Reading and Writing Small Files ← Reading and Writing Text Files → Reading and Writing Binary Files

Reading and Writing Text Files

The `java.nio.file` package supports channel I/O, which moves data in buffers, bypassing some of the layers that can bottleneck stream I/O.

Understanding the Character Handling

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII or UTF-8.

Input and output done with stream classes automatically translates to and from the local character set. Until Java SE 17, a program that uses character streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer. Starting with Java SE 18, the default charset of your Java application is UTF-8.

If internationalization is not a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding.

Reading a Text File by Using Buffered Stream I/O

The newBufferedReader(Path, Charset) method opens a file for reading, returning a BufferedReader that can be used to read text from a file in an efficient manner.

The BufferedReader class gives you a method to read the content of your text file line by line. Starting with Java SE 8, it also gives you a method to create a Stream<String> on the lines of your text file. You can learn more about streams in the Stream Section of this tutorial.

The following code reads your file line by line.

```java
// The closing of the reader and the handling of the exceptions
// have been omitted
// String line = reader.readLine();
long count = 0L;
while (line != null) {
    count++;
    line = reader.readLine();
}
System.out.println("Number of lines in this file = " + count);
```

Note that the line string does not contain the line termination characters of each line. When the end of the file is reached, the line returned is null.

Starting with Java SE 8, you can write the following code.

```java
Path path = Path.of("file.txt");

try (BufferedReader reader = Files.newBufferedReader(path);
     Stream<String> lines = reader.lines();) {

    long count = lines.count();
    System.out.println("count = " + count);
}
```

The reader.lines() method is defined in the BufferedReader class. Because the Stream interface extends the AutoCloseable interface, you can open your stream in a *try-with-resources* statement. In that case, the reader is properly closed.

Writing a Text File by Using Buffered Stream I/O

You can use the `newBufferedWriter(Path, Charset, OpenOption...)` method to write to a file using a `BufferedWriter`.

The following code snippet shows how to create a file encoded in "US-ASCII" using this method:

```
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);
}
```

Last update: *January 25, 2023*

Reading and Writing Small Files ← **Reading and Writing Text Files** → Reading and Writing Binary Files

Home > Tutorials > The Java I/O API > File Operations Basics > Reading and Writing Text Files