The Java™ Tutorials

Trail: Essential Classes
Lesson: Basic I/O
Section: File I/O (Featuring NIO.2)

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

File Operations

The `Files` class is the other primary entrypoint of the `java.nio.file` package. This class offers a rich set of static methods for reading, writing, and manipulating files and directories. The `Files` methods work on instances of `Path` objects. Before proceeding to the remaining sections, you should familiarize yourself with the following common concepts:

- Releasing System Resources
- Catching Exceptions
- Varargs
- Atomic Operations
- Method Chaining
- What *Is* a Glob?
- Link Awareness

Releasing System Resources

Many of the resources that are used in this API, such as streams or channels, implement or extend the `java.io.Closeable` interface. A requirement of a `Closeable` resource is that the `close` method must be invoked to release the resource when no longer required. Neglecting to close a resource can have a negative implication on an application's performance. The `try`-with-resources statement, described in the next section, handles this step for you.

Catching Exceptions

With file I/O, unexpected conditions are a fact of life: a file exists (or doesn't exist) when expected, the program doesn't have access to the file system, the default file system implementation does not support a particular function, and so on. Numerous errors can be encountered.

All methods that access the file system can throw an `IOException`. It is best practice to catch these exceptions by embedding these methods into a `try`-with-resources statement, introduced in the Java SE 7 release. The `try`-with-resources statement has the advantage that the compiler automatically generates the code to close the resource(s) when no longer required. The following code shows how this might look:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

For more information, see The try-with-resources Statement.

Alternatively, you can embed the file I/O methods in a `try` block and then catch any exceptions in a `catch` block. If your code has opened any streams or channels, you should close them in a `finally` block. The previous example would look something like the following using the try-catch-finally approach:

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
} finally {
    if (writer != null) writer.close();
}
```

For more information, see Catching and Handling Exceptions.

In addition to `IOException`, many specific exceptions extend `FileSystemException`. This class has some useful methods that return the file involved (`getFile`), the detailed message string (`getMessage`), the reason why the file system operation failed (`getReason`), and the "other" file involved, if any (`getOtherFile`).

The following code snippet shows how the `getFile` method might be used:

```
try (...) {
    ...
} catch (NoSuchFileException x) {
    System.err.format("%s does not exist\n", x.getFile());
}
```

For purposes of clarity, the file I/O examples in this lesson may not show exception handling, but your code should always include it.

Varargs

Several `Files` methods accept an arbitrary number of arguments when flags are specified. For example, in the following method signature, the ellipses notation after the `CopyOption` argument indicates that the method accepts a variable number of arguments, or *varargs*, as they are typically called:

```
Path Files.move(Path, Path, CopyOption...)
```

When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array (`CopyOption[]`) of values.

In the `move` example, the method can be invoked as follows:

```
import static java.nio.file.StandardCopyOption.*;

Path source = ...;
Path target = ...;
Files.move(source,
           target,
           REPLACE_EXISTING,
           ATOMIC_MOVE);
```

For more information about varargs syntax, see Arbitrary Number of Arguments.

Atomic Operations

Several `Files` methods, such as `move`, can perform certain operations atomically in some file systems.

An *atomic file operation* is an operation that cannot be interrupted or "partially" performed. Either the entire operation is performed or the operation fails. This is important when you have multiple processes operating on the same area of the file system, and you need to guarantee that each process accesses a complete file.

Method Chaining

Many of the file I/O methods support the concept of *method chaining*.

You first invoke a method that returns an object. You then immediately invoke a method on *that* object, which returns yet another object, and so on. Many of the I/O examples use the following technique:

```
String value = Charset.defaultCharset().decode(buf).toString();
UserPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService().
         lookupPrincipalByName("me");
```

This technique produces compact code and enables you to avoid declaring temporary variables that you don't need.

What *Is* a Glob?

Two methods in the `Files` class accept a glob argument, but what is a *glob*?

You can use glob syntax to specify pattern-matching behavior.

A glob pattern is specified as a string and is matched against other strings, such as directory or file names. Glob syntax follows several simple rules:

- An asterisk, `*`, matches any number of characters (including none).
- Two asterisks, `**`, works like `*` but crosses directory boundaries. This syntax is generally used for matching complete paths.
- A question mark, `?`, matches exactly one character.
- Braces specify a collection of subpatterns. For example:
    - `{sun,moon,stars}` matches "sun", "moon", or "stars".
    - `{temp*,tmp*}` matches all strings beginning with "temp" or "tmp".

- Square brackets convey a set of single characters or, when the hyphen character (`-`) is used, a range of characters. For example:
    - `[aeiou]` matches any lowercase vowel.
    - `[0-9]` matches any digit.
    - `[A-Z]` matches any uppercase letter.
    - `[a-z,A-Z]` matches any uppercase or lowercase letter.

  Within the square brackets, `*`, `?`, and `\` match themselves.
- All other characters match themselves.
- To match `*`, `?`, or the other special characters, you can escape them by using the backslash character, `\`. For example: `\\` matches a single backslash, and `\?` matches the question mark.

Here are some examples of glob syntax:

- `*.html` – Matches all strings that end in *.html*
- `???` – Matches all strings with exactly three letters or digits
- `*[0-9]*` – Matches all strings containing a numeric value
- `*.{htm,html,pdf}` – Matches any string ending with *.htm*, *.html* or *.pdf*
- `a?*.java` – Matches any string beginning with `a`, followed by at least one letter or digit, and ending with *.java*
- `{foo*,*[0-9]*}` – Matches any string beginning with *foo* or any string containing a numeric value

---

Note: If you are typing the glob pattern at the keyboard and it contains one of the special characters, you must put the pattern in quotes (`"*"`), use the backslash (`\*`), or use whatever escape mechanism is supported at the command line.

The glob syntax is powerful and easy to use. However, if it is not sufficient for your needs, you can also use a regular expression. For more information, see the Regular Expressions lesson.

For more information about the glob sytnax, see the API specification for the `getPathMatcher` method in the `FileSystem` class.

Link Awareness

The `Files` class is "link aware." Every `Files` method either detects what to do when a symbolic link is encountered, or it provides an option enabling you to configure the behavior when a symbolic link is encountered.

Previous page: Path Operations
Next page: Checking a File or Directory