# UNIT 3 I/O STREAMS

**Structure** **Page No.**



## 3.0 INTRODUCTION

Input refers to any piece of information required for completion of execution of a program ( generally provided by users) while output refers to the information that the program provides to the user. For the development of an application, both input and output are essential. Computer programs can enhance their usefulness when they are able to interact with the rest of the world, which consists of users, machines, computers, etc., by some means. Interaction here refers to input, output, or I/O operations. Previous chapters mainly focus on only one type of interaction i.e. interaction with a user either through command line prompt or through a graphical user interface. But it is only one means of computer program and world interaction in which the user acts as both source and destination of the information. Similarly, one other type of input/output, i.e. Text IO, is used for reading and writing text from a file. In addition to Text IO, Java features a much powerful and flexible input/output framework enabling diverse ways of I/O interaction (socket, files, etc.). Java also features networked communication thereby increasing the program's usefulness.

Since in java, every I/O interaction is carried out via program interface. The following are the points that the java input/output interface must address.

1. Identification of end of the line in a file when a different convention is used by the different operating systems.
2. Handling different encoding systems used by the same operating system
3. Handling of file naming and path naming conventions as they may be different for a different system.

These all create a challenge for Java developers to counter these issues for Java users while performing input/output operations from an external source/destination. Java addresses these issues by integrating the java.io.* package enabling users to worry-free input/output operations.

In this unit, we elaborate on what are streams, different types of streams, how to read data from the input stream, how to send data over the output stream, etc. The unit also discusses the concept of I/O classes and interface, File class, Byte stream, Character stream, and serialization in detail for proper understanding.

## 3.1 OBJECTIVES

After going through this unit, you will be able to:

- explain a clear view of the Java input/output hierarchy,
- use I/O class and interface in your program,
- use File class for handling files and directories,
- use I/O class for reading and writing data,
- Use different types of I/O stream, i.e. Byte stream and Character stream,
- Handling of I/O Exception in the programs, and
- use the concept of Serialization.

## 3.2 INTRODUCTION TO I/O CLASS AND INTERFACES

In java to perform I/O operation, the concept of stream is used. Stream is a sequential flow of input or output data. Stream enables java programs to serially accept input from various data sources such as socket, file, buffer, etc. Stream also helps the programmer to create a file and send data over the internet. The Input stream is used to read data from the input source, while the Output stream helps to write data to a destination. Here source and destination can be anything that generates, consumes and stores the data such as program, disk, peripheral device, etc. The operation of I/O in java is shown in fig.1.

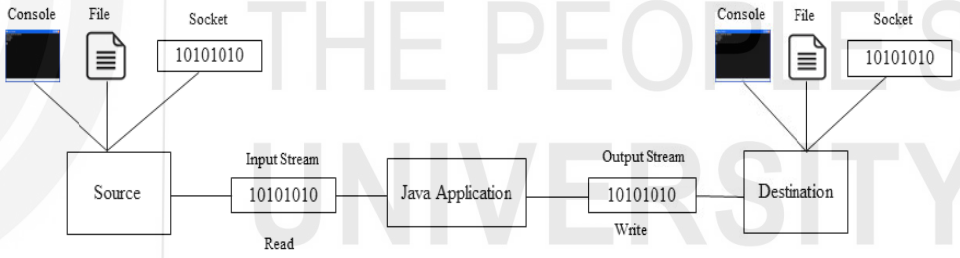



Figure 1: Java I/O

The section discusses different classes and interfaces defined in package java.io. The package mainly classifies the I/O class in 5 different class hierarchy sets: InputStream, OutputStream, Reader, Writer, and File. Input Stream and Output Stream are used for reading and writing byte data, while Reader and Writer classes are used for reading and writing the character data. Based on types of data (byte, file, character) and form of I/O device (file, array) further, these classes are classified. The hierarchy of the I/O classes has been illustrated in Figure 2.

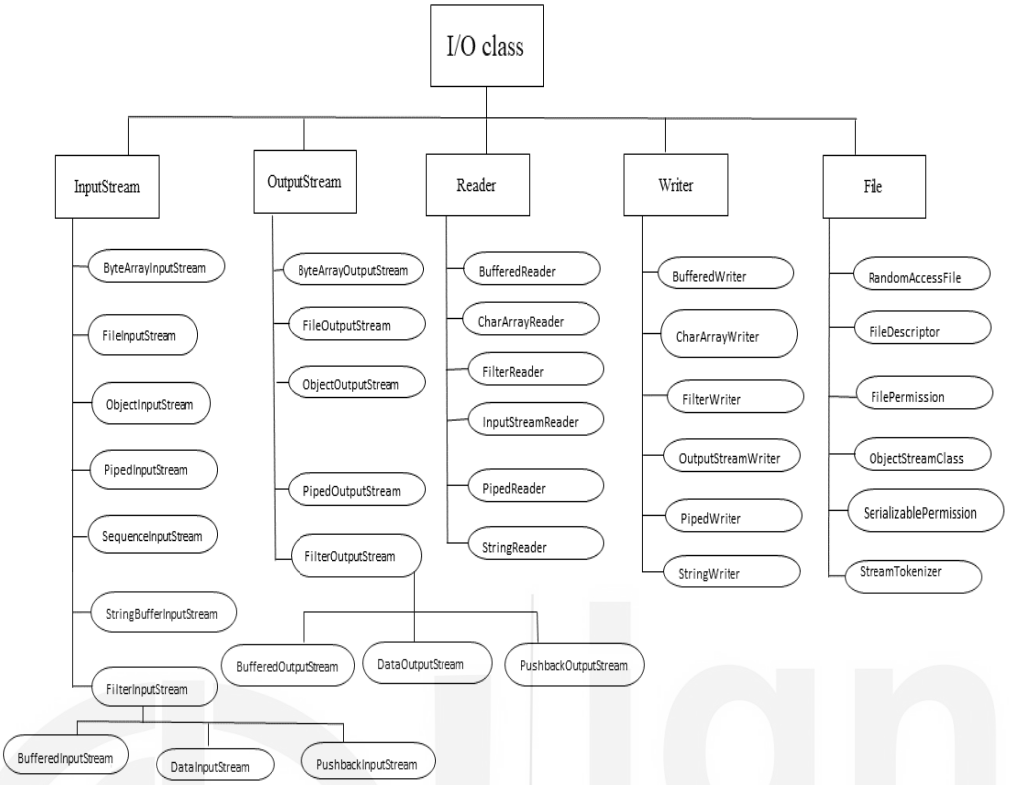



Figure 2: I/O Class hierarchy

Each of these classes defined above serves only one objective but, when combined with other classes can serve complex task objectives. For example, a BufferedReader allows buffering of input while a FileReader allows a method to connect file. When both are used together, the user can perform buffered reading from a connected file.

### 3.2.1 Java Interfaces

The Interface provides a way to perform input and output operations using data streams, file, and serialization. The list of interfaces provided by java.io is summarized in Table 1.

**Table 1**: List of Interfaces

| Sr.No. | Interface & Description |
|---|---|
| 1 | **Closeable**<br>Allows source and destination of data to be closed. |
| 2 | **DataInput**<br>Allows reading of byte data from binary stream and converting these data to any of Java primitive types. |

| | |
|---|---|
| 3 | **DataOutput**<br>Allows converting of Java primitive data to byte and write into binary stream. |
| 4 | **Externalizable**<br>Enables instance of class object to save and restore its content. |
| 5 | **FileFilter**<br>Provides filter for abstract pathnames. |
| 6 | **FilenameFilter**<br>Provides filter for filenames. |
| 7 | **Flushable**<br>Allows destination of data to flushed. |
| 8 | **ObjectInput**<br>Extends DataInput interface for reading of objects and allow deserialization of objects |
| 9 | **ObjectInputValidation**<br>Represents callback interface and enable validation of object within the graph. |
| 10 | **ObjectOutput**<br>Extends DataOutput interface for writing of objects and allow serialization of objects |
| 11 | **ObjectStreamConstants**<br>Enables constant to be written in Object Serialization stream. |
| 12 | **Serializable**<br>Allows serialization of objects/class by implementing the java.io.Serializable interface. |

## 3.3 FILE CLASS AND ITS METHOD

Most of the java.io package classes operate on stream concept, but the File class defined in java.io does not use stream concept. It directly interacts with the file and file system. The file class only describe the file properties but doesn't provide a view of how information related to file are stored or retrieved. For manipulating or obtaining information associated with a file, an object known as File object is created. The properties of file here represent permission, date and time of file creation and directory path of file. Even though there are some restrictions on using file (within applets) due to security issues, but still file is the primary source of storing and sharing information of many programs. The directory in java is a collection of files, whose filenames (list of files) can be fetched using the list() method.

The following are how an object of the File class can be created using a different form of File constructor.

File $f1$ = new File("c:/JavaPrograms/File_Name.txt");

File $f2$ = new File("c:/JavaPrograms", "File_Name.txt");

File $f3$ = new File("JavaPrograms", "File_Name.txt");

In the first form, $f1$ file object is created by File constructor accepting only one string parameter. The string parameter consists of the path of the file and the file name. For $f1$ filepath is c:/JavaPrograms while the name of the file is File_Name.txt. Similarly, $f2$ is created using the File constructor that accepts two parameters. The first parameter represents the path of the directory, while the second parameter represents the file name. Lastly, $f3$ is created using File constructor, which accepts directory name and file name as the input parameter.

### 3.3.1 File Methods

There are many methods defined in the file class that can allow users to examine the properties of the file object. Some of them are list below, along with their addressed function in Table 2.

**Table 2**: Methods of File

| **Method** | **Function** |
|---|---|
| getName() | Returns the name of file. |
| boolean exists() | Returns true if file exists; otherwise, return false. |
| boolean canWrite() | Returns true if file is writable; otherwise, return false. |
| boolean canRead() | Returns true if file is readable; otherwise, return false. |
| boolean isFile() | Returns true if reference is a file and return false if reference is directories. |
| boolean isDirectory() | Returns true if reference is a directory; otherwise, return false. |
| String getAbsolutePath() | Returns application absolute path. |
| boolean renameTo(File newFileName) | Returns true if file name is renamed; otherwise, return false |

To properly understand how the file object is created and how the file method can be accessed, we illustrate the concept by using the following program as an example.

```
//program
import java.io.File;
class DemoFile_1
{
  public static void main(String args[])
  {
    File fil_1 = new File ("/testfile_1.txt");
    System.out.println(" Display File name : " + fil_1.getName());
    System.out.println(" Specify the Path : " + fil_1.getPath());
    System.out.println("Absolute Path of file location: " + fil1.getAbsolutePath());
    System.out.println(fil_1.exists() ? "Do Exists" : "Doesn't exist");
    System.out.println(fil_1.canWrite()?"writable" : " not writable");
    System.out.println(fil_1.canRead() ? " file is readable" :"  file not readable");
    System.out.println("Size of file : " + fil_1.length() + " bytes");
  }
}
```

***Output: (When testfile_1.txt does not exist)***

```
Display File name: testfile_1.txt
Specify the Path: \testfile_1.txt
Absolute Path of file location : C:\testfile_1.txt
Doesn't exist
Not writable
File not readable
Size of file : 0 bytes
```

***Output: (When testfile_1.txt exists)***

```
Display File name : testfile_1.txt
Specify Path : \testfile_1.txt
Absolute Path of file location : C:\testfile_1.txt
Do Exists
writable
file is readable
Size of file : 17 bytes
```

The following program checks whether the filename FileCreated_1.txt is created or not. If the creation of the file is successful, then the output of program is "Creation of new File is successful!". The else part will be generated as output if the file FileCreated_1.txt is already present in the system.

```
import java.io.*;
public class FileCreation1
{
  public static void main(String[] args)
  {
    try
    {
      File f1_ex1 = new File("FileCreated_1.txt");
      if (f1_ex1.createNewFile())
      {
        System.out.println("Creation of new File is successful!");
      }
      else
      {
        System.out.println("File is already present.");
      }
    }
    catch (IOException fileexp)
    {
      fileexp.printStackTrace();
    }
  }
}
```

**Output of the program:** Creation of new File is successful!

The following program is used to check whether the folder jdk-15.0.1 consists of how many files and how many directories. The program use isDirectory() method for checking whether the inputted file is a directory or file. The list() method in the program is used to fetches all files and directories inside the folder jdk-15.0.1.

```
import java.io.File;
class DirCheckList
{
  public static void main(String args[])
  {
    File dir1 = new File("C:\\Program Files\\Java\\jdk-15.0.1");
    if (dir1.isDirectory())
    {
      System.out.println("Directory of " + "Java JDK");
      String str1[] = dir1.list();
      for (int i=0; i < str1.length; i++)
      {
        File chk_f1 = new File("C:\\Program Files\\Java\\jdk-15.0.1"+ "/" + str1[i]);
        if (chk_f1.isDirectory())
        {
          System.out.println(str1[i] + " is a directory");
        }
        else
        {
          System.out.println(str1[i] + " is a file");
        }
      }
    }
    else
    {
      System.out.println("C:\\Program Files\\Java\\jdk-15.0.1" + " is not a directory");
    }
  }
}
```

The output of program displays file in list and directory category.

```
Directory of Java jdk-15.0.1
bin is a directory
conf is a directory
COPYRIGHT is a file
include is a directory
jmods is a directory
legal is a directory
lib is a directory
release is a file
```

## ☞ Check Your Progress-1

1) Why do we need Java I/O Package?

……

……

……

……

2) Write the name of any three interfaces provided in java.io with a description.

……

……

……

……

3) Write the functions of following methods: boolean exists(), boolean canRead(), boolean isDirectory()

……

……

……

……

4) Draw and explain the class hierarchy of Java I/O?

……

……

……

……

## 3.4 THE CLOSEABLE, FLUSHABLE, AND AUTOCLOSEABLE INTERFACES

**Closeable interface:** Closeable interface has only one abstract method i.e. close(). When a call to this method is invoked, system resources held by the stream object is released and these resources can be used further. Generally, stream classes implements Closeable interface and overrides close() method to close the stream. A sub class can use close() method, if its super class implements Closeable interface. E.g. close() method can be used by FileInputStream, because its super class InputStream implements Closeable interface.

**Flushable interface**: Like Closeable interface, Flushable interface also has only one abstract method i.e. flush(). When flush() method is called, data stored in the buffer is flushed out to be written to the destination file. Like Closeable interface, stream classes need to implement Flushable interface in order to override the flush() method.

Both close() and flush() method throws IOExaception.

**AutoCloseable interface:** Like Closeable interface, AutoCloseable interface also includes close() method, but this method is automatically called. An object that implements AutoCloseable interface, when it is done with the held resource with in a try-with-resources block, close() method is automatically called.

## 3.5 I/O EXCEPTIONS

Exceptions refer to the interruption that breaks the normal flow of program execution. For example, if you try to access a file that is not present, it will result in a FilenotFound exception. The following are different types of I/O Exceptions listed in Table 3.

**Table 3**: Different I/O Exceptions

| Sr.No. | Interface & Description |
|---|---|
| 1 | **CharConversionException**<br>Represent the base class responsible for handling character conversion |

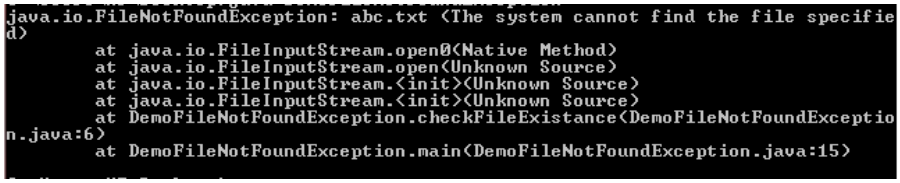
| | |
|---|---|
| | exceptions. |
| 2 | **EOFException**<br>Generate exception when (unexpected) end of stream or end of file is encountered during input. |
| 3 | **FileNotFoundException**<br>Thrown while opening a file using a specified file path, but the file path is not present/invalid. |
| 4 | **InterruptedIOException**<br>Thrown when I/O operation is interrupted. |
| 5 | **InvalidClassException**<br>This is thrown when a problem is detected with a Class during Serialization runtime. |
| 6 | **InvalidObjectException**<br>Thrown when one or more deserializable objects failed validation test. |
| 7 | **IOException**<br>Thrown when some sort of I/O exception occurs. |
| 8 | **NotActiveException**<br>Thrown when serialization and deserialization is inactive. |
| 9 | **NotSerializableException**<br>This is thrown when an instance variable of failed to have Serializable interface. |
| 10 | **ObjectStreamException**<br>Represent superclass of all exceptions occurred in object stream classes. |
| 11 | **OptionalDataException**<br>Throws exception when serialized object of stream performing in read operation result in failure either due to unread primitive data or data end. |
| 12 | **StreamCorruptedException**<br>Thrown when control information read from object stream fails consistency checks. |
| 13 | **SyncFailedException**<br>Thrown when sync operation failed. |
| 14 | **UnsupportedEncodingException**<br>Thrown when the character encoding is not supported. |
| 15 | **UTFDataFormatException**<br>Thrown when unsupported format data( not in UTF-8 format) is read by input stream or by any class that implement data input interface. |

The following program demonstrates that an exception will be thrown if we try to read a file that does not exist.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
public class DemoFileNotFoundException
{
  public void checkFileExistance()
  {
    try
    {
      FileInputStream f = new FileInputStream("abc.txt");
      System.out.println("File Exists");
    }
    catch (FileNotFoundException exceptionFileNotFound)
    {
      exceptionFileNotFound.printStackTrace();
    }
```

```
    }
    public static void main(String[] args)
    {
      DemoFileNotFoundException demo = new DemoFileNotFoundException();
      demo.checkFileExistance();
    }
}
```

**Output:**




## 3.6 INTRODUCTION TO STREAM CLASSES

As discussed above in the chapter, the java program uses the concept of the stream to perform I/O. A stream can be defined as an abstraction which either consumes or produces information. To perform I/O operations with a physical device in Java, the system makes a link connection between the stream and physical device. Even though they are connected to different physical devices, all streams behaved in the same manner. Thus, it enables the input stream to obtain input from different input sources such as a disk file, keyboard, a network socket, etc.

Similarly, it allows the output stream to transmit output to different data destinations such as console, file disk, network connection. Stream provides an efficient way to perform input/output by the program without worrying about different types of physical devices. All the stream classes are implemented in the java.io package.

**Byte Streams and Character Streams**

In Java basically, there are two types of streams: Byte Streams and Character Streams. Byte streams are used to deal with input and output of binary or byte data, while Character streams deal with characters. Though all I/O are byte-oriented at the lowest level but character streams provide a convenient method for handling character using Unicode representation and can be easily internationalized making it more useful.

Java I/O classes are mainly built on four abstract classes i.e. InputStream, OutputStream, Reader, and Writer. The first two classes are categorized as Byte stream class, while the latter two of them are categorized as Character stream class. Byte stream classes are discussed in section 3.7, and Character stream classes are discussed in detail in section 3.8. In order to implement these classes, one must import java.io.

## 3.7 BYTE STREAM CLASSES

Byte Stream class provides a rich repository for the user that wants to perform byte or binary I/O operations. Any type of object can access these stream classes thus making these stream classes important for many programs. Byte Stream classes are further classified into two abstract class hierarchies i.e InputStream and OutputStream. Further, with these two top hierarchy abstract classes, several concrete subclasses are

created, which ease the interaction when encountered with different types of devices such as files, network connection, buffer, etc. Fig 2. shows different types of stream classes. Later in the chapter few of these concrete subclasses are discussed.

Several key methods are defined in these two abstract classes that other types of stream classes can implement. For example, the read() and write() methods defined in InputStream and OutputStream are used for reading and writing byte data. These methods are abstract in nature and are overridden by any derived stream classes.

As InputStream class and OutputStream class are the top hierarchized class of byte stream, we start our discussion with InputStream class.

**InputStream**

InputStream is an abstract class that models the streaming of byte input. As it is a top-hierarchy abstract class, it defines numerous methods that are used/implemented by various other concrete subclasses. Table 4 describes the methods defined in the InputStream class. The methods (all of them) defined in InputStream class throws IOException in case of occurrence of error.

**Table 4**: Methods of InputStream

| **Method** | **Descriptions** |
|---|---|
| int available( ) | Returns the estimation of available number of bytes present for reading. |
| void close( ) | Results in closing the input source and releases the associated system resources. Throws IOException if further attempts of read is made. |
| void mark(int *num_Bytes)* | Marks current point in the input stream, and it remains valid until the size of *num_Bytes* bytes are read. |
| boolean markSupported( ) | Returns **true** if the invoking stream supports **mark( )/reset( ) .** |
| int read( ). | Returns the byte of input stream that is next available in integer form if the end of the file is encountered during reading return -1. |
| int read(byte *buffer*[ ]) | Returns the total size of byte that was successfully read from buffer when an attempt of reading bufferlength byte (input size) is made. If the end of the file is encountered during reading return -1. |
| void reset( ) | Resets the current input pointer to previously set mark. |
| long skip(long *numBytes*) | Skips or Ignores input of size numBytes and return the number of bytes that are skipped. |

**OutputStream**

The next top hierarchy abstract class is the Output stream class. It is an abstract class that models the streaming of byte output. The methods(all of it) defined in this abstract class are void in nature and throws IOException when an error occurs. The methods defined in this class are summarized in Table 5.

**Table 5**: Methods of OutputStream

| Methods | Description |
|---|---|
| void close( ) | Results in closing the output stream and releases associated system resources. Throws IOException if further attempts of write are made. |
| void flush( ) | Clears or flushes the output buffers. |
| void write(int *b*) | Writes b bytes of data to output stream. |
| void write(byte *buffer*[ ]) | Writes an array of bytes to an output stream. |
| void write(byte *buffer*[ ], int *offset*, int *num_Byte*) | Writes an array of bytes of size num_Byte to output stream from the array *buffer* staring from *buffer*[offset]. |

Besides the two top hierarchy classes of Byte stream, other concrete subclasses are defined using the methods defined in InputStream and OutputStream class. Some of these subclasses are discussed as follows.

**FileInputStream**

The FileInputStream class allows users to read bytes from a file. To create an input stream of FileInputStream, the following constructor is used.

FileInputStream(String *pathFile*)

FileInputStream(File *objtFile*)

Both these constructors throw FileNotFoundException in case any exception occurs (when no file is found or invalid path of the file). Here, *pathFile* represents the full pathname of the referenced file, while *objtFile* represents the object of the referenced file. Let's see how to create FileInputStream using these two constructors with an example. In the following examples, both constructors use the same disk file.

```
FileInputStream fl_1 = new FileInputStream("/FileDemo_1.java")

File fl_1 = new File("/FileDemo_1.java");

FileInputStream fl_2 = new FileInputStream(fl_1);
```

As stated in the above example, the first constructor uses file path name while the second uses File object to read byte from file. Commonly the first constructor is used to create FileInputStream, but we can use the second constructor when we want to examine File methods before attaching input streams. Besides mark() and reset() methods of InputStream, this class overrides the other six methods of InputStream class. Any attempt to override mark() and reset() methods, results in the generation of IOException. As soon as FileInputStream is created, it automatically opened.

**FileOutputStream**

The FileInputStream class allows users to write bytes to a file. To create the output stream of File OutputStream, the most common constructor is defined using the following syntax:

FileOutputStream(String *pathFile*)

FileOutputStream(File *objtFile*)

FileOutputStream(String *pathFile*, boolean *app*)

These constructors throw either FileNotFoundException or a SecurityException. Here, pathFile represents the full pathname of the referenced file, while objtFile represents the object of the referenced file. If app is true in the third constructor, then the file is opened in append mode. When the user tried to write byte data to a read-only file, IOException will be thrown.

**ByteArrayInputStream**

ByteArrayInputStream creates an input stream that allows users to read data from byte array(input source). The class uses two types of constructors to create ByteArrayInputStream. In both these constructors, byte array is the source of input data. The syntax of the constructor is of the form:

ByteArrayInputStream(byte *name_of_array*[ ])

ByteArrayInputStream(byte *name_of_array* [ ], int *start*, int *noBytes*)

Here, *name_of_array* is the source of input. In the second constructor, input stream is created from a subset of byte array, i.e. *name_of_array*, starting from index denoted by *start* parameter till the length of *noBytes.*

The following examples are to illustrate how ByteArrayInputStream is created using the above-discussed constructor.

```
// Demonstrating the creation of  byte input stream
import java.io.*;
class ByteArrayInputStream_Demo1
{
   public static void main(String args[]) throws IOException
   {
      String s1 = "howareyouinthishour";
      byte y_1[] = s1.getBytes();
      ByteArrayInputStream input_1 = new ByteArrayInputStream(y_1);
      ByteArrayInputStream input_2 = new ByteArrayInputStream(y_1, 0,3);
   }
}
```

The input_1 contains howareyouinthishour, while input_2 contains only how.

**ByteArrayOutputStream**

ByteArrayOutputStream creates an output stream that allows users to write data to a byte array(output destination). The class uses two types of constructors to create ByteArrayOutputStream. The syntax of the constructor is of the form:

ByteArrayOutputStream()

ByteArrayOutputStream(int *noBytes*)

From the first constructor, a buffer of 32-byte is created, while from the second constructor buffer of size *noBytes* is created. The count field of ByteArrayOutputStream class is used to denote the number of bytes stored in the buffer. The count field is protected in nature.

Following program demonstrates the use of ByteArrayOutputStream class

```
import java.io.*;
class DemoByteArrayOutputStream
{
  public static void main(String args[]) throws IOException
  {
    ByteArrayOutputStream op=new ByteArrayOutputStream();
    byte b[]= “Have a nice day”.getBytes();
    op.write(b);
    System.out.println(op.toString() );
    op.close(); //closing the stream
  }
}
```

**Output:** Have a nice day

**Filtered Byte Streams**

*Filtered streams* are the abstract class that supports basic input or output streams but with additional functionality. These streams are implemented by the class, which requires generic streams. The additional functionality here refers to buffering, character, and raw data translation. **FilterInputStream** and **FilterOutputStream** are types of filtered byte streams.

The constructor of this class is of the form:

FilterInputStream(InputStream *is*)

FilterOutputStream(OutputStream *os*)

InputStream and OutputStream methods and these classes' methods are similar in nature.

**Buffered Byte Streams**

Buffered streams extend the FilterStream class and allow buffering of byte I/O stream by attaching a buffer memory. Due to this buffering, it allows java to perform I/O operation on more than one byte simultaneously, thereby increasing the program's performances. Buffer also allows users to perform skipping, marking and resetting of input streams. BufferedInputStream and BufferedOutputStream are buffered byte stream classes. PushbackInputStream also implements the buffered stream.

**BufferedInputStream**

One of the common ways of performance optimization is achieved by buffering of I/O. In java, this performance is achieved using BufferedInputStream class that enables buffering of input streams. The constructor of this class is of the following form:

BufferedInputStream(InputStream *streamInput*)

BufferedInputStream(InputStream *streamInput*, int *size_Buf*)

In the first form of the constructor, a stream is created and buffered using the default size buffer while in the second form of the constructor , the stream is buffered in

buffer of size *size_Buf*. For significant performance improvement, the buffer is usually taken in terms of a memory page, disk block.

The buffering also enables to trace stream in backward direction due to the storage of input. Hence, it supports mark() and reset() methods defined in Inputstream class. For examining this support, BufferedInputStream.markSupported**( )** is used that returns true when it supports the aforementioned methods.

**BufferedOutputStream**

The BufferedOutput stream behaves similarly to other OutputStream but with an additional flush() method. The flush() method defined here confirms/ensures that the data in buffers are physically written to actual output devices. BufferedOutputStream also tries to improve the performance by decreasing the number of times writes data.

Following are the form of its constructor:

BufferedOutputStream(OutputStream **streamOutput**)

BufferedOutputStream(OutputStream **streamOutput**, int **buf_Size**)

The first one creates a buffer of 512 bytes for a write operation of output streams, while the second one creates a size of the buffer **buf_Size** for writing output streams.

**PushbackInputStream**

Buffering allows pushback operations in java. Pushback allows a byte to read from the input stream and then return this byte to the input stream. It allows users to see what is coming from the input stream without disturbing the flow of the input stream. In java, the pushback of byte is implemented via PushbackInputStream class.

Following are the form of constructor defined in PushbackInputStream:

PushbackInputStream(InputStream *input_stream*)

PushbackInputStream(InputStream *input_stream,* int *num_Bytes*)

The first constructor creates a stream object that allows readied byte to return to the input stream. The second constructor creates a buffer of multiple data of size num_Bytes and returns it to the input stream.

**SequenceInputStream**

When a user wants to concatenate multiple Input streams, it can use **SequenceInputStream** class. The constructor of this class either uses a pair of InputStream or Enumeration of InputStream class. Following are the form of the constructor:

SequenceInputStream(InputStream *first_Stream*, InputStream *second_Stream*)

SequenceInputStream(Enumeration *stream_Enum*)

In the first case, the class starts reading data from the first InputStream until its end and then switch to the next or second InputStream for reading. While in the second case i.e. Enumeration, it will continue reading data until the last stream end is encountered.

**PrintStream**

The PrintStream class offers all types of formatting capabilities for the stream as used from the starting of books, i.e. as in System.out and system filehandle.

Following are its constructor form:

PrintStream(OutputStream *output_Stream*)

PrintStream(OutputStream *output_Stream*, boolean *flush_On_Newline*)

Here *flush_On_Newline* controls the flushing of the output stream whenever a newline(\n) is encountered. *If flush_On_Newline* results in false, flush operation is not invoked automatically; else, it is automatically invoked.

These class objects support both print() and println() methods to accept all types of arguments /parameters, including objects. In both methods, if parameters are not of simple type, a call toString() method is invoked, and then the result is printed.

**RandomAccessFile**

When a user wants to randomly access a file, it can use RandomAccessFile class. This class is not derived from either InputStream or OutputStream class. It implements the DataInput and DataOutput interfaces defined in I/O methods. The class also supports the feature of positioning the file pointer. The defined constructor is of the following forms:

RandomAccessFile(File *file_Obj1*, String *mode_Access*) throws FileNotFoundException

RandomAccessFile(String *name_file*, String *mode_Access*) throws FileNotFoundException

In the first constructor, *file_Obj1* represents the name of the file to be open as a File object while in the second constructor,*name_file* represents the name of the file to be opened. The mode_Access here represents the type of access permitted regarding file access. If 'mode_Access =r', the file can only be read but cannot be written. If 'mode_Access =rw' then the file can be used for both reading and writing operations. If 'mode_Access =rwd', the file is eligible for both read and write operations, and the changes made to files are directly written to the physical device.

To set the file pointer to the current point in the file, seek() method is defined.

void seek(long *new_Pos*) throws IOException

Here, the file pointer is set to a new position, i.e. *new_Pos from the starting* of the file. After this read or write operation is performed from this new position. Another method *setLength()* is used to increase or decrease the length of the file.

Following program writes a string “new end” at the end of the file, i.e. append a string “new end” using RandomAccessFile class.

```
import Java.io.*;
public class DemoRandomAccessFile
{
  public static void main(String args[])
  {
    try
    {
      RandomAccessFile rf = new RandomAccessFile("xyz.txt","rw");
      rf.seek(rf.length());
      raf.writeBytes("\n new end \n");
    }
    catch (IOException e)
    {
      System.out.println("Error Opening a file" + e);
    }
  }
}
```

**Ouput:** Above program will append “new end” in xyz.txt file.

## ☞ Check Your Progress-2

1) What is the difference between Byte Streams and Character Streams?

……

……

……

……

2) What is the difference between closeable and autocloseable interfaces?

……

……

……

……

3) Which of the following classes is used for input and output in Byte streams?

(a) FileInputStream (b) BufferedReader (c) BufferedWriter (d) File

……

……

……

……

4) Which of the following class supports print() and println() methods?

(a) System (b) BufferedOutputStream (c) PrintStream (d) System.out

……

……

……

……

5) Write a program for reading data from two or more input stream using SequnceInputStream class.

……

……

……

……

## 3.8 CHARACTER STREAM CLASSES

**Character Streams**

Even though the byte stream classes are sufficient for handling any type of I/O operation, but the limitation of these classes is that they never directly work with Unicode characters. So, to support one of the main features, i.e. write once and run anywhere of java, support for direct character-oriented I/O is vital. Thus, in this section discussion of different character I/O classes is carried out. As seen earlier in this chapter, character stream classes are top hierarchized by two abstract classes, i.e. Reader and Writer. Many other stream classes use the methods defined in these classes. For example, read() and write() defined here are used by other stream classes to perform read and write character operations. So, let's begin with Reader abstract class.

**Reader**

This is an abstract class that models the streaming of character input in Java. IOExceptions are thrown by every method defined in this class on the occurrence of any error. The methods of this class are described in Table 6.

**Table 6**: Methods of Reader Class

| **Methods** | **Description** |
|---|---|
| void close( ) | Performs closing of source input. Throws I/OException if any further read attempt is carried out. |
| void mark(int *num_Chars*) | A pointer (marker) is placed at current point in input stream, and the pointer is valid until num_ Chars are read. |
| boolean markSupported( ) | Returns true if stream supported mark()/reset(). |
| int read( ) | Reads a character from the input stream and return its integer representation when file end is encountered return -1. |
| int read(char *buffer*[ ]) | Returns the actual number of characters that are successfully read when reading of characters of buffer.length is attempted in buffer, when file end is encountered return -1. |

| | |
|---|---|
| abstract int read(char *buffer*[ ], int *offset,* int *num_Chars)* | Returns the number of characters that were successfully read when the attempt of reading *num_Chars* character into buffer is performed starting from the *buffer [offset]*. When file end is encountered during reading return -1. |
| boolean ready( ) | Returns false if the next input request is not ready(waiting) else, return true. |
| void reset( ) | The input pointer is reset to the previously set mark. |
| long skip(long *noChars*) | Returns the number of characters that are successfully skipped when skip of noChars input character is carried out. |

**Writer Class:**

This is an abstract class that models the streaming of character output. The methods defined in this class are void in nature and throws IOException when an error occurred. The methods defined in this class are described in Table 7.

**Table 7**: Methods of Writer Class

| **Methods** | **Description** |
|---|---|
| abstract void close( ) | Used to perform the closing of the output stream. Throws I/OException if any further write attempt is carried out. |
| abstract void flush( ) | Flushes the output buffers. |
| void write(int *ch*) | Used to perform writing of a single character to the invoked output stream. |
| void write(char *buffer*[ ]) | An array of stream is written to the invoked output stream. |
| abstract void write(char *buffer*[ ], int *offset*, int *numChars*) | A subarray of the character of length num_Chars from buffer array starting from buffer[offset] is written to the invoked output stream. |
| void write(String *str*) | str is written to the invoked output stream. |
| void write(String *str*, int *offset*, int *num_Chars*) | A subrange of *num_Chars* character from array str starting from str[offset] is written to the invoked output stream. |

**FileReader**

The FileReader class allows users to read the contents (input stream) from the file. Syntax of two of its most used constructors are as follows:

FileReader(String *pathFile*)

FileReader(File *objtFile*)

Here *pathFile* represents the path where the file is located, while *objtFile* represent the File object of the invoked file.

In order to demonstrate how to read from a file is described using the following program. The program demonstrates reading of line from file and printing to the output stream.

```
// Demonstrating the reading of file using FileReader
import java.io.*;
class FileReader_Demo1
{
  public static void main(String args[]) throws Exception
```

```
  {
    FileReader rf_1 = new FileReader("FileReader_Demo1.java");
    BufferedReader rb_1= new BufferedReader(rf_1);
    String str;
    while((str = rb_1.readLine()) != null)
    {
      System.out.println(str);
    }
    rf_1.close();
  }
}
```

**FileWriter class**

The class allows users to write characters (input stream) to file. The most common constructors defined in this class are of the following syntax:

FileWriter(String *pathFile*)

FileWriter(String *pathFile,* boolean *app)*

FileWriter(File *objtFile*)

FileWriter(File *objtFile*, boolean *app*)

These constructors defined here throws **IOException.** Here, *pathFile* represents full pathname of the referenced file while *objtFile* represents the object of the referenced file. If *app* is true, then the file is opened with appended output. FileWriter first creates a file by creating an object and then open it for output writing. When a user tries to open read-only file, IOException will be thrown.

**CharArrayReader**

The class provides another method for users to read the input stream (character) from character array (Input source). The constructor of this class uses character array as a source of data reading. The form of constructors is as follows:

CharArrayReader(char *name_ Array*[ ])

CharArrayReader(char *name _Array*[], int *begin*, int *num_Chars*)

Here *name_ Array* is the data source, while in the second constructor, a subset of character of length *num_Chars* is read from *name_ Array* starting with index specified by begin parameter.

**CharArrayWriter**

The class provides another method for users to write the character stream (character)to the character array (data destination). The constructor of this class uses a character array as the destination of data writing. The form of constructor is as follows:

CharArrayWriter( )

CharArrayWriter(int *num_Chars*)

In the first constructor, a default size buffer is created for performing write operations, while in the second constructor, a buffer size equal to *num_Chars* size is created. CharArrayWriter uses buf field to denote/hold buffer information. Automatically the size of the buffer increase when required. A count field defined in the class is used to keep track of the number of characters in the buffer. Both these fields are protected fields.

**BufferedReader**

The class enable the java program to increase its performance by attaching a buffer to the input stream. The buffer allows the user to hold character and allow to perform I/O operation on more than one character at a time. Following are the form of this class constructor.

BufferedReader(Reader *in_Stream*)

BufferedReader(Reader *in_Stream*, int *buf_Size*)

The first constructor creates buffered character stream using default buffer size, while in the second constructor the size of the buffer is set to *buf_Size* for creating buffered character streams. Since we have seen that buffering enable moving stream in the backward direction by implementing mark() and reset() methods and return true when checked by calling **BufferedReader.markSupported( ).**

The following program demonstrates the reading of data using BufferedReader class.

```
import java.io.*;
public class BufferedReaderDemo
{
   public static void main(String[] args) throws IOException
   {
      InputStreamReader abc = new InputStreamReader(System.in);
      BufferedReader xyz = new BufferedReader(abc);
      String str = "";
      System.out.print("Please enter a string: ");
      str = xyz.readLine();
      System.out.println("Entered string is: " + str);
   }
}
```

**Output:**

Please enter a string: abcdef

Entered string is: abcdef

**BufferedWriter**

The BufferedReader stream behaves in a similar fashion as other Writer class but with additional flush() method. The flush() method defined here makes the data in buffers to be physically written to the actual output stream. BufferedWriter also tries to improve the performance by decreasing the number of times actually writes operation is performed.

Following are the form of these constructors:

BufferedWriter(Writer *output_Stream*)

BufferedWriter(Writer *output_Stream*, int *buf_Size*)

The first one creates a buffer of 512 bytes for write operation of output character streams, while the second create a size of the buffer *buf_Size* for writing output streams.

The following program demonstrates the writing of data to a file using BufferedWriter class.

```
import java.io.FileWriter;
import java.io.BufferedWriter;
public class DemoBufferedWriter
{
  public static void main(String args[])
  {
    String str = "This is the output file";
    try
    {
      FileWriter f = new FileWriter("output.txt");
      BufferedWriter op = new BufferedWriter(f);
      op.write(str);
      op.close();
    }
    catch (Exception e)
    {
      e.getStackTrace();
    }
  }
}
```

**PushbackReader**

Buffering enables pushback operations in java. PushbackReader allows one or more character to read from input stream and then return these characters to input stream. Basically, it allows users to see what is coming from input stream without disturbing the flow of input stream. In java, push back of character is implemented via PushbackReader class. Following are the form of its constructor:

PushbackReader(Reader *inStream*)

PushbackReader(Reader *inStream*, int *bufSize*)

In the first constructor, the first character is read and push backed to in_Stream, while the second constructor allows pushback of stream equal to the size of *bufSize.* One must check whether the pushback buffer is full or not before returning a character. If the pushback buffer is full, it results in IO Exception. If we want to return one or more than one characters to the input stream, we can use unread() method of PushbackReader.

**PrintWriter**

PrinterWriter works similar to Print Stream and can be thought of as character oriented version of PrintStream. It provides all types of formatting functionality by defining print() and println() methods. Following are its Constructor forms:

PrintWriter(OutputStream *output_Stream*)

PrintWriter(OutputStream *output_Stream*, boolean *flushOn_Newline*)

PrintWriter(Writer *output_Stream*)

PrintWriter(Writer *outputStream*, boolean *flushOn_Newline*)

Where flushOn_*Newline* controls flushing of output whenever println() is invoked. Flushing of output is done automatically when *flushOn_Newline* is true else flushing is not automatic. This class objects provide support for both print() and println() methods to accept all types of arguments /parameters, including object also. In both methods, if parameters passed are not of simple type, a call toString() method is invoked, and then the result is printed.

## 3.9 SERIALIZATION

Java not only allows users to read and write text but also provides support to read and write objects of file. With serialization, users can write the state of objects to a byte stream. Serialization is useful to users as they allow users to save the content of their program in any non-volatile storage (file, memory) and later allows the users to restore these objects when required or necessary. The restoring process is known as Deserialization.

In Java, Remote Method Invocation (RMI) is implemented with the help of Serialization. With RMI in Java, the object of one machine can invoke methods of another object located on a different machine. In RMI, the object is passed as an argument to the invoked remote method. Before passing this object as a parameter, serialization is used to serialize the passed object and then transmit it. On receiving the serialized object, the destination machine deserialize it.

One important question that arises is that how serialization in Java is easy as compared to serialization in another language. For instance, suppose an object has a reference to another object, and the referenced object has a reference to any other object. In some cases, they may form circular references among objects creating a lot of confusion or chance of inconsistency. Java efficiently handles this situation by doing the following things. If you serialize one object, all its referenced objects are recursively located and serialized. Similarly, when performing deserialization all its referenced objects get restored correctly in the same order. In java if you want to serialize an object, you have to call the writeObject() method defined in ObjectOutputStream, and when you want to deserialize any object, invoke the readObject() method defined in ObjectInputStream. Serialization also enables transferring of object state over the network i.e Marshalling of objects. Implementation of a Serializable interface is also necessary for the serialization of objects.

Following are the interfaces and class that supports serialization.

**Serializable**

The object that implements Serializable interfaces can only be saved and restored using the serialization process. No members are defined in the Serializable interface. To serialize the object of the class, the class must be declared as public and the class must have no parameter constructor. All subclass of the serializable class is also

serializable. Transient and static variables are not saved by serialization facilities. java.io.Serializable interface is by default implemented by the string and all wrapper class. In the following example, Employee class implements Serializable interface. Now object of Employee class can be converted into stream.

```
import java.io.Serializable;
public class Employee implements Serializable
{
  int e_id;
  String e_name;
  public Employee(int e_id, String e_name)
  {
    this.e_id = e_id;
    this.e_name = e_name;
  }
}
```

**Externalizable**

In java, serialization and deserialization have automatically made the state of an object to be stored and restored. But in some cases, the programmer desires to control this automatic process. For example, when implementing encryption and compression techniques, the programmer can use Externalizable interface to control these automatic processes.

**ObjectOutput:**

The interface extends the DataOutput interface and provides support for object serialization, For making object serializable, call to writeObject() method defined in this interface is invoked. Methods defined in this interface throws IOException in case of error occurrence.

**ObjectOutputStream:**

This class implements the ObjectOutput interface along with extending OutputStream class for writing serializable objects into the output stream. Following is the form of this class constructor:

ObjectOutputStream(OutputStream **out_Stream**) throws IOException

Here the serializable objects are written to the output stream, i.e. out_Stream.

**ObjectInput:**

The interface extends the DataInput interface and provides support for object deserialization. For making object deserializable, call to readObject() method defined in this interface is invoked. The method defined in this interface throws IOException in case of error occurrence.

**ObjectInputStream:**

This class implements the ObjectInput interface along with extending OutputStream class for reading serializable objects from the input stream. Following is the form of this class constructor:

ObjectInputStream(InputStream *in_Stream*) throws IOException, StreamCorruptedException

Here from the input stream, i.e. *in_Stream,* the serializable objects are read.

For an illustration of how object serialization and deserialization take place, we use the following program as an example. First, an object of class MyClass is instantiated. The objects define three instance variables of types int, String, and double. For this example, we tried to save and restore the information of these three-instance variables.

First, we create FileOutputStream with the file name "DemoSerial". Then ObjectOutputStream is created for the created file stream. Then for object serialization, we call writeObject() method defined in ObjectOutputStream. After serialization, it's flushed and closed.

Now we create FileInputStream by referring to the same file, i.e. "DemoSerial". Then we create ObjectInputStream for the respective input stream. Then we perform deserialization of object using readObject() method of ObjectOutputStream class. Then this object input stream is closed.

Here we defined MySeriClass that implements Serializable interface. In case MySeriClass does not implement Serializable, it will throw NotSerializableException error.

```
import java.io.*;
public class DemoSerialization1
{
  public static void main(String args[])
  {
    // Object Serialization
    try
    {
      MySeriClass objt_1 = new MySeriClass("Welcome Ram", 85, 99.5);
      System.out.println("First object: " + objt_1);
      FileOutputStream filestr_1 = new FileOutputStream("DemoSerial");
      ObjectOutputStream objtstr_1= new ObjectOutputStream(filestr_1);
      Objtstr_1.writeObject(objt_1);
      Objtstr_1.flush();
      Objtstr_1.close();
    }
    catch(Exception exp)
    {
      System.out.println("Catching Exception: " + exp);
      System.exit(0);
    }

    // Deserialization of object
    try
    {
      MySeriClass objt_2;
      FileInputStream filestr_2= new FileInputStream("DemoSerial");
      ObjectInputStream objtstr_2 = new ObjectInputStream(filestr_2);
      objt_2 = (MySeriClass)objtstr_2.readObject();
      objtstr_2.close();
      System.out.println("Second object: " + objt_2);
    }
```

```
      catch(Exception exp)
      {
        System.out.println("Catching Exception : " + exp);
        System.exit(0);
      }
    }
}
class MySeriClass implements Serializable
{
  String str;
  int a1;
  double db;
  public MySeriClass(String str, int a1, double db)
  {
    this.str = str;
    this.a1 = a1;
    this.db = db;
  }
  public String toString()
  {
    return "str=" + str + "; a1=" + a1 + "; db=" + db;
  }
}
```

**Output:**

First object: str=Welcome Ram; a1=85; db=99.5

Second object: str=Welcome Ram; a1=85; db=99.5

The same output of object instance variables here represents successful serialization and deserialization operation.

## ☞ Check Your Progress-3

1) What are Serialization and Deserialization?

……

……

……

……

2) What is the role of Externalizable?

……

……

……

……

3) Which of the following classes is used for input and output in Character streams?

(a) BufferedReader (b) FileInputStream (c) FileOutputStream (d) InputStream

……

……

……

……

4) Which of the following exceptions, read() method throws?

(a) InterruptedException (b) IOException (c) SystemInputException

(d) SystemException

……

……

……

……

5) Write a program to read only first five character from character array C_arr[]={'E','x','c','e','l','l','e','n','t'} using CharArrayReader Class. Also write a separate program to the write the content of character array CharArr[] to a file name "CharWriteClass.txt" using CharArrayWriter Class.

……

……

……

……

## 3.10 SUMMARY

Generally, java-based applications process some inputs and, based on the input generates output. The source of input and the destination of output may be a file, physical devices, network, etc. Java provides an API, JAVA IO, which helps in reading and writing the data, i.e. input and output. This chapter provides a clear view of the Java input/output hierarchy. This chapter also explained how various classes and interfaces in the IO package are organized and their purpose. Further, the File class is discussed that helps in handling files and directories. Various IO exceptions are discussed and summarized. Java supports two types of streams: Byte Streams and Character Streams. Both streams and associated classes are discussed in the chapter. This chapter discusses serialization that allows us to represent an object in a sequence of bytes, i.e. byte stream.

## 3.11 SOLUTION/ANSWERS TO YOUR PROGRESS

### ☞ Check Your Progress-1

**1)** Java I/O Package consists of input streams and output streams. The Input stream is used to read data from input source, while the Output stream helps to write data to destination. Here source and destination can be anything that generates, consumes and stores the data such as program, disk, peripheral device, etc.

**2)** DataInput: Allow reading of byte data from the binary stream and converting these data to any of Java primitive types.

FileFilter: Provides filter for abstract pathnames.
ObjectOutput: Extends DataOutput interface for the writing of objects and allow serialization of objects

**3)**
boolean exists(): Returns true if file exists otherwise return false.
boolean canRead(): Returns true if file is readable otherwise return false.
boolean isDirectory(): Returns true if reference is a directory otherwise return false.

**4)** The diagram of class hierarchy of I/O is shown in figure 2 in section 3.2.

## ☞ Check Your Progress 2

**1)** Byte streams process the data byte by byte, while Character streams process the data character by character in Unicode.

**2)** Closeable interface has only one abstract method i.e. close(). When a call to this method is invoked, system resources held by stream object are released, which can be used further in the program. Like Closeable interface, AutoCloseable interface also includes close() method but this method is automatically called unlike Closeable interface.

**3)** FileInputStream

**4)** PrintStream

**5)** In order to show how to perform concatenation of multiple input streams, a program is written. The program concatenates two file input streams named BufferReadFile.txt and ReadFile.txt, respectively. The SequenceInputStream first read the input from BufferReadFile.txt and then read input from ReadFile.txt (sequentially one by one in the order mentioned in the program).

```
import java.io.*;
class SequenceInStreamEx
{
   public static void main(String args[])throws Exception
   {
      FileInputStream fin_str1 =new FileInputStream("D:\\BufferReadFile.txt");
      FileInputStream fin_str2=new FileInputStream("D:\\ReadFile.txt");
      SequenceInputStream con_seq=new SequenceInputStream(fin_str1, fin_str2);
      int re;
      while((re=con_seq.read())!=-1)
      {
        System.out.print((char)re);
      }
      con_seq.close();
      fin_str1.close();
      fin_str2.close();
   }
}
```

## ☞ Check Your Progress 3

**1)** With serialization, users can write the state of objects to a byte stream. Serialization is useful to users as they allow users to save the content of their program in any non-volatile storage (file, memory) and later allows the users to restore these objects when required or necessary. The restoring process is known as Deserialization.

**2)** Externalizable is used to customize the serialization. In java, serialization and deserialization have automatically made the state of an object to be stored and

restored. But in some cases, the programmer desires to control this automatic process. For example, when implementing encryption and compression techniques, the programmer can use Externalizable interface to control these automatic processes.

**3)** BufferedReader

**4)** IOException

**5)** Program 1: For reading character using CharArrayReader Class

```
import java.io.CharArrayReader;
public class ProgressCheck3
{
  public static void main(String[] ag) throws Exception
  {
    char[] C_arr = {'E', 'x', 'c', 'e', 'l', 'l', 'e', 'n', 't' };
    CharArrayReader ch_rd1 = new CharArrayReader(C_arr,0,5);
    int k = 0;
    // Read until the end of a file
    while ((k = ch_rd1.read()) != -1)
    {
      char ch_read = (char) k;
      System.out.println(ch_read);
    }
  }
}
```

**Output:-**

```
E
x
c
e
l
```

Prcogram 2: For writing content of C_arr to file name “CharWriteClass.txt” using CharArrayWriter Class.

```
import java.io.CharArrayWriter;
import java.io.FileWriter;
public class CharProgramWrite3
{
   public static void main(String args[])throws Exception
   {
      char[] C_arr = {'E', 'x', 'c', 'e', 'l', 'l', 'e', 'n', 't'};
      String conv_str1 = new String(C_arr);
      CharArrayWriter arr_wr1 =new CharArrayWriter();
      arr_wr1.write(conv_str1);
      FileWriter new_f1=new FileWriter("D:\\CharWriteClass.txt");
      arr_wr1.writeTo(new_f1);
      new_f1.close();
      System.out.println("Successfully Written");
   }
}
```

**Output:** Successfully Written

## 3.12 REFERENCES /FURTHER READING


- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" Vol. 2, Pearson Education, 2013.
- https://docs.oracle.com/javase/tutorial/essential/io/