Manipulating Files and Directories

Checking a File or Directory

You have a `Path` instance representing a file or directory, but does that file exist on the file system? Is it readable? Writable? Executable?

Verifying the Existence of a File or Directory

The methods in the `Path` class are syntactic, meaning that they operate on the `Path` instance. But eventually you must access the file system to verify that a particular `Path` exists, or does not exist. You can do so with the `exists(Path, LinkOption...)` and the `notExists(Path, LinkOption...)` methods. Note that `!Files.exists(path)` is not equivalent to `Files.notExists(path)`. When you are testing a file's existence, three results are possible:

- The file is verified to exist.
- The file is verified to not exist.
- The file's status is unknown. This result can occur when the program does not have access to the file.

If both `exists()` and `notExists()` return `false`, the existence of the file cannot be verified.

Checking File Accessibility

To verify that the program can access a file as needed, you can use the `isReadable(Path)`, `isWritable(Path)`, and `isExecutable(Path)` methods.

The following code snippet verifies that a particular file exists and that the program has the ability to execute the file.

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
Files.isReadable(file) & Files.isExecutable(file);
```

> *Note: Once any of these methods completes, there is no guarantee that the file can be accessed. A common security flaw in many applications is to perform a check and then access the file. For more information, use your favorite search engine to look up TOCTTOU (pronounced TOCK-too).*

Checking Whether Two Paths Locate the Same File

When you have a file system that uses symbolic links, it is possible to have two different paths that locate the same file. The `isSameFile(Path, Path)` method compares two paths to determine if they locate the same file on the file system. For example:

```
Path p1 = ...;
Path p2 = ...;

if (Files.isSameFile(p1, p2)) {
    // Logic when the paths locate the same file
}
```

Deleting a File or Directory

You can delete files, directories or links. With symbolic links, the link is deleted and not the target of the link. With directories, the directory must be empty, or the deletion fails.

The `Files` class provides two deletion methods.

The `delete(Path)` method deletes the file or throws an exception if the deletion fails. For example, if the file does not exist a NoSuchFileException is thrown. You can catch the exception to determine why the delete failed as follows:

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

The `deleteIfExists(Path)` method also deletes the file, but if the file does not exist, no exception is thrown. Failing silently is useful when you have multiple threads deleting files and you do not want to throw an exception just because one thread did so first.

Copying a File or Directory

You can copy a file or directory by using the `copy(Path, Path, CopyOption...)` method. The copy fails if the target file exists, unless the `REPLACE_EXISTING` option is specified.

Directories can be copied. However, files inside the directory are not copied, so the new directory is empty even when the original directory contains files.

When copying a symbolic link, the target of the link is copied. If you want to copy the link itself, and not the contents of the link, specify either the `NOFOLLOW_LINKS` or `REPLACE_EXISTING` option.

This method takes a varargs argument. The following `StandardCopyOption` and `LinkOption` enums are supported:

- `REPLACE_EXISTING` – Performs the copy even when the target file already exists. If the target is a symbolic link, the link itself is copied (and not the target of the link). If the target is a non-empty directory, the copy fails with the `DirectoryNotEmptyException` exception.
- `COPY_ATTRIBUTES` – Copies the file attributes associated with the file to the target file. The exact file attributes supported are file system and platform dependent, but last-modified-time is supported across platforms and is copied to the target file.
- `NOFOLLOW_LINKS` – Indicates that symbolic links should not be followed. If the file to be copied is a symbolic link, the link is copied (and not the target of the link).

If you are not familiar with enums, see the section Enum Types.

The following shows how to use the copy method:

```
1 import static java.nio.file.StandardCopyOption.*;
2
3 Files.copy(source, target, REPLACE_EXISTING);
```

In addition to file copy, the `Files` class also defines methods that may be used to copy between a file and a stream. The `copy(InputStream, Path, CopyOptions...)` method may be used to copy all bytes from an input stream to a file. The `copy(Path, OutputStream)` method may be used to copy all bytes from a file to an output stream.

Moving a File or Directory

You can move a file or directory by using the `move(Path, Path, CopyOption...)` method. The move fails if the target file exists, unless the `REPLACE_EXISTING`

option is specified.

Using Varargs

Several Files methods accept an arbitrary number of arguments when flags are specified. For example, in the following method signature, the ellipses notation after the `CopyOption` argument indicates that the method accepts a variable number of arguments, or *varargs*, as they are typically called:

```
Path Files.move(Path, Path, CopyOption...)
```

When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (`CopyOption[]`) of values.

In the following example, the method can be invoked as follows:

```
Path source = ...;
Path target = ...;
Files.move(source,
           target,
           REPLACE_EXISTING,
           ATOMIC_MOVE);
```

Moving Directories

Empty directories can be moved. If the directory is not empty, the move is allowed when the directory can be moved without moving the contents of that directory. On UNIX systems, moving a directory within the same partition generally consists of renaming the directory. In that situation, this method works even when the directory contains files.

This method takes a varargs argument – the following `StandardCopyOption` enums are supported:

- `REPLACE_EXISTING` – Performs the move even when the target file already exists. If the target is a symbolic link, the symbolic link is replaced but what it points to is not affected.

- `ATOMIC_MOVE` – Performs the move as an atomic file operation. If the file system does not support an atomic move, an exception is thrown. With an `ATOMIC_MOVE` you can move a file into a directory and be guaranteed that any process watching the directory accesses a complete file.

The following shows how to use the move method:

```
1 import static java.nio.file.StandardCopyOption.*;
2
3 Files.move(source, target, REPLACE_EXISTING);
```

Though you can implement the `move()` method on a single directory as shown, the method is most often used with the file tree recursion mechanism. For more information, see the section Walking the File Tree.

Atomic Operations

Several `Files` methods, such as `move()`, can perform certain operations atomically in some file systems.

An atomic file operation is an operation that cannot be interrupted or "partially" performed. Either the entire operation is performed or the operation fails. This is important when you have multiple processes operating on the same area of the file system, and you need to guarantee that each process accesses a complete file.

Link Awareness

The `Files` class is "link aware." Every `Files` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered. For more information

on the way you can handle links on a file system, you can check the Links, Symbolics and Otherwise section.

Last update: *January 25, 2023*

Accessing the File System	←	**Manipulating Files and Directories**	→	Links, Symbolics and Otherwise

Home > Tutorials > The Java I/O API > File System Basics > Manipulating Files and Directories