Theory: Output streams

34 minutes 0 / 0 problems solved

Skip this topic Start practicing

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 to print 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.

2 required topics

✓ What are streams In project

✓ Writing files In project

2 dependent topics

Serialization basics In project

Sockets

```
CharArrayWriter contactWriter = new CharArrayWriter();
FileWriter bc1 = new FileWriter("business_card_1.txt", true);
FileWriter bc2 = new FileWriter("business_card_2.txt", true);

contactWriter.write("Phone: 111-222-333; Address: Java Avenue, 7");
contactWriter.writeTo(bc1);
contactWriter.writeTo(bc2);


char[] array = contactWriter.toCharArray(); // writer content as char[]



bc1.close();

bc2.close();

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.

```
byte[] data = new byte[] {'s', 't', 'r', 'e', 'a', 'm'};

OutputStream outputStream = new FileOutputStream("sample.txt", false);
outputStream.write(data);
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 a string to a file, you have to convert it into bytes first. For instance, you can use the `getBytes()` method for that.

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

Converting `String` to `byte[]` every time you need to write something is inefficient and inconvenient. Moreover, many character streams are based on **byte streams** and are well-optimized. So if you want to write a 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.

Next section

2 sections left Expand all

Table of contents: