Common I/O Tasks in Modern Java

Introduction

This article focuses on tasks that application programmers are likely to encounter, particularly in web applications, such a

- Reading and writing text files
- Reading text, images, JSON from the web
- Visiting files in a directory
- Reading a ZIP file
- Creating a temporary file or directory

The Java API supports many other tasks, which are explained in detail in the Java I/O API tutorial.

This article focuses on API improvements since Java 8. In particular:

- UTF-8 is the default for I/O since Java 18 (since JEP 400: UTF-8 by Default)
- The `java.nio.file.Files` class, which first appeared in Java 7, added useful methods in Java 8, 11, and 12
- `java.io.InputStream` gained useful methods in Java 9, 11, and 12
- The `java.io.File` and `java.io.BufferedReader` classes are now thoroughly obsolete, even though they appear freq

Reading Text Files

You can read a text file into a string like this:

```
String content = Files.readString(path);
```

Here, `path` is an instance of `java.nio.Path`, obtained like this:

```
var path = Path.of("/usr/share/dict/words");
```

Before Java 18, you were strongly encouraged to specify the character encoding with any file operations that read or write

If you want the file as a sequence of lines, call

```
List<String> lines = Files.readAllLines(path);
```

If the file is large, process the lines lazily as a `Stream<String>`:

```
try (Stream<String> lines = Files.lines(path)) {
    . . .
}
```

Also use `Files.lines` if you can naturally process lines with stream operations (such as `map`, `filter`). Note that the strea

There is no longer a good reason to use the readLine method of java.io.BufferedReader.

To split your input into something else than lines, use a java.util.Scanner. For example, here is how you can read word

```
Stream<String> tokens = new Scanner(path).useDelimiter("\\PL+").tokens();
```

The Scanner class also has methods for reading numbers, but it is generally simpler to read the input as one string per lin

Be careful when parsing numbers from text files, since their format may be locale-dependent. For example, the input `100.`

Writing Text Files

You can write a string to a text file with a single call:

```
String content = . . .;
Files.writeString(path, content);
```

If you have a list of lines rather than a single string, use:

```
List<String> lines = . . .;
Files.write(path, lines);
```

For more general output, use a PrintWriter if you want to use the printf method:

```
var writer = new PrintWriter(path.toFile());
writer.printf(locale, "Hello, %s, next year you'll be %d years old!%n", name, age + 1);
```

Note that printf is locale-specific. When writing numbers, be sure to write them in the appropriate format. Instead of usi

Weirdly enough, as of Java 21, there is no PrintWriter constructor with a Path parameter.

If you don't use printf, you can use the BufferedWriter class and write strings with the write method.

```
var writer = Files.newBufferedWriter(path);
writer.write(line); // Does not write a line separator
writer.newLine();
```

Remember to close the `writer` when you are done.

Reading From an Input Stream

Perhaps the most common reason to use a stream is to read something from a web site.

If you need to set request headers or read response headers, use the HttpClient:

```
HttpClient client = HttpClient.newBuilder().build();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://horstmann.com/index.html"))
    .GET()
    .build();
HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

```

```
    String result = response.body();
```
That is overkill if all you want is the data. Instead, use:

```
InputStream in = new URI("https://horstmann.com/index.html").toURL().openStream();
```

Then read the data into a byte array and optionally turn them into a string:

```
byte[] bytes = in.readAllBytes();
String result = new String(bytes);
```

Or transfer the data to an output stream:

```
OutputStream out = Files.newOutputStream(path);
in.transferTo(out);
```

Note that no loop is required if you simply want to read all bytes of an input stream.

But do you really need an input stream? Many APIs give you the option to read from a file or URL.

Your favorite JSON library is likely to have methods for reading from a file or URL. For example, with Jackson jr:

```
URL url = new URI("https://dog.ceo/api/breeds/image/random").toURL();
Map<String, Object> result = JSON.std.mapFrom(url);
```

Here is how to read the dog image from the preceding call:

```
URL url = new URI(result.get("message").toString()).toURL();
BufferedImage img = javax.imageio.ImageIO.read(url);
```

This is better than passing an input stream to the `read` method, because the library can use additional information from tl

The Files API

The `java.nio.file.Files` class provides a comprehensive set of file operations, such as creating, copying, moving, and

Traversing Entries in Directories and Subdirectories

For most situations you can use one of two methods. The `Files.list` method visits all entries (files, subdirectories, syml

```
try (Stream<Path> entries = Files.list(pathToDirectory)) {
    . . .
}
```

Use a *try-with-resources* statement to ensure that the stream object, which keeps track of the iteration, will be closed.

If you also want to visit the entries of descendant directories, instead use the method `Files.walk`

```
Stream<Path> entries = Files.walk(pathToDirectory);
```

Then simply use stream methods to home in on the entries that you are interested in, and to collect the results:

```
try (Stream<Path> entries = Files.walk(pathToDirectory)) {
    List<Path> htmlFiles = entries.filter(p -> p.toString().endsWith("html")).toList();
    . . .
}
```

Here are the other methods for traversing directory entries:

- An overloaded version of `Files.walk` lets you limit the depth of the traversed tree.
- Two `Files.walkFileTree` methods provide more control over the iteration process, by notifying a `FileVisitor` when
- The `Files.find` method is just like `Files.walk`, but you provide a filter that inspects each path and its `BasicFileAtt`
- Two `Files.newDirectoryStream(Path)` methods yields `DirectoryStream` instances, which can be used in enhanced
- The legacy `File.list` or `File.listFiles` methods return file names or `File` objects. These are now obsolete.

Working with ZIP Files

Ever since Java 1.1, the `ZipInputStream` and `ZipOutputStream` classes provide an API for processing ZIP files. But the A

```
try (FileSystem fs = FileSystems.newFileSystem(pathToZipFile)) {
    . . .
}
```

The *try-with-resources* statement ensures that the `close` method is called after the ZIP file operations. That method updat

You can then use the methods of the `Files` class. Here we get a list of all files in the ZIP file:

```
try (Stream<Path> entries = Files.walk(fs.getPath("/"))) {
    List<Path> filesInZip = entries.filter(Files::isRegularFile).toList();
}
```

To read the file contents, just use `Files.readString` or `Files.readAllBytes`:

```
String contents = Files.readString(fs.getPath("/LICENSE"));
```

You can remove files with `Files.delete`. To add or replace files, simply use `Files.writeString` or `Files.write`.

Creating Temporary Files and Directories

Fairly often, I need to collect user input, produce files, and run an external process. Then I use temporary files, which are g

I use the two methods `Files.createTempFile` and `Files.createTempDirectory` for that.

```
Path filePath = Files.createTempFile("myapp", ".txt");
Path dirPath = Files.createTempDirectory("myapp");
```

This creates a temporary file or directory in a suitable location (`/tmp` in Linux) with the given prefix and, for a file, suffix.

Conclusion

Web searches and AI chats can suggest needlessly complex code for common I/O operations. There are often better alte

1. You don't need a loop to read or write strings or byte arrays.
2. You may not even need a stream, reader or writer.
3. Become familiar with the `Files` methods for creating, copying, moving, and deleting files and directories.

4. Use `Files.list` or `Files.walk` to traverse directory entries.
5. Use a ZIP file system for processing ZIP files.
6. Stay away from the legacy `File` class.

Last update: *April 24, 2024*

Back to Tutorial List