Streams

I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data.

The Java library classes for I/O are divided by input and output. You need to import java.io package to use streams. There is no need to learn all the streams just do it on the need basis.

The concept of "streams"

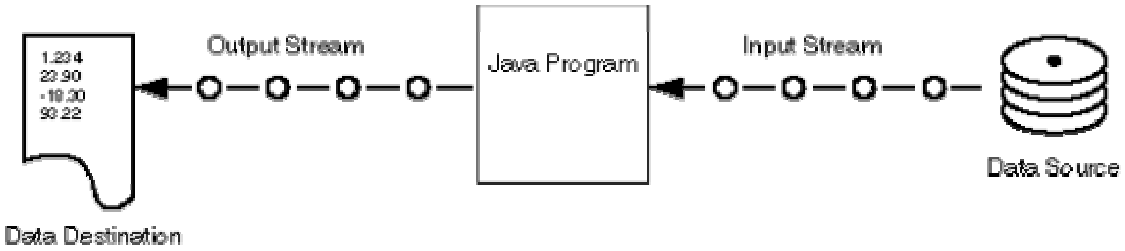
- It is an abstraction of a data source/sink

- We need abstraction because there are lots of different devices (files, consoles, network, memory, etc.). We need to talk to the devices in different ways (sequential, random access, by lines, etc.) Streams make the task easy by acting in the same way for every device. Though inside handling of devices may be quite different, yet on the surface everything is similar. You might read from a file, the keyboard, memory or network connection, different devices may require specialization of the basic stream, but you can treat them all as just "streams". When you read from a network, you do nothing different than when you read from a local file or from user's typing

```
//Reading from console
BufferedReader stdin =  new BufferedReader(new InputStreamReader(
                                                    System.in ));
                                          -------- ( your console)
// Reading from file
BufferedReader br=new BufferedReader(new FileReader("input.txt"));

//Reading from network
BufferedReader br = new BufferedReader(new InputStreamReader
                                           (s.getInputStream()));
                                         ---- "s" is the socket
```

- So you can consider stream as a data path. Data can flow through this path in one direction between specified terminal points (your program and file, console, socket etc.)



Stream classification based on Functionality

Based on functionality streams can be categorized as Node Stream and Filter Stream. Node Streams are those which connect directly with the data source/sick and provide basic functionality to read/write data from that source/sink

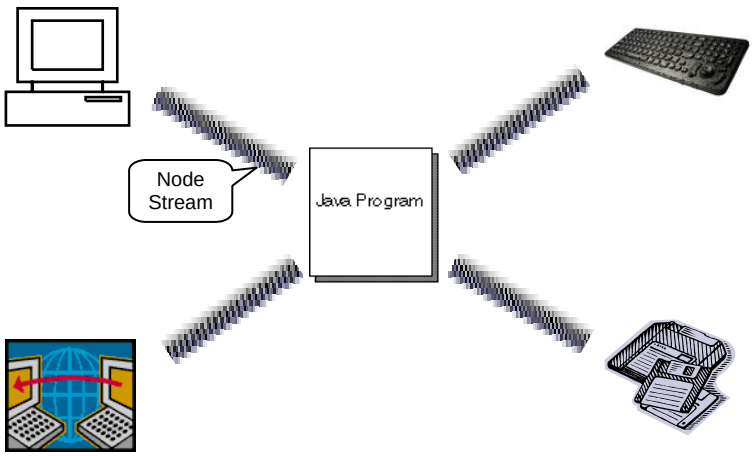
FileReader fr = new FileReader("input.txt");

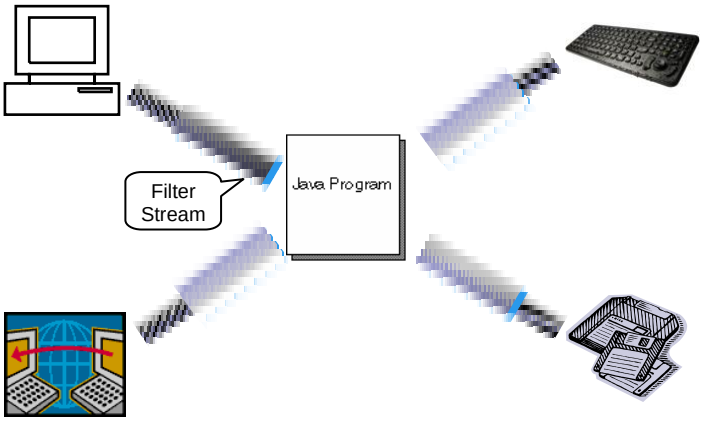
You can see that FileReader is taking a data/source "input.txt" as its argument and hence it is a node stream.

FilterStreams sit on top of a node stream or chain with other filter stream and provide some additional functionality e.g. compression, security etc. FilterStreams take other stream as their input.

BufferedReader bt = new BufferedReader(fr);

BufferedReader makes the IO efficient (enhances the functionality) by buffering the input before delivering. And as you can see that BufferedReader is sitting on top of a node stream which is FileReader..



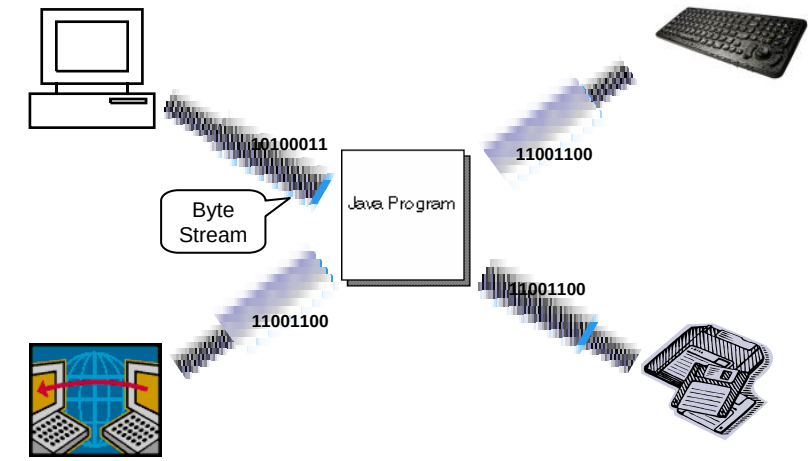

Filter
Stream
Java Program

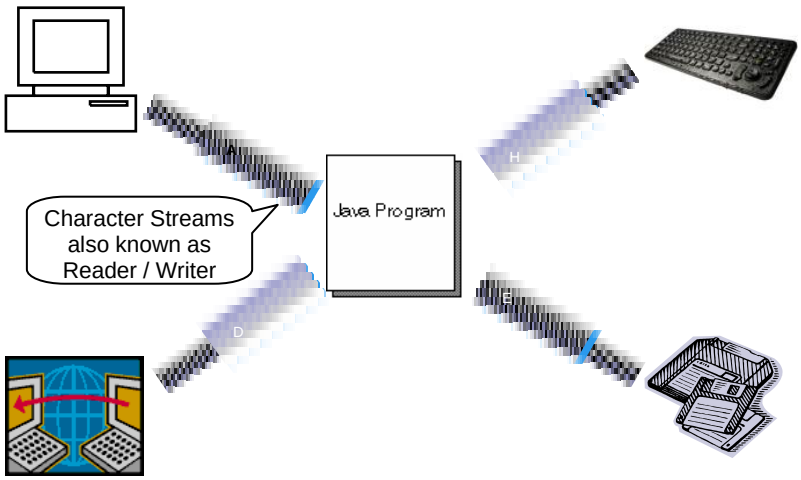
Stream classification based on data

Two type of classes exists.

Classes which contain the word stream in their name are byte oriented and are here since JDK1.0. These streams can be used to read/write data in the form of bytes. Hence classes with the word stream in their name are byte-oriented in nature. Examples of byte oriented streams are FileInputStream, ObjectOutputStream etc.

Classes which contain the word Reader/Writer are character oriented and read and write data in the form of characters. Readers and Writers came with JDK1.1. Examples of Reader/Writers are FileReader, PrintWriter etc





Example Code 8.1: Reading from File

The `ReadFileEx.java` reads text file line by line and prints them on console. Before we move on to the code, first create a text file (*input.txt*) using notepad and write following text lines inside it.

Text File: input.txt

```
Hello World
Pakistan is our homeland
Web Design and Development
```

```
// File ReadFileEx.java

import java.io.*;

public class ReadFileEx {

   public static void main (String args[ ]) {

     FileReader fr = null;
     BufferedReader br = null;

    try {

       // attaching node stream with data source
       fr = new FileReader("input.txt");

       // attatching filter stream over node stream
       br = new BufferedReader(fr);


       // reading first line from file
       String line = br.readLine();

       // printing and reading remaining lines
       while (line != null){
           System.out.println(line);
           line = br.readLine();
       }

       // closing streams
       br.close();
       fr.close();

    }catch(IOException ioex){
       System.out.println(ioex);
    }
  } // end main
    } // end class
```

Example Code 8.2: Writing to File

The `WriteFileEx.java` writes the strings into the text file named "output.txt". If "output.txt" file does not exist, the java will create it for you.

```
// File WriteFileEx.java

import java.io.*;

public class WriteFileEx {

   public static void main (String args[ ]) {

     FileWriter fw  = null;
     PrintWriter pw = null;

    try {

       // attaching node stream with data source
       // if file does not exist, it automatically creates it
       fw = new FileWriter ("output.txt");

       // attatching filter stream over node stream
       pw = new PrintWriter(fw);

       String s1 = "Hello World";
       String s2 = "Web Design and Development";

       // writing first string to file
       pw.println(s1);

       // writing second string to file
       pw.println(s2);

       // flushing stream
       pw.flush();

       // closing streams
       pw.close();
       fw.close();

    }catch(IOException ioex){
       System.out.println(ioex);
    }
  } // end main
    } // end class
```

After executing the program, check the output.txt file. Two lines will be written there.