UNIT 2 I/O IN JAVA

Structure **Page Nos.**

2.0 INTRODUCTION

Input is any information that is needed by a program to complete its execution. *Output* is any information that the program must convey to the user. Input and Output are essential for applications development.

To accept input, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. Whether reading data from a file or from a socket, the concept of serially reading from, and writing to, different data sources is the same. For that very reason, it is essential to understand the features of top-level classes (Java.io.Reader, Java.io.Writer).

In this Unit you will be working on some basics of I/O (InputOutput) in Java such as Files creation through *streams* in Java code. A stream is a linear, sequential flow of bytes of input or output data. Streams are written to the file system to create files. Streams can also be transferred over the Internet.

In this Unit you will learn the basics of Java streams by reviewing the differences between byte and character streams, and the various stream classes available in the Java.io package. We will cover the standard process for standard Input (Reading from console) and standard output (writing to console).

2.1 OBJECTIVES

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

explain the basics of I/O operations in Java;
use stream classes in programming;
take inputs from console;
write output on console;
read from files, and
write to files.

2.2 I/O BASICS

Java input and output are based on the use of *streams*, or sequences of bytes that travel

from a source to a destination over a communication path. If a program is writing to a stream, you can consider it as a stream's *source*. If it is reading from a stream, it is the stream's *destination*. The communication path is dependent on the type of I/O being performed. It can consist of memory-to-memory transfers, a file system, a network, and other forms of I/O.

Streams are powerful because they abstract the details of the communication path from input and output operations. This allows all I/O to be performed using a common set of methods. These methods can be extended to provide higher-level custom I/O capabilities.

Three streams given below are created automatically:

System.out - standard output stream
System.in - standard input stream
System.err - standard error.

An **InputStream** represents a stream of data from which data can be read. Again, this stream will be either directly connected to a device or else to another stream.

An **OutputStream** represents a stream to which data can be written. Typically, this stream will either be directly connected to a device, such as a file or a network connection, or to another output stream.

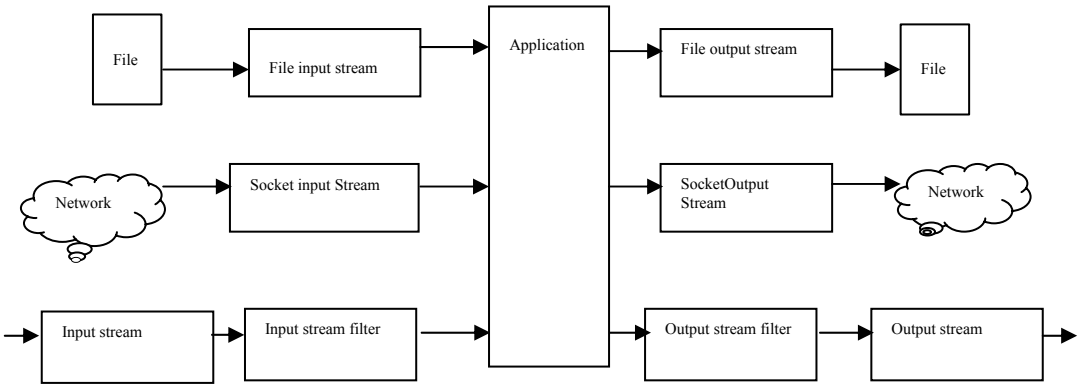

Figure 1: I/O stream basics

Java.io package

This package provides support for basic I/O operations. When you are dealing with the Java.io package some questions given below need to be addressed.

What is the file format: text or binary?
Do you want random access capability?
Are you dealing with objects or non-objects?
What are your sources and sinks for data?
Do you need to use filtering (You will know about it in later section of this unit)?

For example:

If you are using binary data, such as integers or doubles, then use the InputStream and OutputStream classes.
If you are using text data, then the Reader and Writer classes are right.

Exceptions Handling during I/O

Almost every input or output method throws an exception. Therefore, any time you do an I/O operation, the program needs to catch exceptions. There is a large hierarchy of I/O Exceptions derived from IOException class. Typically you can just catch

IOException, which catches all the derived class exceptions. However, some exceptions thrown by I/O methods are not in the IOException hierarchy, so you should be careful about exception handling during I/O operations.

2.3 STREAMS AND STREAM CLASSES

A *stream* is a one-way flow of bytes from one place to another over a communication path.

The Java model for I/O is based entirely on streams.

There are two types of streams: byte streams, and character streams.

Byte streams carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable programs, and byte codes – the class file that runs a Java program.

Character Streams are specialised types of byte streams that can handle only textual data.

Most of the functionality available for byte streams is also provided for character streams. The methods for character streams generally accept parameters of data type *char,* while *byte* streams work with *byte* data types. The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix Reader or Writer and byte-stream classes end with the suffix InputStream and OutputStream.

For example, to read files using character streams use the Java.io.FileReader class, and for byte streams use Java.io.FileInputStream.

Unless you are writing programs to work with binary data, such as image and sound files, use readers and writers (character streams) to read and write information for the following reasons:

They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
They are easier to internationalize because they are not dependent upon a specific character encoding.
They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

Now let us discuss byte stream classes and character stream classes one by one.

2.3.1 Byte Stream Classes

Java defines two major classes of byte streams: InputStream and OutputStream. To provide a variety of I/O capabilities subclasses are derived from these InputStream and OutputStream classes.

InputStream class

The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created. The close() method can explicitly close a stream.

Methods of InputStream class

The basic method for getting data from any InputStream object is the read()method. public abstract int read() throws IOException: This reads a single byte from the input stream and returns it to the stream.

public int read(byte[] bytes) throws IOException: fills an array with bytes read from the stream and returns the number of bytes read.

public int read(byte[] bytes, int offset, int length) throws IOException: fills an array from a stream starting at position offset, up to the length of the bytes. This returns either the number of bytes read or 1 for end of file.

public int available() throws IOException: the read()method always blocks when there is no data available. To avoid blocking, the program might need to ask ahead of time exactly how many bytes it can safely read without blocking. This method returns this number.

public long skip(long n): the skip() method skips over n bytes (passed as argument of skip()method) in a stream.

public synchronized void mark (int readLimit): this method marks the current position in the stream so it can be backed up later.

OutputStream class

The OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when created. An Output stream can be explicitly closed with the close() method.

Methods of OutputStream class

public abstract void write(int b) throws IOException: writes a single byte of data to an output stream.

public void write(byte[] bytes) throws IOException: writes the entire contents of the bytes array to the output stream.

public void write(byte[] bytes, int offset, int length) throws IOException: writes length, number of bytes starting at position offset from the bytes array.

The Java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions. Some of these classes are:

FileInputStream and FileOutputStream: read data from or write data to a file on the native file system.

PipedInputStream and PipedOutputStream : implement the input and output components of a pipe. Pipes are used to channel the output from one program (or thread) into the input of another. A PipedInputStream must be connected to a PipedOutputStream and a PipedOutputStream must be connected to a PipedInputStream.

ByteArrayInputStream and ByteArrayOutputStream : read data from, or write data to a byte array in memory.

ByteArrayOutputStream provides some additional methods not declared for OutputStream. The reset() method resets the output buffer to allow writing to restart at the beginning of the buffer. The size() method returns the number of bytes that have been written to the buffer. The write to () method is new.

SequenceInputStream: concatenate multiple input streams into one input stream.

StringBufferInputStream: allow programs to read from a StringBuffer as if it were an input stream.

Now let us see how Input and Output are handled in the program given below. This program creates a file and writes a string in it, and reads the number of bytes in the file.

```
// program for I/O
import Java.lang.System;
import Java.io.FileInputStream;
import Java.io.FileOutputStream;
import Java.io.File;
import Java.io.IOException;
public class FileIOOperations {
public static void main(String args[]) throws IOException {
 // Create output file test.txt
FileOutputStream outStream = new FileOutputStream("test.txt");
String s = "This program is for Testing I/O Operations";
for(int i=0;i<s.length();++i)
outStream.write(s.charAt(i));
outStream.close();
// Open test.txt for input
FileInputStream inStream = new FileInputStream("test.txt");
int inBytes = inStream.available();
System.out.println("test.txt has "+inBytes+" available bytes");
byte inBuf[] = new byte[inBytes];
int bytesRead = inStream.read(inBuf,0,inBytes);
System.out.println(bytesRead+" bytes were read");
System.out.println(" Bytes read are: "+new String(inBuf));
inStream.close();
File f = new File("test.txt");
f.delete();
}
}
```

Output:
test.txt has 42 available bytes
42 bytes were read
Bytes read are: This program is for Testing I/O Operations.

Filtered Streams

One of the most powerful aspects of streams is that one stream can chain to the end of another. For example, the basic input stream only provides a read()method for reading bytes. If you want to read strings and integers, attach a special data input stream to an input stream and have methods for reading strings, integers, and even floats.

The FilterInputStream and FilterOutputStream classes provide the capability to chain streams together. The constructors for the FilterInputStream and FilterOutputStream take InputStream and OutputStream objects as parameters:

public FilterInputStream(InputStream in)
public FilterOutputStream(OutputStream out)

FilterInputStream has four filtering subclasses, Buffer InputStream,

DataInputStream, LineNumberInputStream, and PushbackInputStream.
BufferedInputStream class: this maintains a buffer of the input data that it receives.

This eliminates the need to read from the stream’s source every time an input byte is needed.

DataInputStream class: this implements the DataInput interface, a set of methods that allow objects and primitive data types to be read from a stream.

LineNumberInputStream class: this is used to keep track of input line numbers.

PushbackInputStream class: this provides the capability to push data back onto the stream that it is read from so that it can be read again.

The FilterOutputStream class provides three subclasses: BufferedOutputStream, DataOutputStream and Printstream.

BufferedOutputStream class: this is the output class analogous to the BufferedInputStream class. It buffers output so that output bytes can be written to devices in larger groups.

DataOutputStream class: this implements the DataOutput interface. It provides methods that write objects and primitive data types to streams so that they can be read by the DataInput interface methods.

PrintStream class: this provides the familiar print() and println() methods.

You can see, in the program given below, how objects of classes FileInputStream, FileOutputStream, BufferedInputStream, and BufferedOutputStream are used for I/O operations.

```
//program
import Java.io.*;
public class StreamsIODemo
{
  public static void main(String args[])
  {
  try
     {
     int a = 1;
     FileOutputStream fileout = new FileOutputStream("out.txt");
     BufferedOutputStream buffout = new BufferedOutputStream(fileout);
     while(a<=25)
     {
     buffout.write(a);
     a = a+3;
     }
     buffout.close();
     FileInputStream filein = new FileInputStream("out.txt");
     BufferedInputStream buffin = new BufferedInputStream(filein);
     int i=0;
     do
      {
     i=buffin.read();
     if (i!= -1)
     System.out.println(" "+ i);
     } while (i != -1) ;
     buffin.close();
     }
      catch (IOException e)
     {
     System.out.println("Eror Opening a file" + e);
     }
     }
     }
```

Output:

1
4
7
10
13
16
19
22
25

2.3.2 Character Stream Classes

Character Streams are defined by using two class **Java.io.Reader and Java.io.Writer** hierarchies.

Both Reader and Writer are the abstract parent classes for character stream based classes in the Java.io package. Reader classes are used to read 16-bit character streams and Writer classes are used to write to 16-bit character streams. The methods for reading from and writing to streams found in these two classes and their descendant classes (which we will discuss in the next section of this unit) given below:

```
 int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Specialised Descendant Stream Classes

There are several specialised stream subclasses of the Reader and Writer class to provide additional functionality. For example, the BufferedReader not only provides buffered reading for efficiency but also provides methods such as "readLine()" to read a line from the input.

The following class hierarchy shows a few of the specialised classes in the Java.io package:

Reader

- BufferedReader
- LineNumberReader
- FilterReader
- PushbackReader
- InputStreamReader
- FileReader
- StringReader

Now let us see a program to understand how the read and write methods can be used.

```
import Java.io.*;
public class ReadWriteDemo
  {
  public static void main(String args[]) throws Exception
  {
   FileReader fileread = new FileReader("StrCap.Java");
   PrintWriter printwrite = new PrintWriter(System.out, true);
   char c[] = new char[10];
   int read = 0;
   while ((read = fileread.read(c)) != -1)
```

```
        printwrite.write(c, 0, read);
      fileread.close();
      printwrite.close();
     }
     }
```

Output:

```
class StrCap
{
public static void main(String[] args)
{
StringBuffer StrB = new StringBuffer("Object Oriented Programming is possible in
Java");
String  Hi = new  String("This morning is very good");
System.out.println("Initial Capacity of StrB is :"+StrB.capacity());
System.out.println("Initial length of StrB is :"+StrB.length());
//System.out.println("value displayed by the expression Hi.capacity() is:
"+Hi.capacity());
System.out.println("value displayed by the expression Hi.length() is: "+Hi.length());
System.out.println("value displayed by the expression Hi.charAt() is:
"+Hi.charAt(10));
 }
}
```

Note: Output of this program is the content of file StrCap.Java.You can refer Unit 3 of this block to see StrCap.Java file.

☞ Check Your Progress 1

1) What is a stream? Differentiate between stream source and stream destination.

……

……

……

2) Write a program for I/O operation using BufferedInputStream and BufferedOutputStream.

……

……

……

3) Write a program using FileReader and PrintWriter classes for file handling.

……...

……

……

2.4 THE PREDEFINED STREAMS

Java automatically imports the Java.lang package. This package defines a class called **System**, which encapsulates several aspects of run-time environment. **System** also contains three predefined stream objects, **in, out, and err.** These **objects** are declared as public and static within System. This means they can be used by other parts of your program without reference to your **System** object.

Access to standard input, standard output and standard error streams are provided via public static System.in, System.out and System.err objects. These are usually automatically associated with a user's keyboard and screen.

System.out refers to standard output stream. By default this is the console.
System.in refers to standard input, which is the keyboard by default.
System.err refers to standard error stream, which also is the console by default.
System.in is an object of InputStream. **System.out and System.err** are objects of **PrintStream.** These are byte streams, even though they typically are used to read and write characters from and to the console.

The predefined PrintStreams, out, and err within system class, are useful for printing diagnostics when debugging Java programs and applets. The standard input stream System.in is available, for reading inputs.

2.5 READING FROM AND WRITING TO, CONSOLE

Now, we will discuss how you can take input from console and see the output on console.

Reading Console Input

Java takes input from the console by reading from System.in. It can be associated with these sources with reader and sink objects by wrapping them in a reader object. For System.in an InputStreamReader is appropriate. This can be further wrapped in a BufferedReader as given below, if a line-based input is required.
BufferedReader br = new BufferedReader (new InputStreamReader(System.in))

After this statement br is a character-based stream that is linked to the console through Sytem.in

The program given below is used for receiving console input in any of your Java applications**.**

```
//program
import Java.io.*;
class ConsoleInput
 {
  static String readLine()
   {
     StringBuffer response = new StringBuffer();
     try
     {
      BufferedInputStream buff = new BufferedInputStream(System.in);
      int in = 0;
      char inChar;
      do
      {
        in = buff.read();
        inChar = (char) in;
        if (in != -1)
        {
         response.append(inChar);
         }
        } while ((in != 1) & (inChar != '\n'));
        buff.close();
        return response.toString();
```

```
        }
        catch (IOException e)
           {
           System.out.println("Exception: " + e.getMessage());
           return null;
           }
    }
    public static void main(String[] arguments)
      {
       System.out.print("\nWhat is your name?");
       String input = ConsoleInput.readLine();
       System.out.println("\nHello, " + input);
      }
      }
```

Output:

C:\JAVA\BIN>Java ConsoleInput

What is your name? Java Tutorial

Hello, Java Tutorial

Writing Console Output

You have to use System.out for standard Output in Java. It is mostly used for tracing errors, or for sample programs. These sources can be associated with a writer and sink objects by wrapping them in writer object. For standard output, an OutputStreamWriter object can be used, but this is often used to retain the functionality of print and println methods. In this case, the appropriate writer is PrintWriter. The second argument to PrintWriter constructor requests that the output will be flushed whenever the println method is used. This avoids the need to write explicit calls to the flush method in order to cause the pending output to appear on the screen. PrintWriter is different from other input/output classes as it doesn't throw an IOException. It is necessary to send check Error message, which returns true if an error has occurred. One more side effect of this method is that it flushes the stream. You can create a PrintWriter object as given below.

PrintWriter pw = new PrintWriter (System.out, true)

The ReadWriteDemo program discussed earlier in this section 2.3.2 of this Unit is for reading and then displaying the content of a file. This program will give you an idea how to use FileReader and PrintWriter classes.

You may have observed that close()method is not required for objects created for standard input and output. You should have a question in mind – whether to use System.out or PrintWriter? There is nothing wrong in using System.out for sample programs, but for real world applications, PrintWriter is easier.

2.6 READING AND WRITING FILES

These streams are most often used for the standard input (keyboard) and the standard output (display). Alternatively, input can arrive from a disk file using "input redirection", and output can be written to a disk file using "output redirection".

I/O redirection is convenient, but there are limitations to it. It is not possible to read data from a file using input redirection *and* to receive user input from the keyboard at same time. Also, it is not possible to read or write multiple files using input redirection. A more flexible mechanism to read or write disk files is available in Java through its *file streams*.

Java has two file streams – the *file reader stream* and the *file writer stream*. Unlike the standard I/O streams, file stream must explicitly "open" the stream before using it.

Although, it is not necessary to close after the operation is over, it is a good practice to "close" the stream.

Reading Files

Let's begin with the FileReader class. As with keyboard input, it is most efficient to work through the BufferedReader class. If input is text to be read from a file, let us say "input.txt," it will be opened as a file input stream as follows:

```
BufferedReader inputFile = new BufferedReader(new FileReader("input.txt"));
```

The line above opens, input.txt as a FileReader object and passes it to the constructor of the BufferedReader class. The result is a BufferedReader object, named inputFile.

To read a line of text from input.txt, use the readLine() method of the BufferedReader class.

```
String s = inputFile.readLine();
```

You can see that input.txt is *not* being read using input redirection. It is explicitly opened as a file input stream. This means that the keyboard is still available for input, so, the user can take the name of a file, instead of "hard coding".

Once you are finished with the operations on the file, the file stream is closed as follows:

```
inputFile.close();
```

Some additional file I/O services are available through Java's File class, which supports simple operations with filenames and paths. For example, if fileName is a string containing the name of a file, the following code checks if the file exists and, if so, proceeds only if the user enters "y" to continue.

```
File f = new File(fileName);
if (f.exists())
{
System.out.print("File already exists. Overwrite (y/n)? ");
if(!stdin.readLine().toLowerCase().equals("y"))
return;
}
```

The program written below opens a text file called input.txt and to count the number of lines and characters in that file.

```
//program
import Java.io.*;
public class FileOperation
{
public static void main(String[] args) throws IOException
{
// the file must be called 'input.txt'
String s = "input.txt"
File f = new File(s);
//check if file exists
if (!f.exists())
{
System.out.println("\'" + s + "\' does not exit!");
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 ‘input.txt’…
Number of lines = 3
Number of characters = 7

Writing Files

You can open a file output stream to which text can be written. For this, use the FileWriter class. As always, it is best to buffer the output. The following code sets up a buffered file writer stream named outFile to write text into a file named output.txt.

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

The object outFile, is an object of PrintWriter class, just like System.out. If a string, s, contains some text, to be written in “output.text”. It is written to the file as follows:
outFile.println(s);

When finished, the file is closed as:
outFile.close();

FileWriter constructor can be used with two arguments, where the second argument is a boolean type specifying an "append" option. For example, the expression, new FileWriter("output.txt", true) opens “output.txt” as a file output stream. If the file currently exists, subsequent output is appended to the file.

One more possibility is of opening an existing *read-only* file for writing. In this case, the program terminates with an “access is denied” exception. This should be caught and dealt with in the 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));
String s = "output.txt";
// check if output file exists
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)");
// read from keyboard, write to file output stream
String s2;
while ((s2 = stdin.readLine()) != null)
outFile.println(s2);
// close disk file
outFile.close();
}
}
```

Output:
Enter some text on the keyboard...
(^z to terminate)
hello students ! enjoying Java Session
^Z
Open out.txt you will find
“hello students ! enjoying Java Session” is stored in it.

☞ Check Your Progress 2

1) Which class may be used for reading from the console?

……

……

……

2) Objects of which class may be used for writing on the console.

……

……

……

3) Write a program to read the output of a file, and display it on the console.

……

……

……

2.7 THE TRANSIENT AND VOLATILE MODIFIERS

Object serialization is very important aspect of I/O programming. Now, we will discuss serialization.

Object serialization

Serialization takes all the data attributes, writes them out as an object, and reads them back in as an object. For an object to be saved to a disk file it needs to be converted to a serial form. An object can be used with streams by implementing the serializable

interface. The serialization is used to indicate that objects of that class can be saved and retrieved in serial form. Object serialization is quite useful when you need to use object persistence. By object persistence, the stored object continues to serve the purpose even when no Java program is running, and stored information can be retrieved in a program so it can resume functioning, unlike the other objects that cease to exist when object stops running.

DataOuputStreams and DataInputStreams are used to write each attribute out individually, and then can read them back in at the other end. But to deal with the entire object, not its individual attributes, store away an object or send it over a stream of objects. Object serialization takes the data members of an object and stores them away or retrieves them, or sends them over a stream.

ObjectInput interface is used for input, which extends the DataInput interface, and ObjectOutput interface is used for output, which extends DataOutput. You are still going to have the methods readInt(), writeInt() and so forth. ObjectInputStream, which implements ObjectInput, is going to read what ObjectOutputStream produces.

Working of object serialization

For ObjectInput and ObjectOutput interface, the class must be serializable. The serializable characteristic is assigned when a class is first defined. Your class must implement the serializable interface. This marker is an interface that says to the Java virtual machine that you want to allow this class to be serializable. You don't have to add any additional methods.

There are a couple of other features of a serializable class. First, there is a zero parameter constructor. When you read the object, it needs to be able to construct and allocate memory for an object, and it is going to fill in that memory from what it has read from the serial stream. The static fields, or class attributes, are not saved because they are not part of an object.

If you do not want a data attribute to be serialized, you can make it transient. That would save on the amount of storage or transmission required to transmit an object. The transient indicates that the variable is not part of the persistent state of the object and will not be saved when the object is archived. Java defines two types of modifiers **Transient** and **Volatile.**

The **volatile** command indicates that the variable is modified asynchronously by concurrently running threads.

Transient Keyword

When an object that can be serialized, you have to consider whether all the instance variables of the object will be saved or not. Sometimes, you have some objects or sub objects which carry sensitive information like a password. If you serialize such objects even if information (sensitive information) is private in that object it can be accessed from outside. To control this you can turn off serialization on a field- by-field basis using the transient keyword.

See the program, given below, to create a login object that keeps information about a login session. In case you want to store the login data, but without the password, the easiest way to do it is to implements **Serializable** and mark the **password** field as **transient**.

```
//Program
import Java.io.*;
import Java.util.*;
public class SerialDemo implements Serializable
{
  private Date date = new Date();
```

```
 private String username;
 private transient String password;
 SerialDemo(String name, String pwd)
 {
  username = name;
  password = pwd;
 }
 public String toString()
  {
  String pwd = (password == null) ? "(n/a)" : password;
  return "Logon info: \n   " + "Username: " + username +
    "\n   Date: " + date +  "\n   Password: " + pwd;
  }
 public static void main(String[] args)
 throws IOException, ClassNotFoundException
  {
  SerialDemo a = new SerialDemo("Java", "sun");
  System.out.println( "Login is = " + a);
 ObjectOutputStream 0 = new ObjectOutputStream( new
 FileOutputStream("Login.out"));
 0.writeObject(a);
 0.close();
 // Delay:
 int seconds = 10;
 long t = System.currentTimeMillis()+ seconds * 1000;
 while(System.currentTimeMillis() < t)
    ;
 // Now get them back:
 ObjectInputStream in = new ObjectInputStream(new
 FileInputStream("Login.out"));
 System.out.println("Recovering object at " + new Date());
 a = (SerialDemo)in.readObject();
 System.out.println( "login a = " + a);
  }
 }
```

Output:
Login is = Logon info:
Username: Java
Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: sun
Recovering object at Thu Feb 03 04:06:32 GMT+05:30 2005
login a = Logon info:
Username: Java

Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: (n/a)

In the above exercise, Date and Username fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, and so it is not stored on the disk. Also the serialization mechanism makes no attempt to recover it. The **transient** keyword is for use with **Serializable** objects only.

Another example of a transient variable is an object referring to a file or an input stream. Such an object must be created anew when it is part of a serialized object loaded from an object stream.

```
// Donot serialize this field
private transient FileWriter outfile;
```

Volatile Modifier

The volatile modifier is used when you are working with multiple threads. The Java language allows threads that access shared variables to keep private working copies of the variables. This allows for a more efficient implementation of multiple threads. These working copies need to be reconciled with the master copies in the shared (main) memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock and conventionally enforcing mutual exclusion for those shared variables. Only variables may be volatile. Declaring them so indicates that such methods may be modified asynchronously.

2.8 USING NATIVE METHODS

You will often feel the requirement that a Java application *must* communicate with the environment outside of Java. This is, perhaps, the main reason for the existence of native methods. Java implementation needs to communicate with the underlying system – such as an operating system (Solaris or Win32, or a Web browser), custom hardware, (such as a PDA, set-top-device), etc. Regardless of the underlying system, there must be a mechanism in Java to communicate with that system. Native methods provide a simple, clean approach to providing this interface between the Java and the non-Java world without burdening the rest of the Java application with special knowledge.

Native Method is a method which is not written in Java, and is outside of the JVM in a library. This feature is not special to Java. Most languages provide some mechanism to call routines written in another language. In C++, you must use the extern "C" statement to signal that the C++ compiler is making a call to C functions.

To declare a native method in Java, a method is preceded with native modifiers much like you use the public or static modifiers, but don't define any body for the method, but simply place a semicolon in its place.

For example:

```
 public native int meth();
```

The following class defines a variety of native methods:

```
public class IHaveNatives
{
native public void Native1(int x) ;
native static public long Native2();
native synchronized private float Native3( Object o ) ;
native void Native4(int[]ary) throws Exception ;
}
```

Native methods can be static methods, thus not requiring the creation of an object (or instance of a class). This is often convenient when using native methods to access an existing C-based library. Naturally, native methods can limit their visibility with the public, private, protected, or unspecified *default* access.

Every other Java method modifier can be used along with native, except *abstract*. This is logical, because the native modifier implies that an implementation exists, and the abstract modifier insists that there is no implementation.

The following program is a simple demonstration of native method implementation.

```
class ShowMsgBox
{
 public static void main(String [] args)
  {
```

```
  ShowMsgBox app = new ShowMsgBox();
  app.ShowMessage("Generated with Native Method");
  }
 private native void ShowMessage(String msg);
  {
  System.loadLibrary("MsgImpl");
 }
 }
```

☞ Check Your Progress 3

1) What is Serialization?

……..

……..

……..

2) Differentiate between Transient and Volatile keyword.

……

……

……..

3) What are native methods?

……

……

……..

2.9 SUMMARY

This Unit covers various methods of I/O streams binary, character and object in Java. This Unit briefs that input and output in the Java language is organised around the concept of streams. All input is done through subclasses of InputStream and all output is done through subclasses of OutputStream except for RandomAccessFile. We have seen, in this Unit, how various streams can be combined together to get the added functionality of standard input and stream input. In this Unit you have also learned the operations of reading from a file and writing to a file. For this purpose objects of FileReader and FileWriter classes are used.

2.10 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) A stream is a sequence of bytes of undetermined length that travel from one place to another over a communication path.
Places from where the streams are picked up are known as stream sources. A source may be a file, input device or a network place-generating stream.
Places where streams are received or stored are known as stream destinations. A stream destination may be a file, output device or a network place ready to receive stream.

2) This IO program is written using FileInputStream, BufferedInputStream, FileOutputStream, and BufferedOutputStream classes.

```
import java.io.*;
public class IOBuffer
{
  public static void main(String args[])
  {
  try
    {
    int x = 0;
    FileOutputStream FO = new FileOutputStream("test.txt");
    BufferedOutputStream BO = new BufferedOutputStream(FO);
    //Writing Data in BO
    while(x<=25)
    {
     BO.write(x);
      x = x+2;
     }
    BO.close();
    FileInputStream FI = new FileInputStream("test.txt");
    BufferedInputStream BI= new BufferedInputStream(FI);
    int i=0;
    do
    {
      i=BI.read();
      if (i!= -1)
      System.out.println(" " +i);
     } while (i != -1) ;
     BI.close();
     }
     catch (IOException e)
      {

      System.out.println("Eror Opening a file" + e);
      }
   }
}
```

3) This program is written using FileReader and PrintWriter classes.

```
import java.io.*;
public class FRPW
{
public static void main(String args[]) throws Exception
    {
    FileReader FR = new FileReader("Intro.txt");
    PrintWriter PW = new PrintWriter(System.out, true);
    char c[] = new char[10];
    int read = 0;
    while ((read = FR.read(c)) != -1)
    PW.write(c, 0, read);
    FR.close();
    PW.close();
   }
  }
```

Check Your Progress 2

1) InputStream class

2) PrintStream class

3) This program reads the content from Intro.txt file and prints it on the console.

```
import java.io.*;
public class PrintConsol
{
 public static void main(String[] args)
 {
  try
   {
    FileInputStream FIS = new FileInputStream("Intro.txt");
     int n;
     while ((n = FIS.available()) > 0)
     {
     byte[] b = new byte[n];
     int result = FIS.read(b);
     if (result == -1) break;
     String s = new String(b);
     System.out.print(s);
      } // end while
     FIS.close();
    } // end try
    catch (IOException e)
    {
    System.err.println(e);
     }
   System.out.println();
  }
 }
```

Check Your Progress 3

1) Serialization is a way of implementation that makes objects of a class such that they are saved and retrieved in serial form. Serialization helps in making objects persistent.

2) Volatile indicates that concurrent running threads can modify the variable asynchronously. Volatile variables are used when multiple threads are doing the work. Transient keyword is used to declare those variables whose value need not persist when an object is stored.

3) Native methods are those methods, which are not written in Java, but they communicate with Java applications to provide connectivity or attach some systems, such as OS, Web browsers, or PDA, etc.