
UNIT 4 EXPLORING JAVA I/O

Structure	Page Nos.
4.0 Introduction	63
4.1 Objectives	63
4.2 Java I/O Classes and Interfaces	63
4.3 I/O Stream Classes	65
4.3.1 Input and Output Stream	
4.3.2 Input Stream and Output Stream Hierarchy	
4.4 Text Streams	72
4.5 Stream Tokenizer	75
4.6 Serialization	76
4.7 Buffered Stream	77
4.8 Print Stream	80
4.9 Random Access File	81
4.10 Summary	83
4.11 Solutions /Answers	83

4.0 INTRODUCTION

Input is any information that is needed by your program to complete its execution and *Output* is information that the program must produce after its execution. Often a program needs to bring information from an external source or to send out information to an external destination. The information can be anywhere in file. On disk, somewhere on network, in memory or in other programs. Also the information can be of any type, i.e., object, characters, images or sounds. In Java information can be stored, and retrieved using a communication system called streams, which are implemented in Java.io package.

The Input/output occurs at the program interface to the outside world. The input-output facilities must accommodate for potentially wide variety of operating systems and hardware. It is essential to address the following points:

- Different operating systems may have different convention how an end of line is indicated in a file.
- The same operating system might use different character set encoding.
- File naming and path name conventions vary from system to system.

Java designers have isolated programmers from many of these differences through the provision of package of input-output classes, java.io. Where data is stored objects of the reader classes of the java.io package take data, read from the files according to convention of host operating system and automatically converting it.

In this unit we will discuss how to work on streams. The discussion will be in two directions: pulling data into a program over an input stream and sending data from a program using an output stream.

In this unit you will work with I/O classes and interfaces, streams File classes, Stream tokenizer, Buffered Stream, Character Streams, Print Streams, and Random Access Files.

4.1 OBJECTIVES

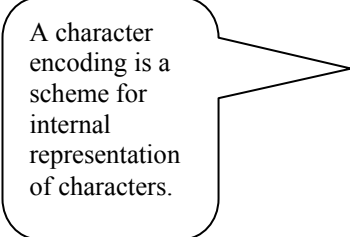
After going through this unit you will be able to:

- explain Java I/O hierarchy;

- handle files and directories using File class;
- read and write using I/O stream classes;
- differentiate between byte stream, character stream and text stream;
- use stream tokenizer in problem solving;
- use Print stream objects for print operations, and
- explain handling of random access files.

4.2 JAVA I/O CLASSES AND INTERFACES

The package java.io provides two sets of class hierarchies-one for reading and writing of bytes, and the other for reading and writing of characters. The InputStream and OutputStream class hierarchies are for reading and writing bytes. Java provides class hierarchies derived from Reader and Writer classes for reading and writing characters.



A character encoding is a scheme for internal representation of characters.

Java programs use 16 bit Unicode character encoding to represent characters internally. Other platforms may use a different character set (for example ASCII) to represent characters. The reader classes support conversions of Unicode characters to internal character storage.

Classes of the *java.io* hierarchy

Hierarchy of classes of Java.io package is given below:

- ◆ InputStream
 - FilterInputStream
 - BufferedInputStream
 - DataInputStream
 - LineNumberInputStream
 - PushbackInputStream
 - ByteArrayInputStream
 - FileInputStream
 - ObjectInputStream
 - PipedInputStream
 - SequenceInputStream
 - StringBufferInputStream
- ◆ OutputStream
 - FilterOutputStream
 - BufferedOutputStream
 - DataOutputStream
 - PrintStream
 - ByteArrayOutputStream
 - FileOutputStream
 - ObjectOutputStream
 - PipedOutputStream
- ◆ Reader
 - BufferedReader
 - LineNumberReader
 - CharArrayReader
 - FilterReader
 - PushbackReader
 - InputStreamReader
 - FileReader
 - PipedReader
 - StringReader

- ◆ Writer
 - BufferedWriter
 - CharArrayWriter
 - FilterWriter
 - OutputStreamWriter
 - FileWriter
 - PipedWriter
 - PrintWriter
 - StringWriter
- File
 - RandomAccessFile
 - FileDescriptor
 - FilePermission
 - ObjectStreamClass
 - ObjectOutputStreamField
 - SerializablePermission
 - StreamTokenizer.

Interfaces of Java.io

Interfaces in Java.io are given below:

- DataInput
- DataOutput
- Externalizable
- FileFilter
- FilenameFilter
- ObjectInput
- ObjectInputValidation
- ObjectOutput
- ObjectOutputStreamConstants
- Serializable

Each of the Java I/O classes is meant to do one job and to be used in combination with other to do complex tasks. For example, a `BufferedReader` provides buffering of input, nothing else. A `FileReader` provides a way to connect to a file. Using them together, you can do buffered reading from a file. If you want to keep track of line numbers while reading from a character file, then by combining the `File Reader` with a `LineNumberReader` to serve the purpose.

4.3 I/O STREAM CLASSES

I/O classes are used to get input from any source of data or to send output to any destination. The source and destination of input and output can be file or even memory. In Java input and output is defined in terms of an abstract concept called stream. A Stream is a sequence of data. If it is an input stream it has a source. If it is an output stream it has a destination. There are two kinds of streams: byte stream and character stream. The `java.io` package provides a large number of classes to perform stream IO.

The File Class

The `File` class is not I/O. It provides just an identifier of files and directories. Files and directories are accessed and manipulated through `java.io.file` class. So, you always remember that creating an instance of the `File` class does not create a file in the

operating system. The object of the File class can be created using the following types of File constructors:

```
File MyFile = new File("c:/Java/file_Name.txt");
File MyFile = new File("c:/Java", "file_Name.txt");
File MyFile = new File("Java", "file_Name.txt");
```

The first type of constructor accepts only one string parameter, which includes file path and file name, here in given format the file name is file_Name.txt and its path is c:/Java.

In the second type, the first parameter specifies directory path and the second parameter denotes the file name. Finally in the third constructor the first parameter is only the directory name and the other is file name. Now let us see the methods of the file class.

Methods

boolean exists() : Return true if file already exists.
boolean canWrite() : Return true if file is writable.
boolean canRead() : Return true if file is readable.
boolean isFile() : Return true if reference is a file and false for directories references.
boolean isDirectory() : Return true if reference is a directory.
String getAbsolutePath() : Return the absolute path to the application

You can see the program given below for creating a file reference.

```
//program
import java.io.File;
class FileDemo
{
    public static void main(String args[])
    {
        File f1 = new File ("/testfile.txt");
        System.out.println("File name : " + f1.getName());
        System.out.println("Path : " + f1.getPath());
        System.out.println("Absolute Path : " + f1.getAbsolutePath());
        System.out.println(f1.exists() ? "Exists" : "doesnot exist");
        System.out.println(f1.canWrite() ? "Is writable" : "Is not writable");
        System.out.println(f1.canRead() ? "Is readable" : "Is not readable");
        System.out.println("File Size : " + f1.length() + " bytes");
    }
}
```

Output: (When testfile.txt does not exist)

```
File name: testfile.txt
Path: \testfile.txt
Absolute Path: C:\testfile.txt
doesnot exist
Is not writable
Is not readable
File Size: 0 bytes
Output: (When testfile.txt exists)
File name : testfile.txt
Path : \testfile.txt
Absolute Path : C:\testfile.txt
Exists
Is writable
```

Is readable

File Size: 17 bytes

4.3.1 Input and Output Stream

Java Stream based I/O builds upon four abstract classes: `InputStream`, `OutputStream`, `Reader` and `Writer`. An object from which we can *read a sequence of bytes* is called an *input stream*. An object from which we can *write sequence of byte* is called *output stream*. Since byte oriented streams are inconvenient for processing. Information is stored in Unicode characters that inherit from abstract *Reader and Writer* superclasses.

All of the streams: The readers, writers, input streams, and output streams are automatically opened when created. You can close any stream explicitly by calling its `close` method. The garbage collector implicitly closes it if you don't close. It is done when the object is no longer referenced. Also, the *direction of flow* and *type* of data is not the matter of concern; algorithms for sequentially reading and writing data are basically the same.

Reading and Writing IO Stream

`Reader` and `InputStream` define similar APIs but for different data types. You will find, that both `Reader` and `InputStream` provide methods for marking a location in the stream, skipping input, and resetting the current position. *For example*, `Reader` and `Input Stream` defines the methods for reading characters and arrays of characters and reading bytes and array of bytes respectively.

Methods

`int read()` : reads one byte and returns the byte that was read, or `-1` if it encounters the end of input source.

`int read(char chrbuf[])`: reads into array of bytes and returns number of bytes read or `-1` at the end of stream.

`int read(char chrbuf[], int offset, int length)`: `chrbuf` – the array into which the data is read. `offset` - is offset into `chrbuf` where the first byte should be placed. `len` - maximum number of bytes to read.

`Writer` and `OutputStream` also work similarly. `Writer` defines these methods for writing characters and arrays of characters, and `OutputStream` defines the same methods but for bytes:

Methods

`int write(int c)`: writes a byte of data

`int write(char chrbuf[])` : writes character array of data.

`int write(byte chrbuf[])` writes all bytes into array `b`.

`int write(char chrbuf[], int offset, int length)` : Write a portion of an array of characters.

Check Your Progress 1

- 1) What is Unicode? What is its importance in Java?

.....

- 2) Write a program which calculates the size of a given file and then renames that file to another name.

.....

- 3) Write a program which reads string from input device and prints back the same on the screen.
-
-

4.3.2 Input Stream and Output Stream hierarchy

There are two different types of Streams found in Java.io package as shown in Figure 1a OutputStream and InputStream. Figure 1b shows the further classification of Inputstream.

The following classes are derived from InputStream Class.

InputStream: Basic input stream.

StringBufferInputStream: An input stream whose source is a string.

ByteArrayInputStream: An input stream whose source is a byte array.

FileInputStream: An input stream used for basic file input.

FilterInputStream: An abstract input stream used to add new behaviour to existing input stream classes.

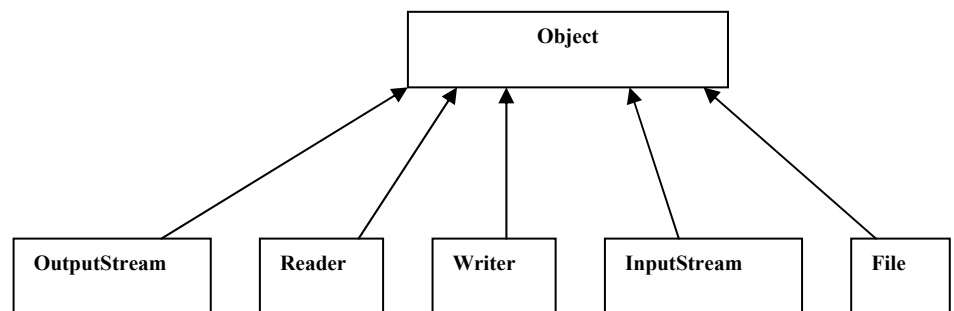


Figure 1(a): Types of stream in Java I/O

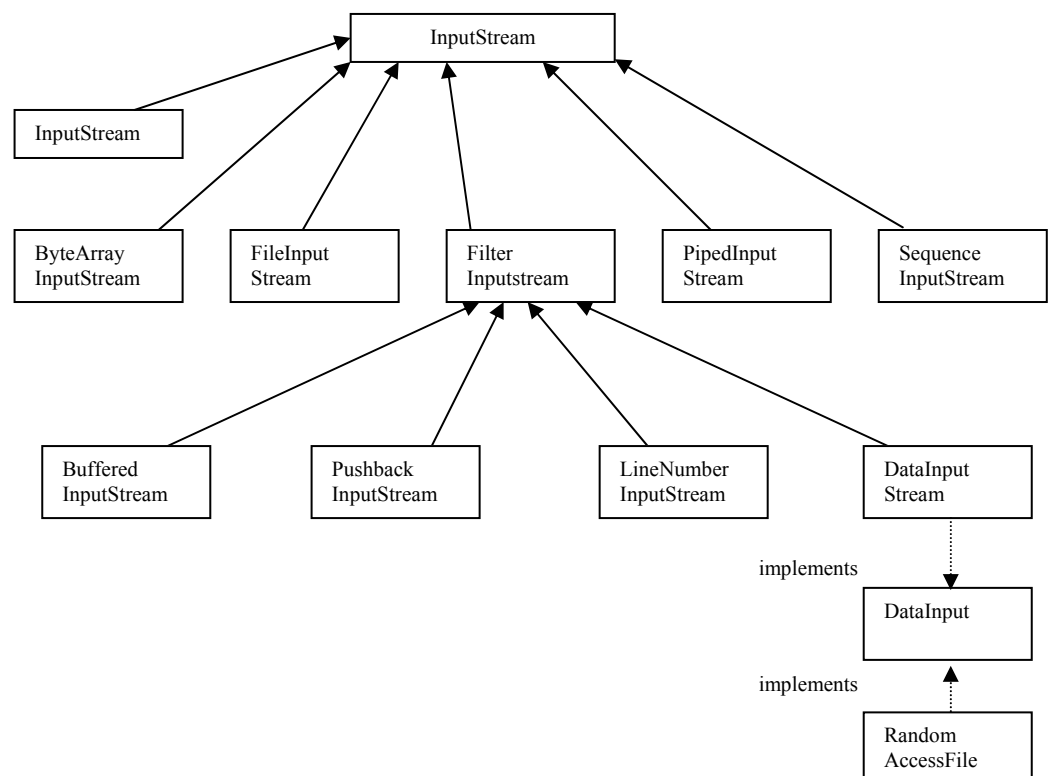


Figure 1(b): Input Stream Class

PipedInputStream: An input stream used for inter-thread communication.
 SequenceInputStream: An input stream that combines two other input streams.
 BufferedInputStream: A basic buffered input stream.
 PushbackInputStream: An input stream that allows a byte to be pushed back onto the stream after the byte is read.
 LineNumberInputStream: An input stream that supports line numbers.
 ataInputStream: An input stream for reading primitive data types.
 RandomAccessFile: A class that encapsulates a random access disk file.

Output streams are the logical counterparts to input streams and handle writing data to output sources. In Figure 2 the hierarchy of output Stream is given. The following classes are derived from outputstream.

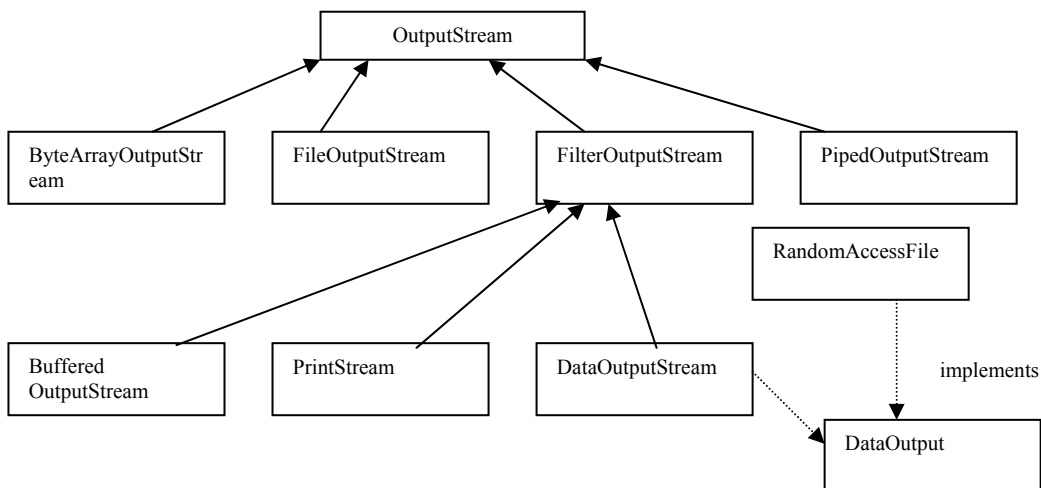


Figure 2: Output Stream Class Hierarchy

OutputStream: The basic output stream.
 ByteArrayOutputStream: An output stream whose destination is a byte array.
 FileOutputStream: Output stream used for basic file output.
 PipedOutputStream: An output stream used for inter-thread communication.
 BufferedOutputStream: A basic buffered output stream.
 PrintStream: An output stream for displaying text.
 DataOutputStream: An output stream for writing primitive data types.
 FilterOutputStream: An abstract output stream used to add new behaviour to existing output stream classes.

Layering byte Stream Filter

Some-times you will need to combine the two input streams. in Java. It is known as **filtered streams** (i.e., *feeding an existing stream to the constructor of another stream*). You should continue layering stream constructor until you have access to the functionality you want.

FileInputStream and *FileOutputStream* give you input and output streams attached to a disk file. For example giving file name or full path name of the file in a constructor as given below:

```
FileInputStream fin = new FileInputStream("Mydat.dat");
```

Input and output stream classes only support reading and writing on byte level, DataInputStream has a method that could read numeric. FileInputStream has no method to read numeric type and the DataInputStream has no method to get data from the file. If you have to read the numbers from file, first create a FileInputStream and pass it to DataInputStream, as you can see in the following code:

```
FileInputStream fin = new FileInputStream ("Mydat.dat");
```

```
DataInputStream din = new DataInputStream (fin)
doubles = din.readDouble();
```

If you want buffering, and data input from a file named Mydata.dat then write your code like:

```
DataInputStream din = new DataInputStream(new BufferedInputStream
                                           (new FileInputStream("employee.dat")));
```

Now let us take one example program in which we open a file with the binary `FileOutputStream` class. Then wrap it with `DataOutput` Stream class. Finally write some data into this file.

```
//program
import java.io.*;
public class BinFile_Test
{
    public static void main(String args[])
    {
        //data array
        double data [] = {1.3,1.6,2.1,3.3,4.8,5.6,6.1,7.9,8.2,9.9};
        File file = null;
        if (args.length > 0 ) file = new File(args[0]);
        if ((file == null) || !(file.exists()))
        {
            // new file created
            file = new File("numerical.txt");
        }
        try
        {
            // Wrap the FileOutputStream with DataOutputStream to obtain its writeInt()
            method
            FileOutputStream fileOutput = new FileOutputStream(file);
            DataOutputStream dataOut  = new DataOutputStream(fileOutput);
            for (int i=0; i < data.length;i++)
                dataOut.writeDouble(data[i]);
        }
        catch (IOException e)
        {
            System.out.println("IO error is there " + e);
        }
    }
}
```

Output:

You can see numeric.txt created in your system you will find something like:

?δììììî?ùTMTMTMTMTM\$@

Reading from a Binary File

Similarly, we can read data from a binary file by opening the file with a **FileInputStream** Object.

Then we wrap this with a **DataInputStream** class to obtain the many **readXxx()** methods that are very useful for reading the various primitive data types.

```
import java.io.*;
public class BinInputFile_Appl
```



```

{
public static void main(String args[])
{
    File file = null;
    if (args.length > 0 ) file = new File(args[0]);
    if ((file == null) || !file.exists())
    {
        file = new File("numerical.dat");
    }
    try
    { // Wrap the FileOutputStream with DataInputStream so that we can use its
      readDouble() method
        FileInputStream fileinput = new FileInputStream(file);
        DataInputStream dataIn  = new DataInputStream(fileinput);
        int i=1;
        while(true)
        {
            double d = dataIn.readDouble();
            System.out.println(i+".data="+d);
        }
    }
    catch (EOFException eof)
    {
        System.out.println("EOF Reached " + eof);
    }
    catch (IOException e)
    {
        System.out.println(" IO erro" + e);
    }
}
}
}

```

Output:

C:\JAVA\BIN>Java BinInputFile_Appl

```

1.data=1.3
1.data=1.6
1.data=2.1
1.data=3.3
1.data=4.8
1.data=5.6
1.data=6.1
1.data=7.9
1.data=8.2
1.data=9.9

```

Reading/Writing Arrays of Bytes

Some time you need to read/write some bytes from a file or from some network place. For this purpose classes Bytes Array InputStream and ByteArrayOutputStream are available.

Constructors of ByteArrayInputStream

ByteArrayInputStream(byte[] input_buf, int offset, int length)

ByteArrayInputStream(byte[] input_buf)

In the first constructor input_buf is the input buffer, offset indicates the position of the first byte to read from the buffer and length tells about the maximum number of bytes to be read from the buffer. Similarly for simple purpose another constructor is defined which uses input_buf its buffer array and also contain information to read this array.

For example in the code given below `ByteArrayInputStream` creates two objects `input1` and `input2`, where `input1` contains all letters while `input2` will contain only first three letters of input stream (“abc”).

```
String tmp = “abcdefghijklmnopqrstuvwxyz”
```

```
byte b[] = tmp.getBytes();
```

```
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
```

```
ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
```

`ByteArrayOutputStream` is an output stream that is used to write byte array. This class has two constructors- one with no parameter and another with an integer parameter that tells the initial size of output array.

```
ByteArrayOutputStream( )
```

```
ByteArrayOutputStream( int buf_length)
```

In the following piece of code you can see how `ByteArrayOutputStream` provides an output with an array of bytes.

```
ByteArrayOutputStream f = new ByteArrayOutputStream();
```

```
String tmp = “abcdefghijklmnopqrstuvwxyz”
```

```
byte b[] = tmp.getBytes();
```

```
f.write(b);
```

PushbackInput

Stream is an
input stream that
can unwrite the
last byte read

You often need to peek at the next byte to see whether it is a value that you expect. Use **PushbackInputStream** for this purpose. You can use this class to unwrite the last byte read is given in the following piece of code:

```
PushbackInputStream pbin = new PushbackInputStream
```

```
(new BufferedInputStream(new FileInputStream (“employee.dat”)));
```

You can read the next byte, as follow

```
int b = pbin.read();
```

```
if (b != '<') pbin.read().unread(b);
```

`Read()` and `Unread()` are the only methods that apply to pushback input stream. If you want to look ahead and also read numbers, you need to combine both a pushback inputstream and a data. Input. stream reference.

```
DataInputStream din = new DataInputStream
```

```
(PushBackInputStream pbin = new PushBackInputStream
```

```
(new bufferedInputStream (newFileInputStream(“employee.dat”))));
```

4.4 TEXT STREAMS

Binary IO is fast but it is not readable by humans. For example, if integer 1234 is saved as binary, it is written as a sequence of bytes **00 00 04 D2** (in hexadecimal notation). In text format it is saved as the string “1234”. Java uses its own character-encoding. This may be single byte, double byte or a variable byte scheme based on host system. If the Unicode encoding is written in text file, then the resulting file will unlikely be human readable. To overcome this Java has a set of stream filters that bridges the gap between Unicode encoded text and character encoding used by host. All these classes are descended from abstract *Reader and Writer classes*.

Reader and Writer classes

You may know that `Reader` and `Writer` are abstract classes that define the java model of streaming characters. These classes were introduced to support 16 bit Unicode

characters in java 1.1 and onwards and provide Reader/Writer for java 1.0 stream classes.

java 1.0

InputStream
OutputStream
FileInputStream
FileOutputStream
StringBufferInputStream

ByteArrayInputStream
ByteArrayOutputStream
PipedInputStream
PipedOutputStream

java 1.1

Reader
Writer
FileReader
FileWriter
StringReader

CharArrayReader
CharArrayWriter
PipedReader
PipedWriter

For *filter classes*, we also have corresponding *Reader* and *Writer* classes as given below:

FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract)
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
DataInputStream	DataInputStream
	BufferedReader (readLine())
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer
PushBackInputStream	PushBackReader

In addition to these classes java 1.1 also provides two extra classes which provide a bridge between byte and character streams. For reading bytes and translating them into characters according to specified character encoding we have InputStreamReader while for the reverse operation translating characters into bytes according to a specified character encoding java 1.1 provides OutputStreamWriter.

Reading Text from a File

Let us see how to use a FileReader stream to read character data from text files. In the following example read () method obtains one character at a time from the file. We are reading bytes and counting them till it meets the EOF (end of file). Here FileReader is used to connect to a file that will be used for input.

```
// Program
import java.io.*;
public class TextFileInput_Appl
{
    public static void main(String args[])
    {
        File file = null;
        if (args.length > 0 ) file= new File(args[0]);
        if (file == null || !file.exists())
        {
            file=new File("TextFileInput_Appl.Java");
        }
        int numBytesRead=0;
        try
        {
            int tmp =0 ;
            FileReader filereader = new FileReader (file);
            while( tmp != -1)
```

```
{
tmp= filereader.read();
if (tmp != -1) numBytesRead++;
}
}
catch (IOException e)
{
System.out.println(" IO erro" + e);
}
System.out.println(" Number of bytes read : " + numBytesRead);
System.out.println(" File.length()= "+file.length());
}
}
```

For checking whether we read all the bytes of the file “TextFileInput_Appl.Java” in this program we are comparing both, length of file and the bytes read by read() method. You can see and verify the result that both values are 656.

Output:

Number of bytes read: 656
File.length()= 656

Writing Text to a file

In the last example we read bytes from a file. Here let us write some data into file. For this purpose Java provides FileWriter class. In the following example we are demonstrating you the use of FileWriter. In this program we are accepting the input data from the keyboard and writes to a file a "textOutput.txt". To stop writing in file you can press ctrl+C which will close the file.

```
// Program Start here
import java.io.*;
public class TextFileOutput_Appl
{
    public static void main(String args[])
    {
        File file = null;
        file = new File("textOutput.txt");
        try
        {
            int tmp= 0;
            FileWriter filewriter = new FileWriter(file);
            while((tmp=System.in.read()) != -1)
            {
                filewriter.write((char) tmp);
            }
            filewriter.close();
        }
        catch (IOException e)
        {
            System.out.println(" IO erro" + e);
        }
        System.out.println(" File.length()="+file.length());
    }
}
```

Output:

C:\Java\Block3Unit3> Java TextFileOutput_Appl
Hi, test input start here.
^Z

4.5 STREAM TOKENIZER

Stream Tokenizer is introduced in JDK 1.1 to improve the `wc()` [Word Count] method, and to provide a better way for pattern recognition in input stream. (do you know how `wc()` method is improved by Stream Tokenizer ?) during writing program. Many times when reading an input stream of characters, we need to parse it to see if what we are reading is a word or a number or something else. This kind of parsing process is called “tokenizing”. A “token” is a sequence of characters that represents some abstract values stream. Tokenizer can identify members, quoted string and many other comment styles.

Stream Tokenizer Class is used to break any input stream into tokens, which are sets of bits delimited by different separator. A few separators already added while you can add other separators. It has two contributors given below:

There are 3 instance variable as defined in `streamtokenizer` **nval**, **sval** and **ttype**. **nval** is public double used to hold number, **sval** is public string holds word, and **ttype** is public integer to indicates the type of token. `nextToken()` method is used with stream tokenizer to parse the next token from the given input stream. It returns the value of token type **ttype**, if the token is word then **ttype** is equal to `TT_WORD`. There are some other values of **ttype** also which you can see in Table 1 given below. You can check in the given program below that we are using `TT_EOF` in the same way as we have used end of file (EOF) in our previous example. Now let me tell you how `wc()` is improved, different token types are available stream tokenizer which we can use to parse according to our choice, which were limitation of `wc()` method also.

Table 1: Token Types

Token Types	Meaning
TT_WORD	String word was scanned (The string field sval contains the word scanned)
TT_NUMBER	A number was scanned (only decimal floating point number)
TT_EOL	End – of - line scanned
TT_EOF	End – of –file – scanned

```
// Program Start here
import java.io.*;
public class tokenizer
{
    public static void main(String args[])
    {
        String sample="this 123 is an 3.14 \n simple test";
        try
        {
            InputStream in;
            in = new StringBufferInputStream(sample);
            StreamTokenizer parser = new StreamTokenizer(in);
            while(parser.nextToken() != StreamTokenizer.TT_EOF)
            {
                if (parser.ttype == StreamTokenizer.TT_WORD)
                    System.out.println("A word"+parser.sval);
                else if (parser.ttype == StreamTokenizer.TT_NUMBER)
                    System.out.println("A number: "+parser.nval);
            }
        }
    }
}
```

```
        else if (parser.ttype == StreamTokenizer.TT_EOL)
            System.out.println("EOL");
        else if (parser.ttype == StreamTokenizer.TT_EOF)
            throw new IOException("End of File");
        else
            System.out.println("other" +(char) parser.ttype);
    }
}
catch (IOException e)
{
    System.out.println(" IO erro" + e);
}
}
```

Output:

A word: this
A number: 123
A word: is
A word: an
A number: 3.14
A word: simple
A word: test

After passing the input stream, each token is identified by the program and the value of token is given in the output of the above program.

4.6 SERIALIZATION

You can read and write text, but with Java you can also read and write objects to files. You can say object I/O is serialization where you read/write state of an object to a byte stream. It plays a very important role sometimes when you are interested in saving the state of objects in your program to a persistent storage area like file. So that further you can De-Serialize to restore these objects, whenever needed. I have a question for you, why is implementing of serialization very difficult with other language and it is easy with Java?

In other languages it may be difficult, since objects may contain references to another objects and it may lead to circular references. Java efficiently implements serialization, though we have to follow different Conditions; we can call them, Conditions for serializability.

If you want to make an object serialized, the class of the object must be declared as public, and must implement serialization. The class also must have a no argument constructor and all fields of class must be serializable.

Java gives you the facility of serialization and deserialization to save the state of objects. There are some more functions which we often used on files like compression and encryption techniques. We can use these techniques on files using Externalizable interface. For more information you can refer to books given in references.

Check Your Progress 2

- 1) Why is PushbackInputStream is used?

.....
.....

- 2) Write a program to print all primitive data values into a File using FileWriter stream wrapped with a Printwriter.

.....

.....

- 3) How would you append data to the end of a file? Show the constructor for the class you would use and explain your answer. .

.....

.....

4.7 BUFFERED STREAM

What is Buffer? Buffer is a temporary storage place where the data can be kept before it is needed by the program that reads or writes data. By using buffer, you can get data without always going back to the original source of data. In JAVA, these Buffers allow you to do input/output operations on more than a byte at a time. It increases the performance, and we can easily manipulate, skip, mark or reset the stream of byte also.

Buffered Stream classes manipulate sequenced streams of byte data, typically associated with binary format files. For example, binary image file is not intended to be processed as text and so conversion would be appropriate when such a file is read for display in a program. There are two classes for bufferstream `BufferInputStream` and `BufferedOutputStream`.

A *Buffered input stream* fills a buffer with data that hasn't been handled yet. When a program needs this data, it looks to the buffer first before going to the original stream source. A *Buffered input stream* is created using one of the two constructors:

BufferedInputStream (InputStream) – Creates a buffered input stream for the specified Input Stream object.

BufferedInputStream (InputStream, int) – Creates a specified buffered input stream with a buffer of int size for a specified Inputstream object.

The simplest way to read data from a buffered input stream is to call its `read ()` method with no argument, which returns an integer from 0 to 255 representing the next byte in the stream. If the end of stream is reached and no bytes are available, `-1` is returned.

A *Buffered output stream* is created using one of the two constructors:

BufferedOutputStream (OutputStream) – Creates a buffered output stream for the specified Output Stream object.

BufferedOutputStream (OutputStream, int) – Creates a buffered output stream with a buffer of int size for the specified outputstream object.

The output stream's `write (int)` method can be used to send a single byte to the stream. `write (byte [], int, int)` method writes multiple bytes from the specified byte array. It specifies the starting point, and the number of bytes to write. When data is directed to a buffered stream, it is not output to its destination until the stream fills or buffered stream `flush()` method is used.

In the following example, we have explained a program for you, where we are creating buffer input stream from a file and writing with buffer output stream to the file.

```
// program
import java.lang.System;
import java.io.BufferedReader;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;
public class BufferedIOApp
{
    public static void main(String args[]) throws IOException
    {
        SequenceInputStream f3;
        FileInputStream f1 = new FileInputStream("String_Test.Java");
        FileInputStream f2 = new FileInputStream("textOutput.txt");
        f3 = new SequenceInputStream(f1,f2);
        BufferedInputStream inStream = new BufferedInputStream(f3);
        BufferedOutputStream outStream = new BufferedOutputStream(System.out);
        inStream.skip(500);
        boolean eof = false;
        int byteCount = 0;
        while (!eof)
        {
            int c = inStream.read();
            if(c == -1) eof = true;
            else
            {
                outStream.write((char) c);
                ++byteCount;
            }
        }
        String bytesRead = String.valueOf(byteCount);
        bytesRead+=" bytes were read\n";
        outStream.write(bytesRead.getBytes(),0,bytesRead.length());
        System.out.println(bytesRead);
        inStream.close();
        outStream.close();
        f1.close();
        f2.close();
    }
}
```

Character Streams

Character streams are used to work with any text files that are represented by ASCII characters set or Unicode set. Reading and writing file involves interacting with peripheral devices like keyboard printer that are part of system environment. As you know such interactions are relatively slow compared to the speed at which CPU can operate. For this reason *Buffered reader* and *Buffered Writer* classes are used to overcome from difficulties posed by speed mismatch.

When a read method of a *Buffered Reader* object, is invoked it might actually read more data from the file then was specifically asked for. It will hold on to additional data read from the file that it can respond to the next read request promptly without making an external read from the file.

Similarly *Buffered Writer* object might bundle up several small write requests in its own internal buffer and only make a single external write to file operation when it considers that it has enough data to make the external write when its internal buffer is full, or a specific request is made via call to its *flush* method. The external write occurs.

It is not possible to open a file directly by constructing a Buffered Reader or Writer Object. We construct it from an existing FileReader or FileWriter object. It has an advantage of not duplicating the file- opening functionality in many different classes

The following constructors can be used to create a BufferedReader:

BufferedReader (Reader) – Creates a buffered character stream associated with the specified Reader object, such as FileReader.

BufferedReader (Reader, int) – Creates a buffered character stream associated with the specified Reader object, and with a buffered of int size.

Buffered character string can be read using read() and read(char[],int,int). You can read a line of text using readLine() method. The readLine() method returns a String object containing the next line of text on the stream, not including the characters or characters that represent the end of line. If the end of stream is reached the value of string returned will be equal to **null**.

In the following program we have explained the use of BufferedReader where we can calculate statistic (No. of lines, No. of characters) of a file stored in the hard disk named “text.txt”.

```
// Program
import java.io.*;
public class FileTest
{
    public static void main(String[] args) throws IOException
    {
        String s = "textOutput.txt";
        File f = new File(s);
        if (!f.exists())
        {
            System.out.println("'" + s + "' does not exist. Bye!");
            return;
        }
        // open disk file for input
        BufferedReader inputFile = new BufferedReader(new FileReader(s));
        // read lines from the disk file, compute stats
        String line;
        int nLines = 0;
        int nCharacters = 0;
        while ((line = inputFile.readLine()) != null)
        {
            nLines++;
            nCharacters += line.length();
        }
        // output file statistics
        System.out.println("File statistics for '" + s + "'...");
        System.out.println("Number of lines = " + nLines);
        System.out.println("Number of characters = " + nCharacters);
        inputFile.close();
    }
}
```

Output:

File statistics for 'textOutput.txt'...

Number of lines = 1

Number of characters = 26

4.8 PRINT STREAM

A *PrintStream* is a grandchild of *OutputStream* in the Java class hierarchy—it has methods implemented that print lines of text at a time as opposed to each character at a time. It is used for producing formatted output.

The original intent of *PrintStream* was to print all of the primitive data types and String objects in a viewable format. This is different from *DataOutputStream*, whose goal is to put data elements on a stream in a way that *DataInputStream* can portably reconstruct them.

Constructors

PrintStream(OutputStream out): *PrintStream(OutputStream out)* creates a new print stream. This stream will not flush automatically; returns a reference to the new *PrintStream* object.

PrintStream(OutputStream out, boolean autoFlush): Creates a new print stream. If *autoFlush* is true, the output buffer is flushed whenever (a) a byte array is written, (b) one of the *println()* methods is invoked, or (c) a newline character or byte ('\n') is written; returns a reference to the new *PrintStream* object.

Methods

flush(): flushes the stream; returns a void

print(char c): prints a character; returns a void

println(): terminates the current line by writing the line separator string; returns a void

println(char c): prints a character and then terminates the line; returns a void

Two constructors of *PrintStream* are deprecated by java 1.1 because *PrintStream* does not handle Unicode character and is thus not conveniently internationalized. Deprecating the constructors rather than the entire class allows existing use of *System.out* and *System.err* to be compiled without generating deprecation warnings. Programmers writing code that explicitly constructs a *PrintStream* object will see deprecating warning when the code is compiled. Code that produces textual output should use the new *PrintWriter* Class, which allows the character encoding to be specified or default encoding to be accepted. *System.out* and *System.err* are the *only* *PrintStream* object you should use in programs.

PrintWriter

To open a file output stream to which text can be written, we use the *PrintWriter* class. As always, it is best to buffer the output. The following sets up a buffered file writer stream named *outFile* to write text into a file named *save.txt*. The Java statement that will construct the required *PrintWriter* object and its parts is:

```
PrintWriter outFile = new PrintWriter(new BufferedWriter(new  
FileWriter("save.txt")));
```

The object *outFile*, above, is of the *PrintWriter* class, just like *System.out*. If a string, *s*, contains some text, it is written to the file as follows:

```
outFile.println(s);
```

When finished, the file is closed as expected:

```
outFile.close();
```

In the following program we are reading text from the keyboard and writing the text to a file called junk.txt.

```
//program
import Java.io.*;
class FileWrite
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
        String s = "junk.txt";
        File f = new File(s);
        if (f.exists())
        {
            System.out.print("Overwrite " + s + " (y/n)? ");
            if(!stdin.readLine().toLowerCase().equals("y"))
                return;
        }
        // open file for output
        PrintWriter outFile = new PrintWriter(new BufferedWriter(new
        FileWriter(s)));
        System.out.println("Enter some text on the keyboard...");
        System.out.println("^z to terminate");
        String s2;
        while ((s2 = stdin.readLine()) != null)
            outFile.println(s2);
        outFile.close();
    }
}
```

Output:

```
Enter some text on the keyboard...
(^z to terminate)
How are you? ^z
```

4.9 RANDOM ACCESS FILE

The character and byte stream are all sequential access streams whose contents must be read or written sequentially. In contrast, Random Access File lets you randomly access the contents of a file. The *Random Access File* allows files to be accessed at a specific point in the file. They can be opened in read/write mode, which allows updating of a current file.

The *Random Access File* class is not derived from *InputStream* and *OutputStream*. Instead it implements *DataInput* and *DataOutput* and thus provides a set of methods from both *DataInputStream* and *DataOutputStream*.

An object of this class should be created when access to binary data is required but in non-sequential form. The same object can be used to both read and write the file.

To create a new Random Access file pass the name of file and mode to the constructor. The mode is either “r” (read only) or “rw” (read and write) *An IllegalArgumentException* will be thrown if this argument contains anything else.

Following are the two constructors that are used to create *RandomAccessFile*:

RandomAccessFile(String s, String mode) throws IOException and s is the file name and mode is "r" or "rw".

RandomAccessFile(File f, String mode) throws IOException and f is an File object, mode is 'r' or 'rw'.

Methods

long length() : returns length of bytes in a file

void seek(long offset) throws IOException: position the file pointer at a particular point in a file. The new position is current position plus the offset. The offset may be positive or negative.

long getFilePointer() throws IOException : returns the index of the current read write position within the file.

void close() throws IOException: close file and free the resource to system.

To explain how to work with random access files lets write a program, here in the following program we are writing a program to append a string "I was here" to all the files passed as input to program.

```
import java.io.*;
public class AppendToFile
{
    public static void main(String args[])
    {
        try
        {
            RandomAccessFile raf = new RandomAccessFile("out.txt", "rw");
            // Position yourself at the end of the file
            raf.seek(raf.length());
            // Write the string into the file. Note that you must explicitly handle line breaks
            raf.writeBytes("\n I was here\n");
        }
        catch (IOException e)
        {
            System.out.println("Error Opening a file" + e);
        }
    }
}
```

After executing this program: "I was here" will be appended in out.txt file.

Check Your Progress 3

- 1) Write a program to read text from the keyboard and write the text to a file called junk.txt.

.....

.....

.....

.....

.....

- 2) Create a file test.txt and open it in read write mode. Add 4 lines to it and randomly read from line 2 to 4 and line 1.

.....

.....

.....

- 3) Write a program using FileReader and FileWriter, which copy the file f1.text one character at a time to file f2.text.

.....

4.10 SUMMARY

This unit covered the Java.io package. Where you start with two major stream of Java I/O, the InputStream and OutputStream, then learnt the standard methods available in each of these classes. Next, you studied other classes that are in the hierarchy derived from these two major classes and their method to allow easy implementation of different stream types in Java.

In the beginning of this unit we studied File class and its methods for handling files and directories. Further we have looked at the different classes of the Java.io package. We have worked with streams in two directions: pulling data over an input stream or sending data from a program using an output stream. We have used byte stream for many type of non-textual data and character streams to handle text. Filters were associated with streams to alter the way information was delivered through the stream, or to alter the information itself.

Let us recall all the uses and need of the classes we just studied in this unit. Classes based upon the InputStream class allow byte-based data to be input and similarly OutputStream class allows the output. The DataInputStream and DataOutputStream class define methods to read and write primitive datatype values and strings as binary data in host independent format. To read and write functionality with streams InputStreamReader and OutputStreamWriter class is available in Java.io package. The RandomAccessFile class that encapsulates a random access disk file allows a file to be both read and written by a single object reference. Reading functionality for system.in can be provided by using input stream reader. Writing functionality for system.out can be provided by using Print writing. The stream tokenizer class simplifies the parsing of input files which consist of programming language like tokens.

4.11 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) Unicode is a 16-bit code having a large range in comparison to previous ASCII code, which is only 7 bits or 8 bits in size. Unicode can represent all the characters of all human languages. Since Java is developed for designing Internet applications, and worldwide people can write programs in Java, transformation of one language to another is simple and efficient. Use of Unicode also supports platform independence in Java.

2)

```
//program
import Java.io.*;
class rename
{
    public static void main(String args[])
    {
        File f1 = new File("out1.txt");
        File f2 = new File("out2.txt");
```

```
if(f1.exists())
{
    System.out.println(f1 + " File exists");
    System.out.println("size of file is: " + f1.length() + "bytes" );
    f1.renameTo(f2);
    System.out.println("Rename file is: " + f2);
    System.out.println("deleting file : " + f1);
    f1.delete();
}
else
    System.out.println("f1" + "file is not existing");
}
```

Output:

```
out1.txt File exists
size of file is: 22bytes
Rename file is: out2.txt
deleting file : out1.txt
```

3)

```
//program
import Java.io.*;
class ReadWrite
{
    public static void main(String args[] )throws IOException
    {
        byte[] c= new byte[10];
        System.out.println("Enter a string of 10 character");
        for (int i=0;i < 10;i++)
            c[i]=(byte)System.in.read();
        System.out.println("Following is the string which you typed: ");
        System.out.write(c);
    }
}
```

Output:

```
C:\Java\Block3Unit3>Java ReadWrite
Enter a string of 10 character
Hi how is life
Following is the string which you typed:
Hi how is
```

Check Your Progress 2

- 1) PushbackInputStream is an input stream that can unread bytes. This is a subclass of FilterInputStream which provides the ability to unread data from a stream. It maintains a buffer of unread data that is supplied to the next read operation. It is used in situations where we need to read some unknown number of data bytes that are delimited by some specific byte value. After reading the specific byte program can perform some specific task or it can terminate.
- 2) This program use FileWriter stream wrapped with a Printwriter to write binary data into file.

```
//program
import Java.io.*;
public class PrimitivesToFile_Appl
{
    public static void main(String args[])
    {
```

```

try
{
    FileWriter filewriter = new FileWriter("prim.txt");
    // Wrap the PrintWriter with a PrintWriter
    // for sending the output to the file
    PrintWriter printWriter = new PrintWriter(filewriter);
    boolean b=true;
    byte by=127;
    short s=1111;
    int i=1234567;
    long l=987654321;
    float f=432.5f;
    double d=1.23e-15;
    printWriter.print(b);
    printWriter.print(by);
    printWriter.print(s);
    printWriter.print(i);
    printWriter.print(l);
    printWriter.print(f);
    printWriter.print(d);
    printWriter.close();
}
catch (Exception e)
{
    System.out.println("IO error" + e);
}
}
}

```

Output:

File prim.txt will contain : true12711111234567987654321432.51.23E-15

- 3) First let us use the FileWriter and BufferedWriter classes to append data to the end of the text file. Here is the FileWriter constructor, you pass in true value in second argument to write to the file in append mode:

```
FileWriter writer = new FileWriter (String filename, boolean append);
```

An alternate answer is to use RandomAccessFile and skip to the end of the file and start writing

```

RandomAccessFile file = new RandomAccessFile(datafile,"rw");
file.skipBytes((int)file.length()); //skip to the end of the file
file.writeBytes("Add this text to the end of datafile"); //write at the end of the file
file.close();

```

Check Your Progress 3

- 1)
- ```

//program
import Java.io.*;
class FileWriteDemo
{
 public static void main(String[] args) throws IOException
 {
 // open keyboard for input

```

```

BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
//output file 'junk.txt'

```

```
String s = "junk.txt";
File f = new File(s);
if (f.exists())
{
 System.out.print("Overwrite " + s + " (y/n)? ");
 if(!stdin.readLine().toLowerCase().equals("y"))
 return;
}
PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter(s)));
System.out.println("Enter some text on the keyboard...");
System.out.println("^z to terminate");
String s2;
while ((s2 = stdin.readLine()) != null)
 outFile.println(s2);
outFile.close();
}
```

2)

```
//program
import Java.lang.System;
import Java.io.RandomAccessFile;
import Java.io.IOException;
public class RandomIOApp
{
 public static void main(String args[]) throws IOException
 {
 RandomAccessFile file = new RandomAccessFile("test.txt", "rw");
 file.writeBoolean(true);
 file.writeInt(123456);
 file.writeChar('j');
 file.writeDouble(1234.56);
 file.seek(1);
 System.out.println(file.readInt());
 System.out.println(file.readChar());
 System.out.println(file.readDouble());
 file.seek(0);
 System.out.println(file.readBoolean());
 file.close();
 }
}
```

Output:

```
123456
j
1234.56
true
```

3)

```
//program
import Java.io.*;
public class CopyFile
{
 public static void main(String[] args) throws IOException,
 FileNotFoundException
 {
 File inFile = new File("out2.txt");
 FileReader in = new FileReader(inFile);
 File outFile = new File("f2.txt");
 FileWriter out = new FileWriter(outFile);
```



```
// Copy the file one character at a time.
int c;
while ((c = in.read()) != -1)
 out.write(c);
in.close();
out.close();
System.out.println("The copy was successful.");
}
}
Output:
The copy was successful.
```