8 - NIO.2

< Previous | Next >

§8.1 Files class

 # `public static Path copy(Path source, Path target, CopyOption... options) throws IOException` :

- Copy a file to a target file. This method copies a file to the target file with the options parameter specifying how the copy is performed. By default, the copy fails if the target file already exists or is a symbolic link, except if the source and target are the same file, in which case the method completes without copying the file. File attributes are not required to be copied to the target file. If symbolic links are supported, and the file is a symbolic link, then the final target of the link is copied. If the file is a directory then it creates an empty directory in the target location (entries in the directory are not copied).
- The options parameter may include any of the following:
    - `REPLACE_EXISTING` - If the target file exists, then the target file is replaced if it is not a non-empty directory. If the target file exists and is a symbolic link, then the symbolic link itself, not the target of the link, is replaced.
    - `COPY_ATTRIBUTES` - Attempts to copy the file attributes associated with this file to the target file. The exact file attributes that are copied is platform and file system dependent and therefore unspecified. Minimally, the last-modified-time is copied to the target file if supported by both the source and target file store. Copying of file timestamps may result in precision loss.
    - `NOFOLLOW_LINKS` - Symbolic links are not followed. If the file is a symbolic link, then the symbolic link itself, not the target of the link, is copied. It is implementation specific if file attributes can be copied to the new link. In other words, the `COPY_ATTRIBUTES` option may be ignored when copying a symbolic link. An implementation of this interface may support additional implementation specific options.
- Copying a file is not an atomic operation. If an IOException is thrown then it possible that the target file is incomplete or some of its file attributes have not been copied from the source file. When the `REPLACE_EXISTING` option is specified and the target file exists, then the target file is replaced. The check for the existence of the file and the creation of the new file may not be atomic with respect to other file system activities.

 # `public static Path move(Path source,  Path target, CopyOption... options) throws IOException` :

- Move or rename a file to a target file. By default, this method attempts to move the file to the target file, failing if the target file exists except if the source and target are the same file, in which case this method has no effect. If the file is a symbolic link then the symbolic link itself, not the target of the link, is moved. Possible options parameters: `StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.ATOMIC_MOVE` .
- When the CopyOption argument of the move method is `StandardCopyOption.ATOMIC_MOVE` , the operation is implementation dependent if the target file exists. The existing file could be replaced or an IOException could be thrown.

 # `public static boolean isSameFile(Path path, Path path2) throws IOException` :

Tests if two paths locate the same file. If both Path objects are equal then this method returns true without checking if the file exists. If the two Path objects are associated with different providers then this method returns false. Otherwise, this method checks if both Path objects locate the same file, and depending on the implementation, may require to open or access both files.

 # Files.lines():

- Two Files.lines methods. One takes just a Path and the second list method allows you the specify the charset of the source file as well:
    - `Stream<String> lines = Files.lines(Paths.get("test.txt"));`
    - `Stream<String> lines = Files.lines(Paths.get("test.txt"), Charset.defaultCharset());`

 # Files.walk():

- `public static Stream<Path> walk(Path start, FileVisitOption... options) throws IOException` Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
- `public static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options) throws IOException` Return a Stream that is lazily populated with Path by walking the file tree rooted at a given starting file.
- The Files class does not have a walk method that takes a string as the second parameter.

 # Files.find():

- `public static Stream<Path> find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options)` Return a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.

 # Files.list():

- `public static Stream<Path> list(Path dir) throws IOException` Return a lazily populated Stream, the elements of which are the entries in the directory. The listing is not recursive.

§8.2 Path Resolution

 # `public Path normalize()` :

- Returns a path that is this path with redundant name elements eliminated.

`public Path resolve(Path other)` :

- Joining Two Paths: You can combine paths by using the resolve method. You pass in a partial path , which is a path that does not include a root element, and that partial path is appended to the original path. Passing an absolute path to the resolve method returns the passedin path.
- If the argument is a relative path (i.e. if it doesn't start with a root), the argument is simply appended to the path to produce the result.
- If the other parameter is an absolute path then this method trivially returns other. If other is an empty path then this method trivially returns this path. Otherwise this method considers this path to be a directory and resolves the given path against this path. In the simplest case, the given path does not have a root component, in which case this method joins the given path to this path and returns a resulting path that ends with the given path. Where the given path has a root component then resolution is highly implementation dependent and therefore unspecified.

`public Path relativize(Path other)` :

- Creating a Path Between Two Paths: This method constructs a path originating from the original path and ending at the location specified by the passedin path. The new path is relative to the original path.
- Constructs a relative path between this path and a given path. Relativization is the inverse of resolution. This method attempts to construct a relative path that when resolved against this path, yields a path that locates the same file as the given path. For example, "a/c" relativize "a/b" is "../b" because "/a/c/../b" is "/a/b". Notice that "c/.." cancel out.
- Where this path and the given path do not have a root component, then a relative path can be constructed.
- A relative path cannot be constructed if only one of the paths have a root component (it will throw an IllegalArgumentException).
- Where both paths have a root component then it is implementation dependent if a relative path can be constructed.
- If this path and the given path are equal then an empty path is returned.
- For any two normalized paths p and q, where q does not have a root component, `p.relativize(p.resolve(q)).equals(q)`
- When symbolic links are supported, then whether the resulting path, when resolved against this path, yields a path that can be used to locate the same file as other is implementation dependent. For example, if this path is "/a/b" and the given path is "/a/x" then the resulting relative path may be "../x". If "b" is a symbolic link then is implementation dependent if "a/b/../x" would locate the same file as "/a/x".
- The relativize method does not take a String as an argument. It takes a Path object!
- The relativize method does not normalize any of the paths!

`public Path resolveSibling(String other)` or `public Path resolveSibling(Path other)` :

- The method resolveSibling is meant for cases in which you are trying to get the absolute path for a file that exists in the same directory as the original file.
- Resolves the given path against this path's parent path. This is useful where a file name needs to be replaced with another file name.
- If this path does not have a parent path, or other is absolute, then this method returns other.
- If other is an empty path then this method returns this path's parent, or where this path doesn't have a parent, the empty path.

`Path.getName(int index)` :

1. Indices for path names start from 0.
2. Root (i.e. c:) is not included in path names.
3. \ is NOT a part of a path name.
4. If you pass a negative index or a value greater than or equal to the number of elements, or this path has zero name elements, java.lang.IllegalArgumentException is thrown. It DOES NOT return null.

`Path getRoot()` :

- Returns the root component of this path as a Path object, or null if this path does not have a root component. (c:\ or /).

`Path.subpath(int beginIndex, int endIndex)` :

1. Indexing starts from 0.
2. Root (i.e. c:) is not considered as the beginning.
3. name at beginIndex is included but name at endIndex is not.
4. paths do not start or end with \.

`Paths.get(URI)` :

- `public static Path get(URI uri)`
- Converts the given URI to a Path object.
- Throws:
  - IllegalArgumentException - if preconditions on the uri parameter do not hold. The format of the URI is provider specific.
  - **FileSystemNotFoundException - The file system, identified by the URI, does not exist and cannot be created automatically, or the provider identified by the URI's scheme component is not installed.**
  - SecurityException - if a security manager is installed and it denies an unspecified permission to access the file system.

< Previous | Next >