PER SCHOLAS

Introduction to Java Programming

Basic Input & Output including java.util

Java IO

- Students should be able to understand:
  - Introduction to I/O
  - FileReader and Closeable
  - BufferedReader
  - Load Big Location and Exits Files
  - Byte Streams
  - Java NIO
  - Writing Objects With Java NIO
  - Reading and Writing with Java NIO
  - Writing Binary Files with Java NIO
  - Reading Files with NIO
  - Absolute and Relative Reads
  - Chained Put Methods

Java IO

- Java IO is an API that comes with Java which is targeted at reading and writing data (input and output). Most applications need to process some input and produce some output based on that input. For instance, read data from a file or over network, and write to a file or write a response back over the network.
- The Java IO API is located in the Java IO package (java.io). If you look at the Java IO classes in the java.io package the vast amount of choices can be rather confusing. What is the purpose of all these classes? Which one should you choose for a given task? How do you create your own classes to plugin? etc. The purpose of this tutorial is to try to give you an overview of how all these classes are grouped, and the purpose behind them, so you don't have to wonder whether you chose the right class, or whether a class already exists for your purpose.

Java IO

Java NIO - The Alternative IO API

- Java also contains another IO API called Java NIO. It contains classes that does much of the same as the Java IO and Java Networking APIs, but Java NIO can work in non-blocking mode. Non-blocking IO can in some situations give a big performance boost over blocking IO.

Java IO

Java NIO - The Alternative IO API

- Java also contains another IO API called Java NIO. It contains classes that does much of the same as the Java IO and Java Networking APIs, but Java NIO can work in non-blocking mode. Non-blocking IO can in some situations give a big performance boost over blocking IO.

Java IO

Input and Output - Source and Destination

- 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? Personally, I found this a bit confusing back in the day when I first started out learning about Java IO.
- 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

Java IO

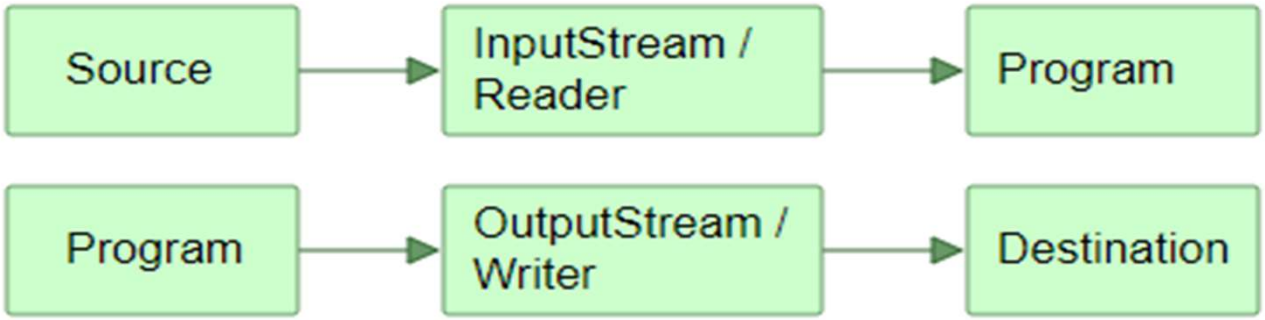
Streams

- **IO Streams:** are a core concept in Java IO. 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).

Java IO

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

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

Java IO

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

Java IO

Java IO Class Overview Table

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			

Java IO

Read and Write Example - Read one character at a time

```
 9 public class IOEx1 {
10     public static void main(String[] args) throws IOException {
11         FileReader inputStream = null;
12         FileWriter outputStream = null;
13         File readin = new File("C:\\Users\\PSAdmin\\Documents\\sample.txt");
14         File writeTo = new File("C:\\Users\\PSAdmin\\Documents\\sampleTo.txt");
15         try {
16             inputStream = new FileReader(readin);
17             outputStream = new FileWriter(writeTo);
18             int c;
19             while ((c = inputStream.read()) != -1) {
20                 outputStream.write(c);
21             }
22         } finally {
23             if (inputStream != null) {
24                 inputStream.close();
25             }
26             if (outputStream != null) {
27                 outputStream.close();
28             }
29         }
30     }
31 }
```

It uses an int variable, "c", to read to and write from. The int variable holds a character value. Using an int as the return type allows read() to use -1 to indicate that it has reached the end of the stream.

Java IO

Always Close Your File

- Closing a stream when it's no longer needed is very important. That is why CopyCharacters uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.
- One possible error is that CopyCharacters was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyCharacters makes sure that each stream variable contains an object reference before invoking close.

Java IO

Read and Write Example - Read one line at a time

- Character I/O is usually processed in units longer than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.
- Let's modify the copyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, BufferedReader and PrintWriter.
- Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.
- The Java PrintWriter class (java.io.PrintWriter) enables you to write formatted data to an underlying Writer. For instance, writing int, long and other primitive data formatted as text, rather than as their byte values.

Java IO

Read and Write Example - Read one line at a time

- Invoking readLine returns a line of text. This line is then output using PrintWriter's println method, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

Java IO

Read and Write Example - Read one line at a time

```
10 public class IOEx2 {
11     public static void main(String[] args) throws IOException {
12         FileReader inputStream = null; FileWriter outputStream = null;
13         BufferedReader bufferStream = null; PrintWriter printStream = null;
14         File readin = new File("C:\\Users\\PSAdmin\\Documents\\sample.txt");
15         File writeTo = new File("C:\\Users\\PSAdmin\\Documents\\sampleTo.txt");
16         try {
17             inputStream = new FileReader(readin); outputStream = new FileWriter(writeTo);
18             bufferStream = new BufferedReader(inputStream);
19             printStream = new PrintWriter(outputStream);
20             String l;
21             while ((l = bufferStream.readLine()) != null) {
22                 printStream.println(l);
23             }
24         } finally {
25             if (inputStream != null) {
26                 inputStream.close();
27             } if (outputStream != null) {
28                 outputStream.close();
29             } if (bufferStream != null) {
30                 bufferStream.close();
31             } if (printStream != null) {
32                 printStream.close(); }
33         }
34     }
35 }
```

Java IO

Scanner

- Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.
  - To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
  - To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
  - To read strings, we use nextLine().
  - To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

Java IO

Scanner Example

```
 9 public class ScannerEx1
10 {
11     public static void main(String[] args) throws IOException {
12         Scanner s = null;
13         File readin = new File("C:\\Users\\PSAdmin\\Documents\\sample.txt");
14         BufferedReader bufferStream = null; FileReader inputStream = null;
15         try {
16             inputStream = new FileReader(readin);
17             bufferStream = new BufferedReader(inputStream);
18             s = new Scanner(bufferStream);
19
20             while (s.hasNext()) {
21                 System.out.println(s.next());
22             }
23         } finally {
24             if (s != null) {
25                 s.close();
26             }
27         }
28     }
29 }
```

Notice that ScannerEX1 invokes Scanner's close method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

Java IO

Scanner Example

- Scanner also supports tokens for all of the Java language's primitive types (except for char), as well as BigInteger and BigDecimal. Also, numeric values can use thousands separators. Thus, Scanner correctly reads the string "32,767" as representing an integer value.

Java IO

Scanner Example

```
 9 public class ScannerEx2
10 {
11     public static void main(String[] args) throws IOException {
12         Scanner s = null;
13         File readin = new File("C:\\Users\\PSAdmin\\Documents\\sample.txt");
14         BufferedReader bufferStream = null; FileReader inputStream = null;
15         Double sum = 0D;
16         try {
17             inputStream = new FileReader(readin);
18             bufferStream = new BufferedReader(inputStream);
19             s = new Scanner(bufferStream);
20             while (s.hasNext()) {
21                 if(s.hasNextDouble()) {
22                     sum += s.nextDouble();
23                 }else {
24                     System.out.println(s.next());
25                 }
26             }
27         } finally {
28             if (s != null) {
29                 s.close();
30             }
31         }
32         System.out.println("Total Sum is " + sum);
33     }
34 }
```

Java IO

Write a Java program to get a list of all file/directory names inside the Documents folder.

```
5  public class ListOfFileNames
6  {
7      public static void main(String a[])
8      {
9          File file = new File("C:\\Users\\PSAdmin\\Documents");
10         String[] fileList = file.list();
11         for(String name:fileList){
12             System.out.println(name);
13         }
14     }
15 }
```

Java IO

Write a Java program to get specific files by extensions from a specified folder.

```
6  public class ListOfFileWithSpecificExtension
7  {
8      public static void main(String a[]){
9          File file = new File("C:\\Users\\PSAdmin\\Documents");
10             String[] list = file.list();
11
12         for(String f:list){
13             if(f.toLowerCase().endsWith(".txt")) {
14                 System.out.println(f);
15             }
16         }
17     }
18 }
```

Java IO

Write Java program to read input from java console.

```
7  public class ConsoleReader
8  {
9          public static void main(String[] args) throws IOException
10         {
11           BufferedReader R = new BufferedReader(new InputStreamReader(System.in));
12           System.out.print("Input your name: ");
13           String name = R.readLine();
14           System.out.println("Your name is: " + name);
15         }
16 }
```

Java IO

Streaming Through the File

- iterate through all the lines in the file – allowing for processing of each line – without keeping references to them – and in conclusion, without keeping them in memory

```
 8 public class StreamEx1 {
 9     public static void main(String[] args) throws IOException {
10         FileInputStream inputStream = null; Scanner sc = null;
11         try {
12 inputStream = new FileInputStream("C:\\Users\\PSAdmin\\Documents\\sample.txt");
13             sc = new Scanner(inputStream, "UTF-8");
14             while (sc.hasNextLine()) {
15                 String line = sc.nextLine();
16                  System.out.println(line);
17             }
18             // note that Scanner suppresses exceptions
19             if (sc.ioException() != null) {
20                 throw sc.ioException();
21             }
22         } finally {
23             if (inputStream != null) {
24                 inputStream.close();
25             }
26             if (sc != null) {
27                 sc.close();
28             }
29         }
30     }
31 }
```

Java IO

File Exist

- If your application needs to create temporary files for some application logic or unit testing, then you would like to make sure that these temporary files are deleted when they are not needed.

```
6  public class DeleteFileOnExit {
7      public static void main(String[] args) {
8         File temp;
9         try
10        {
11           temp = File.createTempFile("C:\\Users\\PSAdmin\\Documents\\myTempFile", ".txt");
12           System.out.println("Temp file created : " + temp.getAbsolutePath());
13
14           //temp.delete(); //For deleting immediately
15
16           temp.deleteOnExit(); //Delete on runtime exit
17
18           System.out.println("Temp file exists : " + temp.exists());
19        } catch (IOException e)
20        {
21           e.printStackTrace();
22        }
23     }
24 }
```