Contents

NIO Streams

Java offers a rich set of `APIs` one can use to manipulate files and directories. Java 7 introduced a set of IO `APIs` called `NIO.2` which stands for - New I/O. It offered convenient ways to perform operations related to a file system. In Java 8 they can also be sued along the Stream API.

Path

All file systems usually form a tree. The file system starts with a root directory that contains files and directories. Each directory in turn may have sub-directories or hold files. To locate a file one just needs to put together the directories from the root directory to the immediate directory containing the file, along with the file name, with a correct file separator. This forms the unique resource name or identifier for the file resource on disk - URN. A path can be absolute or relative. The Absolute path always starts from the root directory, the relative, is well relative to the current working directory or path (that could really mean many things, but in general when a program or a process is executing, it is always started or bound to a certain file system directory by the operating system itself, it also has a way of obtaining the current working directory, which itself is always an absolute path from the root).

Another important mention, is symbolic links. A symbolic link is like a pointer or reference to an actual file. In general symbolic link are transparent to applications, which means operations are performed directly on the files rather than on the links (except certain mutation operations such as deleting or moving), they are also referred to as `sym links` in practice.

The path interface is a programming abstraction for a path. A path objects contains the names of directories and files that form the full path of the file/directory represented by the Path object; the Path abstraction provides methods to extract path elements, manipulate them and append them.

See later that almost all the methods that access files/directories to get information about them or manipulate them use Path objects. The new `path` interface, is the corner stone of the NIO library, and is the building block of pretty much the entire API

Method	Description
Path getRoot()	Returns a Path object representing the root of the given path, or null if the path does not have a root.
Path getFileName()	Returns the file name or directory name of the given path. Note that the file/directory name is the last element or name in the given path.
Path getParent()	Returns the Path object representing the parent of the given path, or null if no parent component exists for the path. int getNameCount() Returns the number of file/directory names in the given path; returns 0 if the given path represents the root.
Path getName(int index)	Returns the ith file/directory name; the index 0 starts from closest name to the root.
Path subpath(int beginIndex, int endIndex)	Returns a Path object that is part of this Path object; the returned Path object has a name that begins at beginIndex and ends with the element at index endIndex - 1. In other words, beginIndex is inclusive of the name in that index and exclusive of the name in endIndex. This method may throw IllegalArgumentException if beginIndex is >= number of elements, or endIndex <= beginIndex, or endIndex > number of elements.
Path normalize()	Removes redundant elements in the path, such as . (dot symbol that indicates the current directory) and .. (double-dot symbol that indicates the parent directory).
Path resolve(Path other)	Resolves a path against the given path. For example, this method can combine the given path with the other path and return the resulting path.
Boolean isAbsolute()	Returns true if the given path is an absolute path; returns false if not (when the given path is a relative path, for example).
Path startsWith(Path path)	Returns true if this Path object starts with the given path, or false otherwise.
Path toAbsolutePath()	Returns the absolute path.

Information

In the following example it is shown how to obtain the full information for a given path, which points to some test file, on the D drive (Windows)

```
// create a Path object by calling static method get() in Paths class
Path testFilePath = Paths.get("D:\\test\\testfile.txt");
// retrieve basic information about path
System.out.println("Printing file information: ");
System.out.println("\t file name: " + testFilePath.getFileName());
System.out.println("\t root of the path: " + testFilePath.getRoot());
System.out.println("\t parent of the target: " + testFilePath.getParent());
// print path elements, path implements Iterable interface
System.out.println("Printing elements of the path: ");
for(Path element : testFilePath) {
    System.out.println("\t path element: " + element);
}
```

The output of this might look something like that, note that the path elements are printed from top to bottom, in this case there are only 2 elements, the root drive is not included. The path instance is obtained through the `Paths` utility class, calling the `get` static final method, which is passed the path location, note that the `get` method, can accept varargs string arguments, which are the path elements, so one can build a path from elements as well as - `Paths.get("D:", "test", "testfile.txt")`

```
Printing file information:
    file name: testfile.txt
    root of the path: D:\
    parent of the target: D:\test

Printing elements of the path:
    path element: test
    path element: testfile.txt
```

The example below shows how to obtain an object to a relative path, based on where this program is started, the actual current path will be used, and this is where the `Test` directory will be resoled to point to i.e. - `<current-absolute-path>\\Test`, note that the call to `toRealPath` will fail, since the `Test` directory probably does not exist, and `toRealPath` tries to resolve the real path to this directory, when it does not exist, it will throw an exception.

```
// get a path object with relative path
Path testFilePath = Paths.get(".\\Test");
System.out.println("The file name is: " + testFilePath.getFileName());
System.out.println("Its URI is: " + testFilePath.toUri());
System.out.println("Its absolute path is: " +
   testFilePath.toAbsolutePath());
System.out.println("Its normalized path is: " + testFilePath.normalize());
// get another path object with normalized relative path
Path testPathNormalized = Paths.get(testFilePath.normalize().toString());
System.out.println("Its normalized absolute path is: " +
   testPathNormalized.toAbsolutePath());
System.out.println("Its normalized real path is: " +
   testFilePath.toRealPath (LinkOption.NOFOLLOW_LINKS));
```

Note the exception coming from calling the last method which is the `toRealPath`, already mentioned why this is happening above. Also take a note at how the different paths are printed out, The `URI` and absolute path include the slash and the dot, however the normalized absolute path does not, this is why it is usually a good practice to normalize a path, especially if it is derived from user input, before any work is done on it.

```
The file name is: Test
Its URI is: file:///D:/OCPJP/programs/NIO2/./Test
Its absolute path is: D:\OCPJP\programs\NIO2\.\Test
Its normalized path is: Test
Its normalized absolute path is: D:\OCPJP\programs\NIO2\Test
Exception in thread "main" java.nio.file.NoSuchFileException:
   D:\OCPJP\programs\NIO2\Test
    at
       sun.nio.fs.WindowsException.translateToIOException(WindowsException.java
    [... stack trace elided ...]
    at PathInfo2.main(PathInfo2.java:16)
```

The toPath()method in java.io.File class returns the Path object this method was added in Java 7. Similarly one can use the toFile()method in the Path interface to get a File object, this is one of the many bridges between the old IO and the NIO which Java provides out of the box

Comparison

The Path interface provides two methods to compare Path objects `equals()` and `compareTo()`. The `equals()` method checks the equality of two Path objects and returns `true or false` the `compareTo()` compares two Path objects character by character and returns and integer 0 - if Paths are equal, or negative integer if this path is `lexicographically` less than the parameter path; and a positive integer if the path is `lexicographically` greater than the parameter path.

```
Path path1 = Paths.get("Test");
Path path2 = Paths.get("D:\\OCPJP\\programs\\NIO2\\Test");
// comparing two paths using compareTo() method
System.out.println("(path1.compareTo(path2) == 0) is: " +
   (path1.compareTo(path2) == 0));
// comparing two paths using equals() method
System.out.println("path1.equals(path2) is: " + path1.equals(path2));
// comparing two paths using equals() method with absolute path
System.out.println("path2.equals(path1.toAbsolutePath()) is " +
   path2.equals(path1.toAbsolutePath()));
```

Assume that the current directory/working directory from where this program is invoked is exactly `D:\\OCPJP\\programs\\NIO2\\`, the equal checks below will provide the following output. Note that comparing the actual path1 and Path2 yields results which imply they are not equal, however once the relative path is converted to an absolute path, then the equal check is passing fine. This is because the Path does not make any assumptions, which are hidden away from the API consumer. Even though semantically and logically those paths are indeed the `same` even though one is relative the other not, the Path instances, data fields do not hold the `same` path, that is why the methods rightfully return the results they do. Which is a good design, one would not want the library to make broad assumptions, such that it would have considered both the relative and absolute path equal.

```
(path1.compareTo(path2) == 0) is: false
path1.equals(path2) is: false
path2.equals(path1.toAbsolutePath()) is true
```

Files

The previous sections discussed how to create and work with Path instance, and extract useful information from it. In this section Path objects are used to manipulate files or directories. Ava 7 offers Files class (in the java.nio.file package). Those can be used to perform various file related operations on files or directories. Note that Files is a utility class, meaning it is final class with private constructor and consists only of static methods. So you can use the Files class by calling the static methods it provides, such as copy() to copy files. This class provides a wide range of functionality.

Method	Description
Path createDirectory(Path dirPath, FileAttribute<?>... dirAttrs) Path createDirectories(Path dir, FileAttribute<?>... attrs)	Creates a file given by the dirPath, and sets the attributes given by dirAttributes. May throw an exception such as FileAlreadyExistsException or UnsupportedOperationException (for example, when the file attributes cannot be set as given by dirAttrs). The difference between createDirectory and createDirectories is that createDirectories creates intermediate directories given by dirPath if they are not already present.
Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs)	Creates a temporary file with the given prefix, suffix, and attributes in the directory given by dir.
Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)	Creates a temporary directory with the given prefix and directory attributes in the path specified by dir.
Path copy(Path source, Path target, CopyOption... options)	Copies the file from source to target. CopyOption can be REPLACE_EXISTING, COPY_ATTRIBUTES, or NOFOLLOW_LINKS. Can throw exceptions such as FileAlreadyExistsException.
Path move(Path source, Path target, CopyOption... options)	Similar to the copy operation, but the source file is removed. If the source and target are in the same directory, it is a file-rename operation.
boolean isSameFile(Path path, Path path2)	Checks whether the two Path objects locate the same file.
boolean exists(Path path, LinkOption... options)	Checks whether a file/directory exists in the given path; can specify LinkOption.NOFOLLOW_LINKS to not to follow symbolic links.
Boolean isRegularFile(Path path, LinkOption...)	Returns true if the file represented by path is a regular file.
Boolean isSymbolicLink(Path path)	Returns true if the file represented by path is a symbolic link.
Boolean isHidden(Path path)	Return true if the file represented by path is a hidden file.
long size(Path path)	Returns the size of the file represented by path in bytes.
UserPrincipal getOwner(Path path, LinkOption...) Path setOwner(Path path, UserPrincipal owner)	Gets/sets the owner of the file.
FileTime getLastModified-Time(Path path, LinkOption...) Path setLastModifiedTime(Path path, FileTime time)	Gets/sets the last modified time for the specified file.
Object getAttribute(Path path, String attribute, LinkOption...) Path setAttribute(Path path, String attribute, Object value, LinkOption...)	Gets/sets the specified attribute of the specified file

Metadata

To quickly check some basic file `metadata` on files. In previous section the example which was trying to figure out if two paths pointed to the same file. There is another way to do this in a more robust way. To do that one can use the `isSameFile` method instead.

```
Path path1 = Paths.get("Test");
Path path2 = Paths.get("D:\\OCPJP\\programs\\NIO2\\Test");
System.out.println("Files.isSameFile(path1, path2) is: " +
   Files.isSameFile(path1, path2));
```

Notice that the first path is relative, to the current directory, however the second one is an absolute one, now also assume that the directory exists on disk. In case the file or directory does not exist in the given path, an exception will occur `NoSuchFileException`. There is an easy way to figure out if a file or directory first exists on the disk. To do that one can use the methods `isDirectory` or `exists`.

```
Path path = Paths.get("Test");
if(Files.exists(path, LinkOption.NOFOLLOW_LINKS)) {
    System.out.println("The file/directory " + path.getFileName() + "
       exists");
    // check whether it is a file or a directory
    if(Files.isDirectory(path, LinkOption.NOFOLLOW_LINKS)) {
        System.out.println(path.getFileName() + " is a directory");
    } else {
        System.out.println(path.getFileName() + " is a file");
    }
}
```

In this example above, the path is first checked for existence, in case it does exist, it can then be checked further if it points to a directory or to a file, note that `exists` alone can not tell if the path is directory or file, neither can the client of Path tell that from the path itself. There are operating systems where both the files and directories can have the same names, and many files have no extensions, which means it is pretty hard to tell for sure, in a reliable way if a given path is a file or directory.

These methods can be used to avoid exceptions being thrown and handled in the code, which is generally a good practice. Further more to check if one has access to read or access given file or directory, the `isReadable` or `isWritable` methods can be used. There is also a method to check if a path points to an executable `isExecutable`

```
// Given the fact that the path here is a directory, which we would assume
   already exists, the readable and writable
// calls should return true, however executable will not
Path path = Paths.get("D:\\OCPJP\\programs\\NIO2\\Test");
System.out.printf( "Readable: %b, Writable: %b, Executable: %b ",
Files.isReadable(path), Files.isWritable(path), Files.isExecutable(path));
```

To obtain a broader view of the attributes of a file, the snippet below obtains some of the more often used file or directory properties, note that it is also possible to obtain OS specific properties such as the dos:hidden one, which is in this case Windows specific.

```
Object object = Files.getAttribute(path, "creationTime",
   LinkOption.NOFOLLOW_LINKS);
System.out.println("Creation time: " + object);
```

```
object = Files.getAttribute(path, "lastModifiedTime",
   LinkOption.NOFOLLOW_LINKS);
System.out.println("Last modified time: " + object);

object = Files.getAttribute(path, "size", LinkOption.NOFOLLOW_LINKS);
System.out.println("Size: " + object);

object = Files.getAttribute(path, "dos:hidden", LinkOption.NOFOLLOW_LINKS);
System.out.println("isHidden: " + object);

object = Files.getAttribute(path, "isDirectory",
   LinkOption.NOFOLLOW_LINKS);
System.out.println("isDirectory: " + object);
```

The tricky part of the example is the second parameter, of the `getAttribute`. One needs to provide a correct attribute name pattern to extract the associated value. The expected string should be specified in view:attribute format, where view is the type of `FileAttributeView`, and attribute is the name of attribute supported by the view. If no view is specified it is assumed to be basic. In this case the `creationTime`, size and `isDirectory`, are base view type attributes, those are cross OS. If the name of the view:attribute is wrong, `UnsupportedOperationException` will be thrown

There is another way to read the attributes, in one shot of a given file or directory element, instead of using the regular method to get them one by one, and using the raw name of the attribute, the Java library provides a way to extract all Basic or Operating System dependent ones.

```
BasicFileAttributes fileAttributes = Files.readAttributes(path,
   BasicFileAttributes.class);
System.out.println("File size: " + fileAttributes.size());
System.out.println("isDirectory: " + fileAttributes.isDirectory());
System.out.println("isRegularFile: " + fileAttributes.isRegularFile());
System.out.println("isSymbolicLink: " + fileAttributes.isSymbolicLink());
System.out.println("File last accessed time: " +
   fileAttributes.lastAccessTime());
System.out.println("File last modified time: " +
   fileAttributes.lastModifiedTime());
```

There are `PosixFileAttributes` and `DosFileAttributes`. For `POSIX` and `DOS` compliant OS-es, which can be used to obtain the OS specific file attributes as well, however before that a check for the current operating system type must be performed, so to use the correct type of attribute types

Copying

Copying a file is relatively straight forward, one can simply use the `copy` method on the Files class, however note that the 3rd argument to the copy method is the file copy flags or attribues, which specify what to in case the file already exists and so on.

```
// imagine one runs the following on a given source - destination paths,
   to copy file from src to dst
Files.copy(pathSource, pathDestination);
// doing the same call again, would yield FileAlreadyExistsException, this
   is because by default files will not be overridden
Files.copy(pathSource, pathDestination);
```

```
// to force override the file even if it exists pass in a standard copy
   option operation to the 3rd argument of copy
Files.copy(pathSource, pathDestination,
   StandardCopyOption.REPLACE_EXISTING);
```

Note that if the destination is a directory path that does not exist, `NoSuchFileException` will be thrown, the full destination where the copy will occur must exist, if a file is to be copied, if a directory is to be copied then the full location to the destination without the target directory must exist

```
Assume that Test is a directory, then the source to be copied has to
   exist, along with the destination location in this case -
   D:\\OCPJP\\programs\\NIO2\\Deep\\Directory\\Location
source -> D:\\OCPJP\\programs\\NIO2\\Test
dest -> D:\\OCPJP\\programs\\NIO2\\Deep\\Directory\\Location\\TestCopy

Assume that test.txt is a file, then the source to be copied has to exist,
   along with the destination location in this csae -
   D:\\OCPJP\\programs\\NIO2\\Deep\\Directory\\Location
source -> D:\\OCPJP\\programs\\NIO2\\test.txt
dest -> D:\\OCPJP\\programs\\NIO2\\Deep\\Directory\\Location\\test-copy.txt
```

```
Note that copy does not copy the files or directories contained in a directory, it will
only copy the top-level directory not the files/directories contained within that directory.
If you copy a directory using the copy method it does not copy the files or directories
contained inside the source directory, they have to be copied recursively/explictily
```

Moving

Moving a file is similar to copying, however it is somewhat different as well, first it is similar to copy operation because it has to take into account that if another file or directory already exists in that location, explicit replacing strategy has to be provided to the move method, otherwise it will fail.

```
// provide explicit instructions to replace the destination if it already
   exists, otherwise exception will be thrown, FileAlreadyExistsException
Files.move(pathSource, pathDestination,
   StandardCopyOption.REPLACE_EXISTING);
```

- if a syn-link is being moved, the symbolic-link will be moved not the file or directory itself, there is a way to move the file itself, by suing the FOLLOW_LINKS option instead.
- a non empty directory can be moved if moving the directory does not require moving the containing files or directories, for instance moving a directory from one location to another may be unsuccessful if moving directory is successful then all the contained files and directories are moved too.
- move can be specified as an atomic operation, ATOMIC_MOVE copy option, with this option either the entire move completes successfully or the source continues to be present, and not moved, if move is not performed as atomic operation, and it fails while in process of moving, the state of both files or directories is unknown, and undefined

Deleting

The Files class provides a delete method as well. To delete a file directory or symbolic link. There are few points to remember about deleting, especially a directory, the delete method should be invoked on an empty

directory, otherwise it will fail. In the case of a symbolic link, the link is deleted not the target file of the link. The file or directory intend to be deleted must exist otherwise a `NoSuchFileException` will be thrown. To silently delete one can instead use the `deleteIfExists` method, which does not complain if the file does not exist and deletes the file if it exists. Also if a file is read-only on some platforms may prevent you from deleting the file

```
// the API of the delete method is quite simple, it accepts only the path
   target to be deleted, no other options or
// actions can be passed to this method or the deleteIfExists one
Files.delete(pathSource);
```

Notes

Some general notes, in relation to the classes already reviewed

- Do not confuse File with Files and Path with Paths: those are very different. File is an old class, that represents file or directory path names, whereas Files was introduced in Java 7 as a utility class with comprehensive support for IO. The Path interface presents a file or directory path unit, and defines a useful list of methods. However Paths is a utility class that offers only two methods - to get construct path objects
- The file or directory represented by a Path object may not exist. Other than methods such as toRealPath methods in Path do not require that the underlying file or directory be present for a Path object.

Streaming

As already mentioned the Stream API, added in Java 8, can be used along with the new Files and Path classes from NIO. There are several new methods in the Files class which produce streams which can then be used to read file contents or metadata

List

The list method, lists all files or directories in the current directory, internally it is using `DirectoryStream`, and hence the close method must be called to release the IO resources.

```
// Note that the list method returns a stream, but also that stream is one
   that holds `IO resources` hence the
// try-with-resources, which will close the Stream, which in this case is
   `DirectoryStream` underneath, it is quite smart
// actually since `DirectoryStream` itself does not extend Stream, however
   the iterator of `DirectoryStream` is wrapped around
// an anonymous Stream implementation
try(Stream<Path> entries = Files.list(Paths.get("."))) {
    entries.forEach(System.out::println);
}
```

Also the list method does not do any recursive traversal, just lists the entries in the current path, there is a recursive version of list, which is walk, which walks the tree formed by the path passed in as argument.

```
// the `FileVisitOption` has one enumeration value - FOLLOW_LINKS, also
   take note of the second argument in the overloaded
// version - `maxDepth`, that is usually useful to take into account when
   using walk. Note that using FOLLOW_LINKS does not
```

```
// guarantee that infinite recursion can not occur. The walk method has no
   built in loop detection. I.e having a symbolic link
// somewhere in the tree formed by the current `path` point to a parent
   directory of `path`
static Stream<Path> walk(Path path, FileVisitOption... options) throws
   IOException
static Stream<Path> walk(Path path, int maxDepth, FileVisitOption...
   options) throws IOException
```

Here is a simple usage, which walks the current directory, along with symbolic links, as well as restricting the depth to a maximum of 4 levels or directories deep.

```
try(Stream<Path> entries = Files.walk(Paths.get("."), 4,
   FileVisitOption.FOLLOW_LINKS)) {
    long numOfEntries = entries.count();
    System.out.printf("Found %d entries in the current path",
       numOfEntries);
}
```

Find

The find method is used to locate a file, somewhere in the current directory tree formed by the `path` argument. Find is built off off walk, it does essentially the same operations, as walk, with the only difference being that only the files that match the predicate are returned by the stream. It is effectively like applying a filter stream operation to `walk`, however it is much more efficient, the method signature of find matches the walk method too

```
// return a Stream that is lazily populated with Path by searching for
   files in a file tree rooted at a given
// starting file. This method walks the file tree in exactly the manner
   specified by  the #walk walk method. For
// each file encountered, the given BiPredicate is invoked with its Path
   and BasicFileAttributes. The Path object is
// obtained as if by
static Stream<Path> find(Path path, FileVisitOption... options) throws
   IOException
static Stream<Path> find(Path path, int maxDepth, FileVisitOption...
   options) throws IOException
```

Lines

There is a way to also obtain the lines from a file, using the `lines` or `readAllLines` methods. Both are slightly different as `lines` will return a `Stream<String>`, which internally is an anonymous implementation that wraps around the NIO operations, the stream can then be used to read the lines lazily. Meaning that depending on the type of operation (findFirst, findAny, etc) there might be a case where not the entire file is read, or for example only parts of the file is read (say first N number of lines, then the stream is closed, which closes the internal file resources held by it). The `readAllLines` however is different it eagerly reads the entire file and all of the lines in the file and returns a `List<String>`

```
// Obtain a stream object to the lines of the file pointed by path, note
   that the returned Stream, is internally
// anonymous implementation which wraps the file resource and IO
   operations to obtain the lines lazily while and when the
```

```
// stream is being iterated over. This can prove more efficient depending
   on what operations are applied on the Stream, the
// example below is however not ideal case, since it goes through the
   entire file and prints all lines. However that
// might not be always the case. Internally the Files.lines is using
   `BufferedReader.lines` (which returns a stream)
try(Stream<String> lines = Files.lines(Paths.get(filePath))) {
    lines.forEach(System.out::println);
} catch (IOException ioe) {
    System.err.println("IOException occurred when reading the file...
       exiting");
    System.exit(-1);
}
```

Summary

Path interface & operations

- A Path object is a programming abstraction to represent the path of a file/directory.
- You can get an instance of Path using the `get()` method of the Paths class.
- Path provides two methods to compare Path objects: `equals()` and `compareTo()`. Even if two Path objects point to the same file/directory, the `equals()` method is not guaranteed to return true.

Read, delete, copy, move, manage `metadata`

- You can check the existence of a file using the `exists()` method of the Files class.
- The Files class provides the methods `isReadable()`, `isWritable()`, and `isExecutable()` to check the ability of the program to read, write, and execute programmatically, respectively.
- You can retrieve the attributes of a file using the `getAttributes()` method.
- You can use the `readAttributes()` method of the Files class to read file attributes in bulk.
- The `copy()` method can be used to copy a file from one location to another. Similarly, the `move()` method moves a file from one location to another.
- While copying, all the directories (except the last one, if you are copying a directory) on the specified path must exist to avoid a `NoSuchFileException`.
- Use the `delete()` method to delete a file; use the `deleteIfExists()` method to delete a file only if it exists.

Stream & NIO.2

- The Files.`list()` method returns a `Stream<Path>`. It does not recursively traverse the directories in the given Path.
- The Files.`walk()` method returns a `Stream<Path>` by recursively traversing the entries from the given Path; in one of its overloaded versions, you can also pass the maximum depth for such traversal and provide `FileVisitOption.FOLLOW_LINKS` as the third option.
- The Files.`find()` method returns a `Stream<Path>` by recursively traversing the entries from the given Path; it also takes the maximum depth to search, a `BiPredicate`, and an optional `FileVisitOption` as arguments.

- Files.`lines()` is a very convenient method to read the contents of a file. It returns a `Stream<String>`.