Input and Output

Contents

Overview

Java refers to file-based **input and output (IO)** as a **stream**.

*(Note: this is not to be confused with **streams**, which are discussed here.)*

A stream represents an *input source* or an *output destination*.

A stream can represent different kinds of sources and destinations, for example; files, devices, other programs, memory arrays, and Web sites.

Streams can support different types of data, including primitive data types and `String`s, and they simply pass on data or manipulate/transform the data into different types of data.

For example you could read in bytes from a file but then process it as a `String`, and then re-write it back as bytes.

Java supports two types of IO streams; `character` and `byte`.

Readers and writers handle the character data, whereas input and output streams handle the byte data.

Tutorial

Handling Streams

A developer will typically use input and output in three ways:

- To interact with files and directories.
- To read from and write to the console.
- Using "socket" based sources to communicate with a remote system.

A socket is similar to a port only it is used for *internal* endpoints as opposed to *external* endpoints - sockets will only accept traffic from clients on the same network, or can be used for incoming traffic by having the port redirect the traffic through the socket.

The base class names are as follows:

- For byte streams - `InputStream` and `OutputStream`
- For character streams - `Reader` and `Writer`

InputStream and OutputStream

▶ Click to expand

In the above method called `copy()`, we are wanting to copy an image, within the same folder and give it a new filename.

To do this we first use a try-with-resources and pass it a `FileInputStream`, which is the file we want to copy, and a `FileOutputStream`, which will be the new file.

Within the `try()`-block we then declare an `int` to store the number of bytes we have read within the loop and a `byte[]` which defines how many bytes we will be reading during each loop iteration.

Then in the `while()`-loop we are setting `numberOfBytesRead` to the number of bytes that `fileInputStream` has read from the `byteArray`. Inside the while loop we write the bytes read by `fileInputStream` to `fileOutputStream`; giving the data (`byteArray`), any offset in the data (`0`), and the number of bytes to write (`numberOfBytesRead`) as parameters.

We can alter the size of `byteArray` freely - the larger we set it, the quicker it will be; however it will also be a lot more memory-intensive.

BufferedReader and BufferedWriter

▶ Click to expand

In the above method called `bufferedCopy()`, we are wanting to copy all text within `textFile.txt` to `newTextFile.txt`.

Within the method we are using a try-with-resources statement and instantiating/passing a `BufferedReader` which has a `FileReader` being instantiated and passed to it, as well as instantiating/passing a `BufferedWriter` which has a `FileWriter` being instantiated and passed to it.

Inside the `try()`-block we are declaring a `String` variable and using a `while()`-loop to iterate over the file.

The `while()`-loop is declared by setting our `String` variable `line` to be equal to the next line within the file that has been given to the `BufferedReader`, and while the next line is not null we will loop over the code within.

Within the `while()`-loop we are then writing the line to our new file that we passed to the `BufferedWriter`.

If an `IOException` is thrown we will catch it and output that we caught the exception along with the exceptions message.

We are passing `FileReader` and `FileWriter` to `BufferedReader` and `BufferedWriter` respectively because `BufferedReader` and `BufferedWriter` implement the Decorator Pattern, meaning it expects a reader in its constructor.

The reason we used `BufferedReader` and `BufferedWriter` instead of using `FileReader` and `FileWriter` directly is because they minimise the number of I/O operations by reading chunks of characters and storing them in an internal buffer; while data is in the buffer the reader or writer will use that instead of directly using the underlying stream.

`BufferedReader` and `BufferedWriter` also provide some nice helper functions; for example `BufferedReader` provides the ability to read files line by line.

Exercises

Person

1. Create a Person class that contains the following attributes:

- Name
- Age
- Job Title

1. Create a method to return all three of these attributes in a formatted String. **(HINT: Override the `toString()` method.)**
2. Create some example objects with this class.
3. Create a List implementation and store those objects inside it.
4. Use a stream to output all of your people to the console.
5. Create a method that can search for a specific Person by their name.

Person Extended With File IO

1. Using your Person class from the above exercises, create a list and populate it with 5 of these objects.
2. Create a loop to iterate through the list, writing each object to one file. (Think about how you want to format this).
3. Separately, create another list and populate it with the data in the file you just wrote to.
   (You are going to have to parse it back in the format you wrote it in).