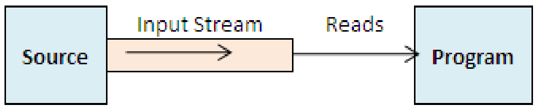
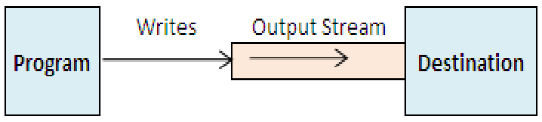
Introduction:

A file is a collection of related records placed in a particular area on the disk. A record is composed of several fields and a field is a group of characters. A flow of data is referred to as a **Data Stream**. A **stream** is an ordered sequence of bytes that has a source (input stream) or a destination (output stream). A stream is a logical device that represents the flow of a sequence of characters. Programs can get inputs from a data source by reading a sequence of characters from the input stream. Similarly, programs can produce outputs by writing a sequence of characters on to an output stream. A Stream can be associated with a file. In java language, all data is written and read using the streams. A stream is a path travelled by the data in a program. A program opens a stream on an information source to bring the information from a file or memory or a socket. The information is read serially. Similarly the program can send information to an external device by opening stream to a destination and writing the information serially.

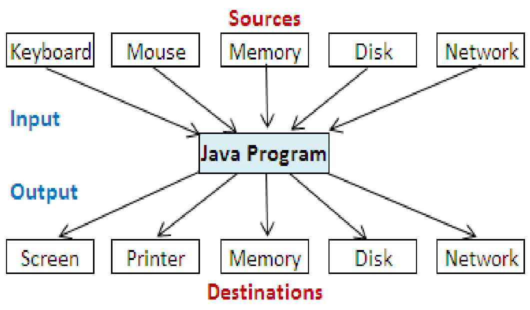
A stream in java is a path along which data flows.



Reading data into a program



Writing data into a program

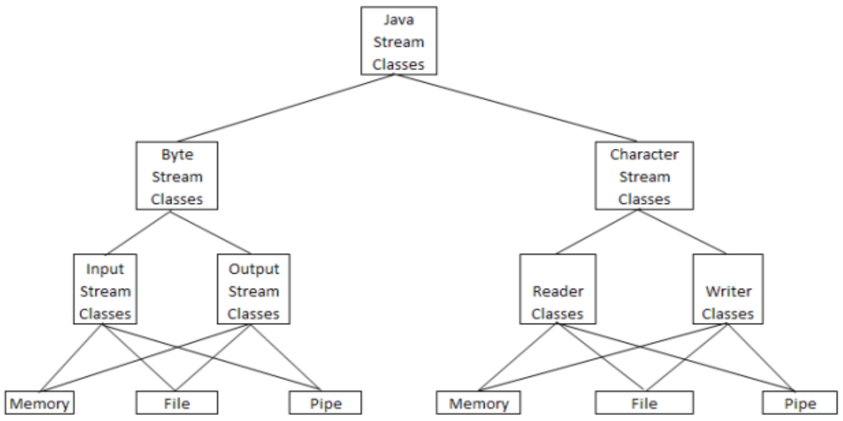


Relationship of java program with I/O devices

There are three types of predefined streams to be used in java are:
System.in: This is the standard input stream and always open and ready to supply input data. This stream is responsive to keyboard.
System.out: This is the standard output stream for display of values. This stream is open and ready to accept output data.
System.err: This is standard error stream used for display or error messages.

Java I/O:

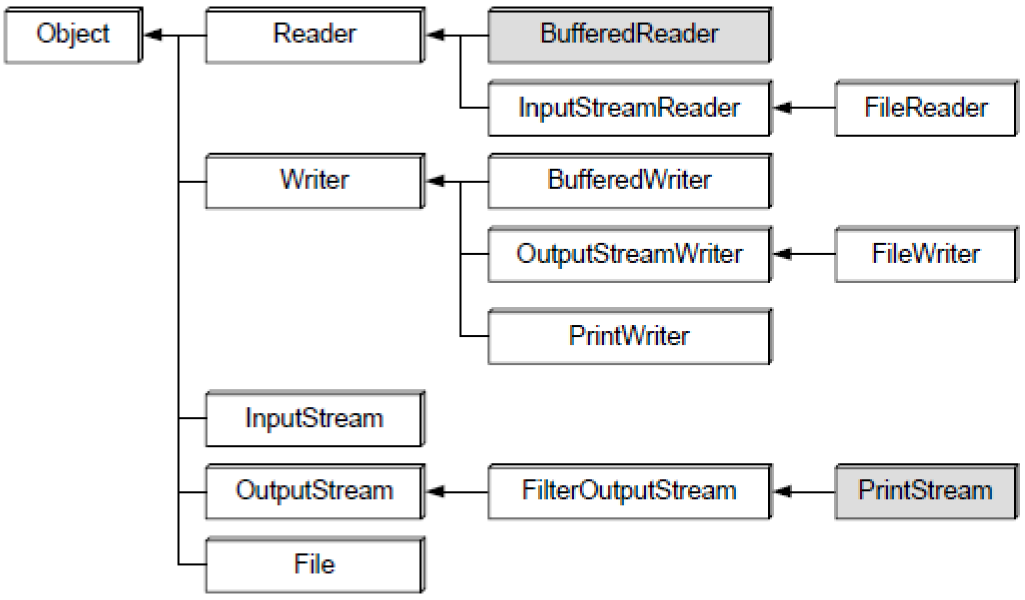
Streams are represented in java as classes. The java.io package defines a collection of stream classes that support input and output (reading and writing). Stream support many kinds of data including a simple bytes, primitive data types, localized characters, and objects. Some Streams simply pass on data, other manipulation and transform the data in useful ways. To use these classes, a program needs to import the java.io package: import java.io.*;



Classification of java stream classes

Steams are classified into two types:

1. **Byte Stream:** This stream is mainly used for binary data handling in which it consider that all the records are maintained in a binary files and in the byte format stream are stored in file.
2. **Character Stream:** These streams are used for handling and dealing with character streams. The data is entered into files which are storing the data in a character format.

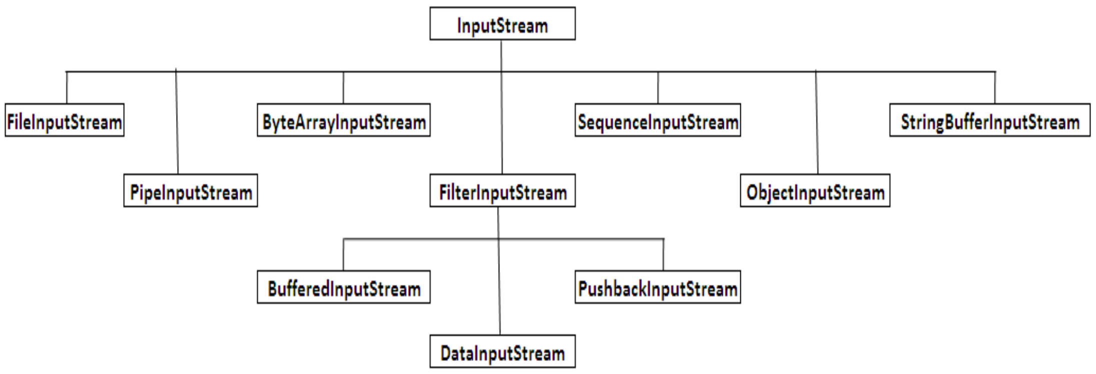


Class hierarchy for stream classes

Byte streams class:

It handles I/O operations on bytes. InputStream and OutputStream classes are operated on bytes for reading and writing, respectively. Byte streams are used in a program to read and write 8-bit bytes. InputStream and OutputStream are the abstract super classes of all byte streams that have a sequential nature. The stream is unidirectional; they can transmit bytes in only one direction. InputStream and OutputStream provide the Application program Interface (API) and partial implementation for input streams (streams that read bytes) and output streams (streams that write bytes).

Input Stream Classes: java.io.InputStream is an abstract class that contains the basic methods for reading raw bytes of data from a stream. The InputStream class defines methods for performing the input functions like: reading bytes, closing streams, marking positions in streams, skipping ahead in a stream and finding the number of bytes in a stream.



InputStream class hierarchy

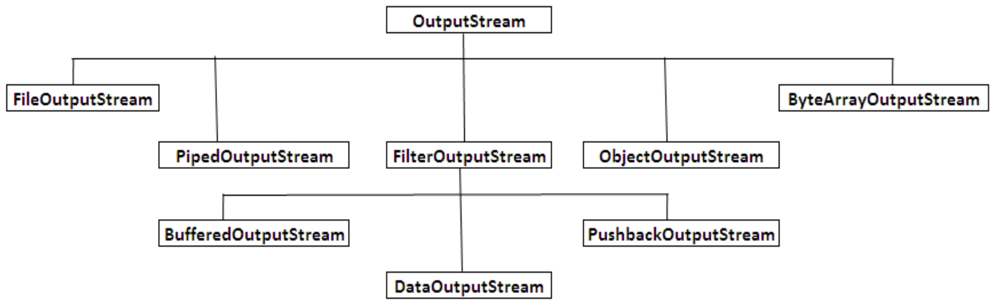
The input stream classes are indicated as below:

Classes	Description
ByteArrayInputStream	In this case the data is read from a byte array which must be specified.
FileInputStream	In this case the data is read as bytes from a file.
FilterInputStream	This is super class of all input stream filter.
BufferedInputStream	A filter which buffered is bytes read from input stream.
DataInputStream	A filter which allows binary representation of java variables.
PipedInputStream	This reads bytes from a pipedoutputstream to each it is connected.
SequenceInputStream	This allows bytes to be read sequentially from two or more input streams.

Input stream class methods:

Methods	Description
int read ()	Returns an integer representation of next available byte of input.-1 is returned at the stream end.
int read (byte buffer[ ])	Read up to buffer.length bytes into buffer & returns actual number of bytes those are read. At the end returns –1.
int read(byte buffer[ ], int offset, int numbytes)	Attempts to read up to numbytes bytes into buffer starting at buffer[offset]. Returns actual number of bytes that are read. At the end returns –1.
void close()-	to close the input stream
void mark(int numbytes)	Places a mark at current point in input stream and remain valid till numbers of bytes are read.
void reset()	Resets pointer to previously set mark/ goes back to stream beginning.
long skip(long numbytes)	Skips number of bytes.
int available()	Returns number of bytes currently available for reading.

Output Stream Classes: The java.io.OutputStream class sends raw bytes of data to a target such as the console or a network server. Like InputStream, OutputStream is an abstract class. The OutputStream includes methods that perform operations like: writing bytes, closing streams, flushing streams etc.



OutputStream class hierarchy

The output stream classes are indicated as below:

Class Name	Description
ByteArrayOutputStream	In this case the data is written from a byte array which must be specified.
FileOutputStream	In this case the data is written as bytes from a file.
FilterOutputStream	This is super class of all Output stream filter.
BufferedOutputStream	A filter which buffered is bytes written from input stream.
DataOutputStream	A filter which allows binary representation of java variables.
PipedOutputStream	This writes bytes from a pipedinputstream to each it is connected.

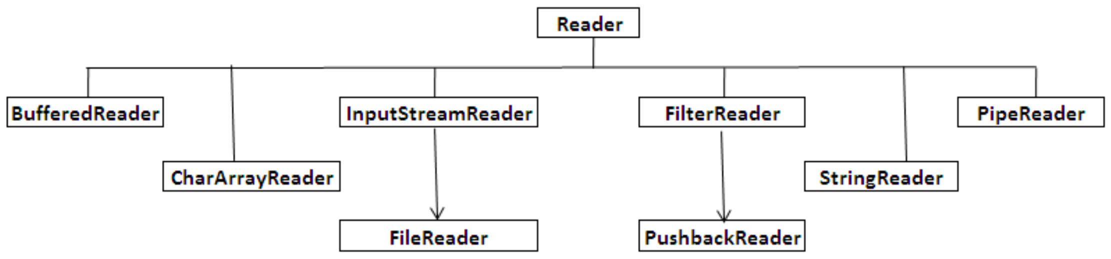
Methods defines by the OutputStream class are

Methods	Description
void close()	To close the OutputStream
void write (int b)	Writes a single byte to an output stream.
void write(byte buffer[ ])	Writes a complete array of bytes to an output stream.
void write (byte buffer[ ], int offset, int numbytes)	Writes a sub range of numbytes bytes from the array buffer, beginning at buffer[offset].
void flush()	Clears the buffer.

Character streams Class:

It provides support for managing I/O operations on characters. Classes Reader and Writer are operated on characters for reading and writing, respectively. Character streams can be used to read and write 16-bit Unicode characters. Reader stream classes are designed to read character from the files and writer stream classes are designed to perform all output operations on files. The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize.

Reader Stream Classes: Reader stream classes are designed to read character from the files. Reader is designed to handle characters; therefore reader classes can perform all the functions implemented by the input stream classes.



Hierarchy of reader stream classes

Various Reader classes and their descriptions are:

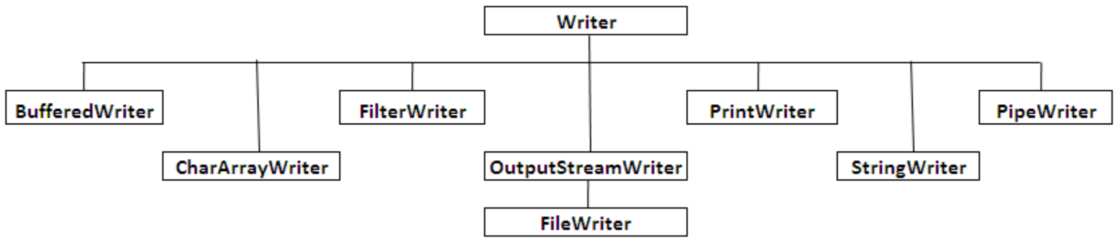
Class Name	Description
BufferedReader	It buffers the characters read from reader. It is a filter stream.
CharArrayReader	The characters are read from a character array.
InputStreamReader	The character is read from byte input stream.
FileReader	This reads characters from a file.
FilterReader	This is used for chaining the reader.
PushbackReader	Allows one character pushback buffer for Reader. That is after a character was read from reader; it is pushed back into the reader.
StringReader	Reads into the string. In StringReader the source for reading is String.
PipeReader	Reading from pipe. Supports the inter-thread communication. In pipe streams, the output from one method could be piped into the next one.
LineNumberReader	It reads character keeping the track of lines read.

All methods in this class will throw an **IOException.**

Methods in Reader Class:

Methods	Description
int read ()	Returns an integer representation of next available character from invoking stream. -1 is returned at the stream end.
int read (char buffer[ ])	Read up to buffer.length chacters to buffer & returns actual number of characters that is successfully read. At the end returns –1.
int read(char buffer[ ], int offset, int numchars)	Attempts to read up to numchars into buffer starting at buffer[offset]. Returns actual number of characters that are read. At the end returns –1.
void close()	to close the input stream
void mark(int numchars)	Places a mark at current point in input stream and remain valid till numbers of characters are read.
void reset()	Resets pointer to previously set mark/ goes back to stream beginning.
long skip(long numchars)	Skips number of characters.
int available()	Returns number of bytes currently available for reading.

Writer Stream Classes: The Writer stream classes are designed to perform all output operations on files. It is an output character stream that writes a sequence of Unicode characters.



Hierarchy of writer stream classes

Various Writer classes and their descriptions are:

Class Name	Description
BufferedWriter	It buffers the characters to be written on writer.
CharArrayWriter	The characters are written from a character array.
FilterWriter	This is used for chaining the writer.
OutputStreamWriter	Supports writing of characters to a byte input stream.
FileWriter	This writes characters on a file.
PrintWriter	Printing values and objects.
StringWriter	Writes into the string. StringWriter writes the output into the StringBuffer.
PipeWriter	Writing to a pipe.
LineNumberWriter	It writes character keeping the track of lines written.

All the methods in Writer class returns a **void** value and throws an **IOException.** *The methods are:*

Methods	Description
void close()	To close the OutputStream
void write (int ch)	Writes a single character to an output stream.
void write(char buffer[ ])	Writes a complete array of characters to an output stream.
void write (char buffer[ ], int offset, int numchars)	Writes a sub range of numchars from the array buffer, beginning at buffer[offset].
void write(String str)	Writes str to output stream.
void write(String str, int offset, int numchars)	Writes a subrange of numchars from string beginning at offset.
void flush()	Clears the buffer.

Difference between Byte Stream Classes and Character Stream Classes:

Byte Stream Class	Character Stream Class
Byte streams access the file byte by byte (8 bits).	A character stream will read a file character by character (16 bits).
Byte stream classes are classified into: 1. Input Stream Classes 2. Output Stream Classes	Character stream classes are classified into: 1. Reader class 2. Writer class
InputStream/OutputStream class is byte-oriented.	The Reader/Writer class is character-oriented.
The methods for byte streams generally work with byte data type.	The methods for character streams generally accept parameters of data type *char* parameters.
Byte-stream classes end with the suffix InputStream and OutputStream.	Character-stream classes end with the suffix Reader or Writer.
It is possible to translate character stream into byte stream with OutputStreamWriter.	It is possible to translate byte stream into a character stream with InputStreamReader.
Byte Streams are dumb. They know absolutely nothing about the data flowing through them. They just know that a data element is a byte.	*Character Streams* are quite dumb, too. But instead of a single byte, their smallest unit of information is a Unicode character.
Byte streams specifically used for reading and writing data in byte format.	The advantage of character streams, that is make it easy to write programs, which is not dependent upon a specific character encoding.
No conversion needed.	Character streams convert the underlying data bytes to Unicode, which is a costly operation.
InputStream and OutputStream are used for reading or writing binary data.	Reader and Writer uses Unicode, hence they can be internationalized. Hence in some cases they are more efficient than byte streams.

Differentiate between Input Stream class and Reader Class

Input Stream Class	Reader Class
Input Streams are used to read bytes from a stream.	Reader classes are used to read character streams.
Input Stream class useful for binary data such as images, video and serialized objects.	Reader classes are best used to read character data.
Input Stream classes are used to read 8 bit bytes.	Reader class is used to read 16 bit Unicode character stream.
An Input Stream is byte-oriented.	A Reader is character-oriented.
Constructor: InputStream()	Constructor: Reader() and Reader(Object lock)

Methods: read();
read(byte[] b);
read(byte[] b, int off, int len);
	Methods: read();
	read(char[] buf);
	read(char[] buf, int off, int len);
	read(charBuffer dest);

Differentiate between Output Stream class and Writer class

Output Stream Class	Writer Class
Output Streams are used to write bytes to stream.	Writer classes are used to writes character streams
An Output Stream is byte-oriented.	A Writer is character-oriented.
OutputStream classes are used to write 8 bit bytes.	Writer class is used to write 16 bit Unicode character stream.
Constructor: OutputStream();	Constructor: Writer(); and Writer(Object lock);

Methods: write(byte[] b);
write(byte[] b, int off, int len);
write(int b);
	Methods: write(char[] buf);
	write(char[] buf, int off, int len);
	write(int c);
	write(String str);
	write(String str, int off, int len);

File Streams:

A file class is defined in the package java.io. A file can be created using File class.
Example:
File f;
f=new File(String filename);
The File class is used to store the path and name of a directory or file. The File object can be used to create, rename or delete file or directory it represents. Each file in java is an object of the File class. A File object is used to manipulate the information associated with a disk file.
A File class has three constructors that are used to create a file object are:

```
File(String pathname);                    //pathname could be file or a directory name
File(String dirPathname, String filename); // specify directory where the file is created and file name
File(File directory, String filename)      // file object as the directory path
```

The File class contains methods to perform various file operations:

- Creating a file
- Opening a file
- Closing a file
- Deleting a file
- Getting the name of the file
- Getting the size of a file
- Renaming a file

- Checking whether the file is readable
- Checking whether the file is writable

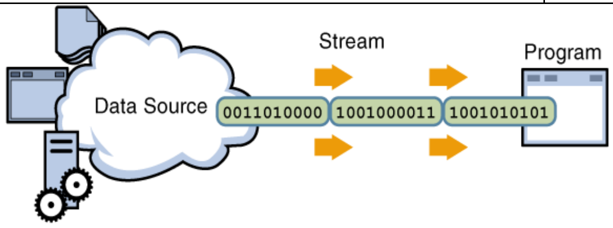
File class methods

Method	Description
String getName()	Returns the name of the file.
String getParent( )	Returns the name of the parent directory.
Boolean exists( )	Returns true if the file exists, false if it does not.
Boolean isHidden( )	Returns true if the invoking file is hidden. Returns false otherwise.
String getPath()	Gets the path of the file.
Boolean isDirectory()	Returns a boolean indicating whether or not a directory file exists.
Boolean isFile()	Returns a boolean indicating whether or not a normal file exists.
Long length()	Returns the length of the file.
String list()	Lists the files in a directory.
Boolean canRead()	Returns a boolean indicating whether or not a readable file exists.
Boolean canWrite()	Returns a boolean indicating whether or not a writable file exists.
boolean createNewFile()	This method atomically creates a new empty file.
boolean delete()	This method deletes the file or directory denoted by this abstract pathname.

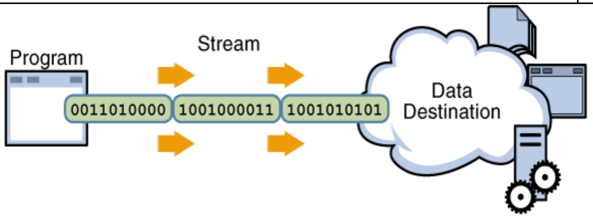
FileInputStream and FileOutputStream:

The File class explains only the file system. Java provides two special types of stream called the FileInputStream and FileOutputStream to read data and write data into the file. These classes operate on the files in the native file system. FileInputStream and FileOutputStream are sub-classes of InputStream and OutputStream respectively. Creation of file includes file name as a parameter in the form of a string, File object or FileDescriptor object.
The constructors of these two streams are:

FileInputStream(String filename);	FileOutputStream(String filename);
FileInputStream(File fileobject);	FileOutputStream(File fileobject);
FileInputStream(FileDescriptor fdesobject);	FileOutputStream(FileDescriptor fdesobject);



A program uses an input stream to read data from a source, one item at a time.



A program uses an output stream to write data to a destination, one item at a time.

DataInputStream and DataOutputStream:

DataInputStream and DataOutputStream is a filtered input and output stream and hence must be attached to some other input and output stream.
constructors to create an InputStream and OutputStream are:
InputStream in = DataInputStream(InputStream in);
DataOutputStream out = DataOutputStream(OutputStream out);
Example of DataInputStream:

```
import java.io.*;
public class FileDataInputStream
{
public static void main(String[] args) throws Exception
{
File file = new File("C:/data.txt");
DataInputStream dis = new DataInputStream(new FileInputStream(file));
while (dis.available() != 0)
{
System.out.println(dis.readLine());
}
dis.close();
}
}
```

Create and save file “data.txt” into c:/

Output:

```
Hai hello
Java Programming
```

Example of DataOutputStream:

```
import java.io.*;
public class FileDataOutputStream
{
public static void main(String[] args) throws Exception
{
try
{
   FileOutputStream fos = new FileOutputStream("data1.dat");
   DataOutputStream dos = new DataOutputStream(fos);
   dos.writeUTF("Vijay Patil");
   dos.writeUTF("Thane, Maharashtra");
   dos.writeInt(400606);
   dos.flush();
   dos.close();
}
catch (Exception e)
{  }
}  }
```

Filter Streams:

A filter stream is a stream which allows for reading and writing of data in java. The FilterInputStream and FilterOutputStream classes define input and output filter stream. A filter stream is constructed on another stream i.e. every filtered stream must be attached to another stream. The subclasses DataInputStream and DataOutputStream implement filters which allow java primitive values to be read and written. The various methods used for reading and writing are:

Data type	Reading method	Writing method
boolean	readBoolean()	writeBoolean(Boolean A)
char	readChar()	writeChar(int A)
byte	readByte()	writeByte(int A)
short	readShort()	writeShort(int A)
int	readInt()	writeInt(int A)
long	readLong()	writeLong(long A)
float	readFloat()	writeFloat(float A)

double	readDouble()	writeDouble(double A)
string	readLine()	writeLine(string S)
string	readUTF()	writeUTF(string S)

PushbackInputStream:

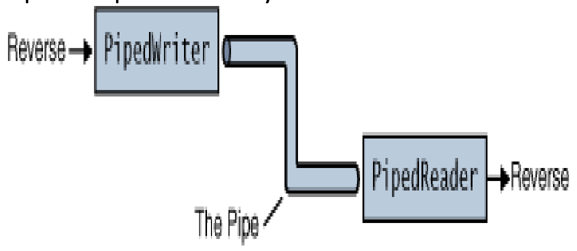
Pushback is used on an input stream to allow a byte to be read and then returned (that is, "pushed back") to the stream. Like all filter streams, PushbackInputStream is also attached to another stream. Sometimes you need to read ahead a few bytes to see what is coming, before you can determine how to interpret the current byte. The PushbackInputStream allows you to do that. It is specifically used to read the first byte of the data from a stream for the purpose of testing the data.
A PushbackInputStream Object can be created as:
PushbackInputStream input = new PushbackInputStream(new InputStream);
Example:
PushbackInputStream input = new PushbackInputStream(new FileInputStream("c:\\data\\input.txt"));
int data = input.read();
input.unread(data);
In order to satisfy the requirement of pushing back a byte of an array into the input stream, this stream has the special method called the unread() method.

1. void unread(int *ch*)
2. void unread(byte *buffer*[ ])
3. void unread(byte *buffer*, int *offset*, int *numChars*)

The first form pushes back the low-order byte of *ch*, which is returned as the next byte when another byte is read. The second and third forms of the method return the byte in a buffer with the difference that the third method pushes back the number of characters (numchars) starting from the offset. PushbackReader also does the same thing on the character stream.

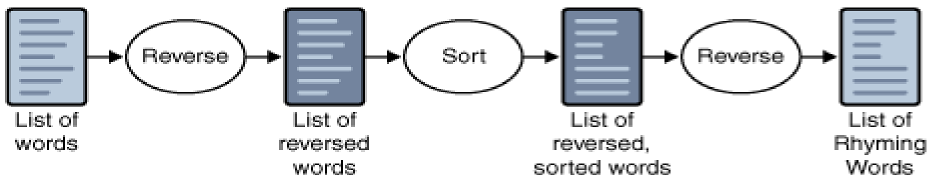
Pipe Stream:

Pipes are used to channel the output from one thread into the input of another. When two threads are communicate each other, one thread sends data as an output to pipe and other thread will receives data from pipe. Creating a pipe using Java IO is done via the PipedOutputStream and PipedInputStream classes. A PipedInputStream should be connected to a PipedOutputStream. The data written to the PipedOutputStream by one thread, can thus be read from the connected PipedOutputStream by another thread.



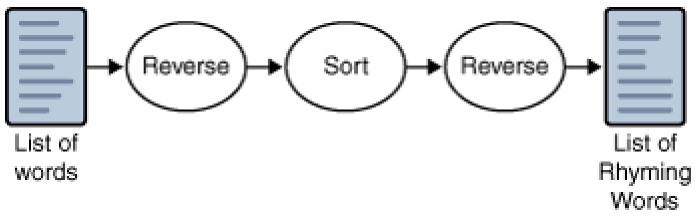
Using PipedWriter and PipedReader to form a pipe.

Example of how to connect a PipedInputStream to a PipedOutputStream:
PipedOutputStream output = new PipedOutputStream();
PipedInputStream input = new PipedOutputStream(output);
Without pipe streams, the program would have to store the results somewhere (such as in a file or in memory) between each step, as shown in the following figure.



Without a pipe, a program must store intermediate results.

With pipe streams, the output from one method could be piped into the next, as shown in this figure.



Random Access File:

The Random access files are used for instant access to files. RandomAccessFile class supported by the java.io package allows us to create files that can be used for reading and writing data with random access. The records in such a file are of fixed length. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the *file pointer*; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written. Output operations that write past the current end of the implied array cause the array to be extended. The file pointer can be read by the getFilePointer() method and set by the seek() method.

Creating a RandomAccessFile:
RandomAccessFile file = new RandomAccessFile("c:\\data\\file.txt", "rw");
The constructor RandomAccessFile receives a filename and file open mode.
Modes of opening a file are:
r=reading
w=writing
rw=reading/writing
Methods:

1. position – Returns the current position
2. position(long) – Sets the position
3. read(ByteBuffer) – Reads bytes into the buffer from the stream
4. write(ByteBffer) – Writes bytes from the buffer to the stream
5. truncate(long) – Truncates the file (or other entity) connected to the stream

File Operations:

Creating file:
File file = new File(“file1.txt”); A File object represents name and path of a file or directory on a Disk.
For creating a new file **File.createNewFile( )** method is used. This method returns a boolean value true if the file is created otherwise return false. If the mentioned file for the specified directory is already exist then the **createNewFile()** method returns the false otherwise the method creates the mentioned file and return true.

Reading of data from file:
The functions are used for reading of data from files are:
“readInt” function reads a integer value from file.
“readUTF” function read a string from the given file.
“readDouble” function read double value from a file.
Writing data into file:
“WriteInt” function writes an integer value on the file.
“WriteUTF” function writes a string on a file.
“WriteDouble” function writes a double value on a file.
Closing of file: input.close(); statement closes the file open by input object.
Seeking position in file: input.seek(n); statement seeks nth position of byte on a file.

Serialization:

Serialization is the process of saving an object in a storage medium (such as a file, or a memory buffer) or to transmit it over a network connection in binary form. The serialized objects are JVM independent and can be re-serialized by any JVM. In this case the "**in memory**" java objects states are converted into a byte stream. This type of the file cannot be understood by the user. It is special types of object i.e. reused by the JVM (Java Virtual Machine). This process of serializing an object is also called **deflating** or **marshalling** an object. Storing the state of the object into a file is referred to as **serialization**. Whereas, reading the object state from a stored file is referred to as **de-serialization**.
The streams ObjectOutputStream and the ObjectInputStream are used for object serialization and de-serialization. These classes allow us to read /write objects from to streams converting from internal to external 8-bit representation. Two methods are useful for object serializations are:
readObject() on ObjectInputStream and writeObject(Object obj) on the ObjectOutputStream.
To make any object serializable, the object of the class must implement the serializable interface of the java.io package.
Syntax: public class implements Serializable

Programs:

1. Program to read characters from file and display contents.

```
import java.io.*;
class readchar
{
public static void main(String args[]) throws IOException
{
int i;
FileInputStream f1=null;
try
{
try
{f1=new FileInputStream(args[0]);
}
catch(FileNotFoundException e)
{
System.out.println("File can't be opened");
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("error in suppy of file name");
return;
}
do
{
i=f1.read();
if(i!=-1)
System.out.print((char)i);
}
while(i!=-1);
f1.close();
}
}
```

```
Input: c:\\input.dat
Output: Vijay Patil
        Thane
```

2. Program to copy contents of one file to another using character stream class.

```
import java.io.*;
class copyf
{
public static void main(String args[])
{
FileInputStream f1=null;
FileOutputStream f2=null;
byte byteRead;
try
{
f1=new FileInputStream("c:\\input.dat");
f2=new FileOutputStream("c:\\output.dat");
do
{
byteRead=(byte)f1.read();
f2.write(byteRead);
} while(byteRead!=-1);
}

catch(FileNotFoundException e)
{
System.out.println("File not Found");
}

catch(IOException e)
{
System.out.println(e.getMessage());
}

finally
{
try
{
f1.close();
f2.close();
}
catch(IOException e)
{ }
}
}
}
```

3. Write a program to read a file (using character stream)

```
import java.io.*;
class charreader
{
public static void main(String args[]) throws Exception
{
FileReader f1=new FileReader("c:\\input.dat");
BufferedReader br=new BufferedReader(f1);
String str;
while((str=br.readLine())!=null)
{
System.out.println(str);
}
f1.close();
}
}
```

4. Program to count occurrences of a string within a text file

```
import java.io.*;
import java.util.*;
public class TextFile
{
public static void main (String arg[])
{
File f = null;
// Get the file from the argument line.
if (arg.length > 0)
f = new File (arg[0]);
if (f == null || !f.exists ())
{
System.exit(0);
}
String string_to_find = arg[1];
int num_lines = 0;
try
{
FileReader file_reader = new FileReader (f);
BufferedReader buf_reader = new BufferedReader (file_reader);
// Read each line and search string
do
{
String line = buf_reader.readLine ();
if (line == null) break;
if (line.indexOf(string_to_find) != -1) num_lines++;
}
while (true);
buf_reader.close ();
}
catch (IOException e)
```

```
 {
System.out.println ("IO exception =" + e );
}
System.out.println ("No of lines containing " + string_to_find + " = " + num_lines);
} // main
} //class TextFileReadApp
```

5. Write a java program to read ‘n’ city names and store them in Array List collection. Display the elements of collection in sorted order

```
import java.util.*;
public class citydetails
{
public static void main(String args[])
{
ArrayList<String> listofcity = new ArrayList<String>();
listofcity.add("Mumbai");
listofcity.add("Thane");
listofcity.add("Nagpur");
listofcity.add("Aurangabad");
listofcity.add("Pune");

/*Unsorted List*/
System.out.println("Before Sorting:");
for(String counter: listofcity)
{
System.out.println(counter);
}

/* Sort statement*/
Collections.sort(listofcity);

/* Sorted List*/
System.out.println("After Sorting:");
for(String counter: listofcity)
{
System.out.println(counter);
}
}
}
```

output

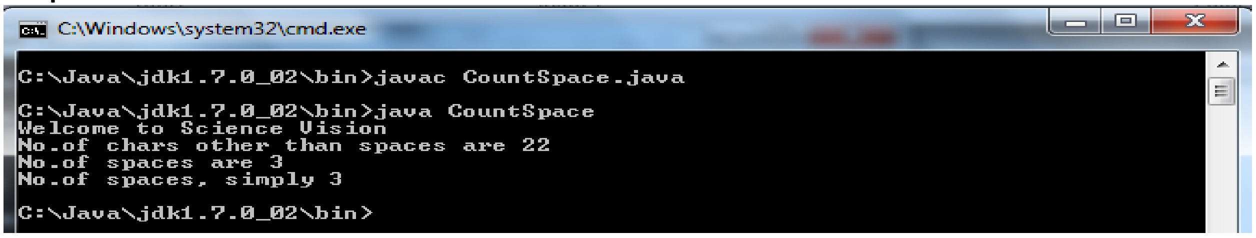
```
Before Sorting:
Mumbai
Thane
Nagpur
Aurangabad
Pune

After Sorting:
Aurangabad
Mumbai
```

Nagpur
Pune
Thane

6. Write a java program to count number of white space character in character stream.

```
import java.util.*;
class CountSpace
{
 public static void main(String args[])
 {
 // Create Scanner object to take input from command prompt
 Scanner s=new Scanner(System.in);

 // Take input from the user and store it in st
 String st=s.nextLine();

 // Initialize the variable count to 0
 int count=0;

 // Convert String st to char array
 char[] c=st.toCharArray();

  // Loop till end of string
  for(int i=0;i<st.length();i++)

  // If character at index 'i' is not a space, then increment count
  if(c[i]!=' ') count++;

 // Print no.of chars other than spcaes
 System.out.println("No.of chars other than spaces are "+count);

 // Print no.of spaces
 System.out.println("No.of spaces are "+(st.length()-count));

 // Simply,
 String words[]=st.split(" ");

 // Print no.of spaces
 System.out.println("No.of spaces, simply "+(words.length-1));
  }
}
```

Output:

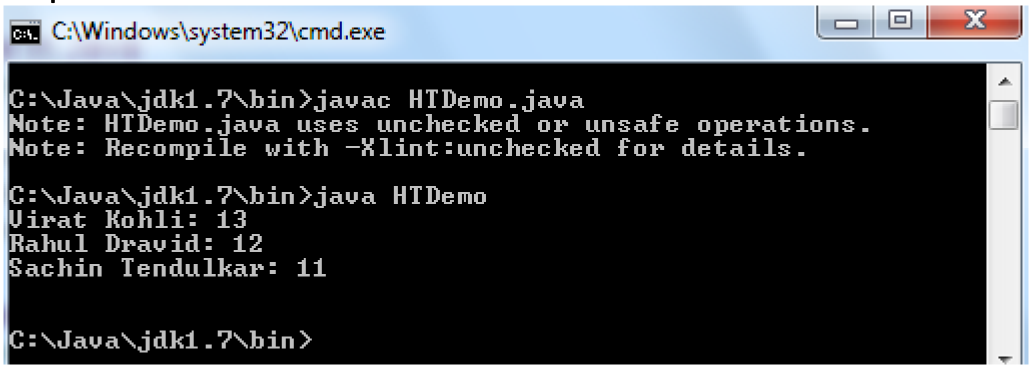


```
C:\Java\jdk1.7.0_02\bin>javac CountSpace.java

C:\Java\jdk1.7.0_02\bin>java CountSpace
Welcome to Science Vision
No.of chars other than spaces are 22
No.of spaces are 3
No.of spaces, simply 3

C:\Java\jdk1.7.0_02\bin>
```

7. Write a java program to accept n employee information (id, name) and store into hash table. Display all employee details.

```
import java.util.*;
class HTDemo
{
public static void main(String args[])
{
Hashtable ht= new Hashtable();
Enumeration names;
String str;
int id;
ht.put("Sachin Tendulkar", new Integer(11));
ht.put("Rahul Dravid",new Integer(12));
ht.put("Virat Kohli",new Integer(13));

// Show all ht in hash table.
names=ht.keys();
while(names.hasMoreElements())
{
str=(String)names.nextElement();
System.out.println(str + ": " + ht.get(str));
}
System.out.println();
}
}
```

Output:



C:\Windows\system32\cmd.exe

```
C:\Java\jdk1.7\bin>javac HTDemo.java
Note: HTDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Java\jdk1.7\bin>java HTDemo
Virat Kohli: 13
Rahul Dravid: 12
Sachin Tendulkar: 11


C:\Java\jdk1.7\bin>
```

8. Write a program to count characters, word and lines from file

```
import java.io.*;
public class countcwl
{
public static void main(String[] args)
{
BufferedReader br=null;

//Initializing charCount, wordCount and lineCount to 0
int charCount = 0;
int wordCount = 0;
int lineCount = 0;
try
{
br= new BufferedReader(new FileReader("input.txt"));
//Reading the first line into currentLine
String currentLine = br.readLine();

while (currentLine != null)
{
//Updating the lineCount
lineCount++;

//Getting number of words in currentLine
String[] words = currentLine.split(" ");

//Updating the wordCount
wordCount = wordCount + words.length;

//Iterating each word
for (String word : words)
{
//Updating the charCount
charCount = charCount + word.length();
}
//Reading next line into currentLine
currentLine = br.readLine();
}
//Printing charCount, wordCount and lineCount
System.out.println("Number Of Chars In A File : "+charCount);
System.out.println("Number Of Words In A File : "+wordCount);
System.out.println("Number Of Lines In A File : "+lineCount);
}
catch (IOException e)
{
e.printStackTrace();
}
finally
{
try
{
br.close();         //Closing the reader
}
catch (IOException e)
{
e.printStackTrace();
} } } }
```