Lab 10: JAVA I/O

Objective(s):

1. Overview of I/O & I/O Streams
2. JAVA I/O Files
3. JAVA I/O Streams

1: Overview of I/O & I/O Streams

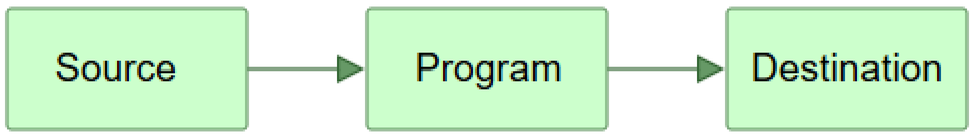
Input/Output – Source/Destination

The terms "input" and "output" can sometimes be a bit confusing. The input of one part of an application is often the output of another. Is an `OutputStream` a stream where output is written to, or output comes out from (for you to read)? After all, an `InputStream` outputs its data to the reading program, doesn't it?

Java's IO package mostly concerns itself with the reading of raw data from a source and writing of raw data to a destination. The most typical sources and destinations of data are these:

- Files
- Pipes
- Network Connections
- In-memory Buffers (e.g. arrays)
- System.in, System.out, System.error

The diagram below illustrates the principle of a program reading data from a source and writing it to some destination:

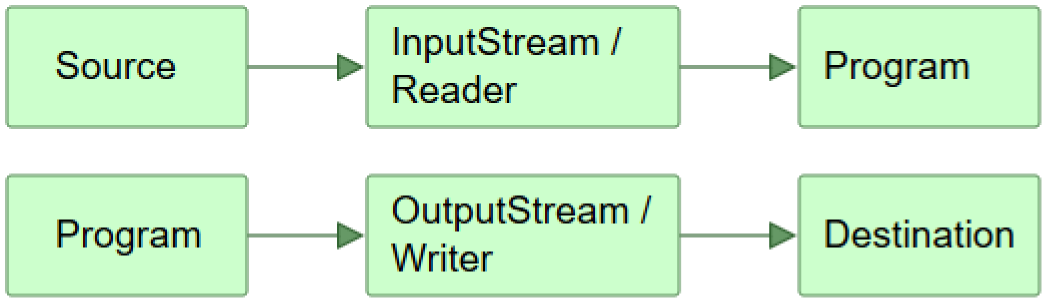


Streams

IO Streams are a core concept in Java IO. In Java, streams are the sequence of data that are read from the source and written to the destination. A stream is a conceptually endless flow of data. You can either read from a stream or write to a stream. A stream is connected to a data source or a data destination. Streams in Java IO can be either byte based (reading and writing bytes) or character based (reading and writing characters).

The InputStream, OutputStream, Reader and Writer

A program that needs to read data from some source needs an `InputStream` or a Reader. A program that needs to write data to some destination needs an `OutputStream` or a Writer. This is also illustrated in the diagram below:



An `InputStream` or Reader is linked to a source of data. An `OutputStream` or Writer is linked to a destination of data.

Java IO Purposes and Features

Java IO contains many subclasses of the `InputStream`, `OutputStream`, `Reader` and `Writer` classes. The reason is, that all of these subclasses are addressing various different purposes. That is why there are so many different classes. The purposes addressed are summarized below:

- File Access
- Network Access
- Internal Memory Buffer Access
- Inter-Thread Communication (Pipes)
- Buffering
- Filtering
- Parsing
- Reading and Writing Text (Readers / Writers)
- Reading and Writing Primitive Data (long, int etc.)
- Reading and Writing Objects

These purposes are nice to know about when reading through the Java IO classes. They make it somewhat easier to understand what the classes are targeting.

Java IO Class Overview Table

Having discussed sources, destinations, input, output and the various IO purposes targeted by the Java IO classes, here is a table listing most (if not all) Java IO classes divided by input, output, being byte based or character based, and any more specific purpose they may be addressing, like buffering, parsing etc.

	Byte Based		Character Based	
	Input	Output	Input	Output
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter
Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
Strings			StringReader	StringWriter
Data	DataInputStream	DataOutputStream		
Data - Formatted		PrintStream		PrintWriter
Objects	ObjectInputStream	ObjectOutputStream		
Utilities	SequenceInputStream			

2: Java I/O Files

Files are a common source or destination of data in Java applications. Therefore this text will give you a brief overview of working with files in Java. It is not the intention to explain every technique in detail here, but rather to provide you with enough knowledge to decide on a file access method.

I/O File Classes

The Java IO API contains the following classes which are relevant to working with files in Java:

- File
- RandomAccessFile
- FileInputStream
- FileReader
- FileOutputStream
- FileWriter

Reading Files via Java IO

If you need to read a file from one end to the other you can use a `FileInputStream` or a `FileReader` depending on whether you want to read the file as binary or textual data. These two classes lets you read a file one byte or character at a time from the start to the end of the file, or read the bytes into an array of `byte` or `char`, again from start towards the end of the file. You don't have to read the whole file, but you can only read bytes and chars in the sequence they are stored in the file.

If you need to jump around the file and read only parts of it from here and there, you can use a `RandomAccessFile`.

Writing File via Java IO

If you need to write a file from one end to the other you can use a `FileOutputStream` or a `FileWriter` depending on whether you need to write binary data or characters. You can write a byte or character at a time from the beginning to the end of the file, or write arrays of `byte` and `char`. Data is stored sequentially in the file in the order they are written.

If you need to skip around a file and write to it in various places, for instance appending to the end of the file, you can use a `RandomAccessFile`.

Random Access to Files via Java IO

As I have already mentioned, you can get random access to files with Java IO via the `RandomAccessFile` class.

Random access doesn't mean that you read or write from truly random places. It just means that you can skip around the file and read from or write to it at the same time in any way you want. This is what is meant by "random" - that the next byte read is not determined by the previous byte read. No particular access sequence is enforced. You can access the bytes in the file "at random" - arbitrarily. This makes it possible to overwrite parts of an existing file, to append to it, delete from it, and of course read from the file from wherever you need to read from it.

File and Directory Info Access

Sometimes you may need access to information about a file rather than its content. For instance, if you need to know the file size or the file attributes of a file. The same may be true for a directory. For instance, you may want to get a list of all files in a given directory. Both file and directory information is available via the `File` class.

3: Java I/O Streams

Java IO streams are flows of data you can either read from, or write to. As mentioned in the Java IO Overview, streams are typically connected to a data source, or data destination, like a file or network connection.

A stream has no concept of an index of the read or written data, like an array does. Nor can you typically move forth and back in a stream, like you can in an array, or in a file using `RandomAccessFile`. A stream is just a continuous flow of data.

Some stream implementations like the `PushbackInputStream` allows you to push data back into the stream, to be re-read again later. But you can only push back a limited amount of data, and you cannot traverse the data at will, like you can with an array. Data can only be accessed sequentially.

Java IO streams are typically either byte based or character based. The streams that are byte based are typically called something with "stream", like `InputStream` or `OutputStream`. These streams read and write a raw byte at a time, with the exception of the `DataInputStream` and `DataOutputStream` which can also read and write `int`, `long`, `float` and `double` values.

The streams that are character based are typically called something with "Reader" or "Writer". The character based streams can read / write characters (like Latin1 or UNICODE characters). See the text Java Readers and Writers for more information about character based input and output.

InputStream

The class `java.io.InputStream` is the base class for all Java IO input streams. If you are writing a component that needs to read input from a stream, try to make our component depend on an `InputStream`, rather than any of it's subclasses (e.g. `FileInputStream`). Doing so makes your code able to work with all types of input streams, instead of only the concrete subclass.

Depending on `InputStream` only isn't always possible, though. If you need to be able to push back data into the stream, you will have to depend on a `PushbackInputStream` - meaning your stream variable will be of this type. Otherwise your code will not be able to call the `unread()` method on the `PushbackInputStream`.

You typically read data from an `InputStream` by calling the `read()` method. The `read()` method returns a `int` containing the byte value of the byte read. If there is no more data to be read, the `read()` method typically returns -1;

Here is a simple example:

```
InputStream input = new FileInputStream("c:\\data\\input-file.txt");

int data = input.read();

while(data != -1){
  data = input.read();
}
```

OutputStream

The class `java.io.OutputStream` is the base class of all Java IO output streams. If you are writing a component that needs to write output to a stream, try to make sure that component depends on an `OutputStream` and not one of its subclasses.

Here is a simple example pushing some data out to a file:

```
OutputStream output = new FileOutputStream("c:\\data\\output-file.txt");
output.write("Hello World".getBytes());
output.close();
```

Combining Streams

You can combine streams into chains to achieve more advanced input and output operations. For instance, reading every byte one at a time from a file is slow. It is faster to read a larger block of data from the disk and then iterate through that block byte for byte afterwards. To achieve buffering you can wrap your `InputStream` in an `BufferedInputStream`. Here is an example:

```
InputStream input = new BufferedInputStream(
                        new FileInputStream("c:\\data\\input-file.txt"));

...
```

Buffering can also be applied to `OutputStream`'s thereby batching the writes to disk (or the underlying stream) up in larger chunks. That provides faster output too. This is done with a `BufferedOutputStream`.

Buffering is just one of the effects you can achieve by combining streams. You can also wrap your `InputStream` in a `PushbackStream`. That way you can push data back into the stream to be re-read later. This is sometimes handy during parsing. Or, you can combine two `InputStream`s into one using the `SequenceInputStream`

There are several other effects that can be achieved by combining input and output streams into chains. You can even write your own stream classes to wrap the standard stream classes that comes with Java. That way you can create your own effects or filters.

For further details

http://tutorials.jenkov.com/java-io/index.html

https://www.programiz.com/java-programming/io-streams

Lab Tasks:

Exercise 1

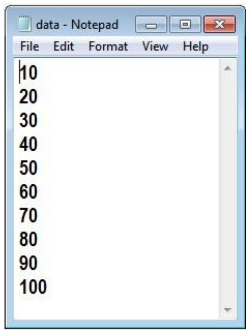
Create following programs.

1. Write a program that get a list of files & directories from a given directory.
2. Write a program to get specific files by extensions from a specified folder.
3. Write a program to check if a file or directory specified by pathname exists or not.
4. Write a program to check if given pathname is a directory or a file.
5. Write a program to delete a given file from a directory.
6. Write program to read input from java console (Don’t use Scanner).
7. Write a program to append text to an existing file.
8. Write a program to read first 3 lines from a file.
9. Write a program to find the longest word in a text file.
10. Write a program to find the largest odd integer number in a text file.

Exercise 2

Write a program to read numbers from a file “data.txt” and calculate their sum and write a sum at the end of a file.

You can create your own “data.txt” (as shown in figure)



```
10
20
30
40
50
60
70
80
90
100
```

Note: If your program does not find your file, use the `getAbsolutePath()` method of File class to find the expected path of your file in your exception handling code.

Create following separate programs.

1. Read data using `Scanner` & write with `PrintWriter`.
2. Read data using `BufferedReader` & write with `BufferedWriter`.

Exercise 3

Write a program to read a line at a time from an input file & prints it to an output file with a line number.

You can create your own file “sample.txt” and add your favorite poem.

You need to use `Scanner, File` & `PrintWriter to complete your program.`

Notes:

- If the file does not exist, it should be created automatically
- If the file exists, it will be overwritten!
- If there is no writing access to the file or its folder, a `SecurityException` is thrown. It is always nice to handle it.
- ALWAYS make sure to close your files! Otherwise your output can be lost!

END