21.8 Stream Operations on Directory Entries

The `Files` class provides static methods that create specialized streams to implement complex file operations on directory entries, which are concise and powerful, taking full advantage of the Stream API. They are also efficient as the lazy execution of the streams ensures that an element is only made available for processing when needed by the terminal stream operation.

Streams we have seen earlier do not use any system resources, as they are backed by data structures and generator functions. Such streams need not be closed, as they are handled as any other objects by the runtime environment with regard to memory management. In contrast, the streams that are backed by *file system resources* must be closed in order to avoid resource leakage—analogous to using I/O streams. As streams implement the `AutoCloseable` interface, the recommended practice is to use the `try`-with-resources statement, ensuring proper and prompt closing of file system resources after the stream operations complete.

Methods that create streams on directory entries throw a checked `IOException`, and in particular, they throw a `NoSuchFileException` if the directory entry does not exist. Any code using these methods is forced to handle these exceptions with either a `try-catch-finally` construct or a `throws` clause.

The examples in this section illustrate both closing of streams and handling of exceptions.

Reading Text Lines Using a Functional Stream

The `lines()` method of the `Files` class creates a stream that can be used to read the lines in a text file. It is efficient as it does not read the whole file into memory, making available only one line at a time for processing. Whereas the `BufferedReader` class provides an analogous method that uses a `BufferedReader`, the stream created by the `Files.lines()` method is backed by a more efficient `FileChannel`.

```
static Stream<String> lines(Path path) throws IOException
static Stream<String> lines(Path path, Charset cs) throws IOException
```

Return a `Stream` of type `String`, where the elements are text lines read from a file denoted by the specified `path`. The first method decodes the bytes into characters using the UTF-8 charset. The charset to use can be explicitly specified, as in the second method.

As an example of using the `lines()` method, we implement the solution to finding palindromes (§13.3, p. 688), where the words are read from a file.

```
Path wordFile = Path.of(".", "project", "wordlist.txt");
System.out.println("Find palindromes, greater than length 2.");
try (Stream<String> stream = Files.lines(wordFile)){
  List<String> palindromes = stream
       .filter(str -> str.length() > 2)
       .filter(str -> str.equals(new StringBuilder(str).reverse().toString()))
       .toList();
  System.out.printf("List of palindromes:   %s%n", palindromes);
  System.out.printf("Number of palindromes: %s%n", palindromes.size());
} catch (IOException e) {
  e.printStackTrace();
}
```

Possible output from the code (*edited to fit on the page*):

```
Find palindromes, greater than length 2.
List of palindromes:   [aba, abba, aga, aha, ...]
Number of palindromes: 90
```

The following code creates a map to count the number of lines with different lengths:

```
Path textFile = Path.of(".", "project", "linesOnly.txt");
try (Stream<String> stream = Files.lines(textFile)) {
  Map<Integer, Long> grpMap =
      stream.collect(Collectors.groupingBy(String::length,
                                           Collectors.counting()));
  System.out.println(grpMap);
} catch (IOException e) {
  e.printStackTrace();
}
```

Possible output from the code:

```
{4=1, 21=2, 10=1, 12=1}
```

Directory Hierarchy Traversal

Tree traversal is an important topic in computer science. Other synonyms used in the literature for traversing such structures are *walking*, *navigating*, and *visiting* a tree.

Note that in order to traverse a directory hierarchy and access its entries, the directory must have *execute* permission.

The `Files` class provides methods to traverse a directory hierarchy which has a tree structure. We will use the directory hierarchy shown below to illustrate these traversal methods. The directory hierarchy below is rooted at directory `a`—that is, this directory is the *start* or the *root* of the directory hierarchy. Traversal of the directory hierarchy starts by visiting directory `a` and is considered to be at *depth* 0. Dropping down each level of directories in the hierarchy increases the depth. The directory `a/b` and the file `a/x.txt` are at depth 1 in relation to the directory `a`, and are said to be *siblings* at depth 1. The directory hierarchy below has a maximum depth of 3, as there are no directories below this depth. Note that the file `a/b/c/dir_link` is a symbolic link to the directory `a/b`, and as we shall see, it can influence the traversal if symbolic links are to be followed.

```
a/
├── b/ ◄------------------------┐
│   ├── c/                      ¦
│   │   ├── dir_link------------┘
│   │   └── z.txt
│   └── d/
│       └── y.txt
└── x.txt
0   1   2   3 ---> Increasing depth
```

Traversal Strategies

There are primarily two main strategies for traversing a tree structure, or as in our case, a directory hierarchy. Both strategies start at the root of the directory hierarchy.

- *Depth-first traversal*

  Any subdirectory encountered at a particular depth level is traversed to its maximum depth before continuing with any sibling of this subdirectory at this depth level. The depth-first traversal of the hierarchy rooted at directory `a` is shown below. Note how at depth level 2, the directory `a/b/c` is traversed completely before traversing its sibling directory `a/b/d`. The traversal methods of the `Files` class use the depth-first traversal strategy.

```
Visited entry      Depth
./a                0
./a/x.txt          1
./a/b              1
./a/b/c            2
./a/b/c/z.txt      3
./a/b/c/dir_link   3
./a/b/d            2
./a/b/d/y.txt      3
```

- *Breadth-first traversal*

  The siblings at each depth level are traversed before moving on to the next depth level—in other words, the traversal is by depth level. The breadth-first traversal of the hierarchy rooted at directory `a` is shown below. Note how entries at each depth are traversed before traversing the entries at the next depth level.

```
Visited entry      Depth
./a                0
./a/x.txt          1
./a/b              1
./a/b/d            2
./a/b/c            2
./a/b/c/z.txt      3
./a/b/c/dir_link   3
./a/b/d/y.txt      3
```

Both strategies have their pros and cons, and each excels at solving particular traversal problems. For example, finding a search goal that is closest to the root is best found by the breadth-first strategy in the shortest amount of time, whereas a goal that is farthest from the root is best found by the depth-first strategy in the shortest amount of time. Depth-first search may need to backtrack when trying to find a solution, whereas breadth-first search is more predictive. Breath-first search requires more memory as entries at previous levels must be maintained. The interested reader should consult the ample body of literature readily available on this important subject of trees and graph algorithms.

Listing the Immediate Entries in a Directory

A common file operation is to list the entries in a directory, similar to basic usage of the DOS command `dir` or the Unix command `ls`. The `list()` method of the `Files` class creates a stream whose elements are `Path` objects that denote the entries in a directory.

```
static Stream<Path> list(Path dir) throws IOException
```

Creates a lazily populated `Stream` whose elements are the entries in the directory—it does *not* recurse on the entries of the directory. The stream does not include the directory itself or its parent. It throws a `java.nio.file.NotDirectoryException` if it is not a directory (*optional specific exception*). Also, it does not follow symbolic links.

The following code lists the entries under the directory `a`. Note that the stream only contains the *immediate* entries in the directory. Any subdirectories under the specified directory are *not* traversed. The `Path` object passed a parameter must denote a directory, or a disgruntled `NotDirectoryException` is thrown.

```
Path dir = Path.of(".", "a");
System.out.printf("Immediate entries under directory \"%s\":%n", dir);
try(Stream<Path> stream = Files.list(dir)) {
  stream.forEach(System.out::println);
} catch (NotDirectoryException nde) {
nde.printStackTrace();
} catch (IOException ioe) {
ioe.printStackTrace();
}
```

Output from the code:

```
Immediate entries under directory "./a":
./a/x.txt
./a/b
```

File Visit Option

In the file operations we have seen so far, the default behavior was to follow symbolic links. Not so in the case of the traversal methods `find()` and `walk()` of the `Files` class—these methods do *not* follow symbolic links, unless instructed explicitly to do so. The `FileVisitOption` enum type defines the constant `FOLLOW_LINKS` (Table 21.13) that can be specified to indicate that symbolic links should be followed when using these methods.

Table 21.13 *File Visit Option*

Enum `java.nio.file.FileVisitOption` constant	Description
`FOLLOW_LINKS`	Indicates that symbolic links should be followed.

Walking the Directory Hierarchy

The `walk()` method of the `Files` class creates a stream to walk or traverse the directory hierarchy rooted at a specific directory entry.

```
static Stream<Path> walk(Path start, FileVisitOption... options)
                         throws IOException
static Stream<Path> walk(Path start, int depth,
                         FileVisitOption... options)
                         throws IOException
```

Return a `Stream` that is lazily populated with `Path` objects by walking the directory hierarchy rooted at the entry specified by the `start` parameter. The `start` parameter can be a directory or a file.

The first method is equivalent to calling the second method with a depth of `Integer.MAX_VALUE`. The methods traverse the entries *depth-first* to a depth limit that is the minimum of the maximum depth of the directory hierarchy rooted at the `start` entry and any depth that is specified or implied.

These methods do *not* follow symbolic links, unless the constant `FileVisitOption.FOLLOW_LINKS` is specified.

Example 21.5 illustrates using the `walk()` method. The hierarchy of directory `a` (p. 1347) is traversed to illustrate different scenarios. The code at (1) creates the symbolic link `a/b/c/dir_link` to the directory `a/b`, if necessary.

Example 21.5 *Traversing the Directory Hierarchy*

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

public class WalkTheWalk {

  public static void main(String[] args) throws IOException {

    // Creating symbolic link.                                   // (1)
    try {
      Path targetPath = Path.of(".", "a", "b");
      Path symbLinkPath  = Path.of(".", "a", "b", "c", "dir_link");
      if (Files.notExists(symbLinkPath, LinkOption.NOFOLLOW_LINKS)) {
        Files.createSymbolicLink(symbLinkPath, targetPath.toAbsolutePath());
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return;
    }

    // Do the walk.                                                // (2)
    Path start = Path.of(".", "a");
    int MAX_DEPTH = 4;
```

```
      for (int depth = 0; depth <= MAX_DEPTH; ++depth)  {               // (3)
        try(Stream<Path> stream = Files.walk(start, depth,              // (4)
                                         FileVisitOption.FOLLOW_LINKS)) {
          System.out.println("Depth limit: " + depth);
          stream.forEach(System.out::println);
        } catch (IOException ioe) {
          ioe.printStackTrace();
        }
      }
    }
  }
```

Output from the program (*edited to fit on the page*):

```
Depth: 0
./a
Depth: 1
./a
./a/x.txt
./a/b
Depth: 2
./a
./a/x.txt
./a/b
./a/b/c
./a/b/d
Depth: 3
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
./a/b/c/dir_link
./a/b/d
./a/b/d/y.txt
Depth: 4
./a
./a/x.txt
./a/b
./a/b/c
./a/b/c/z.txt
Exception in thread "main" java.io.UncheckedIOException:
    java.nio.file.FileSystemLoopException: ./a/b/c/dir_link
...
```

Limiting Traversal by Depth Specification

The `walk()` method at (4) in Example 21.5 is called multiple times in the `for(;;)` loop at (3). From the output, we can see that, for each value of the loop variable `depth`, the `walk()` method starts the traversal at the directory `a` and descends to the depth level given by the loop variable `depth`. For example, when the value of the `depth` variable is `3`, traversal descends from depth level 0 to depth level 3.

If the constant FileVisitOption.FOLLOW_LINKS is *not* specified, so that symbolic links are not followed, the traversal will only go as far as the minimum of the maximum depth of the hierarchy and any specified depth level.

Specifying the depth level can result in a more efficient search if it is known that the result is at an approximate depth level, thus avoiding useless searching farther down in the hierarchy.

Handling Symbolic Links

Specifying the constant FileVisitOption.FOLLOW_LINKS in the walk() method call results in symbolic links being followed.

A symbolic link to a file is not a problem, as after visiting the target file, the traversal can continue with the next sibling of the symbolic link.

A symbolic link to a directory that is *not* a *parent directory* of the symbolic link is also not a problem, as traversal can continue with the next sibling of the symbolic link, after visiting the target directory.

The problem arises when a symbolic link is to a directory that is a *parent directory* of the symbolic link. That creates a *cyclic path dependency*. An example of such a cyclic path dependency is introduced by the symbolic link a/b/c/dir_link to one of its parent directories, a/b, as can be seen in the figure of the directory hierarchy (p. 1347).

When the constant FileVisitOption.FOLLOW_LINKS is specified in the walk() method call, the method detects such cyclic path dependencies by monitoring which entries have been visited. Example 21.5 illustrates this scenario when the constant FileVisitOption.FOLLOW_LINKS is specified and the specified depth is greater than 3. From the output, we can see that, at depth level 4, the method detects that the symbolic link a/b/c/dir_link to its parent directory a/b creates a cyclic path dependency, as the target directory lies on the path to the symbolic link and has been visited before. The method throws a hefty FileSystemLoopException to announce the outcome.

Copying an Entire Directory

The Files.copy() method only copies a single file or creates an empty directory at its destination, depending on whether the source is a file or a directory, respectively. Shown below is the copyEntireDirectory() method that copies an *entire* directory.

```
/** Declared in FileUtils class.
 * Copy an entire directory.
 * @param sourceDir        Directory to copy.
 * @param destinationDir   Directory to which the source directory is copied.
 * @param options          Copy options for all entries.
 */
public static void copyEntireDirectory(Path sourceDir,
                                       Path destinationDir,
                                       CopyOption... options)  {
```

```
    try (Stream<Path> stream = Files.walk(sourceDir)) {                    // (1)
      stream.forEach(entry -> {
        Path relativeEntryPath = sourceDir.relativize(entry);             // (2)
        Path destination  = destinationDir.resolve(relativeEntryPath);    // (3)
        try {
          Files.copy(entry, destination, options);                        // (4)
        } catch (DirectoryNotEmptyException e) {
          e.printStackTrace();
        } catch (IOException e) {
          e.printStackTrace();
        }
      });
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

The method `copyEntireDirectory()` copies each entry with the `Files.copy()` method at (4) as the directory hierarchy is traversed in the stream created by the `Files.walk()` method at (1). For each entry, the destination where the entry should be copied is determined by the code at (2) and (3).

First, the *relative* path between the source directory path and the current entry path is determined by the `relativize()` method at (2):

```
Source directory:      ./a/b
Current entry:         ./a/b/c/d
Relative entry path:   c/d
```

The *destination* path to copy the current entry is determined by joining the destination directory path with the relative entry path by calling the `resolve()` method at (3):

```
Destination directory:   ./x/y
Relative entry path:     c/d
Destination entry paths: ./x/y/c/d
```

In the scenario above, the entry `./a/b/c/d` is copied to the destination path `./x/y/c/d` during the copying of source directory `./a/b` to the destination directory `./x/y`.

```
CopyOption[] options = new CopyOption[] {
    StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES,
    LinkOption.NOFOLLOW_LINKS};
Path sourceDirectory      = Path.of(".", "a", "b");       //      ./a/b
Path destinationDirectory = Path.of(".", "x", "y");       // (5a) ./x/y
// Path destinationDirectory = Path.of(".", "x")
//              .resolve(sourceDirectory.getFileName());  // (5b) ./x/b
FileUtils.copyEntireDirectory(sourceDirectory, destinationDirectory, options);
```

If the destination directory should have the same name as the source directory, we can use the code at (5b) rather than the code at (5a).

A few things should be noted about the declaration of the `copyEntireDirectory()` method. It can be customized to copy the entries by specifying copy options. It closes the stream created by the `walk()` method in a `try`-with-resources statement. Calling the method `copy()` in the body of the lambda expression requires handling any IOException inside the lambda body. Any IOException from calling the walk() method is also explicitly handled.

Searching for Directory Entries

The find() method of the Files class can be used for searching or finding directory entries in the file system.

```
static Stream<Path> find(Path start, int depth,
                         BiPredicate<Path,BasicFileAttributes> matcher,
                         FileVisitOption... options) throws IOException
```

Returns a Stream that is lazily populated with Path objects by searching for entries in a directory hierarchy rooted at the Path object denoted by the start parameter. The start parameter can be a directory or a file.

The method walks the directory hierarchy in exactly the same manner as the walk() method. The methods traverse the entries *depth-first* to a depth limit that is the minimum of the actual depth of the directory hierarchy rooted at the start entry and the depth that is specified.

The matcher parameter defines a BiPredicate that is used to decide whether a directory entry should be included in the stream. For each directory entry in the stream, the BiPredicate is invoked on its Path and its BasicFileAttributes—making it possible to define a customized filter.

This method does *not* follow symbolic links, unless the constant FileVisitOption.FOLLOW_LINKS is specified.

The find() method is analogous to the walk() method in many respects: It traverses the directory hierarchy in the same way, does not follow symbolic links by default, follows symbolic links only if the constant FileVisitOption.FOLLOW_LINKS is specified, and monitors visiting entries to detect cyclic path dependencies since the depth limit is always specified.

Whereas both walk() and find() methods create a stream of Path objects, the find() method also allows a matcher for the search to be defined by a lambda expression that implements the BiPredicate<Path, BasicFileAttributes> functional interface. This matcher is applied by the method and determines whether an entry is allowed in the stream. This is in contrast to an explicit intermediate filter operation on the stream, as in the case of a stream created by the walk() method. The find() method supplies the BasicFileAttributes object associated with a Path—analogous to using the readAttributes() method of the Files class (p. 1328). Given the Path object and its associated BasicFileAttributes object with the basic set of read-only file attributes, it is possible to write complex search criteria on a directory entry. The methods of the BasicFileAttributes interface also do not throw checked exceptions, and are therefore convenient to call in a lambda expression, as opposed to corresponding methods of the Files class.

In the following code, we use the `find()` method to find regular files whose name ends with `".txt"` and whose size is greater than 0 bytes. The `BiPredicate` at (1) defines the filtering criteria based on querying the `Path` object in the stream for its file extension and its associated `BasicFileAttributes` object with read-only file attributes for whether it is a regular file and for its size.

```
System.out.println("Find regular files whose name ends with \".txt\""
                 + " and whose size is > 0:");
Path startEntry = Path.of(".", "a");
int depth = 5;
try (var pStream = Files.find(startEntry, depth,
                              (path, attrs) -> attrs.isRegularFile()       // (1)
                              && attrs.size() > 0
                              && path.toString().endsWith(".txt"))) {
   List<Path> pList = pStream.toList();
   System.out.println(pList);
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Output from the code:

```
Find regular files whose name ends with ".txt" and whose size is > 0:
[./a/x.txt, ./a/b/c/z.txt, ./a/b/d/y.txt]
```

Note that the method `find()` requires the depth of the search to be specified. However, the actual depth traversed is always the minimum of the maximum depth of the directory hierarchy and the depth specified. The maximum depth of the hierarchy rooted at directory `a` is 3 and the specified depth is 5. The actual depth traversed in the directory hierarchy is thus 3. In the code above, different values for the depth can give different results.

If the constant `FileVisitOption.FOLLOW_LINKS` is specified in the `find()` method, its behavior is analogous to the behavior of the `walk()` method. It will keep track of the directories visited, and any cyclic path dependency encountered will unceremoniously result in a `FileSystemLoopException`. The curious reader is encouraged to experiment with the code above by specifying the constant `FileVisitOption.FOLLOW_LINKS` and passing different values for the depth in the `find()` method call.