Reading and Writing Small Files

Choosing the Right I/O Method

There are a wide array of file I/O methods to choose from. To help make sense of the API, the following table shows the file I/O methods available on the `Files` class and their use cases.

Reading	Writing	Comments
`readAllBytes()`, `readAllLines()`	`write()`	Designed for simple, common use cases.
`newBufferedReader()`	`newBufferedWriter()`	Iterate over a stream or lines of text.
`newInputStream()`	`newOutputStream()`	These methods are interoperable with the `java.io` package.
`newByteChannel()`, `SeekableByteChannel`, `ByteBuffer`		

Reading	Writing	Comments
`FileChannel`		Advanced applications, file locking and memory-mapped I/O.

> *Note: The methods for creating a new file enable you to specify an optional set of initial attributes for the file. For example, on a file system that supports the POSIX set of standards (such as UNIX), you can specify a file owner, group owner, or file permissions at the time the file is created. The Managing Metadata section explains file attributes, and how to access and set them.*

The OpenOptions Parameter

Several of the methods in this section take an optional `OpenOption` parameter. This parameter is optional and the API tells you what the default behavior is for the method when none is specified.

Several Files methods accept an arbitrary number of arguments when flags are specified. When you see an ellipses notation after the type of the argument, it indicates that the method accepts a variable number of arguments, or *varargs*. When a method accepts a varargs argument, you can pass it a comma-separated list of values or an array of values.

The following `StandardOpenOption` enums are supported:

- `WRITE` – Opens the file for write access.
- `APPEND` – Appends the new data to the end of the file. This option is used with the WRITE or CREATE options.
- `TRUNCATE_EXISTING` – Truncates the file to zero bytes. This option is used with the WRITE option.

- `CREATE_NEW` – Creates a new file and throws an exception if the file already exists.
- `CREATE` – Opens the file if it exists or creates a new file if it does not.
- `DELETE_ON_CLOSE` – Deletes the file when the stream is closed. This option is useful for temporary files.
- `SPARSE` – Hints that a newly created file will be sparse. This advanced option is honored on some file systems, such as NTFS, where large files with data "gaps" can be stored in a more efficient manner where those empty gaps do not consume disk space.
- `SYNC` – Keeps the file (both content and metadata) synchronized with the underlying storage device.
- `DSYNC` – Keeps the file content synchronized with the underlying storage device.

Commonly Used Methods for Small Files

Reading All Bytes or Lines from a File

If you have a small-ish file and you would like to read its entire contents in one pass, you can use the `readAllBytes(Path)` or `readAllLines(Path, Charset)` method. These methods take care of most of the work for you, such as opening and closing the stream, but are not intended for handling large files. The following code shows how to use the `readAllBytes()` method:

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);
```

Writing All Bytes or Lines to a File

You can use one of the write methods to write bytes, or lines, to a file.

- write(Path, byte[], OpenOption...)
- write(Path, Iterable< extends CharSequence>, Charset, OpenOption...)

The following code snippet shows how to use a `write()` method.

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);
```

Methods for Creating Regular and Temporary Files

Creating Files

You can create an empty file with an initial set of attributes by using the createFile(Path, FileAttribute<?>) method. For example, if, at the time of creation, you want a file to have a particular set of file permissions, use the createFile() method to do so. If you do not specify any attributes, the file is created with default attributes. If the file already exists, createFile() throws an exception.

In a single atomic operation, the createFile() method checks for the existence of the file and creates that file with the specified attributes, which makes the process more secure against malicious code.

The following code snippet creates a file with default attributes:

```
Path file = ...;
try {
    // Create the empty file with default permissions, etc.
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
```

```
    System.err.format("file named %s" +
        " already exists%n", file);
} catch (IOException x) {
    // Some other sort of failure, such as permissions.
    System.err.format("createFile error: %s%n", x);
}
```

POSIX File Permissions has an example that uses `createFile(Path, FileAttribute<?>)` to create a file with pre-set permissions.

You can also create a new file by using the `newOutputStream()` methods, as described in the section Creating and Writing a File using Stream I/O. If you open a new output stream and close it immediately, an empty file is created.

Creating Temporary Files

You can create a temporary file using one of the following `createTempFile()` methods:

- `createTempFile(Path, String, String, FileAttribute<?>)`
- `createTempFile(String, String, FileAttribute<?>)`

The first method allows the code to specify a directory for the temporary file and the second method creates a new file in the default temporary-file directory. Both methods allow you to specify a suffix for the filename and the first method allows you to also specify a prefix. The following code snippet gives an example of the second method:

```
try {
    Path tempFile = Files.createTempFile(null, ".myapp");
    System.out.format("The temporary file" +
        " has been created: %s%n", tempFile)
;
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

The result of running this file would be something like the following:

```
The temporary file has been created: /tmp/509668702974537184.myapp
```

The specific format of the temporary file name is platform specific.

Random Access Files

Random access files permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.

This functionality is possible with the `SeekableByteChannel` interface. The `SeekableByteChannel` interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:

- `position()` – Returns the channel's current position
- `position(long)` – Sets the channel's position
- `read(ByteBuffer)` – Reads bytes into the buffer from the channel
- `write(ByteBuffer)` – Writes bytes from the buffer to the channel
- `truncate(long)` – Truncates the file (or other entity) connected to the channel

Reading and Writing Files With Channel I/O shows that the `Path.newByteChannel()` methods return an instance of a `SeekableByteChannel`. On the default file system, you can use that channel as is, or you can cast it to a `FileChannel` giving you access to more advanced features, such as mapping a region of the file directly into memory for faster access, locking a region of the file, or reading and writing bytes from an absolute location without affecting the channel's current position.

The following code snippet opens a file for both reading and writing by using one of the newByteChannel() methods. The SeekableByteChannel that is returned is cast to a FileChannel. Then, 12 bytes are read from the beginning of the file, and the string "I was here!" is written at that location. The current position in the file is moved to the end, and the 12 bytes from the beginning are appended. Finally, the string, "I was here!" is appended, and the channel on the file is closed.

```
String s = "I was here!\n";
byte data[] = s.getBytes();
ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
    // Read the first 12
    // bytes of the file.
    int nread;
    do {
        nread = fc.read(copy);
    } while (nread != -1 && copy.hasRemaining());

    // Write "I was here!" at the beginning of the file.
    fc.position(0);
    while (out.hasRemaining())
        fc.write(out);
    out.rewind();

    // Move to the end of the file.  Copy the first 12 bytes to
    // the end of the file.  Then write "I was here!" again.
    long length = fc.size();
    fc.position(length-1);
    copy.flip();
    while (copy.hasRemaining())
        fc.write(copy);
    while (out.hasRemaining())
        fc.write(out);
} catch (IOException x) {
    System.out.println("I/O Exception: " + x);

```

```
    }
```

Last update: *January 25, 2023*

Releasing Resources and Catching Exceptions	←	**Reading and Writing Small Files**	→	Reading and Writing Text Files