21.5 Reading and Writing Files Using Paths

The `Files` class provides methods for reading and writing bytes and characters using I/O streams and `Path` objects.

Methods provided for reading and writing files typically close the file after use.

Open Options

The `java.nio.file.OpenOption` interface is implemented by objects that can be specified as options to configure how a file operation should open or create a file. Methods for writing to files can be configured for this purpose by specifying constants defined by the enum type `java.nio.file.StandardOpenOption` that implements the `OpenOption` interface.

Table 21.9 shows the options defined by constants of the `StandardOpenOption` enum type. Such options are specified as values for the variable arity parameter of type `OpenOption` in methods such as `newBufferedWriter()`, `write()`, `writeString()`, `newOutputStream()`, and `newInputStream()` of the `Files` class.

For write operations, if no options are supplied, it implies that the following options for opening and creating a file are present: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`—meaning the file is opened for writing, created if it does not exist, and truncated to size 0.

Table 21.9 *Selected Standard Open Options*

Enum `java.nio.file.StandardOpenOption` implements the `java.nio.file.OpenOption` interface	Description
`READ`	Open the file for read access.
`WRITE`	Open the file for write access.
`APPEND`	If the file is opened for `WRITE` access, write bytes to the end of the file. That is, its previous content is not overwritten.
`TRUNCATE_EXISTING`	If the file already exists and it is opened for `WRITE` access, truncate its length to 0 so that bytes are written from the beginning of the file.

Table 21.9 *Selected Standard Open Options (Continued)*

Enum `java.nio.file.StandardOpenOption` implements the `java.nio.file.OpenOption` interface	Description
`CREATE`	Open the file if it exists; otherwise, create a new file.
`CREATE_NEW`	Fail if the file already exists; otherwise, create a new file.
`DELETE_ON_CLOSE`	Delete the file when the stream is closed. Typically used for temporary files.

Reading and Writing Character Data

The `Files` class provides methods for reading and writing *character data* to files. These methods can be categorized as follows:

- Methods that create character I/O streams (`BufferedReader`, `BufferedWriter`) chained to a `Path` object that denotes a file. The methods of the buffered reader and writer can then be used to read and write characters to the file, respectively.
- Methods that directly use a `Path` object, and read and write characters to the file denoted by the `Path` object.

Reading and Writing Character Data Using Buffered I/O Streams

The `newBufferedReader()` and `newBufferedWriter()` methods of the `Files` class create buffered readers and writers, respectively, that are chained to a `Path` object denoting a file. Interoperability between character I/O streams in the `java.io` package can then be leveraged to chain appropriate I/O streams for reading and writing character data to a file (§20.3, p. 1241).

Previously we have used constructors of the `BufferedReader` class (§20.3, p. 1251) and the `BufferedWriter` class (§20.3, p. 1250) in the `java.io` package to instantiate buffered readers and writers that are chained to a `Reader` or a `Writer`, respectively. Using the methods of the `Files` class is the recommended practice for creating buffered readers and writers when dealing with text files.

```
static BufferedReader newBufferedReader(Path path) throws IOException
static BufferedReader newBufferedReader(Path path, Charset cs)
                                          throws IOException
```

Opens the file denoted by the specified `path` for reading, and returns a `BufferedReader` of a default size to read text efficiently from the file, using either the UTF-8 charset or the specified charset to decode the bytes, respectively. Contrast these methods with the constructors of the `java.io.BufferReader` class (§20.3, p. 1251).

```
static BufferedWriter newBufferedWriter(Path path, OpenOption... options)
                                                    throws IOException
static BufferedWriter newBufferedWriter(Path path, Charset cs,
                      OpenOption... options) throws IOException
```

Opens or creates a file denoted by the specified `path` for writing, returning a `BufferedWriter` of a default size that can be used to write text efficiently to the file, using either the UTF-8 charset or the specified charset to encode the characters, respectively. See also constructors of the `java.io.BufferWriter` class (§20.3, p. 1250).

The code at (1) and at (3) in Example 21.2 illustrates writing lines to a text file using a buffered writer and reading lines from a text file using a buffered reader, respectively. The methods `newBufferedWriter()` and `newBufferedReader()` create the necessary buffered writer and reader at (2) and (4), respectively, whose methods are used to write and read the lines from the file.

Example 21.2 *Reading and Writing Text Files*

```
import java.io.*;
import java.nio.file.*;
import java.util.*;

public class ReadingWritingTextFiles {

  public static void main(String[] args) throws IOException {
    // List of strings:
    List<String> lines = List.of("Guess who got caught?", "Who?",
                                 "NullPointerException.",
                                 "Seriously?", "No. Finally.");
    // Text file:
    String filename = "project/linesOnly.txt";
    Path path = Path.of(filename);

    // Writing lines using buffered writer:                              (1)
    try (BufferedWriter writer =  Files.newBufferedWriter(path)) {   // (2)
      for(String str : lines) {
        writer.write(str);                  // Write a string.
        writer.newLine();                   // Terminate with a newline.
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }

    // Read lines using buffered reader:                                 (3)
    lines = new ArrayList<>();
    try(BufferedReader reader= Files.newBufferedReader(path)) {      // (4)
      String line = null;
      while ((line = reader.readLine()) != null) {  // EOF when null is returned.
        lines.add(line);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
    System.out.printf("Lines read from file \"%s\":%n%s%n", path, lines);
```

```
        // Write the list of strings in one operation:
        Files.write(path, lines);                                    // (5)

        // Write the joined lines in one operation:
        String joinedLines = String.join(System.lineSeparator(), lines);
        Files.writeString(path, joinedLines);                        // (6)

        // Read all contents into a String, including line separators:
        String allContent = Files.readString(path);                  // (7)
        System.out.printf("All lines read from file \"%s\":%n%s%n", path, allContent);

        // Read all lines into a list of String:
        lines = Files.readAllLines(path);                            // (8)
        System.out.printf("List of lines read from file \"%s\":%n%s%n", path, lines);
    }
}
```

Output from the program:

```
Lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
All lines read from file "project/linesOnly.txt":
Guess who got caught?
Who?
NullPointerException.
Seriously?
No. Finally.
List of lines read from file "project/linesOnly.txt":
[Guess who got caught?, Who?, NullPointerException., Seriously?, No. Finally.]
```

Reading and Writing Character Data Using `Path` Objects

The `Files` class provides methods that directly use a `Path` object, and read and write characters to the file denoted by the `Path` object, without the need to specify an I/O stream. These methods also close the file when done.

```
static Path write(Path path, Iterable<? extends CharSequence> lines,
                  OpenOption... options) throws IOException
static Path write(Path path, Iterable<? extends CharSequence> lines,
                  Charset cs, OpenOption... options) throws IOException
```

Open or create a file denoted by the specified `path`, and writes text `lines` to the file, using either the UTF-8 charset or the specified charset, respectively.

No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

```
static Path writeString(Path path, CharSequence csq, OpenOption... options)
            throws IOException
static Path writeString(Path path, CharSequence csq, Charset cs,
                        OpenOption... options) throws IOException
```

Open or create a file denoted by the specified `path`, and writes characters in the `CharSequence csq` verbatim to the file, using either the UTF-8 charset or the specified charset, respectively.

No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

```
static String readString(Path path) throws IOException
static String readString(Path path, Charset cs) throws IOException
```

Read all content from a file denoted by the specified `path` into a string, decoding the bytes to characters using the UTF-8 charset or the specified charset, respectively. The string returned will contain all characters, including line separators. These methods are not recommended for reading large files.

```
static List<String> readAllLines(Path path) throws IOException
static List<String> readAllLines(Path path, Charset cs) throws IOException
```

Read all *lines* from the file denoted by the specified `path`, decoding the bytes to characters using the UTF-8 charset or the specified charset, respectively. These methods are not recommended for reading large files, as these can result in a lethal `java.lang.OutOfMemoryError`.

The code at (5) to (8) in Example 21.2 illustrates methods of the `Files` class that directly write and read character data to a file denoted by a `Path` object.

The `write()` method at (5) writes an `Iterable` (in this case, the `List` of `String`) to the file in one operation. It automatically terminates each string written with a newline.

```
Files.write(path, lines);                                    // (5)
```

The `writeString()` method at (6) writes the contents of a single `CharSequence` (in this case, the string `joinedLines`) to the file. The strings in the `lines` list are joined with an appropriate line separator by the `String.join()` method. The end result written to the file is again lines of text.

```
String joinedLines = String.join(System.lineSeparator(), lines);
Files.writeString(path, joinedLines);                        // (6)
```

The `readString()` method at (7) reads the *whole* file in one operation. It returns all characters read in a string, *including any line separators*.

```
String allContent = Files.readString(path);                  // (7)
```

The `readAllLines()` method at (8) reads all *text lines* in the file in one operation, returning the lines read in a `List` of `String`.

```
lines = Files.readAllLines(path);                            // (8)
```

The methods in the `Files` class for writing and reading directly from a file denoted by a `Path` object should be used with care, as they might not scale up when handling large files. This is especially the case regarding the `readString()` and `readAllLines()` methods that read the whole file in one fell swoop. A better solution for reading text files using streams is provided later in the chapter (p. 1345).

Reading and Writing Bytes

The `Files` class also provides methods for reading and writing *bytes* to files. These methods that can be categorized as follows:

- Methods that create low-level *byte* I/O streams (`InputStream`, `OutputStream`) chained to a `Path` object that denotes a file. The methods of the I/O streams can then be used to read and write bytes to the file.

- Methods that directly use a `Path` object, and read and write bytes to the file denoted by the `Path` object.

Reading and Writing Bytes Using I/O Streams

The `newInputStream()` and `newOutputStream()` methods of the `Files` class create an input stream and an output stream, respectively, that are chained to a `Path` object denoting a file. Interoperability between I/O streams in the `java.io` package can then be leveraged to chain appropriate I/O streams for reading and writing data to a file (§20.2, p. 1234).

```
static InputStream newInputStream(Path path, OpenOption... options)
                                  throws IOException
```

Opens a file denoted by the specified `path`, and returns an input stream to read from the file. No options implies the `READ` option.

```
static OutputStream newOutputStream(Path path, OpenOption... options)
                    throws IOException
```

Opens or creates a file denoted by the specified `path`, and returns an output stream that can be used to write bytes to the file. No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

Previously we have seen how the `copy()` methods of the `Files` class use byte I/O streams for reading and writing bytes to files (p. 1311).

The code at (1) in Example 21.3 is a reworking of Example 20.1, p. 1237, to copy bytes from a source file to a destination file using an explicit byte buffer. The main difference is that the input stream and the output stream on the respective files are created by the `newInputStream()` and `newOutputStream()` methods of the `Files` class, based on `Path` objects that denote the files, rather than on file I/O streams. As before, the methods `read()` and `write()` of the `InputStream` and `OutputStream` classes, respectively, are used to read and write the bytes from the source file to the destination file using a byte buffer.

Example 21.3 *Reading and Writing Bytes*

```
import java.io.*;
import java.nio.file.*;

public class ReadingWritingBytes {
  public static void main(String[] args) {
    // Source and destination files:
    Path srcPath  = Path.of("project", "source.dat");
    Path destPath = Path.of("project", "destination.dat");

    try (InputStream is = Files.newInputStream(srcPath);                    // (1)
         OutputStream os = Files.newOutputStream(destPath))  {
      byte[] buffer = new byte[1024];
      int length = 0;
```

```
        while((length = is.read(buffer, 0, buffer.length)) != -1) {
          os.write(buffer, 0, length);
        }
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }

      try {
        // Reads the file contents into an array of bytes:
        byte[] allBytes = Files.readAllBytes(srcPath);                    // (2)

        // Writes an array of bytes to a file:
        Files.write(destPath, allBytes);                                  // (3)
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
```

Reading and Writing Bytes Using `Path` Objects

The `Files` class provides methods that directly use a `Path` object, and read and write bytes to the file denoted by the `Path` object, without the need to specify a file I/O stream. The method `readAllBytes()` reads all bytes from a file into a `byte` array in one operation, and the method `write()` writes the bytes in a `byte` array to a file. These methods also close the file when done.

> `static byte[] readAllBytes(Path path) throws IOException`
>
> Reads all the bytes from the file denoted by the specified `path`. The bytes are returned in a `byte` array.
>
> `static Path write(Path path, byte[] bytes, OpenOption... options)`
> `        throws IOException`
>
> Writes bytes in a `byte` array to the file denoted by the specified `path`. No options implies the following options: `CREATE`, `TRUNCATE_EXISTING`, and `WRITE`.

The code at (2) and (3) in Example 21.3 shows yet another example of copying the contents of a source file to a destination file. The `readAllBytes()` and `write()` methods accomplish the task in a single call to each method.

```
byte[] allBytes = Files.readAllBytes(srcPath);                    // (2)
...
Files.write(destPath, allBytes);                                  // (3)
```

Note that these methods are meant for simple cases, and not for handling large files, as data is handled using an array of bytes.