8-java-io-fundamentals

Contents

IO Streams

The input and output in Java is vast, and it does not only include console or file input output operations, alone. The public static fields in out and err in `java.lang.System` class respectively represent the standard input output and error streams. System.in is of type `InputStream` and System.out and System.err are of type `PrintStream`.

The example below shows how to read in from the standard `in` descriptor stream (stdin) which is by default usually provided by the `tty`, and is the keyboard, or other input devices

```
System.out.print("Type a character: ");
int val = 0;
try {
    // the return type of read is int, but it returns a byte value!
    val = System.in.read();
} catch(IOException ioe) {
    System.err.println("Cannot read input " + ioe);
    System.exit(-1);
}
System.out.println("You typed: " + val);
```

The most basic example, which reads one byte from the input stream (stdin), the read method reads one byte, in the range of 0 to 255. Hence for example for input character of 5, the print out is actually 53, since that is the ascii code of the digit 5. The reason the read method does not return byte instead, is because the method also returns -1 or can return value outside the 255 range to signal some sort of abnormal result occurred from reading

The standard INPUT and OUTPUT streams are initialized when the JVM starts, Sometimes it is useful to redirect the standard streams by reassigning them. The method System.setIn, takes an `InputStream` object and the methods `System.setOut` and `System.setError` take in `PrintStream`, those are static methods, and allow one to override the default stdin and stdout stream descriptors to point to something else, for example a file.

```
try{
    PrintStream ps = new PrintStream("log.txt");
    System.setOut(ps);
    System.out.println("Test output to System.out");
} catch(Exception ee){
    ee.printStackTrace();
}
```

The snippet above shows a demonstration how all System.out calls can be redirected to a file instead of the standard output, which in a tty environment is usually the display. That is useful for logging purposes, among other things

Console

That class helps reading the data from the standard in, instead of relying on reading byte sized chunks from System.in, the console class takes care of providing a more user friendly interface to do more advanced reading from `stdin` without the general overhead. The console class also provides printing capabilities, meaning it also has reference to `stdout`

Creating

```
Console console = System.console();
if(console == null) {
    System.err.println("Cannot retrieve console object");
    System.exit(-1); // terminate the application
}
// read a line and print it through printf
console.printf(console.readLine());
```

The console object provided by the call to System.console(). The console object that is produced is not a direct wrapper around the objects from System.in or System.out, it directly interacts with the underlying operating system, to obtain the file descriptors to `stdin` and `stdout`, and also provides as mentioned a better API to help usability.

```
Note if this program is started from the command line, it will work just fine, however if
 it is started from an IDE, or through an internal process or indirectly, it will fail to
 obtain console object. That is because the JVM itself, is started indirectly in the IDE
tool, instead of in the console or terminal
```

Method	Description
Reader reader()	Returns the Reader object associated with this Console object; can perform read operations through this returned reference.
PrintWriter writer()	Returns the PrintWriter object associated with this Console object; can perform write operations through this returned reference.
String readLine()	Reads a line of text String (and this returned string object does not include any line termination characters); returns null if it fails (e.g., the user pressed Ctrl+Z or Ctrl+D in the console)
String readLine(String fmt, Object… args)	Same as the readLine() method, but it first prints the string fmt (
char[] readPassword()	Reads a password text and returns as a char array; echoing is disabled with this method, so when the user types the password, nothing will be displayed in the console.
char[] readPassword(String fmt, Object… args)	Same as the readPassword() method, but it first prints the string given as the format string argument before reading the password string.
Console format(String fmt, Object… args)	Writes the formatted string (created based on values of fmt string and the args passed) to the console.
Console printf(String fmt, Object… args)	Writes the formatted string (created based on values of fmt string and the args passed) to the console. This printf method is the same as the format method: This is a "convenience method"-the method printf and the format specifiers are familiar to most C/C++ programmers, so this method is provided in addition to the format method.
void flush()	Flushes any of the data still remaining to be printed in the console object's buffer.

The Console class also supports formatting input and output. These methods are mostly using string-formatting flags to format strings. It is quite similar to the printf() function provided in the library of the C programming language. The first parameter of the `printf` method is a format string. A format string may contain string literals and format specifiers. The actual arguments are passed after the format string. This method can throw `IllegalFormatException` of the passed format is not correct.

Output

```
%[argument_index][flags][width][.precision]datatype_specifier
```

- Each format specifier starts with % sign followed by argument index. flags width and precision information, and ends with a data type specifier

- Argument index refers to the position of the argument in the argument list it is an integer followed by , *asin*1 and 2$ for first and second argument respectively
- Flags are single character symbols that specify the characteristics such as alignment and filling characters, for padding. For instance flag "-" specified left alignment and "0" pads the number types with leading zeroes
- The width specifier indicates the minimum number of characters that will span in the final formatted string, then it is padded with spaces by default, in case the data is bigger than the specified width the full data appears in the output without trimming.
- The precision fields specifies the number precision digits in the output, relevant only for floating point number arguments
- Finally is the data type indicates the type of expected input data. The field is a placeholder for the specified input data.

Let's analyze the following format string - `"%-15s \t %5d \t\t %d \t\t %.1f \n"`

- `%-15s` - a string padded to at least 15 chars
- `%5d` - decimal or integer with at most 5 digits (padded with spaces if less)
- `%.1f` - floating point number that can be displayed with at most 1 digit after the floating point
- `\t` - all formatted arguments are padded or separated with a tabstop, to make some equal space between them

Symbol	Description
%b	Boolean
%c	Character
%d	Decimal integer (signed)
%e	Floating point number in scientific format
%f	Floating point number in decimal format
%g	Floating point number in decimal or scientific format depending on the value passed as argument)
%h	Hashcode of the passed argument
%n	Line separator (new line character)
%o	Integer formatted as an octal value
%s	String
%t	Date/time
%x	Integer formatted as an hexadecimal value

```
All of the format symbols are also supported by the format method in the Console class
as well, the format strings and symbols in general are applicable in general in a wide
variety of places in the java standard library
```

- If you do not specify any string formatting specifier, the printf() method will not print anything from the given arguments!

- Flags such as "-" and "0" make sense only when you specify width with the format specifier string.

- You can also print the % character in a format string; however, you need to use an escape sequence for it. In format specifier strings, % is an escape character, which means you need to use %% to print a single %.

- You can use the argument index feature (an integer value followed by $ symbol) to explicitly refer to the arguments by their index position. For example, the following prints "world hello" because the order of arguments are reversed:

  ```
  console.printf("%2$s %1$s %n", "hello", "world");
  ```

```
// $2 refers to the second argument ("world") and
// $1 refers to the first argument ("hello")
```

- The < symbol in a format string supports relative index with which you can reuse the argument matched by the previous format specifier. For example, assuming console is a valid Console object, the following code segment prints "10 a 12":

```
console.printf("%d %<x %<o", 10);
// 10 - the decimal value, a - the hexadecimal value of 10, and
// 12 - the octal value of 10
```

- If you do not provide the intended input data type as expected by the format string, then you can get an IllegalFormatConversionException. For instance, if you provide a string instead of an expected integer in your printRow() method implementation, you will get following exception:

```
Exception in thread "main" java.util.IllegalFormatConversionException:
d != java.lang.String
```

Input

It is also possible to use the console class to get input, there are some existing methods such as readLine (and other read based ones) In this methods the first arguments is the format specified string, and the following arguments are the values that will be passed to the format specifier string. These two methods return the character data read from the console.

```
Console console = System.console();
if(console != null) {
    String userName = null;
    char[] password = null;
    userName = console.readLine("Enter your username: ");
    password = console.readPassword("Enter password: ");

    // password is a char[]: convert it to a String first, then compare
    if(userName.equals("scrat") && new String(password).equals("nuts")) {
        console.printf("login successful!");
    }
    else {
        console.printf("wrong user name or password");
    }

    // "empty" out the password since its use is over
    Arrays.fill(password, ' ');
}
```

Note that at the end the password is emptied out, this is a common practice, to empty out the read password string once its use is over, it is using the arrays fill method for this purpose. This is secure programming practice to avoid malicious reads of program data to discover passwords strings. That is why the `readPassword` returns a `char[]` instead of a `String`, the String object will live in memory, and is also immutable, meaning there is no easy way to forcefully destroy its contents, once after the password is read, used and no longer needed

Streams

The streams API discussed in earlier chapters are not the same as the IO streams discussed here, they have very different purpose and application. Streams are ordered sequences of data. Java deals with input and output in terms of streams. When one reads a sequence of bytes from a binary file, the reading is done from an input stream (since the flux of information is from the file towards the run-time) when writing to file, one would use output stream (since the flux of information is from the run-time to the file)

Character streams

- meant for reading or writing to character or text based IO such as text files text documents, XML, HTML files.
- the data is usually utf-16 encoded and stored.
- iput and output characters streams are called readers and writers respectively
- the abstract classes of Reader and Writer and their derived classes in the java package provide support for character streams.

Byte streams

- meant for reading or writing to binary data io such as executable files, image files and files in low level file formats such as .zip .class .obj and .exe.
- data dealt with is bytes, units of 8-bit data
- input and output byte streams re simply called input & output streams, respectively
- the abstract classes of InputStream and OutputStream and their derived classes are what provide support for byte streams

```
Mixing different streams for different purposes is not recommended, for example if one
tries to read a .bmp image with a Reader (character stream)since the character streams
read and parse the read bytes as UTF-16, the final result will not be as expected, in this
 case one should use regular InputStream to read the contents of the image
```

Hierarchy

Character streams

```
Writer
+-- BufferedWriter
+-- CharArrayWriter
+-- FilterWriter
|   +-- OutputStreamWriter
|       +-- FileWriter
|   +-- PrintWriter
+-- PipedWriter
+-- StringWriter

Reader
+-- BufferedReader
+-- CharArrayReader
+-- FilterReader
|   +-- PushbackReader
+-- InputStreamReader
|   +-- FileReader
+-- PipedReader
```

Byte streams

```
InputStream
+-- AudioInputStream
+-- ByteArrayInputStream
+-- FileInputStream
+-- FilterInputStream
|   +-- BufferedInputStream
|   +-- DataInputStream
|   +-- PushbackInputStream
+-- ObjectInputStream
+-- PipedInputStream
+-- SequenceInputStream
+-- StringBufferInputStream (Deprecated)

OutputStream
+-- ByteArrayOutputStream
+-- FileOutputStream
+-- FilterOutputStream
|   +-- BufferedOutputStream
|   +-- DataOutputStream
|   +-- PrintStream
+-- ObjectOutputStream
+-- PipedOutputStream
```

Character streams

The example below shows the most basic example of how one would try to read a file on disk, however it is also very inefficient since it is calling read, in a while loop, and read will read char by char, from the file, and then print out. However this approach is no where near robust or even correct, depending on the file's encoding, the characters that the calls to System.out.print might show complete garbage.

The reason is simple, when dealing with `UTF-8` or `UTF-16` encoded files, one 'symbol' or 'character' in the file can be encoded in different widths, between 1 and 4 bytes (in case of UTF-8) or between 2 and 4 bytes (in case of UTF-16). When reading with read() and immediately printing, the call to read might read only half of the surrogate pair in the UTF encoding, where as both surrogates have to be printed and stored together. Therefore this is not robust way to work with files, only if the file is ASCII encoded, meaning it only contains 1 byte wide characters is that approach actually going to work

```java
try (FileReader inputFile = new FileReader("file.txt")) {
    int ch = 0;
    // while there are characters to fetch, read, and print the
    // characters when EOF is reached, read() will return -1,
    // terminating the loop
    while( (ch = inputFile.read()) != -1) {
        // ch is of type int - convert it back to char
        // before printing
        System.out.print( (char)ch );
    }
} catch (FileNotFoundException fnfe) {
    // the passed file is not found ...
    System.err.printf("Cannot open the given file %s ", file);
}
catch(IOException ioe) {
    // some IO error occurred when reading the file ...
    System.err.printf("Error when processing file %s... skipping it", file);
}
```

As mentioned above this example is very crude, and it is not recommended to read files like that, ever. It is slow and prone to errors, due to the fact that the encoding of the file is not-known. This example can be modified to use `BufferedReader` instead, which is a type of reader which buffers the calls to read method, and it can read line by line or also stores the actual contents that are being read into a String

Byte streams

Byte streams are used for processing files that do not contain human readable text. For example a Java source file has human readable content, but a .class file does not A .class file is meant for processing by the JVM, hence you must use byte streams to process the .class file. Each file format usually has some sort of number written out as the first byte, this is done to allow programs to easily distinguish what a given file type is immediately instead of having to parse chunks of it. This is also true for some text files as well, in UTF files, there is something called BOM - byte order mark, which is used to indicate the endian-ness (byte-orderf0 of the file and to signal the encoding scheme of the file (UTF-8, 16, 32).

```
// side note, take a look at the explicit casts of bytes below, which is
   because all numeric literals in java are by
// default ints/integer and down-casts are not implicit in java, every down-
   cast has to be manually specified
byte []magicNumber = {(byte) 0xCA, (byte)0xFE, (byte)0xBA, (byte)0xBE};
try (FileInputStream fis = new FileInputStream(fileName)) {
    // magic number is of 4 bytes -
    // use a temporary buffer to read first four bytes
    byte[] u4buffer = new byte[4];
    // read a buffer full (4 bytes here) of data from the file
    if(fis.read(u4buffer) != -1) { // if read was successful
        // the overloaded method equals for two byte arrays
        // checks for equality of contents
        if(Arrays.equals(magicNumber, u4buffer)) {
            System.out.printf("The magic number for passed file %s matches
                that of a .class file", fileName);
        }
        else {
            System.out.printf("The magic number for passed file %s does not
                match that of a .class file", fileName);
        }
    }
} catch(FileNotFoundException fnfe) {
    System.err.println("file does not exist with the given file name ");
} catch(IOException ioe) {
    System.err.println("an I/O error occurred while processing the file");
}
```

In the example above a read of the binary file is shown where the first 4 bytes are read and compared to the special magic number, in this case this is for .class files, which have the following 4 bytes as a magic number put at the very start of the file - `"0xCAFEBABE"`

To write data out to files one can simply use the `OutputStream` class, the process is pretty much the inverse of reading data from a binary file using the `InputStream` class

Data streams

There are some special types of byte Streams, which provide ease of use for writing primitive types in java. These are the so called Data streams. These streams provide a general API to write out primitive types, such as Byte, Short, Int, Long etc. These are usually represented as multi-byte values in the java run-time, meaning that to write write them out into a file, and then read them special rules or assumptions have to be employed. That is why they are not written out in a human readable format, the actual bytes of the internal representation is dumped in the file. While it is possible for these values to be written out in plain text, it would be much harder to be read back and parsed, especially the float or double (floating point type numbers)

```
try (DataOutputStream dos = new DataOutputStream(new FileOutputStream("temp.
   data"))) {
    // write values 1 to 10 as byte, short, int, long, float and double
    // omitting boolean type because an int value cannot
    // be converted to boolean
    for(int i = 0; i < 10; i++) {
        dos.writeByte(i);
        dos.writeShort(i);
        dos.writeInt(i);
        dos.writeLong(i);
        dos.writeFloat(i);
        dos.writeDouble(i);
    }
}

try (DataInputStream dis = new DataInputStream(new FileInputStream("temp.data
   "))) {
    // the order of values to read is byte, short, int, long, float and
    // double since we've written from 0 to 10,
    // the for loop has to run 10 times
    for(int i = 0; i < 10; i++) {
        // %d is for printing byte, short, int or long
        // %f, %g, or %e is for printing float or double
        // %n is for printing newline
        System.out.printf("%d %d %d %d %g %g %n",
            dis.readByte(),
            dis.readShort(),
            dis.readInt(),
            dis.readLong(),
            dis.readFloat(),
            dis.readDouble());
    }
}
```

In the example above, which has the rest of the code abridged, for brevity, a file is populated with all the numbers from 1 to 10, and their primitive representation, so for each number from 1 to 10, the byte, short, int, long, float and double values of this number `i` is written out to the file, in that order. The next part of the code reads the same values back, note that the order of reading the values is the same, as the one used to write out the numbers, the number of items being read from the file is 10 too.

Why does it matter ? Well the byte ordering is strict, each primitive value has a certain size, bytes are usually 1 byte long, shorts are 2, int is 4, long 8, and so on. Meaning that to read them back correctly the same order has to be retained, otherwise garbage values would be read back if the reading part of the code starts reading double or float first for example.

Object streams

These types of streams are similar to data streams, however they are meant to write out entire java Objects into files, Java in general has quite useful serializtion rules which allows objects instances to be dumped to and read from a file, relatively easily.

```
Map<String, String> presidentsOfUS = new HashMap<>();
presidentsOfUS.put("Barack Obama", "2009 to --, Democratic Party, 56th term")
   ;
presidentsOfUS.put("George W. Bush", "2001 to 2009, Republican Party, 54th
   and 55th terms");
presidentsOfUS.put("Bill Clinton", "1993 to 2001, Democratic Party, 52nd and
   53rd terms");

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("
   object.data"))) {
    oos.writeObject(presidentsOfUS);
} catch(FileNotFoundException fnfe) {
    System.err.println("cannot create a file with the given file name ");
} catch(IOException ioe) {
    System.err.println("an I/O error occurred while processing the file");
}

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("
   object.data"))) {
    Object obj = ois.readObject();
    if(obj != null && obj instanceof Map) {
        Map<?, ?> presidents = (Map<?, ?>) obj;
        System.out.println("President name \t Description");
        for(Map.Entry<?, ?> president : presidents.entrySet()) {
            System.out.printf("%s \t %s %n", president.getKey(),
                president.getValue());
        }
    }
} catch(FileNotFoundException fnfe) {
    System.err.println("cannot create a file with the given file name ");
} catch(IOException ioe) {
    System.err.println("an I/O error occurred while processing the file");
} catch(ClassNotFoundException cnfe) {
    System.err.println("cannot recognize the class of the object - is the
       file corrupted?");
}
```

The example above is similar to the example using the Data stream, in this case one object instance, of type Map, is written out to a file, and then read back. The contents of the map are persistent in the file, and are also serialized along with the rest of the properties.

The serialization process converts contents of the objects in memory with the description of the contents - metadata. When the object has references to other objects the serialization mechanism also includes them as part of the serialized bytes.

When serializing data Java includes not only the object data itself, but also metadata about the class type, which allows Java to reconstruct the exact type of the object during deserialization. This includes objects like Map, List, Set or even custom user created objects.

When an object is serialized the other important property that is included as well is the `serialVersionUID` this is like a crude method to version the class, that way when an object is written out to a file, then read out, the version values in `serialVersionUID` are compared with the ones in the class definition, if there is mismatch the reading will not succeed.

The `serialVersionUID` is a unique identifier that helps ensure that a serialized object is compatible with the class definition it is being `deserialized` into. This is important for maintaining backward compatibility when you change the class over time.While the `Serializable` interface is required for an object to be serialized, the `serialVersionUID` field helps control the versioning and compatibility during serialization/deserialization. Without this `UID`, Java generates it automatically based on the class structure (which can lead to issues when the class changes).

Imagine that a user class was written out to a file with its initial version the very first time the class was declared, some objects of it were serialized, however over time this class definition has changed and new fields were added or old ones removed, there is no way for Java to know how to read this new modified version of the user class from the file. In such situations, it is possible to customize the way an instance is read from the file on a user level

There are special private methods, which are not overriding or overloading any methods existing in the Object class, they are special ones, which are only looked up by the Java serialization pipeline, they are called - `readObject(ObjectInputStream in)` and `writeObject(ObjectInputStream out)`. They give the user the ability to manually write out an object, or read it for that matter, in the example above, the reading/writing can then be tied to the changes in the class, or the actual process of writing/reading can be done in a custom flexible way to handle future changes in the class's declaration

```java
class CustomUserClass implements Serializable {

    // some user defined fields

    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject(); // Write default fields
        out.writeInt(customField); // Write custom field
    }

    private void readObject(ObjectInputStream in) throws IOException,
       ClassNotFoundException {
        in.defaultReadObject(); // Read default fields
        customField = in.readInt(); // Read custom field
    }
}
```

Summary

Read and write data from the console

- The public static fields in, out, and err in java.lang.System class respectively represent the standard input, output and error streams. System.in is of type java.io.InputStream and System.out and System.err are of type java.io.PrintStream.
- You can redirect standard streams by calling the methods `System.setIn`, `System.setOut` and `System .setError`.
- You can obtain a reference to the console using the System.`console()` method; if the `JVM` is not associated with any console, this method will fail and return null.
- Many methods are provided in Console-support formatted I/O. You can use the `printf()` and `format ()` methods available in the Console class to print formatted text; the overloaded `readLine()` and `readPassword()` methods take format strings as arguments.
- The template of format specifiers is: %[flags][width][.precision]datatype_specifier Each format specifier starts with the % sign, followed by flags, width, and precision information, and ending with a data type specifier. In the format string, the flags, width, and precision information are optional but the % sign and data type specifier are mandatory.
- Use the `readPassword()` method for reading secure strings such as passwords. It is recommended to use Array's `fill()` method to "empty" the password read into the character array (to avoid malicious access to the typed passwords).

Use the java.io package classes

- The java.io package has classes supporting both character streams and byte streams.
- You can use character streams for text-based I/O. Byte streams are used for databased I/O.
- Character streams for reading and writing are called readers and writers respectively (represented by the abstract classes of Reader and Writer).
- Byte streams for reading and writing are called input streams and output streams respectively (represented by the abstract classes of `InputStream` and `OutputStream`).
- You should only use character streams for processing text files (or human-readable files), and byte streams for data files. If you try using one type of stream instead of another, your program won't work as you would expect; even if it works by chance, you'll get nasty bugs. So don't mix up streams, and use the right stream for a given task at hand.
- For both byte and character streams, you can use buffering. The buffer classes are provided as wrapper classes for the underlying streams. Using buffering will speed up the I/O when performing bulk I/O operations.
- For processing data with primitive data types and strings, you can use data streams.
- You can use object streams (classes `ObjectInputStream` and `ObjectOutputStream`) for reading and writing objects in memory to files and vice versa.