Theory: Input streams

1 hour

Verify to skip | Start practicing

As a rule, every program consumes some data as a way to communicate with the outer world. It can be a user input from the console, a configuration file, or something else. Java has a common mechanism for consuming data called **input streams**. We have already provided some text as data for our programs, and now it is time to dive deeper and explore the whole mechanism.

§1. Sources

Data can be obtained from many points considered as providers. Besides standard input or files, those can be network connections, in-memory buffers, or even objects. All of them are called **sources** for input streams. In fact, a source is any data that can be consumed and processed by a program. Working with data is quite a specific thing, and each source needs a specialized class.

§2. Character streams

There are several classes for reading text. They are called character input streams and allow reading text data: `char` or `String`. For instance, there are `FileReader`, `CharArrayReader`, `StringReader`, etc.

> The class name indicates what type of source it uses as input and usually ends with *Reader,* since *all* such classes extend the `java.io.Reader` class.

Each class provides a set of useful methods while they also have common methods for reading data:

- `int read()` reads a single character. If the end of the stream is reached, the method returns the value `-1`. Otherwise, it returns the numerical representation of the character according to the current encoding;
- `int read(char[] cbuf)` reads a sequence of characters into the passed array up to its capacity and returns the number of characters that were actually read. It can also return `-1` in case no data was read;
- `int read(char[] cbuf, int off, int len)` reads characters into a portion of an array.

These methods return the number of characters that were actually read or `-1`. They also block the program from running until some input is available or the end of the stream is reached.

Another important method is `void close()` that should be invoked after a stream was used.

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

§3. Example of a character stream

Let's consider `FileReader` as an example of the `Reader` classes. `FileReader` has a set of constructors. Here are some of them:

- `new FileReader(String fileName)`
- `new FileReader(String fileName, Charset charset)`
- `new FileReader(File file)`
- `new FileReader(File file, Charset charset)`

3 required topics

- ✓ What are streams
- ✓ Array — In project 13
- ✗ Writing files

3 dependent topics

- Try with resources
- Serialization basics
- Sockets

As you can see, it can read text data from the file indicated either by a path `String` or as a `File` object.

> A charset is a class that declares the encoding from sequences of bytes to characters. By default, java uses the UTF-16 encoding, suitable for most tasks. However, sometimes the file may have another encoding and you'll have to use a different charset to read the content of the file properly.

Now let's try to read a file. Say we have a file `file.txt` with the following content: `input stream`.

```
Reader reader = new FileReader("file.txt");

char first = (char) reader.read(); // i
char second = (char) reader.read(); // n

char[] others = new char[12];
int number = reader.read(others); // 10
```

After running the code, `others` will contain `['p', 'u', 't', ' ', 's', 't', 'r', 'e', 'a', 'm', '\u0000', '\u0000']`.

Let's explain the result. Since we've read the first two letters into other variables, the first 10 characters of `others` are filled starting from the third letter. When the stream reached the end of the file it stopped reading, so the last two characters are not updated.

When you create an empty array it is actually filled with default values, which are `'\u0000'` for a char array. This is the reason why the last 2 elements of `others` are `'\u0000'`.

The tricky thing here is that `'\u0000'` is interpreted as an empty symbol and not displayed at all, although technically it is present. Remember that when you read data into an array.

Another common way of reading a text data stream is to read it char by char until the stream is closed:

```
FileReader reader = new FileReader("file.txt");

int charAsNumber = reader.read();
while(charAsNumber != -1) {
    char character = (char) charAsNumber;
    System.out.print(character);
    charAsNumber = reader.read();
}
reader.close();
```

When `-1` is returned, it means the end of stream was reached, so that is there's nothing left to read.

§4. Byte streams

Byte streams may seem more complicated so let's start with something you already know. To read some text data from the standard input we may create a `Scanner` instance. You are familiar with the process:

```
Scanner scanner = new Scanner(System.in);
```

`System.in` here is actually an example of a byte input stream. There are other specialized byte stream classes: `ByteArrayInputStream` is used for reading from `byte[]`, `FileInputStream` is dedicated for files, `AudioInputStream` is a way of consuming audio input and there's more.

The class name of a byte stream indicates what type of source it uses as input and usually ends with *InputStream,* since all such classes extend the `java.io.InputStream` class.

All byte stream classes have methods for reading similar to character input streams:

- `abstract int read()` reads a single byte;
- `int read(byte[] b)` reads a number of bytes and stores them in a byte array;
- `byte[] readAllBytes()` reads all bytes.

The method that reads bytes into an array, returns an `int` value. It is the number of bytes that were actually read from the source. If `-1` value is returned it is a sign that no bytes were read.

Each input stream class also has a `void close()` method to release system resources.

§5. Example of a byte stream

Suppose we have a file `file.txt` that contains the following text: `input stream`. Let's read it using the `FileInputStream` class. It can be created by using several constructors, such as:

- `new FileInputStream(String pathToFile)`
- `new FileInputStream(File file)`

Here we create a file input stream by specifying the name of the file:

```
1  FileInputStream inputStream = new FileInputStream("myfile.txt");
```

We are going to read the first five bytes:

```
1  byte[] bytes = new byte[5];
2  int numberOfBytes = inputStream.read(bytes);
3  System.out.println(numberOfBytes); // 5
4  inputStream.close();
```

Now `bytes` contains `['i', 'n', 'p', 'u', 't']`.

The byte-by-byte approach also works here, similar to the character streams example.

§6. Testing input streams

In previous examples, we were considering reading data from files to show you how it is done. You can try it yourself at any moment. However, in the following coding problems, we are going to ask you to read data from standard input by other input streams classes.

And there are some things to keep in mind.

When testing your programs in IDE, you type the text and then press **Enter** to push the typed text into the input stream, like when you did with `Scanner`. In this case, the *end-of-line* symbol is also appended as the last symbol of the typed sequence of characters and will be read into an array if that's what you do.

Pressing Enter doesn't close the input stream. It is still opened and waits for input. To actually close the input stream you need to produce an *end-of-file* event. In IDEA, you should press **Ctrl+D** (Windows and Linux) or **<command>+D** (MacOS).

§7. What type of stream should I use?

Table of contents:

↑ Input streams

§1. Sources

§2. Character streams

§3. Example of a character stream

§4. Byte streams

§5. Example of a byte stream

§6. Testing input streams

§7. What type of stream should I use

The main difference between byte and character streams is that byte ones read input data *as bytes* while character ones work with characters.

A computer understands only sequences of bytes. From this perspective, any data is a set of bytes and byte input streams are a common way of reading any kind of data. For computers, characters are still combinations of bytes defined by a charset specification.

On the contrary, as human beings, we are used to dealing with sequences of characters. Character input streams are aimed to read data which consists of characters. Under the hood, they still read bytes, but they immediately encode bytes to characters.

If you need to read a text, use character input streams. Otherwise, for example, while reading audio, video, zip, etc., use byte input streams.

§8. Conclusion

Input streams provide a way to read data from a source. The source is a data provider like a console, standard input, a file, a string, or even a network connection. There are two types of sources: byte and character ones. Character input streams are intended for reading text only. Byte input streams allow reading sequences of raw bytes. Character input stream classes usually end with *Reader*. Similarly, byte input streams end with *InputStream*.

§8. Conclusion

Discussion