Chapter TWENTY-FIVE

Files and Streams

New Stream Methods in NIO.2

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

static Stream<Path> list(Path dir)

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

static Stream<Path> walk(Path start,
                         FileVisitOption... options)
static Stream<Path> walk(Path start,
                         int maxDepth,
                         FileVisitOption... options)
```

An important thing to notice is that the returned streams are LAZY คือ จะยังไม่ถูก load หรือ read จนกว่าจะถูกเรียกใช้

Files.list()

- วน path ภายใต้ directory ชั้นเดียว (not recursive)
- Return เป็น Stream ของ Path **(ได้มาทั้ง directory และ file)** ไม่รวม directory แรก

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

ตัวอย่างโครงสร้าง

```
/temp/
   /dir1/
      /subdir1/
         111.txt
      /subdir2/
         121.txt
         122.txt
   /dir2/
      21.txt
      22.txt
   file.txt
```

ตัวอย่างการใช้งาน

```
try(Stream<Path> stream =
        Files.list(Paths.get("/temp"))) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

A possible output:

```
/temp/dir1
/temp/dir2
/temp/file.txt
```

Files.walk()

- วนตั้งแต่ directory และ subdirectories ทั้งหมด (recursively traverse the subdirectories.)
- return เป็น path เก็บในรูปแบบของ stream **(ได้มาทั้ง directory และ file)**

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

The following code:

```
try(Stream<Path> stream =
           Files.walk(Paths.get("c:/temp"))) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Will output:

```
/temp
/temp/dir1
/temp/dir1/subdir1
/temp/dir1/subdir1/111.txt
/temp/dir1/subdir2
/temp/dir1/subdir2/121.txt
/temp/dir1/subdir2/122.txt
/temp/dir2
/temp/dir2/21.txt
/temp/dir2/22.txt
/temp/file.txt
```

- Method Files.walk() สามารถกำหนด **depth of subdirectory** ได้ (ถ้าไม่กำหนด default = Integer.MAX_VALUE recursive ครบทุก subdirectory)
    - กำหนดค่า 0 หมายถึง level directory เริ่มต้น

```
try(Stream<Path> stream =
        Files.walk(Paths.get("/temp"), 1)) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

The output is:

```
/temp
/temp/dir1
/temp/dir2
/temp/file.txt
```

- Files.walk() doesn't follow symbolic links by default. To follow symbolic links, just use the argument of type FileVisitOption (preferably, also using the maximum depth argument) this way:

Symbolic Link เป็นการสร้างตัวอ้างอิงจากไฟล์ที่มีอยู่แล้ว ทำให้เมื่อไฟล์ต้นฉบับถูกลบ ข้อมูลในส่วนนั้นก็จะไม่สามารถเข้าถึงได้จาก Link ที่สร้างไว้ได้

Files.find()

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

- Files.find() คล้ายกับ Files.walk() แต่มี argument BiPredicate เพิ่มมาเพื่อใช้ในการ filter files และ directories

BiPredicate : รับ argument 2 ตัว return Boolean

- Argument ที่ 1 รับ Path ของ files หรือ directories
- Argument ที่ 2 BasicFileAttributes เช่น creation time, if it's a file, directory or symbolic link, size, etc.
- return boolean

The following example returns a stream that includes just directories:

```
BiPredicate<Path, BasicFileAttributes> predicate =
    (path, attrs) -> {
        return attrs.isDirectory();
};
int maxDepth = 2;
try(Stream<Path> stream =
        Files.find(Paths.get("/temp"),
                   maxDepth, predicate)) {
    stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

Files.lines()

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

- reads all the lines of a file as a stream of Strings.
- As the stream is lazy, it doesn't load all the lines into memory, only the line read at any given time. If the file doesn't exist, an exception is thrown.
- The file's bytes are decoded using the specified charset or with UTF-8 by default.

For example:

```
try(Stream<String> stream =
            Files.lines(Paths.get("/temp/file.txt"))) {
        stream.forEach(System.out::println);
} catch(IOException e) {
    e.printStackTrace();
}
```

In Java 8, a `lines()` method was added to `java.io.BufferedReader` as well:

```
Stream<String> lines()
```

The stream is lazy and its elements are the lines read from the `BufferedReader`.

Key Points

- In Java 8, new methods that return implementations of the `Stream` interface have been added to `java.nio.file.Files`.
- The returned streams are **LAZY**, which means that the elements are not loaded (or read) until they are used.
- The use of a `try-with-resources` with these methods is recommended so that the stream's `close` method can be invoked to close the file system resources.
- `Files.list()` iterates over a directory to return a stream whose elements are `Path` objects that represent the entries of that directory.
- This method lists directories and files of the specified directory. However, it is not recursive, in other words, it **DOESN'T** traverse subdirectories.
- `Files.walk()` also iterates over a directory in a depth-first strategy to return a stream whose elements are`Path` objects that represent the entries of that directory.
- The difference with `Files.list()` is that `Files.walk()` **DOES** recursively traverse the subdirectories. You can also pass the maximum traversal depth and an option to follow symbolic links.
- `Files.find()` is similar to `Files.walk()`, but takes an additional argument of type `BiPredicate<Path,BasicFileAttributes>` that is used to filter the files and directories.
- `Files.lines()` reads all the lines of a file as a stream of Strings without loading them all into memory.