**Stream**

All fundamental i/o operation in java is based on streams. A stream represents a flow of data, or channel of communication with a writer at one end and reader at other end. when you are working with terminal input and output , reading or writing files, or communicating through sockets in java,you are using a stream of one type to another.

No matter the information is comming from or going to and no matter what type of data is being read or written, the algorithm for reading and writing data is pretty much always the same.

**Reading**

open a stream

while more information

read information

close the stream

**Writing**

open a stream

while more information

write information

close the stream The java.io package contains a collection of stream classes that supports above mentioned algorithms.

**Classification of Stream classes:**

* Byte stream : provides support for handeling i/o operations on bytes.
* Character stream : provides support for managing i/o operations on characters.

It is to be remembered that streams in java are one-way streets.

The java.io input and output classes represent the ends of a simple stream.

For bi-directional conversations, we have to use one of each type of streams.

**Object Stream**

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Using a fixed-length record format is a good choice if you need to store data of the same type. However, objects that you create in an object-oriented program are rarely all of the same type. For example, you may have an array called staff that is nominally an array of Employee records but contains objects that are actually instances of a child class such as Manager. If we want to save files that contain this kind of information, we must first save the type of each object and then the data that defines the current state of the object. When we read this information back from a file, we must:

* Read the object type;
* Create a blank object of that type;
* Fill it with the data that we stored in the file.

It is entirely possible (if very tedious) to do this by hand. Sun Microsystems developed a powerful mechanism that allows this to be done with much less effort. This mechanism is called object serialization, almost completely automates what was previously a very tedious process.

**Zip File Streams**

ZipInputStream is a Java class that implements an input stream filter for reading files in the ZIP file format. It has support for both compressed and uncompressed entries. ZIP is an archive file format that supports lossless data compression. A ZIP file may contain one or more files or directories that may have been compressed.

**FileInputStream**

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available. Following constructor takes a file name as a string to create an input stream object to read the file: InputStream f = new FileInputStream("C:/java/hello"); Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows: File f = new File("C:/java/hello"); InputStream f = new FileInputStream(f); Once we have InputStream object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

**Helper methods:**

* public void close() throws IOException{ } : This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
* protected void finalize() throws IOException { } : This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
* public int read(int r) throws IOException{ } : This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
* public int read(byte[] r) throws IOException{ } : This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
* public int available() throws IOException{} : Gives the number of bytes that can be read from this file input stream. Returns an int.

**FileOutputStream**

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output. Following constructor takes a file name as a string to create an input stream object to write the file: OutputStream f = new FileOutputStream ("C:/java/hello") Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File method as follows: File f = new File("C:/java/hello"); OutputStream f = new FileOutputStream (f); Once we have OutputStream object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

**Helper methods:**

* public void close() throws IOException{ } : This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
* protected void finalize() throws IOException { } : This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
* public void write(int w) throws IOException{ } : This methods writes the specified byte to the output stream.
* public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.