# Byte Streams

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from [InputStream](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) and [OutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank).

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, [FileInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) and [FileOutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank). Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

## Using Byte Streams

We'll explore FileInputStream and FileOutputStream by examining an example program named [CopyBytes](https://docs.oracle.com/javase/tutorial/essential/io/examples/CopyBytes.java" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank), which uses byte streams to copy xanadu.txt, one byte at a time.

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class CopyBytes {

public static void main(String[] args) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("xanadu.txt");

out = new FileOutputStream("outagain.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

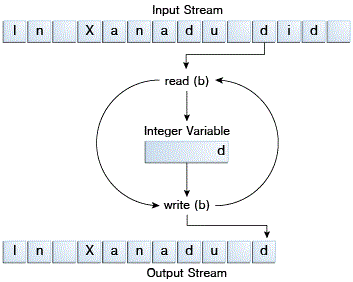
}

}

}

}

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple byte stream input and output.

## Always Close Streams

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

## When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use [character streams](https://docs.oracle.com/javase/tutorial/essential/io/charstreams.html), as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? Because all other stream types are built on byte streams.

# Character Streams

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.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't 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. See the [Internationalization](https://docs.oracle.com/javase/tutorial/i18n/index.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_top) trail for more information.

## Using Character Streams

All character stream classes are descended from [Reader](https://docs.oracle.com/javase/8/docs/api/java/io/Reader.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) and [Writer](https://docs.oracle.com/javase/8/docs/api/java/io/Writer.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank). As with byte streams, there are character stream classes that specialize in file I/O: [FileReader](https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) and [FileWriter](https://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank). The [CopyCharacters](https://docs.oracle.com/javase/tutorial/essential/io/examples/CopyCharacters.java" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) example illustrates these classes.

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CopyCharacters {

public static void main(String[] args) throws IOException {

FileReader inputStream = null;

FileWriter outputStream = null;

try {

inputStream = new FileReader("xanadu.txt");

outputStream = new FileWriter("characteroutput.txt");

int c;

while ((c = inputStream.read()) != -1) {

outputStream.write(c);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

### Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: [InputStreamReader](https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) and [OutputStreamWriter](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStreamWriter.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank). Use them to create character streams when there are no prepackaged character stream classes that meet your needs. The [sockets lesson](https://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_top) in the [networking trail](https://docs.oracle.com/javase/tutorial/networking/index.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_top) shows how to create character streams from the byte streams provided by socket classes.

## Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, [BufferedReader](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) and [PrintWriter](https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank). We'll explore these classes in greater depth in [Buffered I/O](https://docs.oracle.com/javase/tutorial/essential/io/buffers.html) and [Formatting](https://docs.oracle.com/javase/tutorial/essential/io/formatting.html). Right now, we're just interested in their support for line-oriented I/O.

The [CopyLines](https://docs.oracle.com/javase/tutorial/essential/io/examples/CopyLines.java" \t "https://docs.oracle.com/javase/tutorial/essential/io/_blank) example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

import java.io.FileReader;

import java.io.FileWriter;

import java.io.BufferedReader;

import java.io.PrintWriter;

import java.io.IOException;

public class CopyLines {

public static void main(String[] args) throws IOException {

BufferedReader inputStream = null;

PrintWriter outputStream = null;

try {

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

String l;

while ((l = inputStream.readLine()) != null) {

outputStream.println(l);

}

} finally {

if (inputStream != null) {

inputStream.close();

}

if (outputStream != null) {

outputStream.close();

}

}

}

}

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

There are many ways to structure text input and output beyond characters and lines. For more information, see [Scanning and Formatting](https://docs.oracle.com/javase/tutorial/essential/io/scanfor.html).