Streams and Input/Output Files Part I

Introduction

- So far we have used variables and arrays for storing data inside the programs. This approach poses the following limitations:
  - The data is lost when variable goes out of scope or when the program terminates. That is data is stored in temporary/mail memory is released when program terminates.
  - It is difficult to handle large volumes of data.
- We can overcome this problem by storing data on secondary storage devices such as floppy or hard disks.
- The data is stored in these devices using the concept of Files and such data is often called persistent data.

File Processing

- Storing and manipulating data using files is known as file processing.
- Reading/Writing of data in a file can be performed at the level of bytes, characters, or fields depending on application requirements.
- Java also provides capabilities to read and write class objects directly. The process of reading and writing objects is called object serialisation.

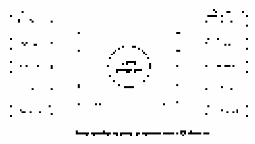
C Input/Output Revision

```
FILE* fp;

fp = fopen(“In.file”, “rw”);
fscanf(fp, ……);
frpintf(fp, …..);
fread(………, fp);
fwrite(……….., fp);
```

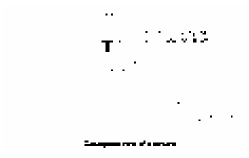
I/O and Data Movement

- The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program.
- The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program.
- Both input and output share a certain common property such as unidirectional movement of data – a sequence of bytes and characters and support to the sequential access to the data.



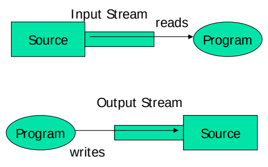
Streams

- Java Uses the concept of Streams to represent the ordered sequence of data, a common characteristic shared by all I/O devices.
- Streams presents a uniform, easy to use, object oriented interface between the program and I/O devices.
- A stream in Java is a path along which data flows (like a river or pipe along which water flows).



Stream Types

- The concepts of sending data from one stream to another (like a pipe feeding into another pipe) has made streams powerful tool for file processing.
- Connecting streams can also act as filters.
- Streams are classified into two basic types:
  - Input Steam
  - Output Stream



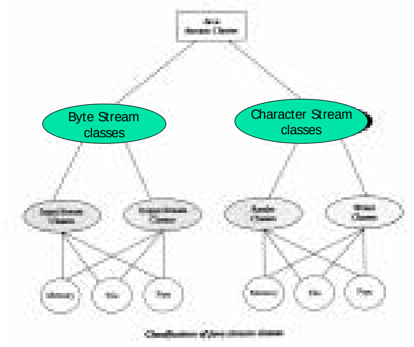
Java Stream Classes

- Input/Output related classes are defined in java.io package.
- Input/Output in Java is defined in terms of streams.
- A *stream* is a sequence of data, of no particular length.
- Java classes can be categorised into two groups based on the data type one which they operate:
  - *Byte streams*
  - *Character Streams*

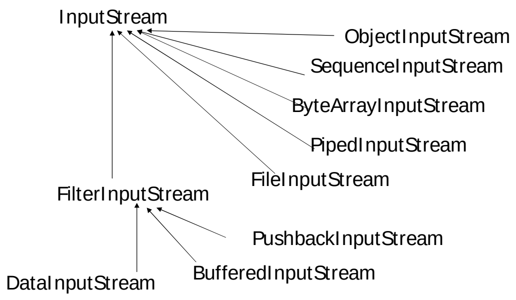
Streams

Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers

Classification of Java Stream Classes



Byte Input Streams



Byte Input Streams - operations

public abstract int read()	Reads a byte and returns as a integer 0-255
public int read(byte[] buf, int offset, int count)	Reads and stores the bytes in buf starting at offset. Count is the maximum read.
public int read(byte[] buf)	Same as previous offset=0 and length=buf.length()
public long skip(long count)	Skips count bytes.
public int available()	Returns the number of bytes that can be read.
public void close()	Closes stream

Byte Input Stream - example

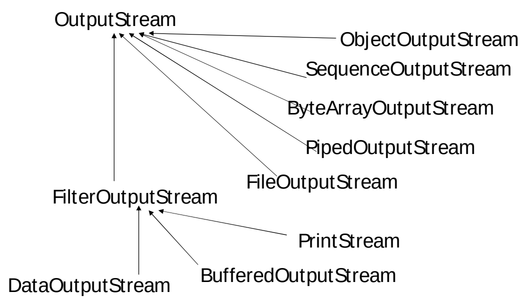
- Count total number of bytes in the file

```
import java.io.* ;

class CountBytes {
        public static void main(String[] args)
            throws FileNotFoundException, IOException
        {
                FileInputStream in;
                in  =  new FileInputStream("InFile.txt");

                int total = 0;
                while (in.read() != -1)
                        total++;
                System.out.println(total + " bytes");
        }
}
```

Byte Output Streams



Byte Output Streams - operations

public abstract void write(int b)	Write *b* as bytes.
public void write(byte[] buf, int offset, int count)	Write *count* bytes starting from *offset* in *buf*.
public void write(byte[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void flush()	Flushes the stream.
public void close()	Closes stream

Byte Output Stream - example

- Read from standard in and write to standard out

```
import java.io.* ;

class ReadWrite {
        public static void main(string[] args)
                throws IOException
        {
                int b;
                while (( b = System.in.read()) != -1)
                {
                        System.out.write(b);
                }
}
```

Summary

- Streams provide uniform interface for managing I/O operations in Java irrespective of device types.
- Java supports classes for handling Input Steams and Output steams via java.io package.
- Exceptions supports handling of errors and their propagation during file operations.