Accessing Resources using Paths → Working with Paths

Accessing Resources using Paths

Introducing the Legacy File Class

There two ways to model a file or a path on a file system in Java. The first, legacy one, is the `File` class. This class is mentioned here, with a word of warning: you should not use it anymore in your code, unless you have very good reasons to do so. You should favor the use of the `Path` interface, also covered in this section. Along with the factory methods from the `Files`, it gives you more features than the `File` class, and better performances, especially to access larger files and directories.

That been said, because it is widely used in legacy code, understanding the `File` may still be important to you. Without diving too deep in it, let us present the main concepts of this class.

An instance of the `File` class can represent anything on a file system: a file, a directory, a symbolic link, a relative path, or an absolute path. This instance is an abstract notion. Creating such an instance does not create anything on your file system. You can query your file system using this class, but you need to do it explicitly.

An instance of a `File` does not allow you to access the content of the file it represents. With this instance, you can check if this file exists or is readable (among other things).

A file is composed of several elements, separated by a separator, which depends on your file system. The first element may be a prefix, such as a disk-drive specifier, a

slash for the UNIX root directory. The other elements are names.

Creating an Instance of File

You can create an instance of the `File` class using several constructors:

- `File(String pathName)`: creates a file from the path you provide.
- `File(String parent, String child)`: create the given file in the `parent` directory.
- `File(File parent, String child)`: create the given file in the `parent` directory, specified as an instance of `File`.
- `File(URI uri)`: create a file from a `URI`.

Getting the Elements of a File

The following methods give you information on the elements of this file:

- `getName()`: returns the name of the file or directory denoted by this file object. This is just the last name of the sequence.
- `getParent()`: returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
- `getPath()`: returns this abstract pathname converted into a pathname string. This method is not related to the `Path` interface.
- `getAbsolutePath()`: returns the absolute pathname string of this abstract pathname. If this abstract pathname is already absolute, then the pathname string is simply returned. Otherwise this pathname is resolved in a system-dependent way.
- `getCanonicalPath()`: returns the canonical pathname string of this abstract pathname. The canonical pathname is absolute and unique and system-dependent. The construction of this canonical pathname typically involves removing redundant names such as `.` and `..` from the pathname, and resolving symbolic links.

Getting Information on a File or a Directory

Some of these methods may require special rights on files or directories. As a legacy class, the `File` does not expose all the security attributes offered by your file system.

- `isFile()`, `isDirectory()`: checks if this abstract pathname denotes an existing file or directory.
- `exists()`, `canRead()`, `canWrite()`, `canExecute()`: checks if this file exists, is readable, if you can modify it, or if you can execute it.
- `setReadable(boolean)`, `setWritable(boolean)`, `setExecutable(boolean)`: allows you to change the corresponding security attribute of the file. These methods return `true` if the operation succeeded.
- `lastModified()` et `setLastModified()`: return or set the time that this file was last modified.
- `length()`: returns the length of the file denoted by this abstract pathname.
- `isHidden()`: tests whether the file named by this abstract pathname is a hidden file.

Manipulating Files and Directories

Several methods allow you to create files and directories on a file system. Most of them are file system dependent. Remember that these methods are legacy methods. You can check the section Refactoring your Code to Using Path to refactor your code to use the equivalent methods from the `Path` interface and the `Files` class.

- `createNewFile()`: tries to create a new file from this pathname. This creation will fail if this file already exists. This method returns `true` if the file was successfully created, and `false` otherwise.
- `delete()`: deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted. This method returns `true` if the file was successfully deleted and `false` otherwise. You should favor the use of the `Files.delete()` method over this one, since it gives you more information in case the deletion fails.
- `mkdirs()` and `mkdir()`: create a directory named by this abstract pathname. `mkdirs()` creates all the intermediate directories if needed.
- `renameTo(file)`: renames the file denoted by this abstract pathname.

Introducing the Path Interface

The `Path` class, introduced in the Java SE 7 release, is one of the primary entrypoint of the `java.nio.file` package. If your application uses file I/O, you will want to learn about the powerful features of this interface.

> *Version Note: If you have pre-JDK7 code that uses `java.io.File`, you can still take advantage of the `Path` interface functionality by using the `File.toPath()` method. See the next section for more information. As its name implies, the `Path` interface is a programmatic representation of a path in the file system. A `Path` object contains the file name and directory list used to construct the path, and is used to examine, locate, and manipulate files.*

A `Path` instance reflects the underlying platform. In the Solaris OS, a `Path` uses the Solaris syntax (`/home/joe/foo`) and in Microsoft Windows, a `Path` uses the Windows syntax (`C:\home\joe\foo`). A `Path` is not system independent. You cannot compare a `Path` from a Solaris file system and expect it to match a `Path` from a Windows file system, even if the directory structure is identical and both instances locate the same relative file.

The file or directory corresponding to the `Path` might not exist. You can create a `Path` instance and manipulate it in various ways: you can append to it, extract pieces of it, compare it to another path. At the appropriate time, you can use the methods in the `Files` class to check the existence of the file corresponding to the `Path`, create the file, open it, delete it, change its permissions, and so on.

Refactoring your Code to Using Path

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 `java.io.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` interface.

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

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

You could modify this code to use the `Files.delete()` factory 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.

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. 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 Interface
java.io.RandomAccessFile	The SeekableByteChannel functionality.	Random Access Files
File.canRead(), File.canWrite(),	Files.isReadable(), Files.isWritable(), and Files.isExecutable(). On UNIX file	Checking a File or

java.io.File Functionality	java.nio.file Functionality	Tutorial coverage
File.canExecute()	systems, the Managing Metadata (File and File Store Attributes) package is used to check the nine file permissions.	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 Files.createFile() method.	Creating Files

java.io.File Functionality	java.nio.file Functionality	Tutorial coverage
File.createTempFile()	Files.createTempFile(String, String, FileAttributes<?>), Files.createTempFile(Path, 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 File.equals()	Path.compareTo() and Path.equals()	Comparing Two Paths
File.getAbsolutePath() and File.getAbsoluteFile()	Path.toAbsolutePath()	Converting a Path, Removing Redundancies From a Path (normalize)
File.getCanonicalPath() and File.getCanonicalFile()	Path.toRealPath() or Path.normalize()	Converting a Path (toRealPath)
File.toURI()	Path.toUri()	Converting a Path
File.isHidden()	Files.isHidden()	Retrieving Information

java.io.File Functionality	java.nio.file Functionality	Tutorial coverage
		About the Path
File.list() and listFiles	Files.newDirectoryStream()	Listing a Directory's Contents
File.mkdir() and mkdirs	Files.createDirectory(Path,FileAttribute)	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

Last update: *January 25, 2023*

Accessing Resources using Paths → Working with Paths