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.NoSuchFileException`, 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:

```
// 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.

```
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.

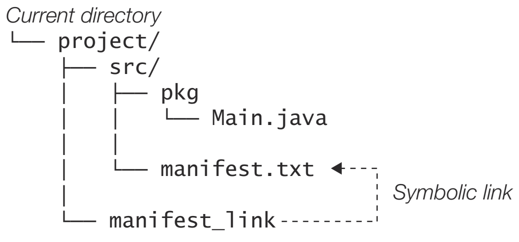
These methods normalize the path, and do not follow symbolic links if the enum constant `LinkOption.NOFOLLOW_LINKS` is specified for the `options` variable arity parameter.

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.



```
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.

```
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.

```
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`.

```
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.

```
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.DirectoryNotEmptyException` is thrown.

Consider the following paths that exist:

```
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.

```
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.

```
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`.

```
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*.

```
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.

```
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.

```
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:

```
./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.

```
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:

```
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.

```
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 InputStream 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.

```
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`.

```
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:

```
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.

```
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.

```
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`).

```
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.

```
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.

```
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`.

```
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);
```

T [redacted]