Input/Output in Java

September 9, 2018

Contents

1 Overview

This note covers how to get data into and out of your Java program. The way you do this depends on the type of the data, the source/destination, and the application.

2 Types of data

2.1 Binary data

Ultimately, all data is just binary: 0's and 1's. When saving to a file or transmitting over a network, that is what is saved or transmitted. You can view any file in its raw binary form using the Unix facility `hexdump` on a Mac or `HexView` on Windows. This shows the bytes in the file in hexadecimal (base-16) format.

```
> more test.txt
This is a text file.
It contains two lines.
> hexdump test.txt
0000000 54 68 69 73 20 69 73 20 61 20 74 65 78 74 20 66
0000010 69 6c 65 2e 0a 49 74 20 63 6f 6e 74 61 69 6e 73
0000020 20 74 77 6f 20 6c 69 6e 65 73 2e 0a
000002c
>
```

Java programs can read or write binary data as a stream of raw bytes without any processing. The lowest-level facilities for this are `java.io.InputStream` and `java.io.OutputStream`. These provide basic mechanisms for reading and writing data one byte at a time or an array of several bytes at a time.

The classes `InputStream` and `OutputStream` are abstract classes, which means they cannot be instantiated directly. One must instantiate them using one of their concrete implementations. There are numerous options, depending on the source or destination of the data: `AudioInputStream`, `ByteArrayInputStream`, `FileInputStream`, `ObjectInputStream`, `StringBufferInputStream`, etc.

These classes are rarely used by themselves, but are usually wrapped in other classes that provide extra functionality, such as buffering or encoding/decoding. More on this below.

A *binary file* is one containing data that is not meant to be interpreted as text; for example, images or audio files.

2.2 Text data

Text data consists of Java strings or character streams. A *string* is a fixed finite sequence of characters and is an instance of `java.lang.String`. A character stream is a sequence of characters of indeterminate length, usually read from some source such as a file or user input.

A *character encoding* is a translation scheme that tells how each character is represented in memory as a sequence of bytes. The most common characters (letters, numbers, whitespace characters, common punctuation) are usually represented as one byte, but some less common characters require more than one byte, and the representations may differ depending on the encoding. There are several character encodings in common use, and the defaults may vary from platform to platform. The most common ones are ISO-8859-1 (also known as Latin1), UTF-8, and UTF-16.

For example, consider converting the string `CS2112` into a sequence of bytes. Each character has a unique identifying number that is fixed and universal, as specified by the Unicode standard. These numbers are called *code points*. The characters in the string `CS2112` have the following code points, listed here in hexadecimal:[1]

character	code point
`C`	0x43
`S`	0x53
`2`	0x32
`1`	0x31

The code point corresponding to a character is an abstract entity. It is not the internal representation of the character in memory. To get the representation in memory, the code point is translated to a sequence of bytes as specified by the character encoding. The ISO-8859-1 encoding supports only code points 0x00–0xFF, and the translation is direct, which means that the internal representation is a single byte and is the same as the code point. Thus the string `CS2112` will be encoded as a sequence of six bytes 0x43, 0x53, 0x32, 0x31, 0x31, 0x32.

Here the translation from code points to bytes is direct, but this is not necessarily so for other character encodings. For example, with the UTF-8 encoding, a byte larger than 0x7F indicates that it is the first byte of a multi-byte sequence.

The lowest-level facilities for reading and writing streams of text data are `java.io.Reader` and `java.io.Writer`. These provide basic mechanisms for reading and writing text data one character at a time or an array of several characters at a time. They do character conversion according to the platform's default character encoding, but perform no other formatting.

Like `InputStream` and `OutputStream`, the classes `Reader` and `Writer` are abstract classes, which means they must be instantiated using one of their concrete implementations: `CharArrayReader`, `InputStreamReader`, `FileReader`, `StringReader`, etc.

Also like `InputStream` and `OutputStream`, the `Reader` and `Writer` classes are rarely used by themselves, but are usually wrapped in other classes that provide extra functionality, such as buffering or further encoding/decoding.

A *text file* is one containing data that is meant to be interpreted as text.

2.3 Formatted data

It may be necessary to translate raw text or binary data to a desired form before it can be used by an application. Such translation is usually done by a *codec*, a coding/decoding scheme particular to the application. This may involve specialized hardware, for example to play an audio stream or to display an image. However, it may also apply to text data. For example, a character

[1]The prefix "0x" indicates a hexadecimal (base-16) numeral.

stream consisting of a sequence of digits needs to be translated to a number before you can do arithmetic on it.

A highly versatile and configurable class for parsing input text in a known format is `java.util.Scanner`. On the output side, `java.io.PrintStream` provides extensive text formatting capabilities.

There are many other examples of text streams requiring specialized software codecs: HTTP[2] commands, serialized Java classes in JSON[3] or XML[4] format, web pages in HTML.[5] We will be seeing some of these in A7.

2.4 Buffering

Access to the underlying input stream may be inefficient if the source is remote or the medium is slow. To improve performance, input streams and readers are often wrapped in a `BufferedInputStream` or `BufferedReader` to provide buffering. The underlying stream is read in large chunks at a time, which occurs only when the buffer becomes empty.

```
InputStream in =
   new BufferedInputStream(new FileInputStream(fileName));

Reader in =
   new BufferedReader(new FileReader(fileName));
```

3 Input sources

There are a number of possible sources from which your Java program may get data. Except for command line arguments, most input data is available in the form of an `InputStream`.

3.1 Command line arguments

When a user runs your program from the console, they can supply arguments on the command line. These arguments are then available to your program in the `String[]` array parameter of the `main` method.

For example, if you type in a console window

```
java MyProgram a b "c d" e
```

and the main method of `MyProgram` looks like

```
public static void main(String[] args) {
   for (String s : args) {
```

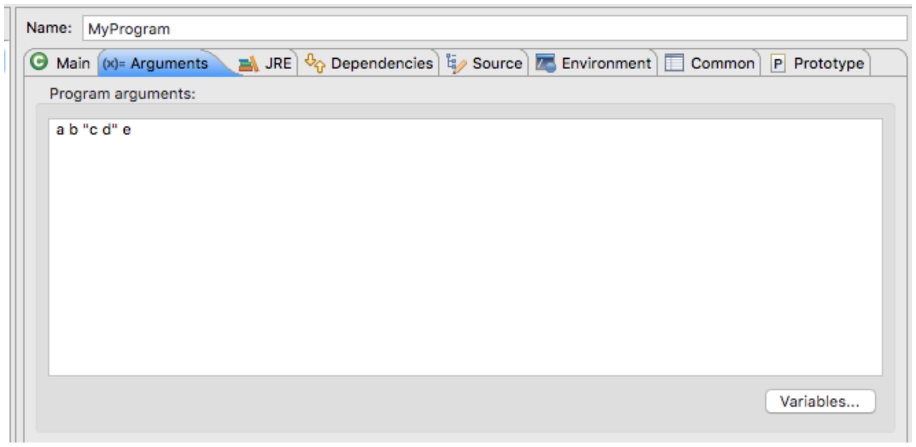
[2]HyperText Transfer Protocol
[3]JavaScript Object Notation
[4]Extensible Markup Language
[5]HyperText Markup Language

```
            System.out.println(s);
        }
    }
```

then you will see the output

```
    a
    b
    c d
    e
```

In Eclipse, you can supply the command line arguments under the Arguments tab in the run configuration.



```
a b "c d" e
```

3.2 Console (user input)

Another possible source of text data is console input typed in at the keyboard by the user while the program is running. There is a built-in `InputStream` just for this purpose, called `System.in`. Often this is wrapped in an instance of `Scanner` to read the input one line at a time.

Say you executed the following code:

```
    Scanner sysin = new Scanner(System.in);
    System.out.print("Please type something: ");
    String s = sysin.nextLine();
    System.out.println("You typed: " + s);
```

Here is what you would see. User input is in green. The program will respond when you hit enter.

```
    Please type something: hi there
    You typed: hi there
```

This is a pretty basic use of `Scanner`, but it has a lot of other useful functionality, such as parsing numbers and other text conforming to a fixed format.

3.3 File system

Often your program would like to read a file from the file system. Typically the user will specify the file to be read, either by selecting it in a `FileChooser` or textually by supplying a path name. Either method results in a path name by which the file can be accessed. The path name will be either *absolute*, starting from the root of the file system (usually C: on Windows and / on Unix-based systems such as Macs) or relative to the current directory. If you run the program from the console, the current directory is the one you are in; if you run it from Eclipse, it is the project directory by default, but you can specify a different one in the run configuration.

You can create an `InputStream` from a binary file or a `Reader` from a text file by supplying the path name to the constructor.

```
InputStream in =
   new FileInputStream("/Users/kozen/Documents/myBinaryFile");

Reader in =
   new FileReader("/Users/kozen/Documents/myTextFile.txt");
```

3.4 Resource files

A *resource* is a file with some data that your program uses internally; for example, a background image, audio clip, or text such as an English dictionary. It is considered an integral part of your program and must always be available whenever and wherever your program runs. It should be packaged up with the program when you create an executable `jar`.

Java looks for resources using your program's `ClassLoader`. This is the system module that finds and loads the classes that your program uses. For this reason, resources must be on the classpath so that the `ClassLoader` can find them.

Here is a way to access a resource file as an `InputStream`:

```
InputStream in =
   ClassLoader.getSystemResourceAsStream("myResource");
```

4 Output destinations

4.1 Console

To write text to the console, use

- `System.out.print` to print a string without a trailing newline,
- `System.out.println` to print a string with a trailing newline, and
- `System.out.format` to print a formatted string.

The object System.out is an instance of java.io.PrintStream, which is a subclass of OutputStream, but does character conversion according to the platform's default character encoding as described in §2.2, and also allows formatted output.

4.2 Error messages

Error messages should go to System.err, which is another PrintStream like System.out, also going to the console. The advantage of using System.err for error messages is that it operates asynchronously with System.out, which means that error messages will appear immediately rather than waiting for any buffered output in System.out to clear.

4.3 File system

An output file can be created with FileOutputStream, a subclass of OutputStream. This can be wrapped in a BufferedOutputStream for efficiency if desired.

To write text files, the OutputStream can be wrapped in a PrintStream or PrintWriter.

5 Examples

The following examples assume that the following field declaration is in scope:

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

Many of these examples use the try-with-resources idiom so that the resources will be automatically closed when you are done with them.

5.1 Reading the command line arguments

See §3.1.

5.2 Printing out user input

```
void displayUserInput() {
   while (true) {
      System.out.print("Please type something: ");
      String s = sysin.nextLine();
      if (s.equals("bye")) break;
      System.out.println("You typed: " + s);
   }
   System.out.println("bye");
}
```

Sample run (user input is in green):

```
Please type something: hi there
You typed: hi there
Please type something: Java is cool!
You typed: Java is cool!
Please type something: bye
bye
```

5.3 Printing out a text file

```
void displayTextFile(String fileName) {
   try (InputStream in =
           new BufferedInputStream(new FileInputStream(fileName));
        Scanner scanner = new Scanner(in)) {
      while (scanner.hasNextLine()) {
         System.out.println(scanner.nextLine());
      }
   } catch (IOException e) {
      System.out.println("File could not be read");
   }
}
```

or

```
void displayTextFile2(String fileName) {
   char[] buffer = new char[256];
   try (Reader in = new BufferedReader(new FileReader(fileName))) {
      for (int n = in.read(buffer); n != -1; n = in.read(buffer)) {
         System.out.print(new String(buffer, 0, n));
      }
   } catch (IOException e) {
      System.out.println("File could not be read");
   }
}
```

5.4 Copying a text file

```
long copyTextFile(File inFile, File outFile) {
   if (!inFile.exists()) {
      System.out.print("Input file does not exist");
      return -1;
   }
   if (outFile.exists()) {
      System.out.print("Output file exists; overwrite [yes/no]? ");
      if (!sysin.nextLine().equals("yes")) return -1;
   }
   try (Reader in = new BufferedReader(new FileReader(inFile));
        Writer out = new BufferedWriter(new FileWriter(outFile))) {
      return in.transferTo(out);
   } catch (IOException e) {
      System.out.println("File was not copied");
      return -1;
   }
}
```

5.5 Copying a binary file

```
void copyBinaryFile(File inFile, File outFile) {
   if (!inFile.exists()) {
      System.out.print("Input file does not exist");
      return;
   }
   if (outFile.exists()) {
      System.out.print("Output file exists; overwrite [yes/no]? ");
      if (!sysin.nextLine().equals("yes")) return;
   }
   try (InputStream in =
           new BufferedInputStream(new FileInputStream(inFile));
        OutputStream out =
           new BufferedOutputStream(new FileOutputStream(outFile))) {
      for (int c = in.read(); c != -1; c = in.read()) {
         out.write(c);
      }
   } catch (IOException e) {
      System.out.println("File was not copied");
   }
}
```

5.6 Printing out a text resource file

```
void displayTextResourceFile(String fileName) {
   InputStream in = ClassLoader.getSystemResourceAsStream(fileName);
   Scanner scanner = new Scanner(in);
   while (scanner.hasNextLine()) {
      System.out.println(scanner.nextLine());
   }
   scanner.close();
}
```