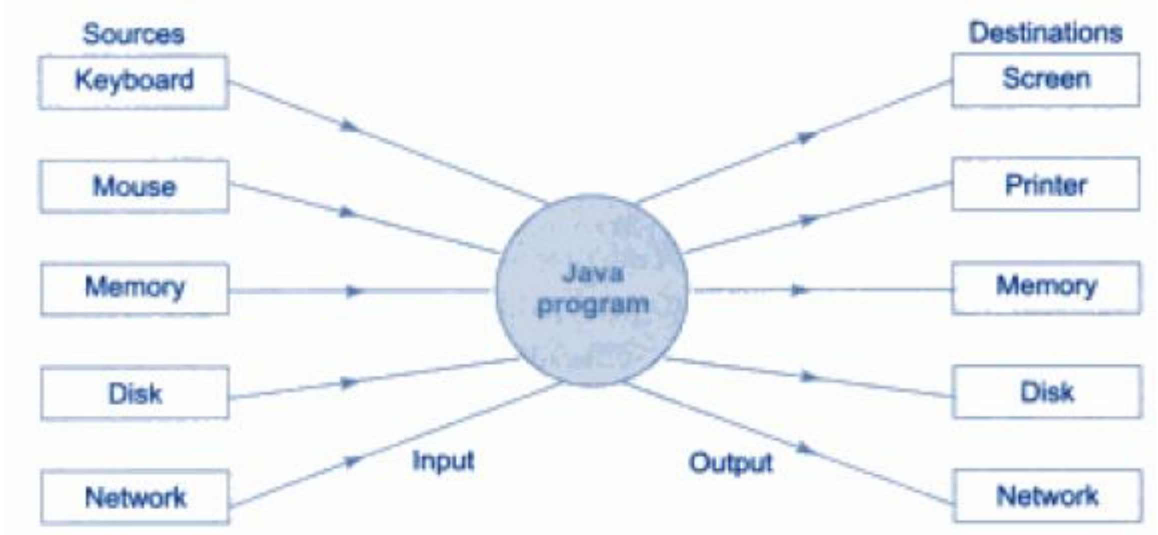
I/O in Java

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. Characters in Java are Unicode characters composed of two bytes, each byte containing eight binary digits, 1 or 0.

Storing and managing data using files is known as file processing which includes task such as creating files, updating files and manipulation of data.

Concept of Streams

In file processing, input refers to the flow of data into a program and output means the flow of data out of a program. Input to a program may come from the keyboard, the mouse, the memory, the disk or another program. Similarly, output from a program may go to the screen, the printer, the memory, the disk or another program as shown below figure:



Relationship of Java program with I/O devices

Java uses the concept of streams to represent the ordered sequence of data, a common characteristics shared by all the input/output devices.

A stream in Java is a path along which data flows. It has a source and a destination. Both the source and the destination may be physical devices or programs or other streams in the same program.

Java streams are classified into two basic types, namely, input stream and output stream. An input stream extracts (i.e. reads) data from the source (file) and sends it to the program. Similarly, an output stream takes data from the program and sends (i.e writes) it to the destination (file). Below figure illustrate the use of input and output streams. The program connects and opens an input stream on the data source and then reads the data serially. Similarly, the program connects and opens an output stream to the destination place of data and writes data out serially. In both the cases, the program does not know the details of end points. (i.e. source and destination)

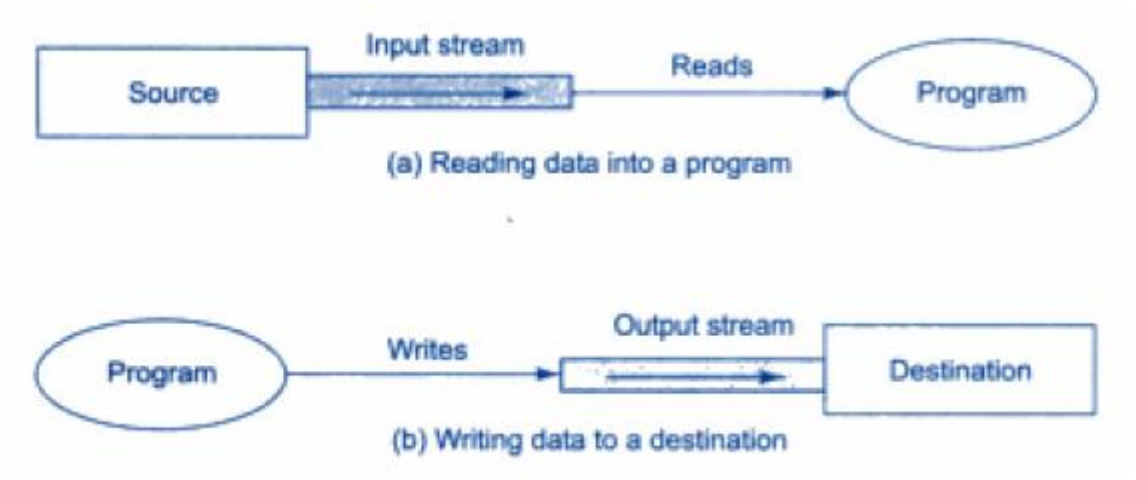


Figure: Using input and output stream

Stream Classes

The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes are categorized into two groups based on the data type on which they operate.

1. **Byte Stream classes:** It provide support for handling I/O operations on bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

   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);

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);

2. **Character stream classes:** It provide support for managing I/O operations on characters. Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

Stream Tokenizer

The Java.io.StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time. The stream tokenizer can recognize identifiers, numbers, quoted strings, and various comment styles.

Class declaration

Following is the declaration for Java.io.StreamTokenizer class −

public class StreamTokenizer extends Object

Buffered Streams

Most of the examples we've seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how we use buffered I/O:

```
inputStream = new BufferedReader(new FileReader("abc.txt"));

outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

Print Stream

A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Unlike other output streams, a PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created so as to flush automatically.
All characters printed by a PrintStream are converted into bytes using the platform’s default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes.

Class declaration

```
public class PrintStream
        extends FilterOutputStream
                implements Appendable, Closeable
```

RandomAccessFile

RandomAccessFile encapsulates a random-access file. It is not derived from InputStream or OutputStream. Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods. It also implements the Closeable interface. RandomAccessFile is special because it supports positioning requests—that is, we can position the file pointer within the file. It has these two constructors:

```
RandomAccessFile(File fileObj, String access)
        throws FileNotFoundException

RandomAccessFile(String filename, String access)
        throws FileNotFoundException
```

In the first form, fileObj specifies the name of the file to open as a File object. In the second form, the name of the file is passed in filename. In both cases, access determines what type of file access is permitted. If it is “r”, then the file can be read, but not written. If it is “rw”, then the file is opened in read-write mode. If it is “rws”, the file is opened for read-write operations and every change to the file’s data or metadata will be immediately written to the physical device. If it is “rwd”, the file is opened for read-write operations and every change to the file’s data will be immediately written to the physical device.

The method seek( ) is used to set the current position of the file pointer within the file:

void seek(long newPos) throws IOException

Here, newPos specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to seek( ), the next read or write operation will occur at the new file position.

RandomAccessFile implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods.

One is setLength( ). It has this signature:

void setLength(long len) throws IOException

This method sets the length of the invoking file to that specified by len.

String Class

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

There are two ways to create string in Java:

String literal

String s= “Assam University, Silchar”;

Using new keyword:

String s= new String (“Assam University, Silchar”);

String Buffer Class

StringBuffer is a peer class of **String** that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.

StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- **StringBuffer( ):** It reserves room for 16 characters without reallocation.

  StringBuffer s=**new** StringBuffer();

- **StringBuffer(int size)**It accepts an integer argument that explicitly sets the size of the buffer.

  StringBuffer s=**new** StringBuffer(20);

- **StringBuffer(String str):** It accepts a **String** argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

  StringBuffer s=**new** StringBuffer("GeeksforGeeks");

String Buffer Methods

Some of the most used methods are:

- length( ) and capacity( ): The length of a StringBuffer can be found by the length( ) method, while the total allocated capacity can be found by the capacity( ) method.
- append( ): It is used to add text at the end of the existence text. Here are a few of its forms:

  StringBuffer append(String str)

  StringBuffer append(int num)

- insert( ): It is used to insert text at the specified index position. These are a few of its forms:

StringBuffer insert(int index, String str)

StringBuffer insert(int index, char ch)

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

- reverse( )**:** It can reverse the characters within a StringBuffer object using **reverse( ).**This method returns the reversed object on which it was called.

- delete( ) and deleteCharAt( ): It can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).The delete( ) method deletes a sequence of characters from the invoking object. The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.These methods are shown here:

StringBuffer delete(int startIndex, int endIndex)

StringBuffer deleteCharAt(int loc)

- replace( ): It can replace one set of characters with another set inside a StringBuffer object by calling replace( ). The substring being replaced is specified by the indexes start Index and endIndex. Thus, the substring at start Index through endIndex–1 is replaced. The replacement string is passed in str. The resulting StringBuffer object is returned.Its signature is shown here:

StringBuffer replace(int startIndex, int endIndex, String str)