CS 240: Streams Transcript

This video shows a split screen of Professor Wilkerson on the right and a PowerPoint screen on the left. The left screen will switch to IntelliJ. Any text displayed or action performed that is not verbalized will be included in italics as visual descriptions.

[00:00:00] **JEROD WILKERSON:** Streams are the most common way that you will work with files, so it's very important that you understand how to read and write with streams.

[00:00:10] We use streams when we need to read an entire file sequentially from beginning to end or if we need to write a bunch of data to a file.

[00:00:21] There really are two choices for working with streams: we can either be working with binary-formatted data or text data.

[00:00:29] If we're working with binary-formatted data, we will use InputStream or one of its subclasses to read data, and we'll use OutputStream or one of its subclasses to write bytes.

[00:00:43] If we're working with characters, it's a lot easier to work with some other classes.

[00:00:48] If we're working with, for example, a character file, then it's easier to work with a Reader to read it.

[00:00:54] Readers work like InputStreams, but they work on character data.

[00:00:58] Similarly, Writers work like OutputStreams, but they work with character data.

[00:01:03] So if we were going to read or write a text file, we'd want to use a Reader or a Writer.

[00:01:09] In early versions of Java, we didn't have Readers or Writers, we just had input streams and output streams, so to work with text data or character data, we had to convert the bytes to text, which was not terribly difficult, but it was a little bit of a pain.

[00:01:25] Now that we have Readers and Writers, we don't have to worry about that.

[00:01:28] That's the primary function of Readers and Writers is to convert the data to character or to text.

[00:01:35] We'll learn first about streams, InputStreams and OutputStreams, and then I'll have a separate video where you learn about Readers and Writers.

[00:01:45] InputStream is actually an interface.

[00:01:48] It's used to read bytes sequentially from a data source, and we have a lot of implementing classes.

[00:01:53] The most common one is FileInputStream, and you've already used that.

[00:01:57] We also have PipedInputStream, which is the class that will allow you to read data from a thread.

[00:02:04] Then we have a few other streams, some of which you'll learn about throughout the course.

[00:02:09] You can get an input stream on a URLConnection.

[00:02:12] That allows you to write a client that can connect to a server and read data from the server.

[00:02:17] We can get one from something called an HTTPExchange.

[00:02:21] That, as you will see, is one way that a server can read data from a client.

[00:02:26] We can get a binary stream from a ResultSet.

[00:02:30] If you think about JDBC, when you learned about that, it's possible that a column contains binary data—maybe it contains an image file, for example.

[00:02:40] The way we can read that is we can get a binary stream from a ResultSet.

[00:02:45] There actually are many more examples of how you can get input streams.

[00:02:49] There are many classes that implement the InputStream interface.

[00:02:55] I'll show you examples, but I need to teach a few more things about streams first.

[00:03:00] There are a lot of features that you might want to enable while reading data from a stream.

[00:03:07] Here's just a small example of that.

[00:03:09] You might want to decompress data.

[00:03:12] If you have compressed data, maybe in a zip file, you might want to decompress it as you read it.

[00:03:17] You might want to decrypt it, if it's encrypted.

[00:03:20] Maybe you want to compute a digest of the stream.

[00:03:22] A digest is just a fixed length value that basically summarizes the data of a stream, so it could be useful to compute one of those.

[00:03:30] There's really no limit to what you might want to do with the data while you're reading it, these are just a few examples.

[00:03:35] Another one is you might want to count the bytes, you might want to count the number of lines, or you might want to buffer the data.

[00:03:41] You might want to keep it in some memory buffer for later use.

[00:03:45] Those are just a few examples of what you might want to do, and that's why we have something called filter input streams.

[00:03:52] Filter input streams are streams that can be attached to another stream.

[00:03:57] You can think of that like a pipeline where you connect different pipes together, and once you connect the pipes together, you just read from the end of the pipe and you get the data, and each stream along the pipeline will do the thing that it was intended to do.

[00:04:10] That actually makes it really easy to do the things on this slide and many other things.

[00:04:17] The way that works is you open an InputStream on some data source.

[00:04:21] If your data source is a file, you would open a file input stream on the file, and then you would attach whatever other streams you want to that by attaching streams that implement the filter input stream class.

[00:04:36] Each stream in that pipeline will read the data from the stream that it's attached to, and it will manipulate that data in some way so the data that comes out the end is manipulated by all the streams along the pipeline.

[00:04:48] That becomes a really powerful concept for getting behavior that would otherwise be difficult to get, and you'll find that it's actually really easy.

[00:04:56] We have the same concept with output streams.

[00:04:59] Writing bytes works the same way as reading them; it's just in reverse.

[00:05:04] We can set up this pipeline of output streams.

[00:05:09] We have basically the same output streams that we have input streams for.

[00:05:14] For example, we have a FileInputStream, we also have a FileOutputStream.

[00:05:19] We have a PipedInputStream, we also have a PipedOutputStream.

[00:05:23] There are many kinds of output streams, and those allow us to write data to different destinations, to a file, to a thread, or various other destinations.

[00:05:34] Just like we had filter input streams, we have filtered output streams, so we can do the reverse.

[00:05:39] We can compress data as we send it out to a file, for example.

[00:05:42] We could encrypt it and compress it.

[00:05:45] We can connect streams together in different ways to get different behavior as we're writing our data, and it's actually very easy to do that.

[00:05:54] This basically says the same thing about filter input streams that I already said about filter output streams.

[00:06:01] With that, let's look at an example to see what I'm talking about here.

[00:06:05] We're going to look at first, an example.

[00:06:07] I think we'll do the compressed example first.

[00:06:11] That is an example that will show how you could write a zip file out using streams, and then we'll look at a decompressed example that would show how you can decompress it as you read it. *(Wilkerson opens the example in Intellij.)*

[00:06:30] First of all, notice I have a bunch of imports from the java.io package.

[00:06:35] Basically, anything you do with input/output is going to be in the IO package.

[00:06:43] In this example, I have a file *(public class)* called Compress.

[00:06:47] I have a *(private)* static final *(int)* variable called CHUNK_SIZE *(that equals 512)*, and that specifies a number of bytes that I can write out at one time.

[00:06:59] We'll start with the main method.

The following code is the main method:

```
public static void main(String [] args) {
        Compress compress = new Compress();
        if (args.length == 2) [
                try {
                        compress.compressFile(args[0], args[1]);
                } catch (IOException e) {
                        e.printStackTrace();
                }
        } else {
                compress.usage();
        }
}
```

End of code.

[00:07:01] I have a main method that creates an instance of this class we're looking at, and then it checks to make sure that it got two parameters.

[00:07:08] It needs to have both an input file name and an output file name.

[00:07:12] If it doesn't get that, it prints out a usage string. If it does, it uses those two parameters and it passes them to a compressFile method.

The following code is the compressFile method:

```
public void compressFile(String inputFilePath, String outputFilePath) throws
IOException {
        File inputFile = new File(inputFilePath);
        File outputFile = new File(outputFilePath);
        try(FileInputStream fis = new FileInputStream(inputFile);
                BufferedInputStream bis = new BufferedInputStream(fis);
                FileOutputStream fos = new FileOutputStream(outputFile);
                BufferedOutputSream bos = new BufferedOutputStream(fos);
                GZIPOutputStream zipos = new GZIPOutputStreamI(bos)) {
                byte [] chunk = new byte[CHUNK_SIZE];
                int bytesRead;
                while((bytesRead = bis.read(chunk)) > 0) {
                        zipos.write(chunk, 0, bytesRead);
                }
        }
}
```

End of code.

[00:07:22] If we look at our compressFile method, it takes these two parameters.

[00:07:27] One represents the input file path, the other one represents the output file path.

[00:07:31] The first thing we do is create two File objects.

[00:07:35] Then within a try-with-resources, my try-with-resources ends right there, so within that code block, I set up the two streams.

[00:07:44] I set up my input stream.

[00:07:47] I want to read a file, so I start with a FileInputStream.

[00:07:50] I want that to be efficient, so I attach a BufferedInputStream to it.

[00:07:54] What that does is typically, with a FileInputStream, I'm either reading one byte at a time or I'm giving it some buffer.

[00:08:03] I'll give it an array and call a read method that will cause the array to be filled, but that can be a little bit tricky and a little bit tedious.

[00:08:10] I might have cases where I would like to read big chunks of data to minimize the number of writes I do against the file so it can be more efficient, but maybe I don't want to process big chunks of data.

[00:08:23] The way I can deal with that is I attach a BufferedInputStream to my FileInputStream.

[00:08:28] The way that works, that will mean that every time I read from my BufferedInputStream, I'll read some number of bytes, and if the BufferedInputStream has that many bytes, it will give them to me, if it has that many bytes or more.

[00:08:40] If it doesn't, it will ask the FileInputStream for some more data and it will ask it for a lot of bytes, probably a lot more than I'm asking for.

[00:08:48] Then it will buffer those, so for the next several reads, it won't have to go to the file system again, so that makes your reads much more efficient.

[00:08:55] We've attached a BufferedInputStream, so now we can read efficiently from a file.

[00:09:02] Now, what we want to do is we want to read from some input file and compress it as we write it out.

[00:09:08] For the output strings, I start with a FileOutputStream again.

[00:09:12] I attach that to my output file that was specified here.

[00:09:17] Then I want to write efficiently, so I'll put a BufferedOutputStream on the FileOutputStream.

[00:09:22] I'll attach those together.

[00:09:23] I would also like to zip my data as I write it out.

[00:09:27] Now I'll attach a GZIPOutputStream to my BufferedOutputStream.

[00:09:31] What I'm going to do is I'm just going to write on the GZIPOutputStream.

[00:09:35] What that will do is it will take the data, zip it, and then pass it to the BufferedOutputStream, which will buffer it in big chunks and it will only write periodically; it won't write every time.

[00:09:47] The rest of this code should look pretty familiar to you.

[00:09:50] You've seen this before, or code just like it.

[00:09:53] First of all, I create a byte array called chunk that has some size; that's how many bytes that I can read at a time.

[00:10:03] I have this variable to keep track of how many I actually do read.

[00:10:06] Remember, when I read a file, there may not be enough bytes.

[00:10:10] There might not be enough bytes to fill this array, so I need to know how many are actually read.

[00:10:15] Then within a while loop, I will just keep calling read on my BufferedInputStream.

[00:10:21] Keep in mind, when I read it, I'm asking for some array of bytes, but the BufferedInputStream is free to grab more bytes than that from the FileInputStream so it doesn't have to do very many reads to the file system.

[00:10:36] Each time I iterate this loop, I'm going to either get enough bytes to fill this array or I'm going to get all the bytes that are available.

[00:10:44] As long as that number is greater than 0, I still have bytes, so I keep reading.

[00:10:48] Then on my GZIPOutputStream, I will write the number of bytes that I got, right from the array that I read into, starting at position 0, and this is how many bytes I'll write, so that just reads in and writes out.

[00:11:03] The cool thing about that is it's zipping the data as it goes and I don't even have to know how to zip the data.

[00:11:08] That is encapsulated within the GZIPOutputStream, so it just works automatically.

[00:11:15] Then remember, we have this try-with-resources concept.

[00:11:19] This means that all of these are going to be closed when I exit the try, so I don't have to worry about closing anything.

[00:11:26] That is the compressed example.

[00:11:29] I also have this legacy compress example and this is exactly the same as compressed, except it's not using a try-with-resources, it's using a try-finally.

[00:11:37] I'm not going to go through the code there, but you can look at that if you want to.

[00:11:42] Let's look and see how we could do the reverse.

[00:11:45] How could we read a compressed file and write it out decompressed? The example is going to be very similar.

[00:11:53] This is my decompressed example. *(Wilkerson switches to another example in IntelliJ.)*

[00:11:55] Here, the main method looks about the same.

Begin visual description. The differences between this and the previous example is that the class is called Decompress instead of Compress, the instance created is: Decompress compress = new Decompress();. In the try block, compress calls the decompressFile method. End visual description.

[00:11:58] Now, we're creating an instance of Decompress.

[00:12:01] Probably should have changed the name of that reference.

[00:12:06] It just grabs it two parameters and passes them to decompressFile.

Begin visual description. The differences between the decompressFile and compressFile methods is that the prior has a GZIPInputStream in the try-with-resources while the latter had a GZIPOutputStream in the try-with-resources. In

the while loop, decompressFile sets bytesRead equal to zipis.read(chunk) instead of bis.read(chunk) in compressFile. Also in the while loop, decompressFile has bos.write instead of zipos.write like in compressFile. End visual description.

[00:12:12] Here we create two File objects and within my try-with-resources, I set up the readers or the input stream and the output stream again.

[00:12:22] Here I create a FileInputStream attached to the input stream File object.

[00:12:28] I want to read efficiently, so I attach a BufferedInputStream.

[00:12:31] But now I'm expecting that I'm going to read zipped bytes.

[00:12:35] I want to unzip them.

[00:12:37] The way to unzip them is attach a GZIPInputStream, and that will unzip the data as you read from the end of it.

[00:12:43] Then I want to write it out unzipped, so I create a new FileOutputStream to attach to the output file.

[00:12:50] I attach a BufferedOutputStream to that and now the read looks about the same as before.

[00:12:56] Now, I'm reading from the zip input stream that will unzip the bytes, and then I just write unzip bytes out from my example.

[00:13:04] Again, I have a legacy decompress and the only difference is that here, I'm not using try-with-resources, I'm using try-finally, so I have to clean up.

[00:13:18] That's an example, just one example of how I can use streams and create these pipelines to get really sophisticated behavior that I don't necessarily even know how to write.

[00:13:29] I can just use streams that are already written and connect them together and get the behavior that I want.

[00:13:35] That's a really powerful concept in input/output with Java.

[00:13:39] Okay, just a couple of other things to learn about reading and writing from streams.

[00:13:46] There is a class that allows me to read and write data types.

[00:13:50] It's not really that convenient to work with bytes most of the time.

[00:13:54] If I really want bytes, if I'm reading an image file or something, that's what I need, but often I'm reading a file that contains datatypes, so for example, it might contain some ints and floats and doubles, some strings and characters, and so it would be nice if I didn't have to worry about converting from binary to each of those data types.

[00:14:11] If I attach a DataOutputStream or a DataInputStream—if I'm talking about reading, if I attach a DataInputStream, that gives me methods for each of the datatypes and I can just call read int, for example.

[00:14:24] Now that assumes that I know the order that that data appears in the file, but if you created the file format, you would know that.

[00:14:32] I can do the same thing by attaching a DataOutputStream to an output stream. Now I can write datatypes instead of bytes, which can be much more convenient.

[00:14:42] That is how to use input streams and output streams, and that will be something that you use a lot as a Java programmer.