The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Basic I/O
Section: File I/O (Featuring NIO.2)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Legacy File I/O Code

Interoperability With Legacy Code

Prior to the Java SE 7 release, the `java.io.File` class was the mechanism used for file I/O, but it had several drawbacks.

- Many methods didn't throw exceptions when they failed, so it was impossible to obtain a useful error message. For example, if a file deletion failed, the program would receive a "delete fail" but wouldn't know if it was because the file didn't exist, the user didn't have permissions, or there was some other problem.
- The `rename` method didn't work consistently across platforms.
- There was no real support for symbolic links.
- More support for metadata was desired, such as file permissions, file owner, and other security attributes.
- Accessing file metadata was inefficient.
- Many of the `File` methods didn't scale. Requesting a large directory listing over a server could result in a hang. Large directories could also cause memory resource problems, resulting in a denial of service.
- It was not possible to write reliable code that could recursively walk a file tree and respond appropriately if there were circular symbolic links.

Perhaps you have legacy code that uses `java.io.File` and would like to take advantage of the `java.nio.file.Path` functionality with minimal impact to your code.

The `java.io.File` class provides the `toPath` method, which converts an old style `File` instance to a `java.nio.file.Path` instance, as follows:

```
Path input = file.toPath();
```

You can then take advantage of the rich feature set available to the `Path` class.

For example, assume you had some code that deleted a file:

```
file.delete();
```

You could modify this code to use the `Files.delete` method, as follows:

```
Path fp = file.toPath();
Files.delete(fp);
```

Conversely, the `Path.toFile` method constructs a `java.io.File` object for a `Path` object.

Mapping java.io.File Functionality to java.nio.file

Because the Java implementation of file I/O has been completely re-architected in the Java SE 7 release, you cannot swap one method for another method. If you want to use the rich functionality offered by the `java.nio.file` package, your easiest solution is to use the `File.toPath` method as suggested in the previous section. However, if you do not want to use that approach or it is not sufficient for your needs, you must rewrite your file I/O code.

There is no one-to-one correspondence between the two APIs, but the following table gives you a general idea of what functionality in the `java.io.File` API maps to in the `java.nio.file` API and tells you where you can obtain more information.

java.io.File Functionality	java.nio.file Functionality	Tutorial Coverage
`java.io.File`	`java.nio.file.Path`	The Path Class
`java.io.RandomAccessFile`	The `SeekableByteChannel` functionality.	Random Access Files
`File.canRead`, `canWrite`, `canExecute`	`Files.isReadable`, `Files.isWritable`, and `Files.isExecutable`. On UNIX file systems, the Managing Metadata (File and File Store Attributes) package is used to check the nine file permissions.	Checking a File or Directory Managing Metadata
`File.isDirectory()`, `File.isFile()`, and `File.length()`	`Files.isDirectory(Path, LinkOption...)`, `Files.isRegularFile(Path, LinkOption...)`, and `Files.size(Path)`	Managing Metadata
`File.lastModified()` and `File.setLastModified(long)`	`Files.getLastModifiedTime(Path, LinkOption...)` and `Files.setLastMOdifiedTime(Path, FileTime)`	Managing Metadata
The `File` methods that set various attributes: `setExecutable`, `setReadable`, `setReadOnly`, `setWritable`	These methods are replaced by the `Files` method `setAttribute(Path, String, Object, LinkOption...)`.	Managing Metadata
`new File(parent, "newfile")`	`parent.resolve("newfile")`	Path Operations
`File.renameTo`	`Files.move`	Moving a File or Directory
`File.delete`	`Files.delete`	Deleting a File or Directory
`File.createNewFile`	`Files.createFile`	Creating Files
`File.deleteOnExit`	Replaced by the `DELETE_ON_CLOSE` option specified in the `createFile` method.	Creating Files
`File.createTempFile`	`Files.createTempFile(Path, String, FileAttributes<?>)`, `Files.createTempFile(Path, String, String, FileAttributes<?>)`	Creating Files Creating and Writing a File by Using Stream I/O

		Reading and Writing Files by Using Channel I/O
`File.exists`	`Files.exists` and `Files.notExists`	Verifying the Existence of a File or Directory
`File.compareTo` and `equals`	`Path.compareTo` and `equals`	Comparing Two Paths
`File.getAbsolutePath` and `getAbsoluteFile`	`Path.toAbsolutePath`	Converting a Path
`File.getCanonicalPath` and `getCanonicalFile`	`Path.toRealPath` or `normalize`	Converting a Path (`toRealPath`) Removing Redundancies From a Path (`normalize`)
`File.toURI`	`Path.toURI`	Converting a Path
`File.isHidden`	`Files.isHidden`	Retrieving Information About the Path
`File.list` and `listFiles`	`Path.newDirectoryStream`	Listing a Directory's Contents
`File.mkdir` and `mkdirs`	`Files.createDirectory`	Creating a Directory
`File.listRoots`	`FileSystem.getRootDirectories`	Listing a File System's Root Directories
`File.getTotalSpace`, `File.getFreeSpace`, `File.getUsableSpace`	`FileStore.getTotalSpace`, `FileStore.getUnallocatedSpace`, `FileStore.getUsableSpace`, `FileStore.getTotalSpace`	File Store Attributes

Previous page: Other Useful Methods

Next page: Summary