

Theory: Output streams

🕒 27 minutes 0 / 5 problems solved

Skip this topic

Start practicing

1667 users solved this topic. Latest completion was about 5 hours ago.

Although you might think you haven't worked with streams yet, surely you've already used similar constructions for printing data to console:

```
1 | System.out.println("Text to display");
```

In Java terminology `System.out` is an **output stream**, which allows programmers printing data to the console. In this way, you are familiar with output streams. It is time now to learn how streams work in more detail and consider other examples.

§1. Destination

As we learned from the previous topic output stream allows you to write data to a **destination**. Some obvious destinations that you probably already worked with are console and file. Disks, memory buffer, web sockets or other network locations can be a destination as well. Generally speaking, the destination is a target endpoint that data sent to output stream reaches.

Java standard library provides a wide variety of classes to represent an output stream. Quite a large number of these classes is the result of several factors. One of them is that each destination requires a specific way to write to it. Indeed, writing to a file differs from writing to a web socket!

§2. Character streams

Character output streams allow writing text data: `char` or `String`. You might have already used such streams as `FileWriter` and `PrintWriter` earlier for writing text data to files. Both of them, as well as other character output streams, have a common abstract ancestor `java.io.Writer`. Let's look at it closely.

The class contains a group of methods for writing. Some of them are listed here:

- `void write(char[] cbuf)` writes a char array
- `void write(char[] cbuf, int off, int len)` writes a portion of a char array
- `void write(int c)` writes a single character
- `void write(String str)` writes a string
- `void write(String str, int off, int len)` writes a portion of a string

Another important method is `close()`. It should be invoked for preventing resource leaks.

If you're familiar with **try-with-resources** construction, you know it is a better way to prevent resource leaks. For now, we're skipping it due to learning purposes

`Writer` has several direct subclasses for different purposes in the standard library. For example, `FileWriter` is intended for writing to files. `StringWriter` is designed to construct a string. `CharArrayWriter` uses `char[]` as a destination.

Let's consider `CharArrayWriter` class. Besides methods inherited from `Writer` the class has its own `toCharArray()` and `writeTo` methods. The former provides `char[]` with content. The latter writes content to another writer.

Imagine the case when you need to create two types of business cards. Each type of card has its own content, but the contact sections are the same. Here it will be convenient to implement `writeTo` method of `CharArrayWriter` to supply both cards with contact sections content.

Current topic:

[Output streams](#) ...

Topic depends on:

✗ [What are streams](#) ...

✗ [Writing files](#) ...

Topic is required for:

[Serialization basics](#) ...

[Sockets](#) ...

Table of contents:

[1 Output streams](#)

[§1. Destination](#)

[§2. Character streams](#)

[§3. Byte streams](#)

[§4. Character vs byte streams](#)

[§5. Buffered streams](#)

[§6. Conclusion](#)

[Feedback & Comments](#)

```
1  CharArrayWriter contactWriter = new CharArrayWriter();
2  FileWriter bc1 = new FileWriter("business_card_1.txt", true);
3  FileWriter bc2 = new FileWriter("business_card_2.txt", true);
4
5  contactWriter.write("Phone: 111-222-333; Address: Java Avenue, 7");
6  contactWriter.writeTo(bc1);
7  contactWriter.writeTo(bc2);
8
9  char[] array = contactWriter.toCharArray(); // writer content as char[]
10
11
12  bc1.close();
13
14  bc2.close();
15
16  contactWriter.close();
```

Here we've created `FileWriter` objects in append mode by passing `true` to an `append` parameter.

§3. Byte streams

From a computer's point of view, any data is just a sequence of bits: 0 or 1, which are usually assembled to bytes of 8 digits. In other words, any data is represented as a serial set of bytes. This means that images, audio, videos and so on have a binary format, i.e. represented as a sequence of bytes. Actually, text files have byte representation too: if you remember, characters are combinations of bytes.

Java has a set of classes called **byte output streams** to write bytes.

Byte output stream classes from the standard library extend `java.io.OutputStream` abstract class. The class contains three methods for writing:

- `void write(byte[] b)` writes a byte array
- `void write(byte[] b, int off, int len)` writes a portion of a byte array
- `abstract void write(int b)` writes a single byte

Just like character streams, byte streams have `void close()` that should be invoked in a similar way.

Let's look at some direct subclasses of `OutputStream` from the standard library. `FileOutputStream` is intended for writing data to a file as a destination. `ByteArrayOutputStream` as you may guess allows writing to `byte[]` destination. Such classes like `FilterOutputStream` or `PipedOutputStream` have no endpoint destination and write data to other output streams. These classes are supposed to be intermediate streams for data transformation or possibly providing additional functionality.

Let's look at an example where we write something to a file using `FileOutputStream`. The class has a set of constructors. Some of them are:

- `FileOutputStream(String fileName)`
- `FileOutputStream(String fileName, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

Parameter `append` indicates whether to append (*true*) or overwrite (*false*) an existing file.

It is useful to be aware that `FileOutputStream` will create a file with the name provided if one does not exist yet. It creates a file right after `FileOutputStream` is initialized, even if you have not tried to write into it.

Let's look at the snippet now.

```
1  byte[] data = new byte[] {'s', 't', 'r', 'e', 'a', 'm'};
2  OutputStream outputStream = new FileOutputStream("sample.txt", false);
3  outputStream.write(data);
4  outputStream.close();
```

After running this code you will see a `sample.txt` file with content `stream` in it.

§4. Character vs byte streams

Note that all methods of byte streams considered above allow you to only write bytes. It means that you can't directly write strings, you must convert them to `byte[]` before. So if you want to write stream to a file you have to convert it into bytes first. For instance, you can use `getBytes()` method for that.

```
1 String str = "stream";
2 byte[] strAsBytes = str.getBytes(); // convert String to byte[]
```

Converting `String` to `byte[]` every time you need writing something is inefficient and inconvenient. Moreover, many character streams are based on **byte streams** and well-optimized. So if you want to write text, do not reinvent the wheel: use character output streams.

On the other hand, you'll need to use byte streams when you will work with binary files, for example, .jpg image or .pdf file.

§5. Buffered streams

Output streams have 2 classes from the standard library which do buffering.

`BufferedOutputStream` is based on the buffering principle. It has only two constructors:

- `BufferedOutputStream(OutputStream out)`
- `BufferedOutputStream(OutputStream out, int size)`

Same works for `BufferedWriter`:

- `BufferedWriter(Writer out)`
- `BufferedWriter(Writer out, int size)`

These classes are intermediate output streams. They take an output stream as an input and do buffering before delegating to another stream. An additional parameter `size` is the size of the buffer. If you want to release all data from the buffer by writing it to a destination, you can use `flush()` method. It is usually called automatically when your buffer is full or before stream closing.

§6. Conclusion

The output stream is a way to write data to a destination. The destination is a target endpoint of data, that can be a file, a console or even a web socket. Streams are divided into byte and character ones. Byte output streams allow writing sequences of bytes. It is necessary for working with binary files. Character output streams are intended for writing text. Character output stream classes usually end with *Writer*, because they extend one abstract `java.io.Writer` class as a rule. Similarly, byte output streams end with *OutputStream*. Some streams use buffering under the hood. It is a widely used optimization that tries to minimize costly interaction with a destination.

 Report a typo

168 users liked this theory. 8 didn't like it. What about you?



Start practicing