I/O STREAMS

java.io PACKAGE   io->input/output

Stream:
i) Constant flow of water is called as stream.
ii) In computer stream is a flow of data items of unlimited length.

*** A Set of classes is used by some programmers or some program users, memory as its source as well as its destination.

IO Stream (or) java.io package consists of a set of classes and interfaces which can be used by the java programmers to read the (take) data from multiple input sources and write the data to different destinations.

IS-->Input Source
OD-->Output Destination

```
IS1
read(){
………………
……………}
read(k){
………… }
close(){
……….}
```

```
IS2

read(){
………….}
read(k){
………… }
close(){
……….}
```

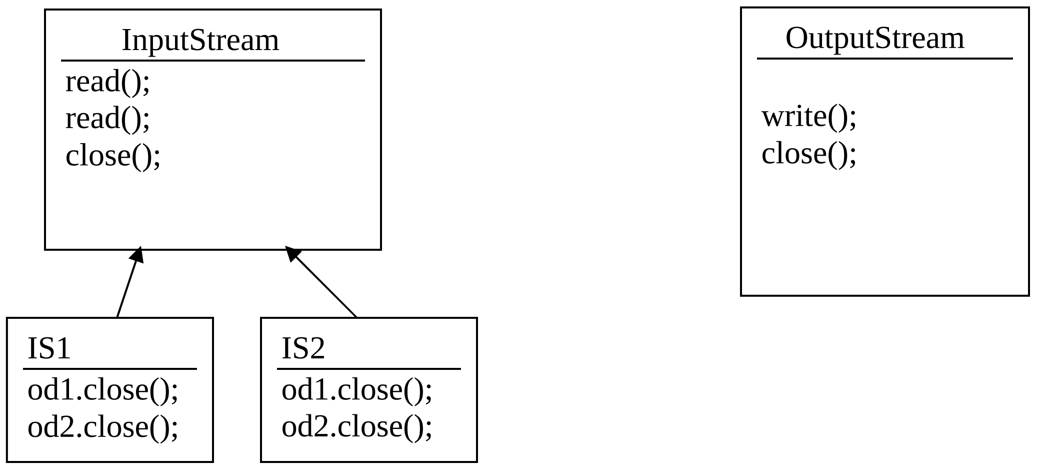
An abstract method can be called from the concrete method.

EX:

```
abstract class InputStream{
abstract read();
read(k,i){
        for(int n=0;n<i;n++)
        k[n]=read();
        }
}
```

** In input & output Streams library we have two abstract classes:
(i) *InputStream* (ii) *OutputStream*, which are extended by the sub-classes to provide the implementation of reading from multiple sources & writing multiple destinations.

The advantage of the design of having the super-classes Input and Output Streams is, any one can provide their own implementation of a new class to read or write data extending



the classes input and output streams and the other advantage of this design is a developer can write a generic code as shown below:

InputStream IS=new InputStream();
IS.read();
OutputStream OS=new OutputStream();
OS.write();
OS.close();

* After Java1.0,JavaSoft has developed
reader-->read character by character.
writer-->write character by character.

The classes (or) the sub-classes of I/O Streams allow the programmer to read or write byte by byte Format. From Java1.1 Onwards JavaSoft has provided another set of classes to read and write character data .The Super-classes for the set of classes that reads and writes character data are: Reader and Writer.

JavaSoft Followed a very simple technique naming convention to easily identify the name of the classes that acts as input stream (or) reader.

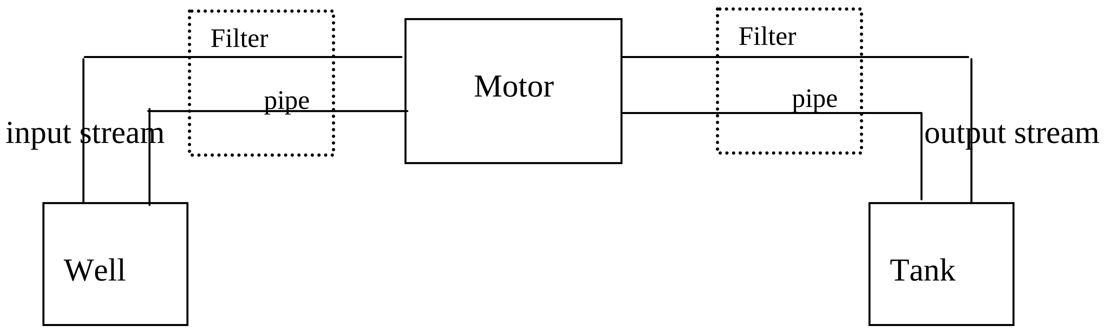
* end of Stream -> End is represented by special # -1.

When we apply a read method in the Input Streams, the Input streams read the bytes, but returns the bytes as integers. All the positive numbers from 0 to 127are returned as 0 to 127.But the negative numbers are returned as numbers greater than 127.So this is used to differentiate between byte -1 in the Input Stream .And a Special Value -1 which is an end of the stream.

Every Output stream puts a limit on no. of Files that can be opened by a process.

In Dos -- \r\n-->represents end of line.

In Windows --- \n-->represents end of line.
Most of the programs on DOS O/S stores '\r\n' to represents end of line, but in windows '\n' represents as end of line.
File Output Stream allows us to create a file and adds the data, and it also allows us to open a file that already exists and append the data to that file.
Most of the editors on Windows O/S are designed to interpret both'\r\n' and '\n' as the end of line.

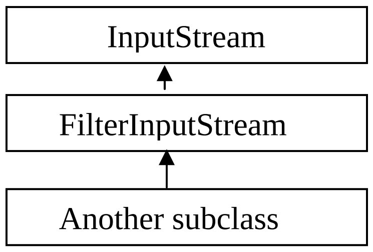


Our program is a motor
Our data is a well
I/O stream is a pipe
File is a tank

Filter: Filter is used to perform some sort of conversion as per our requirement.(convert the way the data can be understood by the programmer).

Filter gets the data in one form i.e. byte by byte and converts it into integer/float/long or any other data types.

Data requires filtering action. Type of filtering depends upon the developer’s requirement. Filtering may be in the outside i.e. output side or in the input side. When we write a program in java or any kind of languages, we need to take input and produce an output. Using the sub classes of input and output streams, we can read the data byte by byte or write the data byte by byte. But in some cases it would be convenient for the java programmer to read the data in a different format. In order to allow this, JavaSoft has designed two different classes,
1. FilterInputStream
2. FilterOutputStream
Subclasses of FilterInput/output streams are used to perform filtering actions.



**every filter requires another stream.
**Any FilterInputStream requires some other InputStream as a parameter and any FilterOutputStream requires some other output stream as a parameter.
In the Input/Output classes we have two classes with the name ByteArrayInput/OutputStreams, which are used to represents a memory as the source and memory as the destination.

Source file is replaced by memory.
 * FileInputStream requires file as input.
 * ByteArrayInputStream requires memory as input.

Ex:
```
  class iostream
   {
     public static void main(String args[])
      {
         byte c[]={22,23,24,25,.....,32};
         InputStream in=new ByteArrayInputStream(c);

        /* creation of file InputStream is different from that of ByteArrayInputStream   */

         in.mark(10);      //read limit is 10
         System.Out.println("is mark supported: "+in.markSupported());

        /* markSupported returns true/false, if it is true, this will support the marks
(bookmarks)
          otherwise this will not support.*/

        System.Out.println("next available byte is="+in.read());
        // this returns the next available byte in an array.

        byte b[]=new byte[20];
        in.read(b,0,3);
        int i;
        for(i=0;i<3;i++)
         {
           System.Out.println(b[i]," ");
           System.Out.println();
           in.reset();

           System.Out.println("bytes read are: "+in.read(b,0,3));
           System.Out.println("in.available()" +in.available());
           System.Out.println(in.read());
         }
        /* InputStream.available() returns the number of bytes available in the next read
call */      }
```

When we use multi byte read, the return type value indicates the no of bytes read and in some cases it returns -1 to indicate the end of stream is reached.

-1 returns 255
Byte >127 should be a negative value.

When we use InputStream.read(), the read method converts a negative byte value into a positive value i.e., greater than 127.
[-1=255, -2=254, -3=253, ..., -128=127]

* When we use single byte read in the InputSteram, all the negative byte values converted into positive integers by using the below formula.

* As a java programmer we need to check always the return values of single byte read and then apply appropriate logic to get actual values read from the stream.

* When we use multi byte read, the values are returned as it is. In case of single byte read, the above logic is used to differential between a byte value -1 and the end of stream -1 value.

In case of InputStream which supports the mark operation, we can use the methods:

1. mark() --> to place a bookmark
2. reset() --> to come back to the bookmark
3. readLimit() --> to limit the bytes read
4. skip(n) --> to skip n number of bytes

note: we should not use the mark and reset operations on the stream which does not return true for marksupported method. In some of the streams it may be possible to reset back after crossing the readlimit. But it is always recommended to reset back before crossing the readlimit.

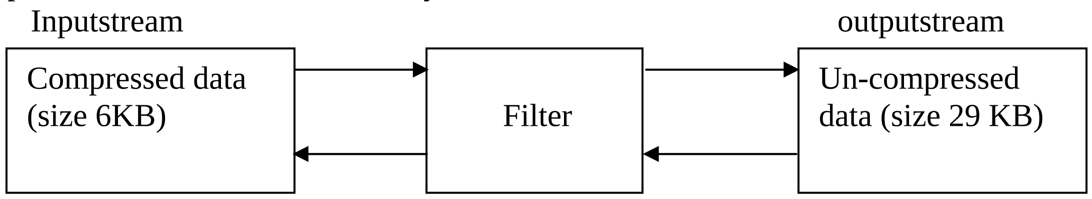
* In case of InputStream we can use the mark method to put a single mark.
* In.available() returns number of bytes are available to the next read call and in some cases In.available() may return a value x, but when we execute In.read() as shown, it may return a value greater than x if the input is the network program.

```
In.available();
/* if there are 10 bytes in an array */
In.read(b.0,x+99);
/* here it is should return 10, but it is returning the value greater than x.*/
```

* In case of ByteArrayInputStream where the total data available is the memory itself. we can able to predict the return values for available methods.
* Experiment with ByteArrayInputStream which allows us to write the output to memory itself.

We can use FilterStreams to compress and uncompress the data.

The other filters allow us to read byte by byte and convert them as integers, strings (primitive data types). A java programmer can use a filter and write java primitive data types which will be converted to bytes.



There are standard algorithms to compress and uncompress the data. ex: WinZip
    java.util.zip --- package

Inflator() -- expands the data
Deflator() -- compress the data

If you read the data from compressed file while InputStream is connected to the filter, then some data is not available to the FilterStream. If you want to read the data from FilterStream you should not get the complete data. If you want the data you must read either from FileInputStream or from the Filter. But don't read the data from the two.

ex: /* program to uncompress the compressed data */

```
class uncompress
 {
   //Filter on the input side
     public static void main(String args[])throws Exception
      {
        FileInputStream fis=new FileInputStream("com.bmp");
        FileOutputStream fos=new FileOutputStream("uncom.bmp");
        InflaterInputStream iis=new InflaterInputStereeam("fis");
        int data;

        while((data=fis.read())!=-1)
          {
            fis.write(data);
           }
        fis.close();
       }
   }
```

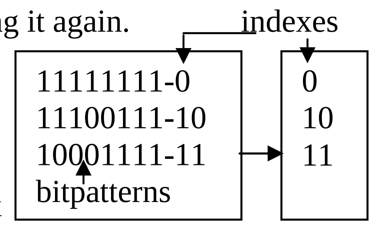
```
/* program to compress the uncompressed file */

   class compress
    {
       //Filter on the output stream
```

```
            public static void main(String args[])throws Exception
             {
               FileInputStream fis= new FileInputStream("uncom.bmp:);
               FileOutputStream fos=new FileOutputStream("com.bmp");
               DeflaterOutputStream dos=new DeflaterOutputStream("fos");

               int data;
               while((data=fos.read())!=-1)
                {
                  fos.write(data);
                 }
               fos.close();
              }
          }
```

note: experiment with the compressed file by compressing it again.



How the compression takes place:
In the algorithm first,
* Bit patterns are selected.
* For each bit pattern of 8 bytes, they will create an index with one or more digits.
* At the output they read the file according to the indexes created for the bit patterns.
* When compared to the original file, this will be very small.
* There are so many standard algorithms to compress and uncompress the data.
* In some cases, when we compress the data which is already compressed, the size of the new data will be more than the size of data that is compressed.

/* program to display all files in c:\windows i.e., in all the folders and sub folders*/

```
    class Listfiles
     {            // by using the recursion function
        static int nof;
        public static void main(String args[])throws Exception
         {
           Listfiles("c:\windows");
           System.Out.println("number of files=" +nof);
         }

        private static void Listfiles(String args[] dirname)
         {
           File Myfile = new File(dirname);
           int i;
           File flist[];
           flist=Myfile.Listfiles;

           for(i=0; i<flist.length; i++)
```

```
            {
              if(flist[i].isDirectory())
               {
                 System.Out.println("D**" +flist.getAbsolutepath());
                 Listfiles(flist[i].getAbsolutepath());
                }
               else
                 System.Out.println(flist[i].getAbsolutepath());
               nof++;
             }
           }

         }
```

* converting recursive methods to iterative methods is possible

Even though it is very convenient to write some methods by using recursion we have an overhead in executing the recursive methods. When ever we use a recursive method, a method will be calling itself and when a method call is executed, a new stack frame is created which is an expensive operation and in some cases we will see stack overflows.

So in most of the cases we need to prefer an iterative mechanism than using recursion mechanism.

/* Algorithm how the list of files are displayed */

```
  Listfiles(fname)
    {
           A: get the list of files from a particular folder
       for(i=0; i<n; i++)   // n is list of files
         {
           if(it is not a directory)
            {
             print all the files;
             if(it is a directory) goto A;
            }
         }
    }
```

* We have different FilterStreams available which allows us to read the data in java primitive data types format and write the data in java primitive data types format.

DataInputStreams are the subclasses of filter input/output streams

* Data Input/Output Streams allows us to read the data using java primitive data type format and writing java primitive data type format.

* Using DataInputStream we can convert the underlying ByteStream to java primitive datatypes
Using DataOutputStream we can write the data using java primitive datatypes which will be converted by DOS as bytes and they are written to the underlying stream.

Data input/output example:

We can not use the following methods because they are not part of FileInputStreams contract.

```
        System.Out.println(fis.readInt());
        System.Out.println(fis.readFloat());
        System.Out.println(fis.reset());
```

/* how the data input/output stream works */
ex: ByteArray as input parameter to the inputstream

```
class iostreams
 {
   public static void main(String args[])throws Exception
    {
      byte c1={22,23,24,.....,32};
      byte c2={2,3,4,5,6,7,8,9,0,1,2};

      InputStream in=new ByteInputStream(c1,0,10);
      FileInputStream fis=new DataInputStream(in);
      System.out.println(fis.read());
      DataInputStream dis=(DataInputStream)fis;

      fis.mark(10);
      System. Out.println(dis.readInt());
      fis.reset();
      System.Out.println(dis.readFloat());
      System.Out.println(dis.readInt());
      System.Out.println(dis.readDouble());
    }
 }
```

note: in this example we are reading 10 bytes from an input stream. From this 4 bytes are used to convert to float and 4 bytes are to double data type we will get an error saying that EOF. This is occurred because the remaining bytes are 2 only. But we need 8 bytes to convert double data type.

* observe the output the output will be displayed in a binary form.

```
output:  22
         387455258
         1.2913426 to -22
         Exception in thread "main" java.io.EOFException
```

The data input stream takes the appropriate number of bytes from the underlying stream and converts it into java primitive data type. For ex, to create an integer, DataInputStream takes 4 bytes from the underlying stream and converts into integers.

Incase of DataOutputStream the DataOutputStream can be fed with any outputstream as a parameter and we can apply the methods writeInt(), writeFloat().

```
/* read the file and display the contents in a file */

   class readline
    {
      public static void main(String args[])throws Exception
       {
         FileReader fr=new FileReader("testfile.txt");
         LineNumberReader lnr=new LineNumberReader(fr);
         for(int io=0; i<3; i++)     {
             String s=lnr.redLine();
             System.Out.println(s);
         }
         fr.close();
     }
```

FileReader: reads the data from a file as character by character.
LineNumberReader: provides a convenient way to read line by line.

```
\r\n -- to terminate the line in dos o/s
\n -- to terminate the line in unix o/s
```

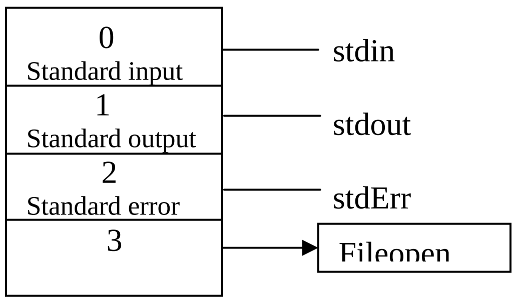
* The LineNumberReader(LNR) reads the data till reaches \r\n or \n from the underlying reader and returns the lines by skipping the line terminators.

while running the program, give like this:

```
c:\> java reader>in.txt   //returns the out put to in.txt
c:\> java reader<in.txt   //reads the input from in.txt
```

* The input/outputs will be reads/writes from/to the file given at the command line.

* The o/s will take care. how this managed is



0,1,2,3 are called as filedescriptors. (this will gives an information about a specific file or this describes a specific file)

FileTable: this is assumed as an array but it is going to keep track of the files opened by a specific program.

* File descriptions are the indexes to the table. In the o/s first three slots are reserved for fileTable as shown in the above figure. If you open a new file then that file is stored in the slot number 3. if you open another file without closing the earlier file, then the new file will override the old file and stores it in the slot number 3 only.

* UNIX guys treat every device as a file and represented the file descriptor as follows:
  First FD(0) represents the keyboard
  Second FD(1) represents the monitor
  Third FD(2) represents the monitor

The out put and error messages are displayed in the monitor.

* When ever the process done by an o/s, the program is responsible for opening up a file table in which it is going to store the information about stdin, stdout and stderr.

ex:

```
class ex
 {
   public static void main(String args[]) throws Exception
    {
       System.Out.println("this is sent to stdout");
       System.Err.println("this is sent to stderr");
       System.in.read();
       System.in.read();
     }
 }
```

observe the output:

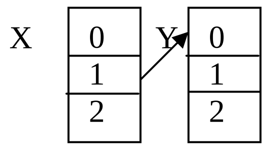
```
this is sent to stdout
this is sent to stderr        will appear.
```

Options:
* c:\>java exp<testfile.txt
in this case, the program should not wait for the keyboard input.

* c:\>java exp>out
this will goes to the stderr(2) but this will be displayed in the monitor i.e. output will be on your console.
* c:\>java exp 2>eee
this will store the file description eee to the stderr(s). to manipulate this send it to the stdout(1). then the output will be displayed is: this is sent to stdout.

* Try to find out how to redirecxt both stdout and stderr of a program

* Depending on the configuration of o\s we have certain limit on number files that can be used by a particular program. so it is always advisable to close the files after using the files.



in UNIX o/s X|Y means

stdout of first program is the stdin of second program.

When we use a pipe symbol between two programs in unix o/s, the unix o/s is responsible to connect stdOut of first program to stdIn of second program.

Note: FileAllocation Table and FileTable are not the same FAT used in the way the files written on the disk.
* Print Streem provides the methods like print() and println().
* Nulloutput-> It takes some bytes, and writes nothing (or) not stored any where.
* By default printstream connected to the FileDescriptor (stdout). Which is by default connect to the monitor.

Ex:- /* Print Stream with Nullop Example*/
```
class iostreamg
{
        public static void main(String args[])throws Exception
        {
                nullop nop=new nullop();
                PrintStream ps=new PrintStream(nop);
                System.out.println("Before setting the new out Stream");
                System.setout(ps);
                System.out.println("After setting the new out stream");
                FileOutputStream out=mew FileOutputStream(FileDescriptor.out);
                System.out.println(new PrintStream(new BufferedOuputStream(out)));
                System.out.println("After Re-setting the New Out out stream");
                out.close();
                //if this is used s.o.flush() should
                System.out.flush();
                //notwork
        }
}
```
Output:- Before setting the New out stream

After Re-setting the new outstream. If System.out.flush(); is not used, the output will beonly
one sentence.
In nullcp class:
```
 class nullop extends outputstream
 {
        public void write(int i)
        {
        }
 }
```

How the out variable working in the java language at the back:-

Out variable of the System class is of type PrintStream. This PrintStream object by default is connected to the FileDescriptor1(stdout)
By default System.in variable will be pointing to FileDescriptor o (stin), System.out will be pointing to FileDescriptor1(stdout) and the System.err will be pointing to the FileDescriptor2 (stderr).
But these variables can be programmatically modified by using System.SetIn(), System.setOut and System.setErr().

SequenceInpputStream() SIS:- Provides a convenient mechanism of combining multiple streams and treating them as a single stream. They can read the File2, when File1 is completely read and so on.
When we used SIS, the SIS reads the First input Stream and then moves on to the second input stream. When we execute the available() method on SIS, it simply executes the available() method on SIS, it simply executes the available() method on the current inputstream (i.e. from which it is reading).

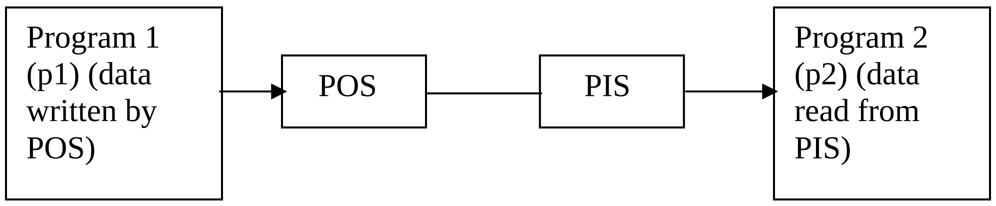
Example:-
/* Sequence Input Stream Example*/

```
class iostream2
{
        public static void main(String args[])throws Exception
        {
                byte c1[]={22,23,24,25.......32};
                byte c2[]={2,3,4,5,6,7,8,9,0,1,2};
                ByteArrayInputStream b1=new ByteArrayInputStream(c1);
                ByteArrayInputStream b2=new ByteArrayInputStream(c2));
```

```
 //SequenceInputStream(InputStream,InputStream);

   SequenceInputStream s=new SequenceInpuStream(b1,b2);
             for(int i=0;i<15;i++)
             {
                         s.o.p(s.read());
                         if(i==s)
                         s.skip(5);
                         s.o.p("........"+s.available());
             }
}
}
```
Note: we can use all the methods that are used in the ByteArrayInputStream Example.

PipedInput\Output Streams (PIS/POS):- PIS & POS's provides us a mechanism of setting up a pipe, which allows us to write to POS which can be read from the other end PIS.


Program 1 (p1) (data written by POS)
POS
PIS
Program 2 (p2) (data read from PIS)

P2 requires input from P1. POS is available on PIS.

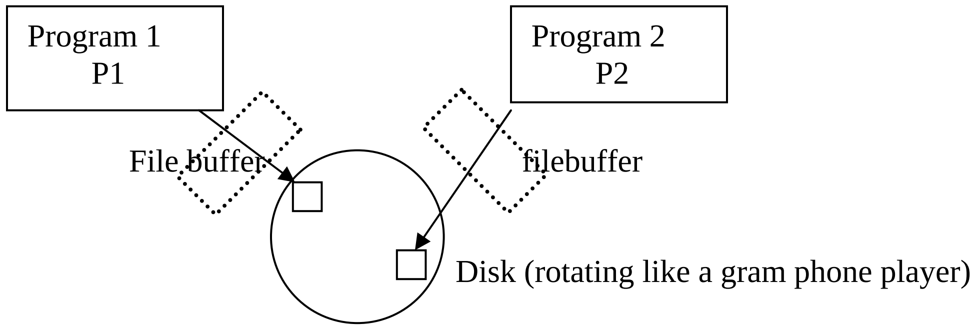
When we use PIS and POS's, the data from the PIS can be read in the same sequence in which the data is written by POS.

Ex:- /* piped input/output streams example */
```
class ioStream6
{
            p.s.v.m(String args[])
            throws Exception
            {
                        byte c1[]={22,23,24,.........32}
                        byte c2[]={2,3,4,5,...9};
                        PipedInputStream pis;
                        PipedOutputStream pos;
//create an outputstram to which we can write
                        pos=new PipedOutputStream();
//create an inputstream to read, and connect it to previously created outputstream.
                        pis=new PipedInputStream(pos);
                        pos.write(c1,o,c1.length());
                        s.o.p("data available in pis="+pis.available());
                        pos.write(c2,o,c2.length());
                        s.o.p("data available in pis="+pis.available());
                        pis.read(c1,o,4);
                        for(int i=0; i<6; i++)
                        s.o.p(pis.read());
                        s.o.p("data available in pis="+pis.available());
            /*   pis.read(c2,o,c2.length());
                for(int i=0; i<8; i++)
                s.o.p(c2[i]);
                s.o.p("data available in pis="+pis.available()); */
            }
}
output:-
            data available in pis=11
            data available in pos=19
            25

            data available in pis=14
(because 5 bytes are already read by the pis.read(c1,o,4) & s.o.p(pis.read()); reads one)
```

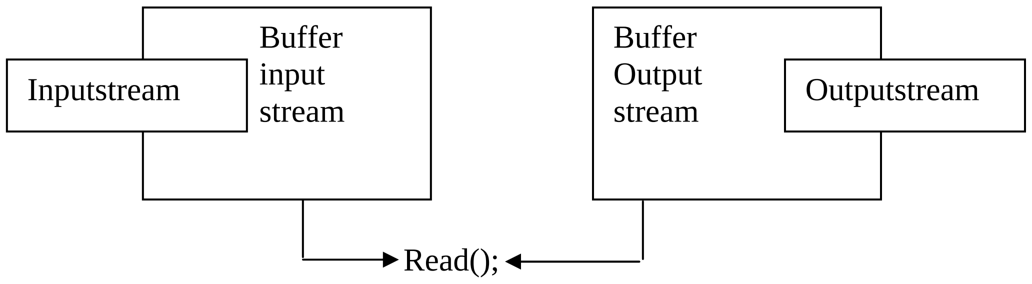
File Buffers: - these are used to speed up the processing of files/file input & output.

Q: - what happens if fileBuffers are not managed by the O/S.

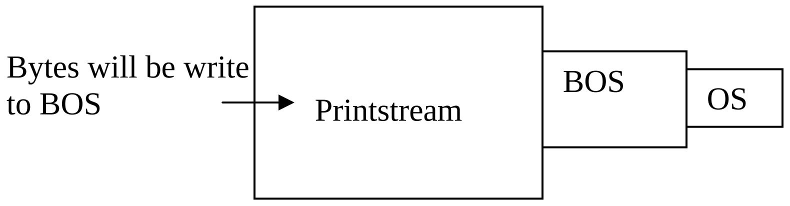


A: -   As shown in the figure, if p1 is storing the data in the disk, and p2 is also storing the data in some other location in the disk. If p1 is again used to store same more data in the previous location, when the disk is rotating. So it will take much time to do this. So, in order to speed up the fileInput/outputs fileBuffers are used. In every o/s uses the fileBuffers in order to special up the file input/outputs. So write the content to the file. the file will not be directly updated, instead of updating the file directly the O/S stores this data written by a program temporarily in a buffer and the O/S updates the disc with the content of these buffers periodically when the program closes the file when we executes a flush method.

Q:- Mark is not supported by the stream, but you want to work with mark in your program. How?
A:- A buffer input /output streams are the sub-classes of filter input /output streams. these streams acts as intermediate buffers(temporary storage) . using this stream you can able to do the mark operation in your program.



Advgs:-
1. inputstream will be read from the BufferInputStream, but not from the disc to speedup the process. BufferInputStream supports mark() method, so you can use this in your program.
2. output should not be writing to the disc immediately.
3. Most of the people say that BIS/BOS 's are used to increase the performance of an application Since the Buffer is used to temporarily collect the data. But in the case of FileInput/output Streams , already the o/s provides buffering mechanism in order to improve the performance of an application.
4.A BufferedInputStream can be used to provide mark operations, For the InputStreams, which does not directly provide the mark operation.

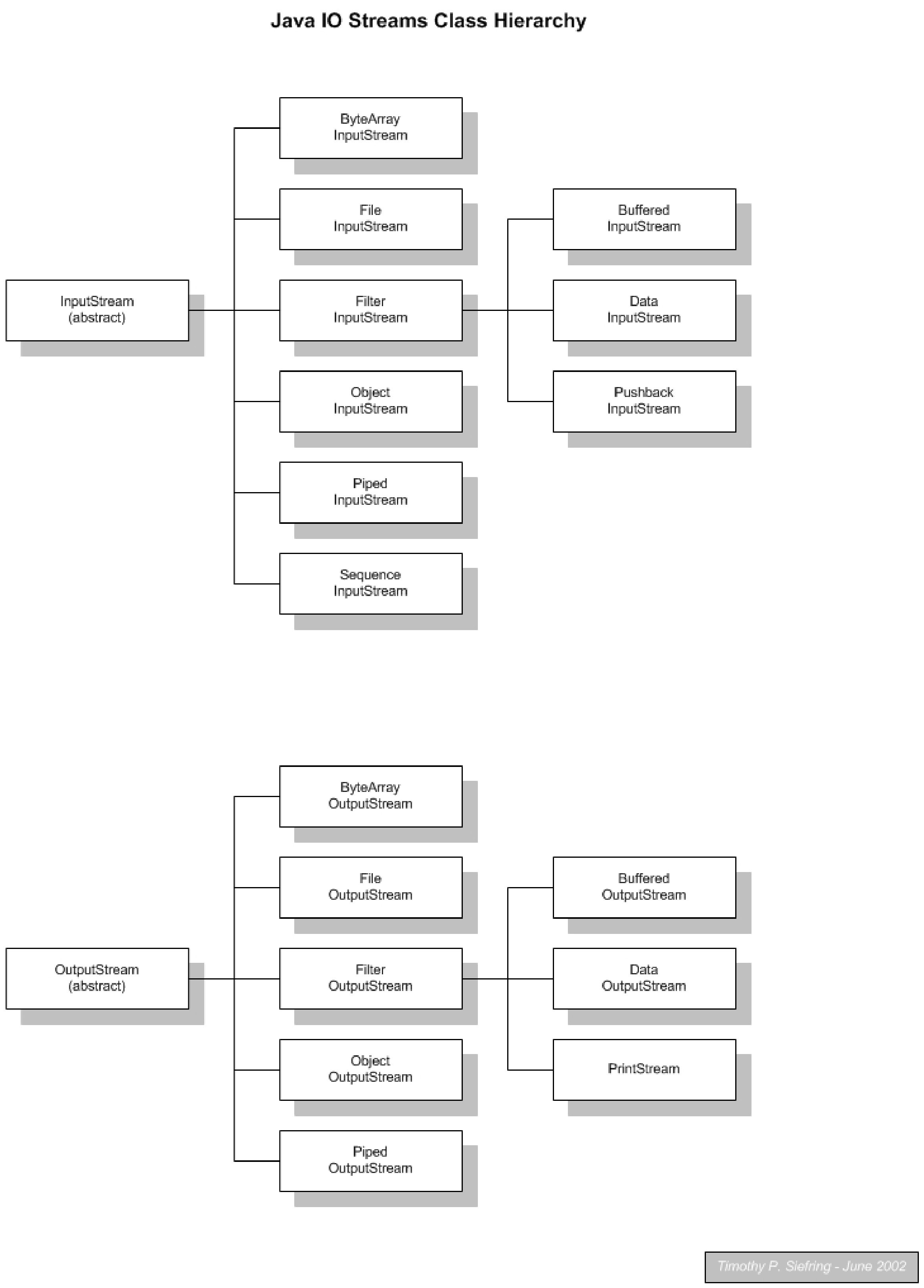


which internally returns to the os?
The output will be in the Buffer. If you use, System.out.flush();-->you will get the output from the buffer.
ex:-

```
class iostream{
        public static void main(String ars[])throws Exception{
                nullop nop=new nullop();
                printStream ps=new printStream(nop);
                System.out.println("Before setting new outstream");
                System.setOut(ps);
                System.out.println("After setting new out stream");
                FileOutputStream out=new FileOutputStream(file.Descriptor.out);
                System.setOut(new printStream(new BufferedOutputStream(out)));
                System.out.println("After  Resulting New out Stream");
                out.close();
               System.out.flush();
 }
}
```

```
// create a FilOutputStream, then use
      PrintStream ps = new PrintStream(fos);
      ps.print(100000);
      fos.close();
// Take the back up of the file, and replace dos
               dod ds =  new dos(fos);
               ds.readInt();
               ds.readFloat();
```
// Observe the outputs of both and these are occurring like this ?

Find the difference between ps & dos ?



Class hierarchy of Input and OutputStreams in java.io

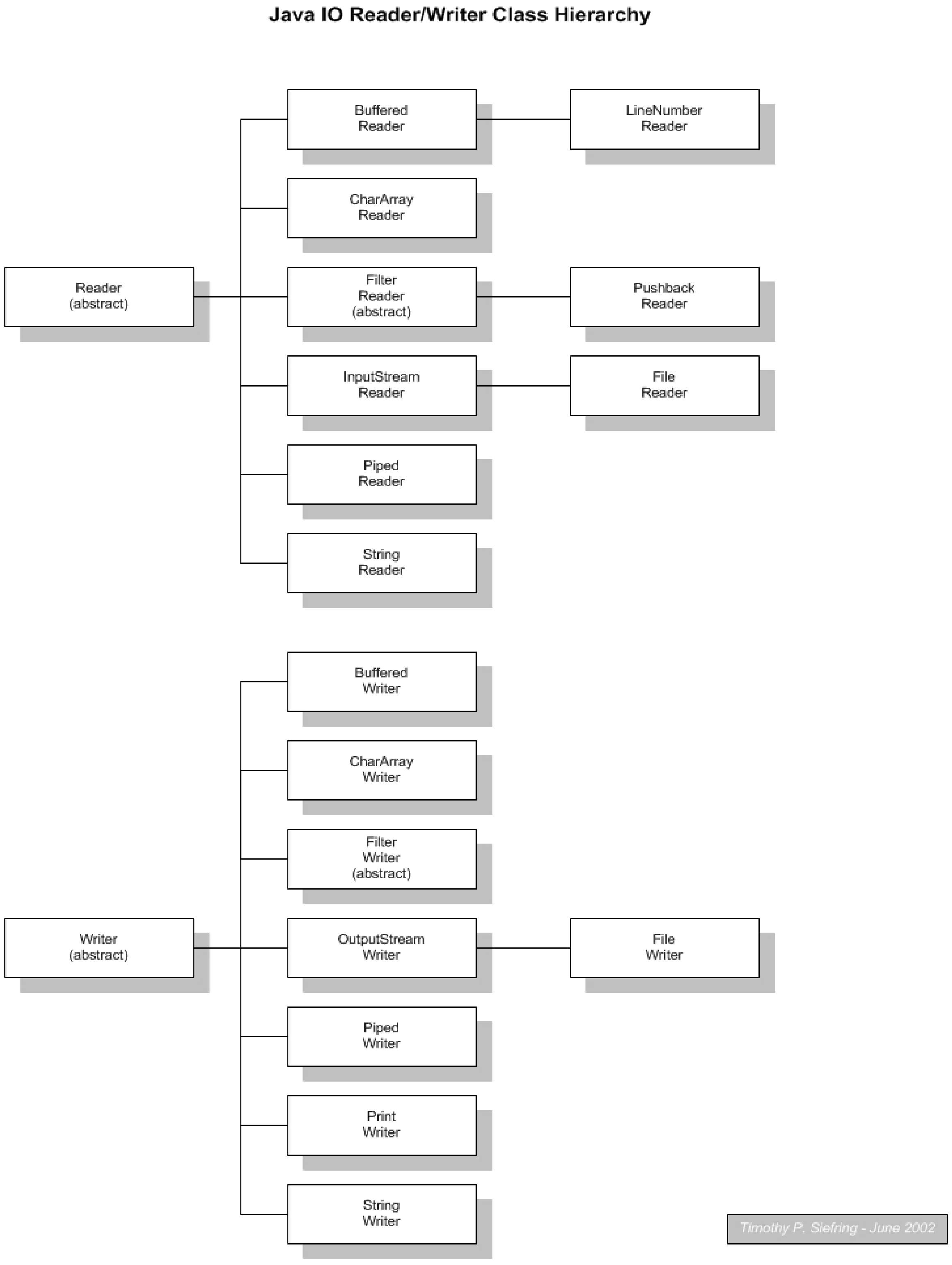
Ex:- /* How to read/write to a Random Access file */

```
            class Raf {
                        public static void main(String args[]) throws Exception {
            RandomAccessFile ras = new RandomAccessFile("ras.txt","rw");
            // We can also pass File & fileDescriptor objects to fis.
            byte b[] = {1,2,3,4,5,65,6,3,33,52,32,2,33,4}; //photo
            writeRecord(ras,"x",100,10000f,b);
            System.out.println("Record pointer is at :"+ras.getFilepointer());
            writeRecord(ras,"y",101,10000f,b);
            System.out.println("Record pointer is at :"+ras.getFilepointer());
            writeRecord(ras,"a",102,10000f,b);
            writeRecord(ras,"b",103,10000f,b);
            writeRecord(ras,"c",104,10000f,b);
            printRecords(ras,1);
            printRecords(ras,2);
            printRecords(ras,20);
            ras.close();
    }

    private static void writeRecord(RandomAccessFile r, String name, int eno,
                                                float sal, byte photo[]) throws Exception {
            r.write UTF(name);
            r.write Int(eno);
            r.write Float(sal);
            r.write UTF(photo,0,photo.length);
            r.write UTF("\n");
     }

    private static void printrecords(RandomAccessFile r, int recno) throws Exception {
       long curpos = r.getFilepointer();
    //set the file pointer to appropriate position
    long newpos = (recno-1)*30;
    r.seek(newpos);
    System.out.println("r.readUTF() + "   ");
    System.out.println("r.readInt() + "   ");
    System.out.println("r.readFloat() + "  ");
    //System.out.println(ras.readBytes(phto,0,photo.length);
    //ras.writeString("\n");
    r.seek(curpos);
   }
  }
```

class hierarchy for Readers/writers in Java.io:-


Java IO Reader/Writer Class Hierarchy
Reader (abstract)
Buffered Reader
LineNumber Reader
CharArray Reader
Filter Reader (abstract)
Pushback Reader
InputStream Reader
File Reader
Piped Reader
String Reader
Writer (abstract)
Buffered Writer
CharArray Writer
Filter Writer (abstract)
OutputStream Writer
File Writer
Piped Writer
Print Writer
String Writer
Timothy P. Siefring - June 2002