Contents

IO

All input and output operations in java are performed by streams, these stream are usually defined by two major types of data - Byte and Character streams. Rembmer that in Java byte is of size 8bits or 1 byte, while character is usually 2 bytes wide to be able to represent utf-16 character and text encoding. Thus when reading a text file, it is more likely we would like to use the Character stream api instead of the Byte one, however if we are reading output from a device or something that is not really a text content one must use the Byte stream.

Byte streams

At the very top of the class hierarchy one can find the two major classes representing the input and output streams which are - `InputStream` and `OutputStream`, these are the two base classes from which all byte streams extend off of.

Class	Applicaton
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
********	****
BufferedInputStream	Buffered input stream, wraps around other stream objects and buffers their content
BufferedOutputStream	Buffered output stream, wraps around other stream objects and buffers their content
********	****
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
********	****
DataInputStream	An input stream that contains methods for reading the Java standard data primitive types
DataOutputStream	An output stream that contains methods for writing the Java standard data primitive types
********	****
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
********	****
FilterInputStream	Used to wrap other stream objects and proxy all calls to the wrapped stream
FilterOutputStream	Used to wrap other stream objects and proxy all calls to the wrapped stream
********	****
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
********	****
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
********	****

Class	Applicaton
PushbackInputStream	Input stream that supports one-byte “unget,” which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Char streams

At the very top of this class hierarchy stay the two major classes representing the input and output streams which are - `Reader` and `Writer`, these are the two base classes from which all char streams extend off of.

Class	Applicaton
Reader	Abstract class that describes character stream input
Writer	Abstract class that describes character stream output
********	****
BufferedReader	Buffered input, wraps around other stream objects and buffers their content
BufferedWriter	Buffered output, wraps around other stream objects and buffers their content
********	****
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
********	****
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
********	****
FilterReader	Used to wrap other stream objects and proxy all calls to the wrapped stream
FilterWriter	Used to wrap other stream objects and proxy all calls to the wrapped stream
********	****
InputStreamReader	Input stream that translates bytes to characters
OutputStreamWriter	Output stream that translates characters to bytes
********	****
PipedReader	Input pipe
PipedWriter	Output pipe
********	****
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
********	****
PrintWriter	Output stream that contains print( ) and println( )
PushbackReader	Input stream that allows characters to be returned to the input stream
********	****
LineNumberReader	Input stream that counts lines

Predefined

By default the language provides some predefined instantiated streams which are coming from the `System` object which is located in `java.lang` which is imported by default implicitly, these are the `in, out and err` streams on the System object, these represent the input, output and error streams which are by default linked to the For a lower level understanding of how this works, is a list of how the JVM links these streams to an actual file descriptor controlled and managed by the OS, which takes care to actually interact with the output and input devices

Mapping

- System.in: Is mapped to the underlying standard input stream (stdin) of the OS (like reading from the terminal).
- System.out: Is mapped to the underlying standard output stream (stdout) of the OS (like writing to the terminal).
- System.err: Is mapped to the underlying standard error stream (stderr) of the OS (like writing error messages to the terminal).

Interaction

- On Linux/Unix, the JVM uses native system calls like read() and write() to interact with file descriptors for stdin (0), stdout (1), and stderr (2).
- On Windows, the JVM uses the ReadFile and WriteFile using in the default os file descriptor handles - STD_INPUT_HANDLE and STD_OUTPUT_HANDLE

In the system class the in, out and err streams are defined like that

```
// these static variables defined in the System class are automatically
   initialized by the JVM, when it starts, ready to be used immediately
public static final InputStream in;
public static final PrintStream out;
public static final PrintStream err;
```

Reading

To read characters from the standard input one can use the System.in stream wrapped around/in `BufferedReader`, the reason behind this is that, the `BufferedReader` represents a character stream, while `System.in` is simple byte stream

- it is `InputStream`, remember. For example calling `readLine` on the `BufferedReader` will buffer read chunks from the `System.in` until it reads a new line or at most 8KB of data, further more it will make sure that while it reads bytes from the `System.in` stream, will convert them to readable `2-byte` wide characters with `UTF` encoding.

Nothing is really stopping somebody from directly reading from System.in however one will be reading plain bytes, meaning that one has to correctly convert these to a readable String as well as correctly parse line feeds and so on, wrapping one stream into another helps us `translate` one type of data to another easily, and also bridge the gap between a byte stream and a character stream, which is no small feat

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
br.readLine(); // blocking the program until a new line is read from stdin
br.read(); // read a single char, 2-byte wide from stdin instead of newline
```

```
System.in is buffered on new lines, this is done to prevent too many sys calls between
the JVM and operating systems, it would not be very good if on each new byte the OS was
polled to write to the System.in stream
```

Writing

To write characters or text the easiest way is to use the default System.out, which is of type `PrintStream`, directly connected to `stdout`. The `PrintStream` in this case is directly connected to he OS file descriptor which represents the standard output, the reason Java is not using plain `OutputStream`, is again due to buffering, the run time does not want to write each byte to the output issuing system calls to the operating system on every byte, rather the implementation is using `PrintStream` to wrap the `OutputStream` linked to

std out, to buffer the input, usually it is buffered on line feed character/bytes or on some maximum capacity of the buffer (8KB)

Keep in mind that `PrintStream` can be used to wrap around a `FileOutputStream`, meaning that one can instead write to a file, which is what some of the logging frameworks actually do, instead of attaching to `STDOUT`, or in addition to `stdout` they route the same logs to files too, those logs could be routed over pipes and so on, the interface JAVA provides is quite flexible in that regard

```
System.out.println("Enter lines of text."); // write an entire line along
   with a new-line
System.out.println("Enter 'stop' to quit."); // write an entire line along
   with a new-line

System.out.write(b); // write a single byte to output
System.out.write('\n'); // trigger flushing of the stdout
```

`When creating an instance of PrintStream, one can also specify the autoFlush strategy, which basically forces flush on new line characters by default otherwise flushing is performed when the buffer reaches a certain size, then the contents of the internal buffer stored in the java run-time are sent down to the underling operating system`

Files

To read and write and overall interact with files on a very basic level java exposes the two byte stream based classes - `FileInputStream` and `FileOutputStream`. Both of which by default have a constructor which takes the file name of the file to be opened. What java does internally is to obtain the file descriptor which used to issue native or in other words system calls to the operating system to read or write bytes to the file

```
int i = 0; // hold the result byte of of reading from the file
try(FileInputStream fin = new FileInputStream("file.txt")) {
    do {
        i = fin.read(); // read will return negative if the read failed or
           EOF was reached
        if(i != -1)  {
            System.out.print((char) i); // print out all valid output to
               the stdout
        }
    } while(i != -1); // means the reading probably reached the end of the
       file
} catch(IOException e) {
    System.out.println("Error Reading File");
}
```

Notice that the return type of `read` is `integer`, not `byte`, this is because all positive values from `0-255` are valid bytes which the file might contain, however the return type still needs to have a way to identify reaching end of file or if the file is not able to be read further for whatever reason, where the return type is `-1`, if the return of `read` was byte, there is no way for java to represent `EOF`.

`When you are done with a file, you must close it. This is done by calling the close()method, which is implemented by both FileInputStream and FileOutputStream.`

```
try(FileOutputStream fout = new FileOutputStream("file.txt")) {
    fout.write(0xff); // write a single byte to the file
    fout.write(0xad); // write another byte to the file
```

```
} catch(IOException e) {
    System.out.println("Error Reading File");
}
```

When opening a file for writing one can also specify additional boolean argument `append` after the file name, which tells the run-time to open the file in write-append mode, meaning the cursor is positioned at the end of the file, and that is the position from where the writing starts, if no append flag is provided the cursor is put at the start of the file effectively deleting the original content. The `fout` output stream can be manually flushed based on the use case, however it is pretty much guaranteed that the stream will be flushed when the stream is closed, in the example above using try-with-resources, the stream will be closed once the flow exits the try block

Serialization

In java by default all types of classes and their data can be serialized, however one might wish to serialize only part of the class, in that case the usage of transient is required, marking a member field in a class type as transient will make sure that when that class is serialized then all transient fields will NOT be written out, or if reading object from a file, they will NOT be read in.