Unit 4: I/O Files in Java

4

Unit Structure

4.1 LEARNING OBJECTIVE

After learning this unit, students,will be able to:

- Define streams
- Describe the use of character streams
- Describe the use of byte streams
- Describe RandomAccessFile, StreamTokenizer
- Access File

4.2 OUTCOMES

After learning the contents of this chapter, the students will be able to:

- Define Streams
- Differentiate byte stream and character stream
- Implement buffered based input and output operation apart from other important stream classes like object input and output, data input and output, piped input and output etc.
- Implement File handling operation to read and write content from and to the file.
- Perform random read and write operation on the file

4.3 INTRODUCTION

Java I/O stands for Java Input / Output and is contained in java.io package. This package has an Input Stream and Output Stream classes. Input Stream classes are used for reading the stream, byte stream and array of byte stream. This can be used for memory allocation. The Output Stream classes are used for writing byte and array of bytes.

In this chapter, we are going to dicuss and learn the use of streams that can handle all kinds of data including primitive values to advanced objects.

4.4 CONCEPTS OF STREAM

Streams are the sequence of data or information. The other streams help in adding capabilities, like the ability to read a whole chunk of data at once for performance reasons (BufferedInputStream) or converting data from one kind of character set to Java's native unicode (Reader), or where the data is coming from (FileInputStream, SocketInputStream and ByteArrayInputStream, etc.).

Some input-output stream initialized automatically by the JVM and these streams are available in System class. These strems are,

1. System.out: it is a standard output stream. It refers to the default output device, i.e. console.
2. System.in: It is a standard input stream. It refers the default input device, i.e. keyboard.
3. System.err: It is a standard error stream. It refers to the default output device, i.e. console.

Two types of streams are there, Input Streams and Output Streams.

- **Input Streams**: It is used to read the data from different input devices like keyboard, file, network etc.
- **Output Streams**: It is used to write the data to different output devices like monitor, file, network etc.

Based on data, streams are divided in two types:

1. **Byte Stream**: Byte stream performs input and output on 8-bit bytes. Byte stream classes are used to read or write byte data. InputStream is used to read and Output Stream is used to write byte data. InputStream and Output Stream class are abstract classes and they are the super classes of all the input byte streams and output byte streams.
2. **Character Stream**: Character stream is used to read and write data in 16 bit Unicode characters. These classes are used for reading or writing character data. Reader and Writer are abstract classes.

- **Exceptions Handling during I/O in Java**

Exception is an abnormal condition and it must be avoided. In java IO almost all input or output method throws an exception. Therefore, it is required to enclose I/O operation in the try and catch block. All the I/O exceptions are derived from IOException class. Generally you can catch IOException a super class, which will catch all the derived class exceptions. For some exceptions thrown by I/O which are not in super class, we have to take extra care to catch them while wrting IO programs.

4.5 DIFFERENCE BETWEEN CHARACTERSTREAMS AND BYTESTREAMS

By definition:

Character Stream performs input and output operations of 16-bit Unicode while Byte Stream performs input and output of 8-bit bytes.

By use:

Character stream is used to read character either from Socket or text file. Byte streams should only be used for the primitive I/O.

By datatype:

Character oriented streams can read only string type or character type while byte oriented streams are not tied to any datatype. Data of any datatype can be read in byte stream (except string).

By access:

Character oriented stream reads character by character while Byte oriented stream reads byte by byte.

By encoding:

Character oriented streams use character encoding scheme (UNICODE) while byte oriented do not use any encoding scheme.

By associated classes:

Character oriented streams are reader and writer streams while Byte oriented streams are data streams i.e. Data input stream and Data output stream.

- **Check Your Progress 1**

1) Divide the classes in Low level vs High Level to read / write data from files.

……

……

2) Define Stream, Readers / Writers and Buffer.

……

……

3) Differentiate Byte Stream and Character Stream.

……

……

4.6 CHARACTER STREAMS

Character Stream contains classes that are used to read characters from the source file and write characters to destination file. The following table depicts different classes for Character Streams.

Stream class	Description
Reader	This class is an abstract class that define character stream input.
Writer	This class is an abstract class that define character stream output.
BufferedReader	This class handles buffered input stream. - LineNumberReader is extends BufferedReader
BufferedWriter	This class handles buffered output stream.

FileReader	This class handles input stream that reads from file. It extends InputStreamReader
FileWriter	This class handles output stream that writes to file. It extends OutputStreamWriter
InputStreamReader	This class handles input stream that translate byte to character
OutputStreamWriter	This class handles output stream that translate character to byte.
PrintWriter	This class handles output Stream that contain print() and println() method.
FilterReader	This class is used to perform filtering operation on reader stream. It is an abstract class. - PushBackReader class extends FilterReader
FilterWriter	This class is an abstract class used to write filtered character streams.
CharArrayReader	This class is consists of two words: CharArray and Reader. It is used to read character array as a reader (stream). It extends Reader class.
CharArrayWriter	This class is used to write common data to multiple files. This class extends Writer class.
PipedReader	This class is used to read the contents of a pipe as a stream of characters. It is used generally to read text.
PipedWriter	This class is used to write data to a pipe as a stream of characters. It is used generally for writing text.
StringReader	This class is a character stream with string as a source. It accepts an input string and changes it into character stream. It extends Reader class.

StringWriter	This class is a character stream that collects output from string buffer, which can be used to construct a string. The StringWriter class extends the Writer class.

Table-12 Classes for Character Streams

There are two types of Character Stream classes: Reader and Writer classes.

1. Reader Classes:

These classes are subclasses of an abstract class Reader and they are used to read characters from a source like file, memory or console. Being abstract class we can't create its object but we can use its subclasses for reading characters from the input stream.

2. Writer Classes:

These classes are subclasses of an abstract class Writer and they used to write characters to a destination like file, memory or console. Being abstract class we can't create its object but we can use its subclasses for writing characters to the output stream.

The main methods for reading from and writing to character streams found in reader and writer classes and their child classes are given below:

- int read()
- int read(char cbuff[ ])
- int read(char cbuff[ ], int offset, int length)
- int write(int ch)
- int write(char cbuff[ ])
- int write(char cbuff[ ], int offset, int length)

4.6.1 Inputstreamreader Class And Outputstreamwriter

InputStreamReader class is wrapped around an inputstream to read data in the form of characters from it, so InputStreamReader class acts as a converter of bytes to characters.

Constructor:

InputStreamReader (InputStream inst)

This constructor creates an InputStreamReader object wrapped around an InputStream to read data from it in the form of characters.

Example:

```
FileInputStream fis = new FileInputStream("D://Test.txt");

InputStreamReader isread = new InputStreamReader(fis);
```

Here, In this example we have wrapped an InputStream i.e. FileInputStream, inside InputStreamReader. FileInputStream class reads data from a file Test.txt as bytes and then this data is converted to characters, when it is read using InputStreamReader class.

- **OutputStreamWriter**

OutputStreamWriter class is a subclass of Writer class. Using OutputStreamWriter class allows us to convert a character, character arrays or a String to bytes before it is written to an output stream. OutputStreamWriter class works as a converter of characters to bytes.

Constructor:

```
OutputStreamWriter (OutputStream outstr)
```

This constructor creates an OutputStreamWriter object wrapped around an OutputStream to write data to this OutputStream in the form of bytes.

Example-:

```
char data[ ] ={‘B’, ‘A’, ‘O’, ‘U’};

FileOutputStream fostm = new FileOutputStream(“D://Test.txt”);

OutputStreamWriter oswt = new OutputStreamWriter (fostm);
```

Here, FileOutputStream object is wrapped inside the OutputStreamWriter. Only bytes can be written through FileOutputStream. OutputStreamWriter class will first convert the characters in a character array data, to bytes before writing them to a file Test.txt using FileOutputStream.

Now, we will try to understand read and write operation with help of programs.

// Program to write a String and character array using OutputStreamWriter and reading back the same file using InputStreamReader.

```
import java.io.*;
public class Outstreamwriter
{
	public static void main(String[ ] arg)
	{
		String str=" BAOU";
		char[] arrdata= {'V','I','D','Y','A','P','I','T','H'};
		try
		{
			FileOutputStream fos= new FileOutputStream("Test1.txt");
			OutputStreamWriter osw= new OutputStreamWriter(fos);

			// writing each character of character array using for-each loop
				for(char ch : arrdata)
				{
					osw.write(ch);
				}
			//writing a String
			osw.write(str);
			osw.flush();
			osw.close();
			}
		catch(IOException e)
		{
			System.out.println(e);
		}
		try
		{
			FileInputStream fis= new FileInputStream("Test1.txt");
			InputStreamReader isr= new InputStreamReader(fis);
			int data;
```

```
                        while( (data=isr.read())!=-1)
                        {
                                System.out.print((char)data);
                        }
                }
                catch(IOException e)
                {
                        System.out.println(e);
                }
        }
}
```

This program will create a file named Test1.txt, writes a character array first and then string in to it. After that, the same file is read by using InputStreamReader. The out put of the program is shown below.

Output:

VIDYAPITH BAOU

4.6.2 BufferedReader and BufferedWriter

BufferedReader and BufferedWriter class in java are classified as buffered I/O streams. Buffered input stream reads text from a memory area i.e. buffer and buffered output stream writes data to a buffer. For unbuffered Input and Output stream, every read or write request is handled directly by the underlying Operating System. This makes a program less efficient as every request involves disk access, network activity etc. So, it is adviced to use buffered I/O streams as opposed to Scanner and PrintWriter classes. The buffer size may be specified. If not specified then the default size will be used.

BufferedReader and BufferedWriter achieve greater efficiency through the use of buffers. A data buffer is generally a temporarily a region in memory. BufferedWriter doesn't write on a file directly, rather, it stores data in a buffer and writes it onto the file when you want it to execute a flush operation. Flushing tells the BufferedWriter to write everything onto the output file. Use of a buffer is what makes both BufferedReader and BufferedWriter fast and efficient.

BufferedReader Constructors

1. BufferedReader (Reader rd)
   This constructor allows to create a buffering input stream that uses a default size for input buffered.
2. BufferedReader (Reader rd, int size)
   This constructor allows to create a buffering input stream that uses a specified size for input buffered.

Example:

```
FileReader readfile = new FileReader("Test.txt");
BufferedReader bufread = new BufferedReader(readfile);
```

Above example will buffer the input from the specified file. Without buffering, each call to read() or readLine() method could cause bytes to be read from the file, converted into characters, and then returned. This can be very inefficient.

A **BufferedWriter** writes text to a character-output stream, while buffering characters to provide for the efficient writing of single characters, strings and arrays. Unlike byte stream (convert data into bytes), bufferwriter writes the strings, arrays or character data directly to a file.

Constructors:

1. BufferedWriter(Writer wout):
   This allows us to create a buffered character-output stream that uses a default sized output buffer with specified Writer object.
2. BufferedWriter(Writer wout, int sz):
   This allows us to create a buffered character-output stream that uses an output buffer of specified size with specified Writer object.

```
// Program to writes a data in a file using BufferedWriter and reads the content back from the same file using BufferedReader.

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
```

```
import java.io.FileWriter;
import java.io.IOException;

public class buferReadWriter
{
	public static void main(String[] args)
	{
		File buffile = new File("buff.txt");

		/*Writing file using BufferedWriter*/
		FileWriter filewrite = null;
		BufferedWriter buffwrite =null;
		try {
			filewrite=new FileWriter(buffile);
			buffwrite =new BufferedWriter(filewrite);
			buffwrite.write("Babasaheb Ambedkar Open University \n");
			buffwrite.write("Gujarat Vidyapith \n");
			buffwrite.write("Dept. of Computer Science");
			buffwrite.flush();
		} catch (IOException ioe)
		{
			System.out.println(ioe);
		}
		finally {
			try {
				if(filewrite!=null){
					filewrite.close();
				}
				if(buffwrite!=null){
					buffwrite.close();
				}
			} catch (IOException ioe) {
				System.out.println(ioe);
			}
		}
```

```
                /*Reading file using BufferedReader*/
                FileReader fileread=null;
                BufferedReader buffRead=null;
                try {
                        fileread =new FileReader(buffile);
                        buffRead=new BufferedReader(fileread);
                        String data=null;
                        while((data=buffRead.readLine())!=null){
                                System.out.println(data);
                        }
                } catch (IOException ioe) {
                        System.out.println(ioe);
                }finally {
                        try {
                                if(fileread!=null){
                                        fileread.close();
                                }
                                if(buffRead!=null){
                                        buffRead.close();
                                }
                        } catch (IOException ioe)
                        {
                                System.out.println(ioe);
                        }
                }
        }
}
```

This program creates a file named buff.txt, writes a few data in to it. After that, the same file is read by using BufferedReader. The out put of the program is shown below.

Output:

Babasaheb Ambedkar Open University

4.6.3 PipedWriter and PipedReader

PipedWriter and PipedReader classes are connected to each other to create a communication link called pipe. PipedWriter and PipedReader class works on character output and input stream. PipedWriter is the Sending end while PipedReader is the receiving end. If pipe is broken, IOException will be thrown. The pipe reader and pipe writer are connected with each other but both are processed by two different threads.

PipedReader Constructor:

1. PipedReader()
   This constructor allows us to create the piped reader object.
2. PipedReader(int pSize)
   This constructor allows us to create the piped reader object with specified size of buffer or pipe.
3. PipedReader(PipedWriter src, int pSize)
   This constructor allows us to create the piped reader object with specified size of buffer or pipe with the specified connection to piped writer instance.

PipedWriter Constructor:

1. PipedWriter()

   This constructor allows us to create the piped writer object and not connected with piped reader.

2. PipedWriter(PipedReader pread)

   This constructor allows us to create the piped writer object which is connected to the specified piped reader instance.

```
import java.util.*;
import java.io.*;
public class PipeThreadRdWtr {
```

```
  public static void main(String[] args) throws Exception {
    PipedWriter owner = new PipedWriter();
    PipedReader user = new PipedReader(owner);

    DigitOwner dit = new DigitOwner(owner);
    DigitUser du = new DigitUser(user);

    dit.start();
    du.start();
  }
}
class DigitOwner extends Thread{
  BufferedWriter bw;
  public DigitOwner(Writer w)
  {
    this.bw = new BufferedWriter(w);
  }

  // thread continually generates random votes
  public void run() {
    try {
      Random r = new Random();
      while (true) {
        String vote = "" + Math.abs((r.nextInt() % 10));

        bw.write(vote);
        bw.newLine();
        bw.flush();
          sleep(20);
      }
    }
    catch(IOException e) {
      System.err.println(e);
    }
    catch(InterruptedException e) {
```

```
        System.err.println(e);
      }
   }
}
class DigitUser extends Thread{
   BufferedReader br;
   int[] votes = new int[10];

   public DigitUser(Reader r) {
      br = new BufferedReader(r);
   }
     public void run() {
      try {
         String data;
         int count = 0;
         while ((data = br.readLine()) != null) {
            int member = Integer.parseInt(data);
            votes[member]++;
            count++;
            if (count % 100 == 0){
                        for (int i=0; i<votes.length; i++)
                        {
                                System.out.println("Member ->" + i + ": " + votes[i]);
                        }
                        System.out.println("****");
              }
         }
      }
      catch(IOException e) {
         System.err.println(e);
      }
   }
}
```

In this example, one thread takes off the behavior of scores for 10 members by generating random numbers between 0 to 10. Another thread keeps track of the total votes per members. The output of the program is shown below.

Output:

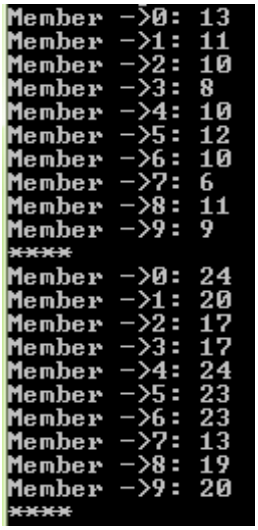

Figure-125: Output of program

- **Check Your Progress 2**

1) Define Filter Stream.

……

……

2) Write a code to append the new content to the end of a file using PrintWriter.

……

……

3) What is the functionality of SequenceInputStream?

……

……

4.7 BYTE STREAMS

There are various important classes' falls under the umbrella of Byte Streams.

Stream class	Description
InputStream	This class is an abstract class that describe stream input. This is a super class of all InputStream class.
OutputStream	This class is an abstract class that describe stream output. This is a super class of all OutputStream class.
FileInputStream	This class is used for Input stream that reads from a file.
FileOutputStream	This class is used for Output stream that write to a file.
FilterInputStream	This class contains different sub classes as BufferedInputStream, DataInputStream, LineNumberInputStream and PushBackInputStream for providing additional functionality.
FilterOutputStream	This class provides different sub classes such as BufferedOutputStream and DataOutputStream and PrintStream to provide additional functionality.
SequenceInputStream	This class is used to read data from multiple streams. It allows us to reads data sequentially (one by one).
ByteArrayInputStream	This class is cosnsists of two words: ByteArray and InputStream. it can be used to read byte array as input stream.It contains an internal buffer which is used to read byte array as stream. The data is read from a byte array.
ByteArrayOutputStream	This class is used to write common data into multiple files. The data is written into a byte array and it will be written to multiple streams lateron. It contains a copy of data and forwards it to multiple streams.
ObjectInputStream	This class is used to read the primitive data type and Java object from an input stream.
ObjectOutputStram	This class is used to store the primitive data type and

	Java object to an output stream. Those objects whose class implements java.io.Serializable interface are written to stream.
PipedInputStream PipedOutputStream	Both classes can be used to read and write data simultaneously. Both streams are connected with each other using the connect() method of the PipedOutputStream class.
StringBufferInputStream	This class helps in creating an Input Stream where, one can read bytes from the string. We can only read lower 8 bits of each character present in the string. This class has been deprecated.
PrintStream	This class is used for Output Stream that contain print() and println() method

Table-13 Classes for Byte Streams

ByteStream contains classes that are used to read bytes from the source file and write bytes to destination file. There are two types of Byte Stream classes: Input and Output stream classes.

Byte Streams can be used for all types of files except Strings or text files.

4.7.1 FileInputStream and FileOutputStream

FileInputStream class is used to read the data from file. It is used for reading streams of raw byte. The FileInputStream class establishes the connection with the disk file.

Constructors:

1. FileInputStream(File bytefile)

   This constructor allows us to create a FileInputStream object to read a file specified by the File object.

Example:

File bytefile= new File(“D:\\Test.txt”);

FileInputStream fis= new FileInputStream(bytefile);

2. FileInputStream(String filepath)

   This constructor allows us to create a FileInputStream to read a file which is accessed by the path specified in the argument of this constructor.

   Example:

   FileInputStream fis= new FileInputStream(“D:\\Test.txt”);

Both the above constructors have created a FileInputStream object to create an input stream to read a file called “Test.txt” which is located in the D drive.

FileOutputStream class is used for writing the data to a File.

Constructor:

1. FileOutputStream(File bytefile)

   This constructor allows us to create a FileInputStream object to read a file specified by the File object.

   Example:

   File bytefile = new File(“D:\\Test.txt”);

   FileInputStream fis= new FileInputStream(bytefile);

2. FileOutputStream(String filepath)

   This constructor allows us to create a FileOutputStream to write to a file which is accessed by the path specified in the argument of this constructor.

   Example:

   FileOutputStream fis= new FileOutputStream(“D:\\Test.txt”);

   Both the above constructors have created a FileInputStream object to create an input stream to read a file called “Test.txt” which is located in the D drive.

```
//Program to write to and read from the file
import java.io.*;
class fileInoutStream{
```

```
        public static void main(String args[])
        {
                FileInputStream fin;
                BufferedReader br = null;
                try
                {
                        //Writing in to file
                         FileOutputStream fout=new FileOutputStream("foutest.txt");
                        fout.write(50);
                        fout.write('V');
                        fout.write('D');
                           fout.close();
                        //Reading from the file
                        fin=new FileInputStream("foutest.txt");
                        BufferedInputStream bin=new BufferedInputStream(fin);
                        int i;
                        while((i=bin.read())!=-1)
                        {
                                System.out.print((char)i);
                        }
                        bin.close();
                        fin.close();
                }
                catch(Exception ex)
                {
                        System.out.println(ex);
                }
        }
}
```

In the above program first we are writing the character / byte in to the file through FileOutputStream. After that we reads the same file by wrapping the FileinputStream in BufferedInputStream and displays the content.

Output:

2VD

4.7.2 DataInputStream and DataOutputStream

InputStream classes always reads data in the form of bytes but DataInputStream class is used to read data in the form of primitive data types such as char, int, float, double, Boolean, short from an input stream. This class is a filter class used to wrap any input stream to read primitive data types out of it. It is a subclass of FilterInputStream class which in turn is a subclass of InputStream class.

Constructor:

DataInputStream(InputStream dis)

This constructor takes an InputStream object dis as its argument to read data out of this input stream.

Example:

```
FileInputStream disfis=new FileInputStream("D://Test.txt");
DataInputStream disread =new DataInputStream(disfis);
```

In the above code, we have created a DataInputStream object to read primitive data types out of a file D:\\Test.txt, pointed by FileInputStream object disfis.

OutputStream classes write data only in terms of bytes but **DataOutputStream** class allows us to write data of primitive types such as char, int, float, double, boolean, short to an output stream. This class is a filter class which is used to wrap any output stream, to write primitive data to it.

Constructor:

DataOutputStream(OutputStream disout)

This constructor takes an OutputStream object disout in the parameters to write data to this output stream.

Example:

```
FileOutputStream disfos=new FileOutputStream("D:\\Test.txt");
DataOutputStream doswrite =new DataOutputStream(disfos);
```

In the above code, we have created a DataInputStream object to write data to a file D:\\Test.txt, pointed by FileOutputStream object reference disfos.

```
import java.io.*;
public class DataInOut {
   public static void main(String[] args) throws IOException {
      //Writing to the file
                FileOutputStream datafile = new FileOutputStream("dataout.txt");
                  DataOutputStream data = new DataOutputStream(datafile);
                   data.write(50);
                 data.write('V');
                 data.write('L');
                 data.write('D');
                 data.flush();
                 data.close();

                //Reading from the file
                InputStream inputdata = new FileInputStream("dataout.txt");
                DataInputStream datainst = new DataInputStream(inputdata);
                int count = inputdata.available();
                byte[] arydata = new byte[count];
                datainst.read(arydata);
                for (byte vd : arydata)
                {
                        char ch = (char) vd;
                        System.out.print("->"+ch);
                }
  }
}
```

In the above program first we are writing the character / byte in to the file through DataOutputStream. After that we reads the same file by wrapping the FileinputStream in DataInputStream and displays the content.

Output:

->2->V->L->D

4.8 OTHER CLASSES

There are various other classes apart from discussed above. They are,

4.8.1 RANDOMACCESSFILE

Java allows us to access the contents of a file in random order i.e. data items can be read and written in any fashion. This is especially very important and helpful in direct access applications like banking systems, airline reservation systems, Automatic Teller Machine (ATM) etc. Random access files are similar to arrays, where each data is accessed directly by its index number. In Java, java.io.RandomAccessFile class enables us to perform random access file input and output operations as opposed to sequential file I/O offered by ByteStream and CharacterStream classes.

Constructor:

public RandomAccessFile(String fileName, String mode) throws IOException

This constructor allows us to create a random access file stream to read from, and optionally to write to, a file with the specified file name. The mode argument must either be equal to “r” or “rw”, stating either to open the file for read or for both read and write.

When a data file is opened for random read and write access, an internal file pointer will be set at the beginning of the file. When we read or write data to the file, the file pointer will move forward to the next data item. For example, when reading an int data using readInt(), 16 bytes are read from the file and the file pointer moves 16 bytes forward from the previous file pointer position. Similarly, when reading a double data using readDouble (), 8 byte are read from the file pointer and the file pointer moves 8 bytes forward from the previous file pointer position.

4.8.2 STREAMTOKENIZER

The StreamTokenizer class is used to break an object of type Reader into tokens based on different identifiers, numbers, quoted strings and various comment styles. The next token will be obtained from the Reader by calling nextToken() method. It will return the type of token. StreamTokenizer class defines four int constants: TT EOF, TT EOL, TT NUMBER and TT WORD.

Apart from these constant there are three instance variables named nval, sval and ttype. The nval hold the values of numbers, sval hold the value of any words (string) and the ttype is a public int that has just been read by the nextToken( ) method.

If the token will be a word or string, ttype equals TT WORD. If the token will be a number, ttype equals TT NUMBER. If the token will be a single character, ttype contains its value. When an end of line condition has been encountered, ttype will equal TT EOL. When the end of the stream has been encountered, ttype will equal TT EOF.

Constructor:

The constructor for java.io.StreamTokenizer which works on an InputStream has been deprecated in favor of the constructor that works on a Reader. We can still tokenize an InputStream by converting it to a Reader:

```
Reader rd = new BufferedReader(new InputStreamReader(insr));

StreamTokenizer strtoken = new StreamTokenizer(rd);
```

Method StreamTokenizer(InputStream) is deprecated and the Alternative method is StreamTokenizer(Reader).

4.8.3 FILE

Java.io package also provides a **File** class that provide support for creating and manipulation of files. Means, the File class does not specify how information is retrieved from or stored in files rather it describes the properties of a file itself. A File Object is used to obtain or manipulate the data associated with a disk file. It will provide the permission, directory path and so on.

Constructor:

1. File(File superstr, String substr)

   This constructor allows us to create a new File instance from a superstr pathname and a substr pathname string.

2. File(String path)

This constructor allows us to create a new File instance by converting the given path string into an abstract pathname.

3. File(String superstr, String substr)

   This constructor allows us to create a new File instance from a superstr path string and a substr path string.

4. File(URI uripath)

   This constructor allows us to create a new File instance by converting the given file URI into an abstract pathname.

File class defines many methods. For example, **getName()** method returns the name of the file, **getPath()** method returns the path of the file, **getParent( )** method returns the name of the parent directory, **exists()** method returns true if the file exists and false if it does not. isFile() method returns true if invoked on a file and false if invoked on a directory. The **mkdir( )** method allows us to create a directory, returns true on success and false on failure. The **createNewFile()** method allows us to create a new empty file, return true on success and false on failure. It is written in a try-catch block. This is must because the createNewFile( ) method throws an IO exception, if the file cannot be created because of the entire path does not exist. If we fail to catch the exception, program will not compile.

```
import java.io.File;
import java.io.*;
public class Filehandling {

   public static void main(String[] args) {

      File f1 = new File("D:\\College\\BAOU\\BAOU\\Writing Book\\Program\\Book\\
                Test.txt");
      System.out.println("Folder Name is       : "+f1.getName());
      System.out.println("Full Path  is        : "+f1.getPath());
      System.out.println("Parent of  file      : "+f1.getParent());
      System.out.println("Book Folder is    : "+f1.exists());
      System.out.println("Book is a File    : "+f1.isFile());
```

```
        System.out.println("Test.txt is writeable   : "+f1.canWrite());
        System.out.println("Test.txt is readable    : "+f1.canRead());
        System.out.println("Test.txt size in Bytes  : "+f1.length());
        System.out.println("Absolute Location is   : "+f1.toString());
        System.out.println("Test.txt is Hidden file     : "+f1.isHidden());

         //Creating a new Directory
         File        f2        =        new        File("D:\\College\\BAOU\\BAOU\\Writing
                                        Book\\Program\\newDir");
         if(f2.mkdir())
         {
           System.out.println("Directory Created : Success");
         }else
          {
           System.out.println("Directory Created : Unsuccess");
           }
          //New file creation
          File f3 = new File("D:\\College\\BAOU\\BAOU\\Writing Book\\Program\\
                                  new.txt");
           try{
           if(f3.createNewFile())
           {System.out.println("File Created : Success");}
           else
           {System.out.println("File Created : Unsuccess"); }
        }catch (IOException io){}
    }
}
```

Above example shows the basic function related to File handling using File class. The output of the above program is as shown below:

Output:

Folder Name is   : Test.txt

Full Path is   : D:\College\BAOU\BAOU\Writing Book\Program\Book\Test.txt

Parent of file   : D:\College\BAOU\BAOU\Writing Book\Program\Book

Book Folder is   : true

Book is a File : true

Test.txt is writeable : true

Test.txt is readable : true

Test.txt size in Bytes : 816

Absolute Location is : D:\College\BAOU\BAOU\Writing Book\Program\Book\Test.txt

Test.txt is Hidden file : false

Directory Created : Unsuccess

File Created : Unsuccess

4.8.4 READING DATA FROM CONSOLE

There are three different techniques to read the input values from Java Console. They are:

1. Using Java Bufferedreader Class
2. Scanner Class in Java
3. Console Class in Java

We have already discussed the use of BufferedReader class in Character Stream classes. So, now we will discuss the remaining two.

- **Scanner Class**

This is easy and widely used technique to take input. The primary reason for the Scanner class is to parse primitive types and strings utilizing general expressions.

```
import java.util.*;
public class studentInput{
    public static void main(String []args){
      String Stuname;
```

```
        int Stuage;
        float Stuheight;
        //creating object of Scanner class
        Scanner input = new Scanner(System.in);
        System.out.print("Enter student name: ");
        Stuname = input.next();
        System.out.print("Enter student age: ");
        Stuage = input.nextInt();
        System.out.print("Enter student height: ");
        Stuheight = input.nextFloat();
        System.out.println("Name: " + Stuname + ", Age: "+ Stuage + ", height: "+
      Stuheight);
    }
}
```

Output:

Enter student name: Vinod

Enter student age: 35

Enter student height: 6

Name: Vinod, Age: 35, height: 6.0

- **Console Class in Java**

The java.io.Console class provides convenient methods for reading input and writing output to the standard input (keyboard) and output streams (display) in command-line (console) programs. The following program depicts the use of Console class to read input data from the user and print output:

```
import java.io.*;
import java.util.*;
public class ConsoleReadWrite {
    public static void main(String[] args) throws IOException {
        Console console = System.console();
        if (console == null) {
```

```
            System.out.println("Console is not supported");
            System.exit(1);
        }
        String Stuname = console.readLine("What's the student name? ");
        String Stuage = console.readLine("How old are the student is? ");
        String Stucity = console.readLine("Where do the student lives? ");

        //console.format("%s, a %s year-old student is living in %s", Stuname, Stuage,
         Stucity);
          console.printf("%s, a %s year-old student is living in %s", Stuname, Stuage,
         Stucity);
    }
}
```

console.printf () and console.format () prints the same results with applied formats.

Output:

What's the student name? Ved

How old are the student is? 10

Where do the student lives? Gandhinagar

Ved, a 10 year-old student is living in Gandhinagar

- **Check Your Progress 3**

1)Write a code of ObjectInputStream and ObjectOutputStream classes to demonstrate the working of java IO on objects.

……

……

2) Write a code to read byte array from a file using RandomAccessFile.

……

……

3) Differentiate various console based input options of Java.

……

……

4.9 LET US SUM UP

I/O in Java is based on streams. A stream represents a flow of data or a channel of communication. The package java.io contains streams-binary, character and object to handle fundamental input and output operations in Java. The I/O classes can be grouped as follows: All input related process is performed through subclasses of InputStream and all output related process is performed through subclasses of OutputStream. In this unit we have discussed various streams combined together to perform the added functionality of standard input and stream input. In this we have also discussed the operations of reading from a file and writing to a file. We have also discussed the classes which performs input – output through Pipes.

4.10 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

➢ **Check Your Progress 1**

1. We can divide the classes into two groups. They are,

   Low level: In this group FileInputStream, FileOutputStream, FileReader, FileWriter covered

   High level: In this group BufferedInputStream, BufferedReader, ObjectInputStream and their accompanying output classes are covered.

2.

- Streams: It deals with one byte at a time. It is good for binary data.
- Readers/Writers: it deals with one character at a time. It is good for text data.
- Buffered: It deals with many bytes/characters at a time. It is used always.

3. Byte streams are suggested for normal input and output.

   Character streams are suggested exclusively for character data.

   Basically, all data consist of bits grouped into 8-bit bytes. So, logically all streams could be called “byte streams”. Whenever the streams which are intended for bytes and represent characters are known as “character streams” and rest are called “byte streams”.

➢ **Check Your Progress 2**

1. Filter streams are used to manipulate data reading from an underlying stream. The read method in a readable filter stream reads input from the underlying stream, filters it, and then forward on the filtered data to the caller. The write method in a writable filter stream, filters the data and then writes the data to the underlying stream.

2.

```
import java.io.*;

public class FileAppend

{
```

```
public static void main(String[] args)
{
 try {
    PrintWriter pout = new PrintWriter(new BufferedWriter(new
    FileWriter("fAppnd.txt", true)));        //the true will append the new content
    pout.println("Welcome to BAOU, Ahmedabad.");
    pout.close();
 } catch (IOException ex) {
    System.out.println(ex);
 }  } }
```

3.

This class is very useful to copy multiple source files into one destination file with very less code.

- **Check Your Progress 3**

1.

```
String str = "Gujarat";
   byte[] byt = {'V', 'i', 'd', 'y', 'a', 'p', 'i', 't', 'h'};
  try {
     // create a new file with an ObjectOutputStream
     FileOutputStream out = new FileOutputStream("test.txt");
     ObjectOutputStream oout = new ObjectOutputStream(out);

     // write something in the file
     oout.writeObject(str);
     oout.writeObject(byt);
     oout.flush();
```

```
// create an ObjectInputStream for the file we created before
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"));

// read and print an object and cast it as string
System.out.println("" + (String) ois.readObject());

// read and print an object and cast it as string
byte[] read = (byte[]) ois.readObject();
String str1 = new String(read);
System.out.println("" + str1);
```

2.

```
RandomAccessFile ramacc = new RandomAccessFile("Test.txt", "r");
ramacc.seek(1);
byte[] byt = new byte[5];
ramacc.read(byt);
ramacc.close();
System.out.println(new String(byt));
```

3.

I.Using Buffered Reader Class

II. Using Scanner Class

III. Using Console Class

4.11 FURTHER READING

1) Core Java for Beginners: 1 (X-Team) by Sharanam Shah, Vaishali Shah 1st edition
2) Java Programming for Beginners by Mark Lassoff
3) Core Java Programming-A Practical Approach by Tushar B. Kute
4) Java: The Complete Reference by Schildt Herbert. Ninth Edition
5) https://www.decodejava.com/
6) https://www.javatpoint.com/

4.12 ASSIGNMENTS

1) Define Stream. Explain BufferedInputStream and BufferedOutputStream with example.
2) Discuss the different filter classes of IO streams.
3) Explain StringTokenizer class with proper example.
4) What is Console IO? Explain Scanner class with proper example.
5) Define Random access. State its benefit with respect to file access.