean isHidden()
```

Determine whether the file entry is of a specific kind.

File Attribute Views

Table 21.12 summarizes the *file attribute views* that different interfaces provide for *readable or updatable* access to file attributes, in contrast to the file attributes interfaces in Table 21.11, p. 1330, that allow readonly access. The `BasicFileAttribute-View` interface allows access to the basic set of file attributes that are common to many file systems. Its two subinterfaces, `PosixFileAttributeView` and `DosFileAttributeView`, additionally allow access to file attributes associated with POSIX-based and DOS-based file systems, respectively.

An object of the appropriate view interface pertaining to a specific directory entry is returned by the `Files.getFileAttributeView()` method. All file interface views provide a `readAttributes()` method that returns the readonly file attributes object associated with the view.

Table 21.12 *Selected File Attribute Views*

Updatable file attribute view interfaces in the `java.nio.file.attribute` package.	**Note that when the view is created, it is opened on a specific directory entry in the file system. The view provides information about file attributes associated with this directory entry.**
`AttributeView`	Can read or update non-opaque values associated with directory entries in a file system.
`FileAttributeView extends AttributeView`	Can read or update file attributes.
`FileOwnerAttributeView extends FileAttributeView`	Can read or update the owner.
`BasicFileAttributeView extends FileAttributeView` *Corresponding readonly file attributes interface:* `BasicFileAttributes`	Can read or update a basic set of file attributes. Can obtain a readonly `BasicFileAttributes` object via the view. Can set a timestamp for when the directory entry was last modified, last accessed, and created. (p. 1334)
`PosixFileAttributeView extends BasicFileAttributeView, FileOwnerAttributeView` *Corresponding read-only file attributes interface:* `PosixFileAttributes`	Can read or update POSIX file attributes. Can obtain a read-only `PosixFileAttributes` object via the view. Can set group and file permissions, and update the owner. (p. 1336)
`DosFileAttributeView extends BasicFileAttributeView` *Corresponding read-only file attributes interface:* `DosFileAttributes`	Can read or update DOS file attributes. Can obtain a read-only `DosFileAttributes` object via the view. Can set archive, hidden, read-only, and system attributes. (p. 1338)

The `BasicFileAttributeView` Interface

The `java.nio.file.attribute.BasicFileAttributeView` interface defines a file attribute view for the basic set of file attributes.

Click here to view code image

```
interface BasicFileAttributeView extends FileAttributeView

String name()
```

Returns the name of the attribute view, which in this case is the string `"basic"`.

Click here to view code image

```
BasicFileAttributes readAttributes()
```

Reads the basic file attributes as a bulk operation. The `BasicFileAttributes` object can be used to read the values of the basic file attributes (p. 1330). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328).

Click here to view code image

```
void setTimes(FileTime lastModifiedTime,
              FileTime lastAccessTime,
              FileTime createTime)
```

Updates any or all timestamps for the file's last modified time, last access time, and creation time attributes. If any parameter has the value `null`, the corresponding timestamp is not changed. Note that apart from the `Files.setLast-Modified()` method, there are no methods in the `Files` class for the last access and creation times for a directory entry.

The code below obtains a `BasicFileAttributeView` object at (6) that pertains to the file denoted by the `path` reference. A `BasicFileAttributes` object is obtained at (7), providing read-only access to the basic file attributes, whose values are printed by calling the `printBasicFileAttributes()` method. The last modified time of the directory entry is explicitly read by calling the `lastModifiedTime()` method of the `BasicFileAttributes` object.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
BasicFileAttributeView bfaView = Files.getFileAttributeView(path,      // (6)
                                             BasicFileAttributeView.class);
```

```
  System.out.printf("Using view: %s%n", bfaView.name());

  // Reading the basic set of file attributes:                           (7)
  BasicFileAttributes bfa2 = bfaView.readAttributes();
  FileUtils.printBasicFileAttributes(bfa2);
  FileTime currentLastModifiedTime = bfa2.lastModifiedTime();

  // Updating timestamp for last modified time using view:               (8)
  long newLMTinMillis = currentLastModifiedTime.toMillis() + 15*60*1000L;
  FileTime newLastModifiedTime = FileTime.fromMillis(newLMTinMillis);
  bfaView.setTimes(newLastModifiedTime, null, null);

  // Reading the updated last modified time:                             (9)
  out.println("updated lastModifiedTime (incorrect): "
                                          + bfa2.lastModifiedTime());   // (10)
  out.println("updated lastModifiedTime: "
                                    + Files.getLastModifiedTime(path)); // (11)
  out.println("updated lastModifiedTime: " + Files.getAttribute(path,    // (12)
                                                "basic:lastModifiedTime"));
```

Possible output from the code:

Click here to view code image

```
File: project/src/pkg/Main.java
Using view: basic
Printing basic file attributes:
...
lastModifiedTime: 2021-07-26T15:15:46.813Z
...
updated lastModifiedTime (incorrect): 2021-07-26T15:15:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
updated lastModifiedTime: 2021-07-26T15:30:46.813Z
```

The `BasicFileAttributeView` object allows the last modified, last access, and creation times of the directory entry to be updated by calling the `setTimes()` method. The code at (9) shows how the last modified time of the directory entry can be updated to a new value via the view. A `FileTime` object is created representing the new last modified time by first converting the current last modified time to milliseconds and incrementing it by 15 minutes. In the call to the `setTimes()` method, only the last modified time is specified. The other timestamps are specified as `null`, indicating that they should not be changed.

In order to verify the new last modified time, we might be tempted to use the current `BasicFileAttributes` object associated with the view, but its copies of the file attribute values are not updatable. We can create a new `BasicFileAttributes` object that reflects the new values of the file attributes, or alternately use the `getLastModifiedTime()` or the `getAttribute()` methods of the `Files` class, as shown at (11) and (12), respectively.

The *PosixFileAttributeView Interface*

As the `PosixFileAttributeView` interface is a subinterface of the `BasicFileAttribute-View` interface, it allows both the basic set of file attributes and the POSIX-specific file attributes to be read and updated.

Click here to view code image

```
interface PosixFileAttributeView
          extends BasicFileAttributeView, FileOwnerAttributeView

String name()
```

Returns the name of the attribute view, which in this case is the string `"posix"`.

Click here to view code image

```
PosixFileAttributes readAttributes()
```

Retrieves the basic and POSIX-specific file attributes as a bulk operation into a `PosixFileAttributes` object whose methods can be used to read the values of these file attributes (p. 1332). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328).

Click here to view code image

```
void setGroup(GroupPrincipal group) throws IOException
```

Updates the group of the directory entry. Note that there is no analogous method in the `Files` class for handling the group.

Click here to view code image

```
void setPermissions(Set<PosixFilePermission> perms)
```

Updates the file permissions. This method is analogous to the `Files.setPosixFilePermissions()` method (p. 1325).

The `PosixFileAttributeView` interface extends the `java.nio.file.attribute.File-OwnerAttributeView` interface that defines the methods for reading and updating the owner of the directory entry. See also analogous methods relating to ownership in the `Files` class (p. 1325).

Click here to view code image

```
interface java.nio.file.attribute.FileOwnerAttributeView
          extends FileAttributeView

UserPrincipal getOwner() throws IOException
void          setOwner(UserPrincipal owner) throws IOException
```

Return or update the owner of a directory entry, respectively. These methods are analogous to the methods in the `Files` class (p. 1325).

A `PosixFileAttributeView` object can thus read both the basic set of file attributes and the POSIX-specific file attributes, and can update the owner, group, file permissions, and time-stamps for the last modified, last access, and creation times for a directory entry.

The code below obtains a `PosixFileAttributeView` object at (13) that pertains to the file denoted by the `path` reference. The associated `PosixFileAttributes` object is obtained at (14), providing read-only access to the basic file attributes and the POSIX-specific file attributes, whose values are printed by calling the methods `printBasicFileAttributes()` and `printPosixFileAttributes()` in the utility class `FileUtils`, respectively.

Click here to view code image

```
Path path = Path.of("project", "src", "pkg", "Main.java");
out.println("File: " + path);
PosixFileAttributeView pfaView = Files.getFileAttributeView(path,      // (13)
                                              PosixFileAttributeView.class);
System.out.printf("Using view: %s%n", pfaView.name());

// Reading the basic + POSIX set of file attributes:                   // (14)
PosixFileAttributes pfa2 = pfaView.readAttributes();
FileUtils.printBasicFileAttributes(pfa2);
FileUtils.printPosixFileAttributes(pfa2);

// Updating owner and group file attributes using view.                // (15)
FileSystem fs = path.getFileSystem();
UserPrincipalLookupService upls = fs.getUserPrincipalLookupService();
UserPrincipal newUser = upls.lookupPrincipalByName("javadude");
GroupPrincipal newGroup = upls.lookupPrincipalByGroupName("admin");
pfaView.setOwner(newUser);
pfaView.setGroup(newGroup);

//Updating file permissions using view.                                // (16)
Set<PosixFilePermission> newPerms = PosixFilePermissions.fromString("r--r--r--");
pfaView.setPermissions(newPerms);

//Updating last access time using view.                                // (17)
```

```
  FileTime currentAccessTime = pfa2.lastAccessTime();
  long newLATinMillis = currentAccessTime.toMillis() + 10*60*1000L;
  FileTime newLastAccessTime = FileTime.fromMillis(newLATinMillis);
  pfaView.setTimes(null, newLastAccessTime, null);

  // Reading the updated file attributes:                               // (18)
  pfa2 = pfaView.readAttributes();
  FileUtils.printBasicFileAttributes(pfa2);
  FileUtils.printPosixFileAttributes(pfa2);
```

The code from (15) to (17) shows how the `PosixFileAttributeView` object can be used to update various file attributes. Keep in mind that this view inherits from the `BasicFileAttributeView` and the `FileOwnerAttributeView` interfaces.

The code at (15) updates the owner and the group of the directory entry via the view. An owner and a group are looked up in the appropriate lookup services, and updated by the `setOwner()` and `setGroup()` methods of the `PosixFileAttributeView` interface. See also corresponding methods in the `Files` class (p. 1325).

The code at (16) updates the file permissions of the directory entry via the view, analogous to the `Files.setPosixFilePermissions()` method (p. 1325). File permissions are set to read-only for the owner, the group, and other users.

The code at (17) updates only the last access time of the directory entry via the view, analogous to updating the last modified time via the `BasicFileAttributeView` object (p. 1334).

Updated file attribute values can be read using the appropriate methods of the `Files` class, or by obtaining a new `PosixFileAttributes` object, as shown at (18).

The *DosFileAttributeView Interface*

As the `DosFileAttributeView` interface is a subinterface of the `BasicFileAttributeView` interface, it allows both the basic set of file attributes and the DOS-specific file attributes to be read and updated. Its usage is analogous to the `PosixFileAttributeView` interface discussed earlier (p. 1336).

Click here to view code image

```
interface DosFileAttributeView extends BasicFileAttributeView

String name()
```

Returns the name of the attribute view, which in this case is the string `"dos"`.

Click here to view code image

```
DosFileAttributes readAttributes()
```

Reads the basic file attributes as a bulk operation. The `DosFileAttributes` object can be used to read the values of the basic and DOS-specific file attributes (p. 1333). This method is analogous to the `readAttributes()` method of the `Files` class (p. 1328)

Click here to view code image

```
void setReadOnly(boolean value)
void setSystem(boolean value)
void setArchive(boolean value)
void setHidden(boolean value)
```

Update the value of the appropriate attribute. These methods are all implementation specific.

21.7 Creating Directory Entries

The `Files` class provides methods for creating files and directories. These methods can accept a variable arity parameter of type `FileAttribute<?>`. The interface `java.nio.file.attribute.FileAttribute<T>` defines an object that encapsulates the value of a file attribute that can be set when a file or a directory is created by these methods. For a POSIX-based file system, the `PosixFilePermissions.asFileAttribute()` method creates a `FileAttribute` that encapsulates a set of type `PosixFilePermission` (p. 1325) `.`

The methods for creating regular files and directories throw a `FileAlreadyExists-Exception` if the directory entry with that name already exists. All methods for creating directory entries can throw an `IOException`, and should be called in a `try` block or the exception should be specified in a `throws` clause.

The steps to check whether the directory entry exists and to create the new directory entry if it does not exist are performed as a single *atomic operation*.

The following code calls the `printDirEntryInfo()` method in the utility class `FileUtils` on a path to print what kind of directory entry it denotes and its file permissions.

Click here to view code image

```
public static void printDirEntryInfo(Path path) throws IOException {
  String fmt = Files.isSymbolicLink(path)? "Symbolic link: %s%n":
      Files.isRegularFile(path)? "File: %s%n":
        Files.isDirectory(path)? "Directory: %s%n":
          "Directory entry: %s%n";
  out.printf(fmt, path);
  Set<PosixFilePermission> perms = Files.getPosixFilePermissions(path);
  String permStr = PosixFilePermissions.toString(perms);
  out.println(permStr);
}
```

Creating Regular and Temporary Files

The following methods of the `Files` class can be used to create regular and temporary files, and symbolic links.

Click here to view code image

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

Creates an empty file that is denoted by the `path` parameter, but fails by throwing a `FileAlreadyExistsException` if the file already exists. It does *not* create nonexistent parent directories, and throws a `NoSuchFileException` if any parent directory does not exist.

Click here to view code image

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

Create an empty file in the *default temporary-file directory* or in the *specified directory* denoted by the `Path` object, respectively, using the specified `prefix` and `suffix` to generate its name. The default temporary-file directory and naming of temporary files is file-system-specific.

Click here to view code image

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

Creates a symbolic link to a target (*optional operation*). When the target is a *relative path* then file system operations on the target are relative to the path of the symbolic link. The target of the symbolic link need not exist. Throws a `File-AlreadyExistsException` if a directory entry with the same name as the symbolic link already exists.

Click here to view code image

```
static Path readSymbolicLink(Path symbLink) throws IOException
```

Returns a `Path` object that denoted the target of a symbolic link (*optional operation*). The target of the symbolic link need not exist.

For creating files, we will use the following `FileAttribute` object denoted by the `fileAttr` reference (p. 1325). It specifies read and write permissions for the owner, but only read access for the group and others.

Click here to view code image

```
Set<PosixFilePermission> filePerms = PosixFilePermissions.fromString("rw-r--r--");
FileAttribute<Set<PosixFilePermission>> fileAttr =
        PosixFilePermissions.asFileAttribute(perms);
```

Creating Regular Files

The `createFile()` method of the `Files` class fails to create a regular file if the file already exists, or the parent directories on the path do not exist. The code below at (1) creates a file with the path `project/docs/readme.txt` relative to the current directory under the assumption that the file name does not exist and the parent directories exist on the path. However, running this code repeatedly will result in a `FileAlreadyExistsException`, unless the file is deleted (e.g., calling `Files.deleteIfExists(regularFile)`) before rerunning the code.

Click here to view code image

```
try {
  Path regularFile  = Path.of("project", "docs", "readme.txt");
  Path createdFile1 = Files.createFile(regularFile, fileAttr);        // (1)
  FileUtils.printDirEntryInfo(createdFile1);
} catch (NoSuchFileException | FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code:

```
File: project/docs/readme.txt
rw-r--r--
```

Creating Temporary Files

Programs usually create temporary files during execution, and delete such files when done in the interest of good housekeeping. The JVM defines a system property for the default temporary-file directory where temporary files are created, if a specific location is not specified. This location is printed by the code below at (1).

How the name of the temporary file is generated is file-system specific. The code at (2) creates a temporary file under the default temporary-file directory, that has default file permissions.

The code at (3) creates temporary files under a specific directory. How the specification of file name prefix and suffix affect the generated file name can be seen in the output, where the `null` value as the prefix omits the prefix, and the `null` value as the suffix appends the default file name extension `".tmp"`.

The NIO.2 API does not define a method to request that a file be deleted when the JVM terminates.

Click here to view code image

```
try {
  // System property that defines the default temporary-file directory.      (1)
  String tmpdir = System.getProperty("java.io.tmpdir");
  System.out.println("Default temporary directory: " + tmpdir);

  // Create under the default temporary-file directory.                      (2)
  Path tmpFile1 = Files.createTempFile("events", ".log");
  FileUtils.printDirEntryInfo(tmpFile1);

  // Create under a specific directory:                                      (3)
  Path tmpFileDir = Path.of("project");
  Path tmpFile2 = Files.createTempFile(tmpFileDir, "proj_", ".dat", fileAttr);
  Path tmpFile3 = Files.createTempFile(tmpFileDir, "proj_", null,   fileAttr);
  Path tmpFile4 = Files.createTempFile(tmpFileDir, null,    ".dat", fileAttr);
  Path tmpFile5 = Files.createTempFile(tmpFileDir, null,    null,   fileAttr);
  FileUtils.printDirEntryInfo(tmpFile2);
  FileUtils.printDirEntryInfo(tmpFile3);
  FileUtils.printDirEntryInfo(tmpFile4);
  FileUtils.printDirEntryInfo(tmpFile5);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code (*edited to fit on the page*):

Click here to view code image

```
Default temporary directory: /var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/
File:
/var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/events4720093907665196131.log
rw-------
File: project/proj_6062790522710209175.dat
```

```
rw-r--r--
File: project/proj_957015125593453845.tmp
rw-r--r--
File: project/3032983609251590109.dat
rw-r--r--
File: project/8205471872222375044.tmp
rw-r--r--
```

Creating Symbolic Links

The code below illustrates creating symbolic links to directory entries. We assume that the symbolic link at (1) below does not exist; otherwise, a resounding `File-AlreadyExistsException` will be thrown by the `createSymbolicLink()` method of the `Files` class. However, the target path need *not* exists, but that might limit the file operations that can be performed using the symbolic link. We assume that the target path at (2) exists when the code below is run.

The `createSymbolicLink()` method can be called with a relative path or the absolute path of the target, as shown at (3a) and (3b), respectively. In both cases, the symbolic link will be created with the default file permissions, as we have not specified the optional `FileAttribute` variable arity parameter. Note that the symbolic link is created to a file or a directory, depending on the kind of the target.

The method `readSymbolicLink()` method of the `Files` class returns the path of the target denoted by the symbolic link.

Click here to view code image

```
try {
  Path symbLinkPath  = Path.of(".", "readme_link");                     // (1)
  Path targetPath    = Path.of(".", "project", "backup", "readme.txt"); // (2)

  Path symbLink = Files.createSymbolicLink(symbLinkPath, targetPath);   // (3a)
//Path symbLink = Files.createSymbolicLink(symbLinkPath,
//                                         targetPath.toAbsolutePath());// (3b)
  Path target = Files.readSymbolicLink(symbLink);                       // (4)

  FileUtils.printDirEntryInfo(symbLink);
  FileUtils.printDirEntryInfo(target);
} catch (FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code:

Click here to view code image

```
Symbolic link: ./readme_link
rw-r--r--
File: ./project/backup/readme.txt
rw-r--r--
```

Creating Regular and Temporary Directories

The `Files` class provides methods to create regular and temporary directories.

Click here to view code image

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

Creates a new directory denoted by the `Path` object. It does *not* create nonexistent parent directories, and throws a `NoSuchFileException` if any parent directory does not exist. It also throws a `FileAlreadyExistsException` if the directory entry with that name already exists.

Click here to view code image

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

Creates a directory by creating all nonexistent parent directories first. It does not throw an exception if any of the directories already exist. If the method fails, some directories may have been created.

Click here to view code image

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

Create a new directory in the *default temporary-file directory* or in the *specified directory* denoted by the `Path` object, using the mandatory non-`null prefix` to append to its generated name. A `NoSuchFileException` is thrown by the second method if the specified location does not exist.

For creating directories, we will use the following `FileAttribute` object denoted by the `dirFileAttr` reference (p. 1325). It specifies read, write, and execute permissions for all users: the owner, the group, and others—often called *full permissions*.

Click here to view code image

```
Set<PosixFilePermission> dPerms = PosixFilePermissions.fromString("rwxrwxrwx");
FileAttribute<Set<PosixFilePermission>> dirFileAttr =
        PosixFilePermissions.asFileAttribute(dPerms);
```

Creating Regular Directories

The `Files` class provides two methods to create regular directories. The `createDirectory()` method fails to create a regular directory if the directory already exists, or if the parent directories on the path do not exist. The code below creates a directory with the path `project/bin` relative to the current directory, under the assumption that the directory name `bin` does not exist and the parent directory `./project` exists on the path. The directory is first created with the default file permissions at (1), then deleted at (2) and created again at (3) with specific file permissions. The second call to the `createDirectory()` method at (3) would throw a `FileAlreadyExistsException` if we did not delete the directory before creating it with specific file permissions.

The astute reader will have noticed from the output that the *write* permission for the group and others is not set at creation time by the `createDirectory()` method, regardless of the file permissions specified. This can be remedied by setting the file permissions explicitly after the directory has been created, as at (4).

Click here to view code image

```
try {
  Path regularDir  = Path.of("project", "bin");
  Path createdDir = Files.createDirectory(regularDir);                     // (1)
  FileUtils.printDirEntryInfo(createdDir);

  if (Files.deleteIfExists(regularDir)) {                                  // (2)
    System.out.printf("Directory deleted: %s%n", regularDir);
  }
  Path newDir = Files.createDirectory(regularDir, dirFileAttr);            // (3)
  FileUtils.printDirEntryInfo(newDir);
  Files.setPosixFilePermissions(newDir, dPerms);                           // (4)
  FileUtils.printDirEntryInfo(newDir);
} catch (NoSuchFileException | FileAlreadyExistsException fe) {
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code:

Click here to view code image

```
Directory: project/bin
rwxr-xr-
Directory deleted: project/bin
Directory: project/bin
rwxr-xr-x
Directory: project/bin
rwxrwxrwx
```

The method `createDirectories()` first creates all nonexistent parent directories in the path if necessary, and only creates the directory if it does not exist—a convenient way to ensure that a directory always exists. In the code below at (5), the parent directories (in the path `project/branches/maintenance)` are created from top to bottom, relative to the current directory if necessary, and the directory `versions` is only created if it does not exist from before. A `FileAlreadyExistsException` at (6) is thrown if a directory entry named `versions` exists, but is not a directory.

Click here to view code image

```
try {
  Path regularDir2  = Path.of("project", "branches", "maintenance", "versions");
  Path createdDir2 = Files.createDirectories(regularDir2, dirFileAttr);  // (5)
  FileUtils.printDirEntryInfo(createdDir2);
} catch (FileAlreadyExistsException fe) {                              // (6)
  fe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Output from the code:

Click here to view code image

```
Directory: project/branches/maintenance/versions
rwxrwxrwx
```

Creating Temporary Directories

Analogous to temporary files, temporary directories can be created by the `create-TempDirectory()` methods of the `Files` class. Again, these directories can be created in the default temporary-file directory, or in a specific location if one is specified in the method call, with either the default file permissions or specific permissions.

The code at (7) creates a temporary directory named `"log_dir"` under the default temporary-file directory. A temporary directory with the same name is also created under a specific location (the `./project` directory) at (8). A `NoSuchFileException` is thrown if the specified location does not exist. A prefix can be specified for the directory name or it can be `null`.

Click here to view code image

```
try  {
  // Create under the default temporary-file directory.                     (7)
  Path tmpDirPath1 = Files.createTempDirectory("log_dir", dirFileAttr);
  FileUtils.printDirEntryInfo(tmpDirPath1);

  // Create under a specific location:                                      (8)
  Path tmpDirLoc = Path.of("project");
  Path tmpDirPath2 = Files.createTempDirectory(tmpDirLoc, "log_dir", dirFileAttr);
  Path tmpDirPath3 = Files.createTempDirectory(tmpDirLoc, null, dirFileAttr);
  FileUtils.printDirEntryInfo(tmpDirPath2);
  FileUtils.printDirEntryInfo(tmpDirPath3);

  Files.setPosixFilePermissions(tmpDirPath3, dPerms);                    // (9)
  FileUtils.printDirEntryInfo(tmpDirPath3);
} catch (NoSuchFileException nsfe) {
  nsfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Possible output from the code (*edited to fit the page*):

Click here to view code image

```
Directory:
/var/folders/cr/wk7fqcjx07z95d9vxcgjnrtc0000gr/T/log_dir18136008118052819366
rwxr-xr-x
Directory: project/log_dir14908011762674796217
rwxr-xr-x
Directory: project/18428782809319921018
rwxr-xr-x
Directory: project/18428782809319921018
rwxrwxrwx
```

We notice from the output that the *write* permission for the group and others is not set at creation time by the `createTempDirectory()` method, regardless of the file permissions specified. Again, this can be remedied by setting the file permissions explicitly after the temporary directory has been created, as at (9).

21.8 Stream Operations on Directory Entries

The `Files` class provides static methods that create specialized streams to implement complex file operations on directory entries, which are concise and powerful, taking full advantage of the Stream API. They are also efficient as the lazy execution of the streams ensures that an element is only made available for processing when needed by the terminal stream operation.

Streams we have seen earlier do not use any system resources, as they are backed by data structures and generator functions. Such streams need not be closed, as they are handled as any other objects by the runtime environment with regard to memory management. In contrast, the streams that are backed by *file system resources* must be closed in order to avoid resource leakage—analogous to using I/O streams. As streams implement the `AutoCloseable` interface, the recommended practice is to use the `try`-with-resources statement, ensuring proper and prompt closing of file system resources after the stream operations complete.

Methods that create streams on directory entries throw a checked `IOException`, and in particular, they throw a `NoSuchFileException` if the directory entry does not exist. Any code using these methods is forced to handle these exceptions with either a `try`-`catch`-`finally` construct or a `throws` clause.

The examples in this section illustrate both closing of streams and handling of exceptions.

Reading Text Lines Using a Functional Stream

The `lines()` method of the `Files` class creates a stream that can be used to read the lines in a text file. It is efficient as it does not read the whole file into memory, making available only one line at a time for processing. Whereas the `BufferedReader` class provides an analogous method that uses a `BufferedReader`, the stream created by the `Files.lines()` method is backed by a more efficient `FileChannel`.

Click here to view code image

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

Return a `Stream` of type `String`, where the elements are text lines read from a file denoted by the specified `path`. The first method decodes the bytes into characters using the UTF-8 charset. The charset to use can be explicitly specified, as in the second method.

As an example of using the `lines()` method, we implement the solution to finding palindromes (§13.3, p. 688), where the words are read from a file.

Click here to view code image

```
Path wordFile = Path.of(".", "project", "wordlist.txt");
System.out.println("Find palindromes, greater than length 2.");
try (Stream<String> stream = Files.lines(wordFile)){
  List<String> palindromes = stream
       .filter(str -> str.length() > 2)
       .filter(str -> str.equals(new StringBuilder(str).reverse().toString()))
       .toList();
  System.out.printf("List of palindromes:   %s%n", palindromes);
  System.out.printf("Number of palindromes: %s%n", palindromes.size());
} catch (IOException e) {
  e.printStackTrace();
}
```

Possible output from the code (*edited to fit on the page*):

Click here to view code image

```
Find palindromes, greater than length 2.
List of palindromes:   [aba, abba, aga, aha, ...]
Number of palindromes: 90
```

The following code creates a map to count the number of lines with different lengths:

Click here to view code image

```
Path textFile = Path.of(".", "project", "linesOnly.txt");
try (Stream<String> stream = Files.lines(textFile)) {
  Map<Integer, Long> grpMap =
      stream.collect(Collectors.groupingBy(String::length,
                                           Collectors.counting()));
  System.out.println(grpMap);
} catch (IOException e) {
  e.printStackTrace();
}
```

Possible output from the code:

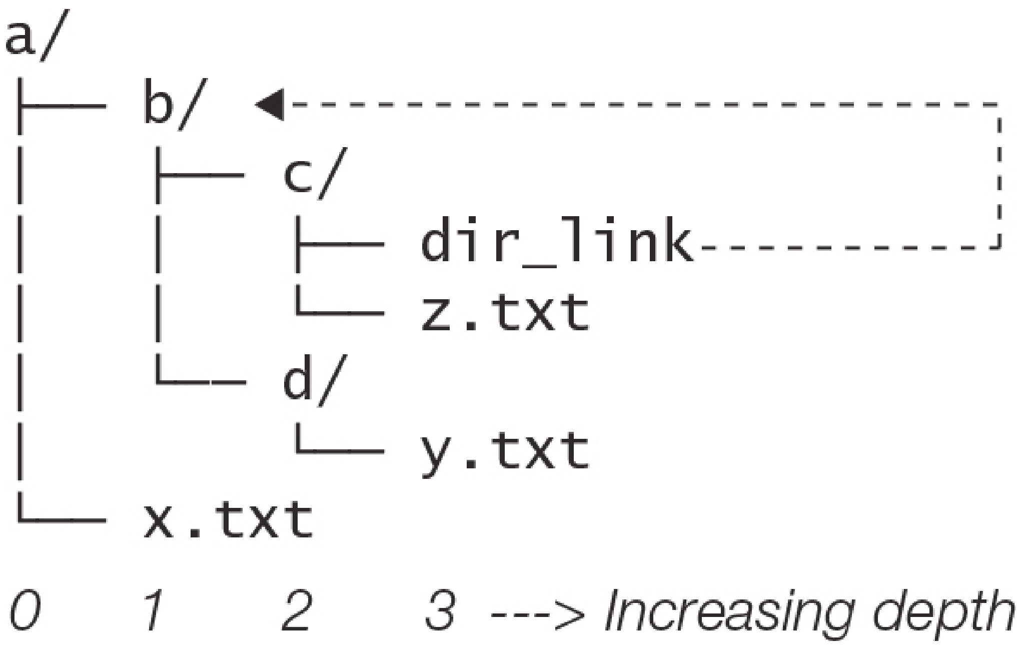
```
{4=1, 21=2, 10=1, 12=1}
```

Directory Hierarchy Traversal

Tree traversal is an important topic in computer science. Other synonyms used in the literature for traversing such structures are *walking*, *navigating*, and *visiting* a tree.

Note that in order to traverse a directory hierarchy and access its entries, the directory must have *execute* permission.

The `Files` class provides methods to traverse a directory hierarchy which has a tree structure. We will use the directory hierarchy shown below to illustrate these traversal methods. The directory hierarchy below is rooted at directory `a`—that is, this directory is the *start* or the *root* of the directory hierarchy. Traversal of the directory hierarchy starts by visiting directory `a` and is considered to be at *depth* 0. Dropping down each level of directories in the hierarchy increases the depth. The directory `a/b` and the file `a/x.tx`t are at depth 1 in relation to the directory `a`, and are said to be *siblings* at depth 1. The directory hierarchy below has a maximum depth of 3, as there are no directories below this depth. Note that the file `a/b/c/dir_link` is a symbolic link to the directory `a/b`, and as we shall see, it can influence the traversal if symbolic links are to be followed.



Traversal Strategies

There are primarily two main strategies for traversing a tree structure, or as in our case, a directory hierarchy. Both strategies start at the root of the directory hierarchy.

- *Depth-first traversal*
  Any subdirectory encountered at a particular depth level is traversed to its maximum depth before continuing with any sibling of this subdirectory at this depth level. The depth-first traversal of the hierarchy rooted at directory `a` is shown below. Note how at depth level 2, the directory `a/b/c` is traversed completely before traversing its sibling directory `a/b/d`. The traversal methods of the `Files` class use the depth-first traversal strategy.
  Click here to view code image

```
Visited entry        Depth
./a                  0
./a/x.txt            1
./a/b                1
./a/b/c              2
./a/b/c/z.txt        3
./a/b/c/dir_link     3
./a/b/d              2
./a/b/d/y.txt        3
```

- *Breadth-first traversal*
  The siblings at each depth level are traversed before moving on to the next depth level—in other words, the traversal is by depth level. The breadth-first traversal of the hierarchy rooted at directory `a` is shown below. Note how entries at each depth are traversed before traversing the entries at the next depth level.
  Click here to view code image

  ```
  Visited entry        Depth
  ./a                  0
  ./a/x.txt            1
  ./a/b                1
  ./a/b/d              2
  ./a/b/c              2
  ./a/b/c/z.txt        3
  ./a/b/c/dir_link     3
  ./a/b/d/y.txt        3
  ```

Both strategies have their pros and cons, and each excels at solving particular traversal problems. For example, finding a search goal that is closest to the root is best found by the breadth-first strategy in the shortest amount of time, whereas a goal that is farthest from the root is best found by the depth-first strategy in the shortest amount of time. Depth-first search may need to backtrack when trying to find a solution, whereas breadth-first search is more predictive. Breath-first search requires more memory as entries at previous levels must be maintained. The interested reader should consult the ample body of literature readily available on this important subject of trees and graph algorithms.

Listing the Immediate Entries in a Directory

A common file operation is to list the entries in a directory, similar to basic usage of the DOS command `dir` or the Unix command `ls`. The `list()` method of the `Files` class creates a stream whose elements are `Path` objects that denote the entries in a directory.

Click here to view code image

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

Creates a lazily populated `Stream` whose elements are the entries in the directory—it does *not* recurse on the entries of the directory. The stream does not include the directory itself or its parent. It throws a `java.nio.file.NotDirectoryException` if it is not a directory (*optional specific exception*). Also, it does not follow symbolic links.

The following code lists the entries under the directory `a`. Note that the stream only contains the *immediate* entries in the directory. Any subdirectories under the specified directory are *not* traversed. The `Path` object passed a parameter must denote a directory, or a disgruntled `NotDirectoryException` is thrown.

Click here to view code image

```
Path dir = Path.of(".", "a");
System.out.printf("Immediate entries under directory \"%s\":%n", dir);
try(Stream<Path> stream = Files.list(dir)) {
  stream.forEach(System.out::println);
} catch (NotDirectoryException nde) {
nde.printStackTrace();
} catch (IOException ioe) {
ioe.printStackTrace();
}
```

Output from the code:

Click here to view code image

```
Immediate entries under directory "./a":
./a/x.txt
./a/b
```

File Visit Option

In the file operations we have seen so far, the default behavior was to follow symbolic links. Not so in the case of the traversal methods `find()` and `walk()` of the `Files` class—these methods do *not* follow symbolic links, unless instructed explicitly to do so. The `FileVisitOption` enum type defines the constant `FOLLOW_LINKS` (Table 21.13) that can be specified to indicate that symbolic links should be followed when using these methods.

Table 21.13 *File Visit Option*

Enum java.nio.file.FileVisitOption constant	Description
`FOLLOW_LINKS`	Indicates that symbolic links should be followed.

Walking the Directory Hierarchy

The `walk()` method of the `Files` class creates a stream to walk or traverse the directory hierarchy rooted at a specific directory entry.

Click here to view code image

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

Return a `Stream` that is lazily populated with `Path` objects by walking the directory hierarchy rooted at the entry specified by the `start` parameter. The `start` parameter can be a directory or a file.

The first method is equivalent to calling the second method with a depth of `Integer.MAX_VALUE`. The methods traverse the entries *depth-first* to a depth limit that is the minimum of the maximum depth of the directory hierarchy rooted at the `start` entry and any depth that is specified or implied.

These methods do *not* follow symbolic links, unless the constant `FileVisitOption.FOLLOW_LINKS` is specified.

Example 21.5 illustrates using the `walk()` method. The hierarchy of directory `a` (p. 1347) is traversed to illustrate different scenarios. The code at (1) creates the symbolic link `a/b/c/dir_link` to the directory `a/b`, if necessary.

Example 21.5 *Traversing the Directory Hierarchy*

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class WalkTheWalk {

  public static void main(String[] args) throws IOException {

    // Creating symbolic link.                                       // (1)
    try {
      Path targetPath = Path.of(".", "a", "b");
      Path symbLinkPath  = Path.of(".", "a", "b", "c", "dir_link");
      if (Files.notExists(symbLinkPath, LinkOption.NOFOLLOW_LINKS)) {
        Files.createSymbolicLink(symbLinkPath, targetPath.toAbsolutePath());
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return;
    }

    // Do the walk.                                                   // (2)
    Path start = Path.of(".", "a");
    int MAX_DEPTH = 4;

    for (int depth = 0; depth <= MAX_DEPTH; ++depth)  {                // (3)
      try(Stream<Path> stream = Files.walk(start, depth,               // (4)
                                           FileVisitOption.FOLLOW_LINKS)) {
        System.out.println("Depth limit: " + depth);
        stream.forEach(System.out::println);
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
Depth: 0
./a
Depth: 1
./a
./a/x.txt
./a/b
Depth: 2
./a
./a/x.txt
./a/b
./a/b/c
./a/b/d
Depth: 3
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
./a/b/c/dir_link
./a/b/d
./a/b/d/y.txt
Depth: 4
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
Exception in thread "main" java.io.UncheckedIOException:
    java.nio.file.FileSystemLoopException: ./a/b/c/dir_link
...
```

Limiting Traversal by Depth Specification

The `walk()` method at (4) in Example 21.5 is called multiple times in the `for(;;)` loop at (3). From the output, we can see that, for each value of the loop variable `depth`, the `walk()` method starts the traversal at the directory `a` and descends to the depth level given by the loop variable `depth`. For example, when the value of the `depth` variable is `3`, traversal descends from depth level 0 to depth level 3.

If the constant `FileVisitOption.FOLLOW_LINKS` is *not* specified, so that symbolic links are not followed, the traversal will only go as far as the minimum of the maximum depth of the hierarchy and any specified depth level.

Specifying the depth level can result in a more efficient search if it is known that the result is at an approximate depth level, thus avoiding useless searching farther down in the hierarchy.

Handling Symbolic Links

Specifying the constant `FileVisitOption.FOLLOW_LINKS` in the `walk()` method call results in symbolic links being followed.

A symbolic link to a file is not a problem, as after visiting the target file, the traversal can continue with the next sibling of the symbolic link.

A symbolic link to a directory that is *not* a *parent directory* of the symbolic link is also not a problem, as traversal can continue with the next sibling of the symbolic link, after visiting the target directory.

The problem arises when a symbolic link is to a directory that is a *parent directory* of the symbolic link. That creates a *cyclic path dependency*. An example of such a cyclic path dependency is introduced by the symbolic link `a/b/c/dir_link` to one of its parent directories, `a/b`, as can be seen in the figure of the directory hierarchy (p. 1347).

When the constant `FileVisitOption.FOLLOW_LINKS` is specified in the `walk()` method call, the method detects such cyclic path dependencies by monitoring which entries have been visited. Example 21.5 illustrates this scenario when the constant `FileVisitOption.FOLLOW_LINKS` is specified and the specified depth is greater than 3. From the output, we can see that, at depth level 4, the method detects that the symbolic link `a/b/c/dir_link` to its parent directory `a/b` creates a cyclic path dependency, as the target directory lies on the path to the symbolic link and has been visited before. The method throws a hefty `FileSystemLoopException` to announce the outcome.

Copying an Entire Directory

The `Files.copy()` method only copies a single file or creates an empty directory at its destination, depending on whether the source is a file or a directory, respectively. Shown below is the `copyEntireDirectory()` method that copies an *entire* directory.

Click here to view code image

```
/** Declared in FileUtils class.
 * Copy an entire directory.
 * @param sourceDir        Directory to copy.
 * @param destinationDir   Directory to which the source directory is copied.
 * @param options          Copy options for all entries.
 */
public static void copyEntireDirectory(Path sourceDir,
                                       Path destinationDir,
                                       CopyOption... options)  {
  try (Stream<Path> stream = Files.walk(sourceDir)) {                     // (1)
    stream.forEach(entry -> {
      Path relativeEntryPath = sourceDir.relativize(entry);               // (2)
      Path destination  = destinationDir.resolve(relativeEntryPath);     // (3)
      try {
        Files.copy(entry, destination, options);                          // (4)
      } catch (DirectoryNotEmptyException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
    });
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

The method `copyEntireDirectory()` copies each entry with the `Files.copy()` method at (4) as the directory hierarchy is traversed in the stream created by the `Files.walk()` method at (1). For each entry, the destination where the entry should be copied is determined by the code at (2) and (3).

First, the *relative* path between the source directory path and the current entry path is determined by the `relativize()` method at (2):

Click here to view code image

```
Source directory:       ./a/b
Current entry:          ./a/b/c/d
Relative entry path:    c/d
```

The *destination* path to copy the current entry is determined by joining the destination directory path with the relative entry path by calling the `resolve()` method at (3):

Click here to view code image

```
Destination directory:   ./x/y
Relative entry path:     c/d
Destination entry paths: ./x/y/c/d
```

In the scenario above, the entry `./a/b/c/d` is copied to the destination path `./x/y/c/ d` during the copying of source directory `./a/b` to the destination directory `./x/y`.

Click here to view code image

```
CopyOption[] options = new CopyOption[] {
    StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES,
    LinkOption.NOFOLLOW_LINKS};
Path sourceDirectory      = Path.of(".", "a", "b");       //      ./a/b
Path destinationDirectory = Path.of(".", "x", "y");       // (5a) ./x/y
// Path destinationDirectory = Path.of(".", "x")
//              .resolve(sourceDirectory.getFileName());  // (5b) ./x/b
FileUtils.copyEntireDirectory(sourceDirectory, destinationDirectory, options);
```

If the destination directory should have the same name as the source directory, we can use the code at (5b) rather than the code at (5a).

A few things should be noted about the declaration of the `copyEntireDirectory()` method. It can be customized to copy the entries by specifying copy options. It closes the stream created by the `walk()` method in a `try`-with-resources statement. Calling the method `copy()` in the body of the lambda expression requires handling any `IOException` inside the lambda body. Any `IOException` from calling the `walk()` method is also explicitly handled.

Searching for Directory Entries

The `find()` method of the `Files` class can be used for searching or finding directory entries in the file system.

Click here to view code image

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

Returns a `Stream` that is lazily populated with `Path` objects by searching for entries in a directory hierarchy rooted at the `Path` object denoted by the `start` parameter. The `start` parameter can be a directory or a file.

The method walks the directory hierarchy in exactly the same manner as the `walk()` method. The methods traverse the entries *depth-first* to a depth limit that is the minimum of the actual depth of the directory hierarchy rooted at the `start` entry and the `depth` that is specified.

The `matcher` parameter defines a `BiPredicate` that is used to decide whether a directory entry should be included in the stream. For each directory entry in the stream, the `BiPredicate` is invoked on its `Path` and its `BasicFileAttributes` — making it possible to define a customized filter.

This method does *not* follow symbolic links, unless the constant `FileVisitOption.FOLLOW_LINKS` is specified.

The `find()` method is analogous to the `walk()` method in many respects: It traverses the directory hierarchy in the same way, does not follow symbolic links by default, follows symbolic links only if the constant `FileVisitOption.FOLLOW_LINKS` is specified, and monitors visiting entries to detect cyclic path dependencies since the depth limit is always specified.

Whereas both `walk()` and `find()` methods create a stream of `Path` objects, the `find()` method also allows a matcher for the search to be defined by a lambda expression that implements the `BiPredicate<Path, BasicFileAttributes>` functional interface. This matcher is applied by the method and determines whether an entry is allowed in the stream. This is in contrast to an explicit intermediate filter operation on the stream, as in the case of a stream created by the `walk()` method. The `find()` method supplies the `BasicFileAttributes` object associated with a `Path`—analogous to using the `readAttributes()` method of the `Files` class (p. 1328). Given the `Path` object and its associated `BasicFileAttributes` object with the basic set of read-only file attributes, it is possible to write complex search criteria on a directory entry. The methods of the `BasicFileAttributes` interface also do not throw checked exceptions, and are therefore convenient to call in a lambda expression, as opposed to corresponding methods of the `Files` class.

In the following code, we use the `find()` method to find regular files whose name ends with `".txt"` and whose size is greater than 0 bytes. The `BiPredicate` at (1) defines the filtering criteria based on querying the `Path` object in the stream for its file extension and its associated `BasicFileAttributes` object with read-only file attributes for whether it is a regular file and for its size.

Click here to view code image

```
System.out.println("Find regular files whose name ends with \".txt\""
                 + " and whose size is > 0:");
Path startEntry = Path.of(".", "a");
int depth = 5;
try (var pStream = Files.find(startEntry, depth,
                              (path, attrs) -> attrs.isRegularFile()       // (1)
                              && attrs.size() > 0
                              && path.toString().endsWith(".txt"))) {
   List<Path> pList = pStream.toList();
   System.out.println(pList);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Output from the code:

Click here to view code image

```
Find regular files whose name ends with ".txt" and whose size is > 0:
[./a/x.txt, ./a/b/c/z.txt, ./a/b/d/y.txt]
```

Note that the method `find()` requires the depth of the search to be specified. However, the actual depth traversed is always the minimum of the maximum depth of the directory hierarchy and the depth specified. The maximum depth of the hierarchy rooted at directory `a` is 3 and the specified depth is 5. The actual depth traversed in the directory hierarchy is thus 3. In the code above, different values for the depth can give different results.

If the constant `FileVisitOption.FOLLOW_LINKS` is specified in the `find()` method, its behavior is analogous to the behavior of the `walk()` method. It will keep track of the directories visited, and any cyclic path dependency encountered will unceremoniously result in a `FileSystemLoopException`. The curious reader is encouraged to experiment with the code above by specifying the constant `FileVisitOption.FOLLOW_LINKS` and passing different values for the depth in the `find()` method call.

Review Questions

21.1 Assume that the current directory has the absolute path `/wrk`.

Which of the following statements are true about the following program?

Click here to view code image

```
import java.io.IOException;
import java.nio.file.Path;

public final class Filing {
  public static void main (String[] args) throws IOException {
    Path file = Path.of("./document", "../book/../chapter1");
    System.out.println(file.toAbsolutePath());   // (1)
    System.out.println(file.toRealPath());       // (2)
    System.out.println(file.normalize());        // (3)
    System.out.println(file.toString());         // (4)
    System.out.println(file.getParent());        // (5)
    }
  }
```

Select the two correct answers.

a. The line at (1) will print `/wrk/document/book/chapter1`.

b. The line at (2) will print `/wrk/chapter1`.

c. The line at (3) will print `chapter1`.

d. The line at (4) will print `./document/book/chapter1`.

e. The line at (5) will print `./document/book/`.

21.2 Given the following program:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class ListingFiles {
  public static void main(String[] args) throws IOException {
    Path dirPath = Path.of(".", "wrk");
    printFiles1(dirPath);
    printFiles2(dirPath);
    printFiles3(dirPath);
  }

  public static void printFiles1(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.list(dirPath)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }

  public static void printFiles2(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.walk(dirPath)) {
      paths.forEach(System.out::println);
    }
    System.out.println();
  }

  public static void printFiles3(Path dirPath) throws IOException {
    try (Stream<Path> paths = Files.find(dirPath, Integer.MAX_VALUE,
                                         (p, a) -> true)) {
      paths.forEach(System.out::println);
    }
```

```
      System.out.println();
    }
  }
```

Assume that the directory `./wrk` has the following directory hierarchy:

```
./wrk
└── src
    └── readme.txt
```

Which statement is true about the program?

Select the one correct answer.

a. All three methods `printFiles1()`, `printFiles2()`, and `printFiles3()` will produce the same output.

b. Only the methods `printFiles1()` and `printFiles2()` will produce the same output.

c. Only the methods `printFiles2()` and `printFiles3()` will produce the same output.

d. Only the methods `printFiles1()` and `printFiles3()` will produce the same output.

e. The program will fail to compile because the `list()` method does not exist in the `Files` class.

21.3 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class RQ1 {
  public static void main(String[] args) {

    Path earth = Path.of("/", "planets", "earth");
    Path moonOrbit = earth.resolve(Path.of("moon", "orbit.param"));
    Path mars = earth.resolveSibling("mars");
    Path fromMarsToMoon = mars.relativize(moonOrbit);
    System.out.println(fromMarsToMoon);
  }
}
```

What is the result?

Select the one correct answer.

a. `../earth/moon/orbit.param`

b. `/planets/mars/../earth/moon/orbit.param`

c. `/planets/earth/moon/orbit.param`

d. `./mars/../earth/moon/orbit.param`

e. The program will throw an exception at runtime.

21.4 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class RQ2 {
  public static void main(String[] args) {
    /* Assume current directory path is /planets. */
    Path path = Path.of("./mars/../earth").normalize().toAbsolutePath();
    System.out.println(path.getNameCount());
  }
}
```

What is the result?

Select the one correct answer.

a. `1`

b. `2`

c. `5`

d. The program will throw an exception at runtime.

21.5 Given the following code:

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
public class RQ3 {
  public static void main(String[] args) {
      try (var stream = Files.list(Path.of("/test")
                                          .toRealPath(LinkOption.NOFOLLOW_LINKS))) {
        stream.filter(p -> p.getFileName().toString().endsWith("txt"))
              .forEach(p -> System.out.println(p));
    } catch (IOException ex) {
      ex.printStackTrace();
```

```
        }
    }
}
```

and the following hierarchy for the `/test` directory:

```
/test
├── a.txt
│   └── b.txt
├── c
│   └── d.txt
├── e.txt
└── f.txt
```

What is the result?

Select the one correct answer.

a.

```
/test/a.txt
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

b.

```
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

c.

```
/test/a.txt/b.txt
/test/e.txt
/test/f.txt
```

d.

```
/test/a.txt
/test/e.txt
/test/f.txt
```

e.

```
a.txt/b.txt
c/d.txt
e.txt
f.txt
```

f.

```
b.txt
d.txt
e.txt
f.txt
```

g.

```
a.txt/b.txt
e.txt
f.txt
```

h.

```
a.txt
b.txt
d.txt
e.txt
f.txt
```

21.6 Given the following code:

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
public class RQ4 {
  public static void main(String[] args) {

    try (var stream = Files.walk(Path.of("/test")
                                        .toRealPath(LinkOption.NOFOLLOW_LINKS))) {
      stream.map(p -> p.getFileName().toString())
            .filter(p -> p.endsWith("txt"))
            .sorted()
            .forEach(System.out::println);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    }
}
```

and the following hierarchy for the `/test` directory:

```
/test
├── a.txt
│   └── b.txt
├── c
│   └── d.txt
├── e.txt
└── f.txt
```

What is the result?

Select the one correct answer.

a.

```
/test/a.txt
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

b.

```
/test/a.txt/b.txt
/test/c/d.txt
/test/e.txt
/test/f.txt
```

c.

```
/test/a.txt/b.txt
/test/e.txt
/test/f.txt
```

d.

```
/test/a.txt
/test/e.txt
/test/f.txt
```

e.

```
b.txt
d.txt
e.txt
f.txt
```

f.

```
b.txt
e.txt
f.txt
```

g.

```
a.txt
b.txt
d.txt
e.txt
f.txt
```

21.7 Which of the following statements is true?

Select the one correct answer.

a. The `Files.createDirectories()` method throws an exception when the directory to create already exists.

b. The `Files.delete()` method throws an exception when trying to delete a nonexistent path.

c. The `Files.move()` method throws an exception when attempting to move a non-empty directory in the same file system.

d. The `Files.exists()` method throws an exception when a path does not exist in the file system.

21.8 Given the following code:

Click here to view code image

```
import java.io.*;
import java.nio.file.*;
public class RQ6 {
  public static void main(String[] args) {

    try {
      Path p1 = Path.of("/users/joe/test/a.jpg");
      Path p2 = Path.of("/users").resolve(p1.getName(1).resolve("test/a.jpg"));
```

```
        Files.move(p1, p2);
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
```

Assuming that the relevant paths exist, what is the result?

Select the one correct answer.

a. The file `a.jpg` is moved to the `/users` directory.

b. The file `a.jpg` is moved to the `/users/joe/test` directory.

c. No action is taken when trying to move the file `a.jpg`.

d. An `IOException` is thrown when trying to move the file `a.jpg`.

21.9 Given the following code:

Click here to view code image

```
import java.nio.file.*;
public class RQ7 {
  public static void main(String[] args) {
    Path p1 = Path.of("/test");
    Path p2 = Path.of("store");
    System.out.println(p1.resolve(p2));
    System.out.println(p2.resolve(p1));
  }
}
```

What is the result?

Select the one correct answer.

a.

```
/test/store
/store/test
```

b.

```
/test
/store
```

c.

```
/test/store
/test
```

d.

```
/test/store
/test/store
```

21.10 Given the following code:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
public class RQ8 {
  public static void main(String[] args) {
    try {
      Path source = Path.of("/test/readme.txt");
      Path destination = Path.of("/backup/readme_save.txt");
      Files.copy(source, destination);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Assuming that both source and destination files exist and are accessible, what is the result?

Select the one correct answer.

a. The destination file will be overwritten with the content of the source file.

b. The content of the source file will be appended to the end of the destination file.

c. The content of the source file will be inserted at the beginning of the destination file.

d. The program will throw an exception at runtime.

21.11 Given the following code:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
public class RQ9 {
  public static void main(String[] args) {
    Path p = Path.of("/test/test.html");
    try (var stream = Files.lines(p)){
      String result = stream.filter(s->s.startsWith("<"))
                            .map(s->s.substring(s.indexOf(">"), s.indexOf(">")+1))
                            .reduce("", (s1,s2) -> s1+s2);
      System.out.println(result);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

and that the `/test/test.html` file contains the following lines:

```
<!DOCTYPE html>
<html>
<body>
Hello World
</body>
</html>
```

What is the result?

Select the one correct answer.

a. An empty string

b. <<<<<

c. <><><><><>

d. `>>>>>`

e. The program will throw an exception at runtime.

21.12 Given the following code:

Click here to view code image

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.*;

public class RQ10 {
  public static void main(String[] args) {
    Path directory = Path.of("/test/data");
    Path file = Path.of("/test/data/info.txt");
```

```
      try {
        Set<PosixFilePermission> permissions = new HashSet<>();
        permissions.add(PosixFilePermission.OWNER_READ);
        Files.setPosixFilePermissions(directory, permissions);

        permissions.add(PosixFilePermission.OWNER_WRITE);
        Files.setPosixFilePermissions(file, permissions);
        try (var stream = Files.walk(directory)) {
          stream.forEach(System.out::println);
        }
      } catch(IOException e) {
        System.out.println(e);
      }
    }
  }
```

assume that the directory `/test/data` exists and only contains the `info.txt` file, and that full permissions are set for both the directory and the file prior to program execution.

What is the result?

Select the one correct answer.

a. `/test/data`

b. `/test/data/info.txt`

c.

Click here to view code image

```
java.nio.file.AccessDeniedException:/test/data/info.txt
```

d.

Click here to view code image

```
java.nio.file.AccessDeniedException:/test/data
```

21.13 Given the following code:

Click here to view code image

```
import static java.nio.file.attribute.PosixFilePermission.*;
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.util.Set;

public class RQ11 {
  public static void main(String[] args) {
    try {
      Path file = Path.of("/test/data/info.txt");
      Set<PosixFilePermission> perms
              = PosixFilePermissions.fromString("---------");
      Files.setPosixFilePermissions(file, perms);

      PosixFileAttributeView pfaView = Files.getFileAttributeView(file,
          PosixFileAttributeView.class);
      PosixFileAttributes pfa = pfaView.readAttributes();

      perms = Set.of(OWNER_READ, GROUP_WRITE, OTHERS_READ);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      perms.add(OWNER_WRITE);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      perms.remove(GROUP_WRITE);
      pfaView.setPermissions(perms);

      perms = pfa.permissions();
      System.out.println(perms);

    } catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

What is the result?

Select the one correct answer.

a.

Click here to view code image

```
[OWNER_READ, OWNER_WRITE, GROUP_WRITE, OTHERS_READ]
```

b.

Click here to view code image

```
[OWNER_READ, OWNER_WRITE, OTHERS_READ]
```

c. `[OWNER_WRITE]`

d. `[]`

e. The program will throw an exception at runtime.

21.14 Given the following code:

Click here to view code image

```
import java.io.*;
import java.net.*;
import java.nio.file.*;
public class RQ12 {
  public static void main(String[] args) {
    try {
      URI uri = URI.create("file:///test/ora.html");
      Path p1 = Path.of(uri);
      Path p2 = Path.of("/test/ora.html");
      Files.copy(p1, p2);
      File file = p2.toFile();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Assume that the file `/test/ora.html` exists. What is the result?

Select the one correct answer.

a. No output is produced.

b. An exception is thrown when converting a `URI` object to a `Path` object.

c. An exception is thrown when copying path `p1` to path `p2`.

d. An exception is thrown when converting a `Path` object to a `File` object.