CHAPTER

24

STREAMS AND FILES

Streams facilitate transporting data from one place to another. Different streams are needed to send or receive data through different sources, such as to receive data from keyboard, we need a stream and to send data to a file, we need another stream. Without streams, it is not possible to move data in Java.

Stream

A stream carries data just as a water pipe carries water from one place to another (Figure 24.1). Streams can be categorized as 'input streams' and 'output streams'. Input streams are the streams which receive or read data while output streams are the streams which send or write data. All streams are represented by classes in `java.io` (input-output) package.

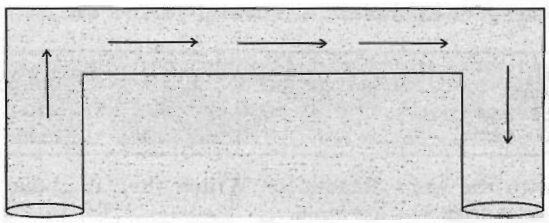

Figure 24.1 A stream to move data

Now, let us see how a stream works. We know an input stream reads data. So, to read data from keyboard, we can attach the keyboard to an input stream, so that the stream can read the data typed on the keyboard.

```
DataInputStream dis = new DataInputStream(System.in);
```

In the above statement, we are attaching the keyboard to `DataInputStream` object. The keyboard is represented by `System.in`. Now, `DataInputStream` object can read data coming from the keyboard. Here, `System` is a class and `in` is a field in `System` class. In fact, the `System` class has the following 3 fields:

- `System.in`: represents `InputStream` object. This object represents the standard input device, that is keyboard by default.
- `System.out`: represents `PrintStream` object. This object by default represents the standard output device, that is monitor.

- `System.err`: represents `PrintStream` object. This object by default represents the stand output device, that is monitor.

So, we can also use `System.err` to print something on the monitor, just like `System.out`.

Important Interview Question

What is the difference between System.out and System.err ?

Both are used to display messages on the monitor. System.out is used to display normal messages as:

System.out.println("Hello");

System.err is used to display any error messages in the program as:

System.err.println("This is an error");

Please observe that the keyboard is represented by `System.in` which internally cre `InputStream` object. It means the keyboard is an `InputStream`. Similarly, the monitor represented by `PrintStream`. In this way, streams represent input/output devices in Java. Eve we change the keyboard or monitor, we can still use the same streams to handle those devices this way, streams are useful to handle the input/output devices irrespective of their make.

Important Interview Question

What is the advantage of stream concept?

Streams are mainly useful to move data from one place to another place. This concept can be used to receive data from an input device and send data to an output device.

Another classification of streams is 'byte streams' and 'text streams'. Byte streams represent dat the form of individual bytes. Text streams represent data as characters of each 2 bytes. If a cl name ends with the word 'Stream', then it comes under byte streams. `InputStream` reads bytes `OutputStream` writes bytes. For example:

```
FileInputStream
FileOutputStream
BufferedInputStream
BufferedOutputStream
```

If a class name ends with the word 'Reader' or 'Writer' then it is taken as a text stream. Rea reads text and Writer writes text. For example,

```
FileReader
FileWriter
BufferedReader
BufferedWriter
```

Byte streams are used to handle any characters (text), images, audio, and video files. For exam to store an image file (`.gif` or `.jpg`), we should go for a byte stream. The important classes of streams are shown in Figure 24.2(a) and 24.2(b).

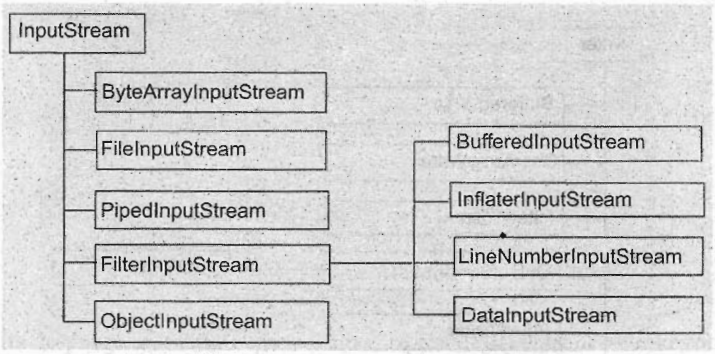

Figure 24.2(a) byte stream classes for reading data

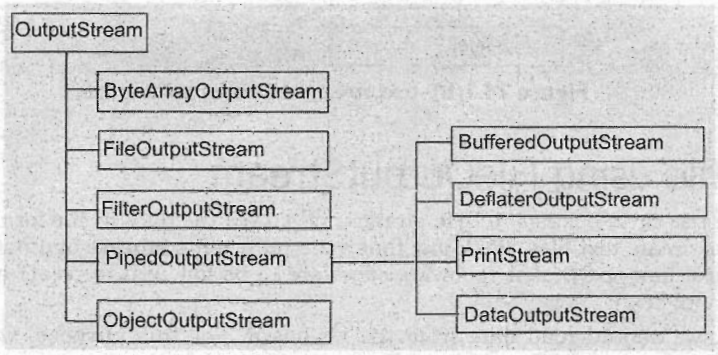

Figure 24.2(b) byte stream classes for writing data

Character or text streams can always store and retrieve data in the form of characters (or text) only. It means text streams are more suitable for handling text files like the ones we create in Notepad. They are not suitable to handle the images, audio, or video files. The important classes of character streams are shown in Figure 24.3(a) and 24.3(b).

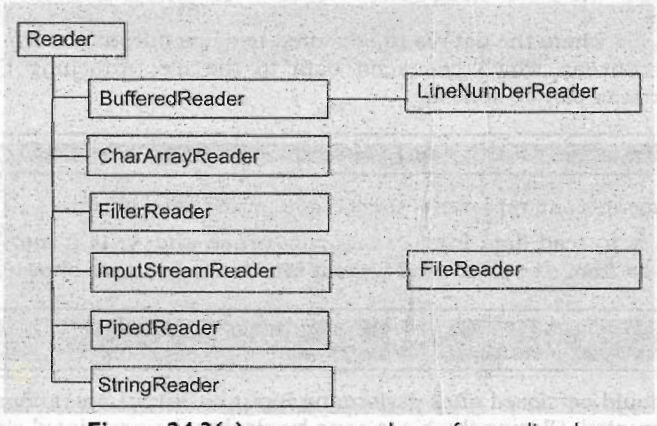

Figure 24.3(a) text stream classes for reading data

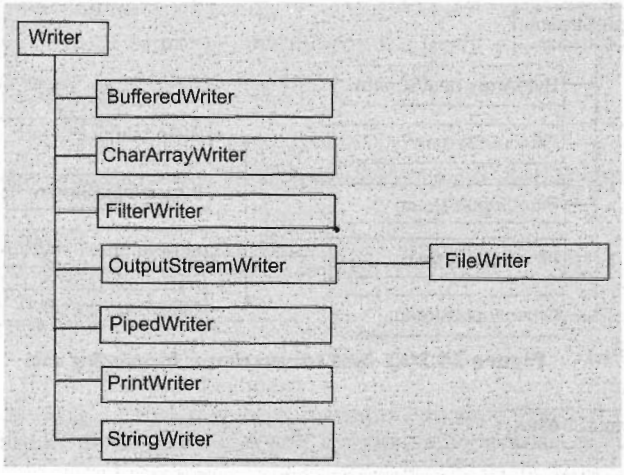

Figure 24.3(b) text stream classes for writing data

Creating a file using FileOutputStream

`FileOutputStream` class belongs to byte stream and stores the data in the form of individual b It can be used to create text files. We know that a file represents storage of data on a second st media like a hard disk or CD. The following steps are to be followed to create a text file that s some characters (or text):

- First of all, we should read data from the keyboard. For this purpose, we should attach keyboard to some input stream class. The code for using `DataInputStream` class for rea data from the keyboard is as:

```
DataInputStream dis = new DataInputStream(System.in);
```

Here, `System.in` represents the keyboard which is linked with `DataInputStream` object, th `dis`.

- Now, attach a file where the data is to be stored to some output stream. Here, we take the of `FileOutputStream` which can send data to the file. Attaching the file `myfile.t` `FileOutputStream` can be done as:

```
FileOutputStream fout = new FileOutputStream("myfile.txt");
```

In the above statement, `fout` represents the `FileOutputStream` object.

- The next step is to read data from `DataInputStream` and write it into `FileOutputStr` means read data from `dis` object and write it into `fout` object, as shown here:

```
ch =(char)dis.read();  //read one character into ch
fout.write(ch);  //write ch into file
```

Finally, any file should be closed after performing input or output operations on it, else the d the file may be corrupted. Closing the file is done by closing the associated streams. For exam `fout.close();` will close the `FileOutputStream`, hence there is no way to write data into the These steps are shown in Figure 24.4 and implemented in Program 1.

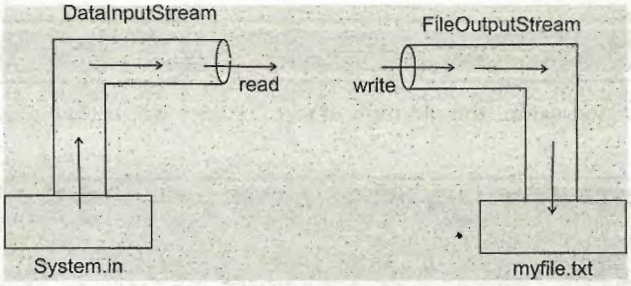

Figure 24.4 Creating a text file

Program 1: Write a program which shows how to read data from the keyboard and write it to `myfile.txt` file.

```
//Creating a text file using FileOutputStream
import java.io.*;
class CreateFile
{
      public static void main(String args[ ])
      throws IOException
      {

            //attach keyboard to DataInputStream
            DataInputStream dis = new DataInputStream(System.in);

            //attach myfile to FileOutputStream
            FileOutputStream fout = new FileOutputStream("myfile.txt");

            System.out.println("Enter text (@ at the end): ");

            char ch;

            //read characters from dis into ch. Then write them into fout.
            //repeat this as long as the read character is not @
            while((ch =(char)dis.read()) != '@')
            fout.write(ch);

            //close the file
            fout.close();
      }
}
```

Output:

```
C:\> javac CreateFile.java
C:\> java CreateFile
Enter text (@ at the end):
This is my file line one
This is my file line two
@

C:\> type myfile.txt
This is my file line one
This is my file line two
```

In this program, we read data from the keyboard and write it to `myfile.txt` file. This program accepts data from the keyboard till the user types `@` when he does not want to continue.

Here, we stored two lines of text into `myfile.txt`. To view the contents of `myfile.txt`, we can use `type` command in DOS or `cat` command in UNIX. For example,

```
Type myfile.txt
will display the contents of myfile.txt.
```

If Program 1 is executed again, the old data of `myfile.txt` will be lost and any recent data is only stored into the file.

```
C:\> java CreateFile
Enter text (@ at the end):
This is my third line
@
C:\> type myfile.txt
This is my third line
```

Notice that the file `myfile.txt` has now stored only the third line which has been entered in the preceding code output. The previous two lines have been deleted from the file and the file has been created as a fresh file. If we do not want to lose the previous data of the file, and just append the new data at the end of already existing data, then we should open the file by writing `true` along with the filename as:

```
FileOutputStream fout = new FileOutputStream("myfile.txt", true);
```

When the above statement is used—even though the program is run several times—all previous data will be preserved and new data will be added to the old data.

Improving Efficiency using BufferedOutputStream

Normally, whenever we write data into a file using `FileOutputStream` as:

```
fout.write(ch);
```

we call `write()` method to write a character `(ch)` into the file. The Operating system is invoked to write the character into the file. So, if we write 10 characters, the underlying Operating system would be called 10 times and it will write the characters into the file. This wastes a lot of time.

On the other hand, if Buffered classes are used, they provide a buffer (temporary block of memory) which is first filled with characters and then all the characters from the buffer are at once written into the file by the Operating system. This takes less time and hence efficiency is more. See the behavior of Buffered classes in Figure 24.5.

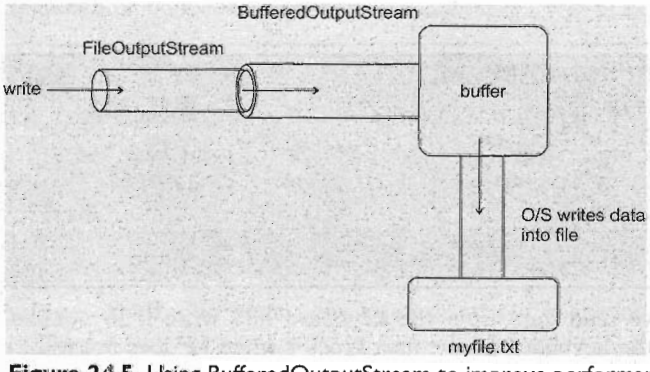

Figure 24.5 Using BufferedOutputStream to improve performance

In Program 1, we can attach `FileOutputStream` to `BufferedOutputStream` as:

```
BufferedOutputStream bout = new BufferedOutputStream(fout, 1024);
```

Here, the buffer size is declared as 1024 bytes. If the buffer size is not specified, then a default buffer size of 512 bytes is used.

Important Interview Question

What is the default buffer size used by any buffered class?

512 bytes.

Program 2: Write a program to improve the efficiency of writing data into a file using `BufferedOutputStream`.

Note

Here, we are rewriting Program 1 using BufferedOutputStream where the file is opened for appending data

```
//Creating a text file using BufferedOutputStream
import java.io.*;
class CreateFile
{
      public static void main(String args[ ])
      throws IOException
      {

            //attach keyboard to DataInputStream
            DataInputStream dis = new DataInputStream(System.in);

            //attach myfile to FileOutputStream in append mode
            FileOutputStream fout = new FileOutputStream("myfile.txt",true);

            //attach FileOutputStream to BufferedOutputStream
            BufferedOutputStream bout = new BufferedOutputStream(fout, 1024);

            System.out.println("Enter text (@ at the end): ");

            char ch;

            //read characters from dis into ch. Then write them into bout.
            //repeat this as long as the read character is not @
            while((ch =(char)dis.read()) != '@')
            bout.write(ch);

            //close the file
            bout.close();
      }
}
```

Output:

```
C:\> javac CreateFile.java
C:\> java CreateFile
Enter text (@ at the end):
This is my file line four
This is last line
@

C:\> type myfile.txt
This is my third line
This is my file line four
```

```
This is last line
```

Reading Data from a File using FileInputStream

FileInputStream is useful to read data from a file in the form of sequence of bytes. It is possible read data from a text file using `FileInputStream`. Let us see how it is done:

- First, we should attach the file to a `FileInputStream` as shown here:

```
FileInputStream fin = new FileInputStream("myfile.txt");
```

This will enable us to read data from the file. Then, to read data from the file, we should read d from the `FileInputStream` as:

```
ch= fin.read();
```

When the `read()` method reads all the characters from the file, it reaches the end of the file. Wh there is no more data available to read further, the `read()` method returns `-1`.

- Then, we should attach the monitor to some output stream, example `PrintStream`, so that output stream will send data to the monitor. For displaying the data, we can use `System.` which is nothing but `PrintStream` object.

```
System.out.print(ch);
```

- Finally, we read data from the `FileInputStream` and write it to `System.out`. This will disp all the file data on the screen.

These steps are shown in Figure 24.6 and implemented in Program 3.

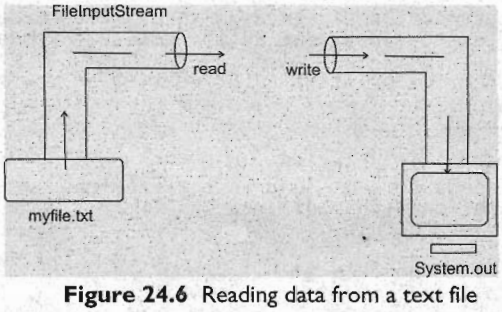

Figure 24.6 Reading data from a text file

Program 3: Write a program to read data from `myfile.txt` using `FileInputStream` and displ on the monitor.

```
//Reading textfile using FileInputStream
import java.io.*;
class ReadFile
{
      public static void main(String args[ ])
      throws IOException
      {

            //attach the file to FileInputStream
            FileInputStream fin = new FileInputStream("myfile.txt");
```

```
                System.out.println("File contents: ");

        //read characters from FileInputStream and write them
        //to monitor. Repeat this till the end of file.
        int ch;

        while((ch= fin.read()) != -1)
          System.out.print((char)ch);

        //close the file
        fin.close();
      }
    }
```

Output:

```
C:\> javac ReadFile.java
C:\> java ReadFile
File contents:
This is my third line
This is my file line four
This is last line
```

There are some improvements suggestible in the above program. First, the above program works with `myfile.txt` only. To make this program work with any file, we should accept the filename from the keyboard. For this purpose, create `BufferedReader` object as:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Here, the keyboard `(System.in)` is attached to `InputStreamReader` which is attached to `BufferedReader`. So if we read data from the `BufferedReader`, then that is actually read from the keyboard. Using `readLine()` method of `BufferedReader` class we can read the filename from the keyboard as:

```
String fname = br.readLine();
```

This filename should be attached to `FileInputStream` for reading data as:

```
FileInputStream fin = new FileInputStream(fname);
```

What happens if the file being opened is not available? There would be `FileNotFoundException`. By handling this exception, it is possible to know whether the file is available or not.

```
try{
      fin = new FileInputStream(fname);
}
catch(FileNotFoundException fe) {
      System.out.println("File not found");
      return;
}
```

As another improvement, we can use `BufferedInputStream` to read a buffer full of data at a time from the file. This improves the speed of execution. These improvements can be found in Program 4.

Program 4: Write a program which is used to read data from any text file.

```
//Reading data from text file - version 2
import java.io.*;
class ReadFile
```

```
{
    public static void main(String args[ ])
    throws IOException
    {
            //to accept filename from keyboard
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.print("Enter file name: ");
        String fname = br.readLine();

        //attach the file to FileInputStream
        FileInputStream fin = null;  //assign nothing to fin

        //check if file exists or not
        try{
            fin = new FileInputStream(fname);
        }
        catch(FileNotFoundException fe) {
        System.out.println("File not found");
        return;
        }

        //attach FileInputStream to BufferedInputStream
        BufferedInputStream bin = new BufferedInputStream(fin);

        System.out.println("File contents: ");

        //read characters from BufferedInputStream and write them
        //to monitor. Repeat this till the end of file.
        int ch;

        while((ch= bin.read()) != -1)
        System.out.print((char)ch);

        //close the file
        bin.close();
    }
}
```

Output:

```
C:\> javac ReadFile.java
C:\> java ReadFile
Enter file name: myfile.txt
File contents:
This is my third line
This is my file line four
This is last line
C:\> java ReadFile
Enter file name: jjjjjj
File not found
```

Creating a File using FileWriter

`FileWriter` is useful to create a file by writing characters into it. The following program depicts how to create a text file using `FileWriter`.

Program 5: Write a program to create a text file using FileWriter.

```
//Creating a text file using FileWriter
import java.io.*;
class CreateFile1
{
    public static void main(String args[ ])
```

```
        throws IOException
        {
                //take a string
                String str="This is a book on Java."+"\nIam a learner of Java.";

                //attach file to FileWriter
                FileWriter fw = new FileWriter("text");

                //read character wise from string and write into FileWriter
                for(int i=0; i<str.length(); i++)
                fw.write(str.charAt(i));

                //close the file
                fw.close();
        }
}
```

Output:

```
C:\> javac CreateFile1.java
C:\> java CreateFile1
C:\>
C:\> type text
This is a book on Java.
Iam a learner of Java.
```

In this program, we are taking a string from where the characters are read and written into a `FileWriter`, which is attached to a file named `text`.

Here, we could also use `BufferedWriter` along with `FileWriter` to improve speed of execution as:

```
BufferedWriter bw = new BufferedWriter(fw, 1024);
```

Reading a File using FileReader

`FileReader` is useful to read data in the form of characters from a 'text' file.

Program 6: Write a program to show how to read data from the 'text' file using FileReader.

```
//Reading data from a file using FileReader
import java.io.*;
class ReadFile1
{
        public static void main(String args[ ])
        throws IOException
        {
                //var
                int ch;

                //check if file exists or not
                FileReader fr = null;

                //check if file exists or not
                try{
                        fr = new FileReader("text");
                }
                catch(FileNotFoundException fe) {
                        System.out.println("File not found");
                        return;
                }

                //read from FileReader till the end of file
                while((ch=fr.read()) != -1)
```

```
                System.out.print((char)ch);

                //close the file
                fr.close();
        }
}
```

Output:

```
C:\> javac ReadFile1.java
C:\> java ReadFile1
This is a book on Java.
Iam a learner of Java.
```

In this program, the 'text' file is attached to `FileReader` for reading data. It is read using re method and is displayed on the monitor.

Here, we could also use `BufferedReader` to improve the speed of execution as:

```
BufferedReader br = new BufferedReader(fr, 512);
```

The data will be then read from the `BufferedReader` object `br`, instead of the `FileReader` fr.

Zipping and Unzipping Files

We know that some software like 'winzip' provide zipping and unzipping of file data. In zippin file contents, following two things could happen:

- The file contents are compressed and hence the size will be reduced.
- The format of data will be changed making it unreadable.

While zipping a file content, a zipping algorithm (logic) is used in such a way that the algorithm finds out which bit pattern is most often repeated in the original file and replaces that bit pa with a `0`. Then the algorithm searches for the next bit pattern which is most often repeated i input file. In its place, a `1` is substituted. The third repeated bit pattern will be replaced by `0` fourth by `10`, the fifth by `100`, and so on. In this way, the original bit patterns are replaced by number of bits. This file with lesser number of bits is called 'zipped file' or 'compressed file'.

To get back the original data from the zipped file, we can follow a reverse algorithm, substitutes the original bit pattern wherever particular bits are found. This is shov Figure 24.7.

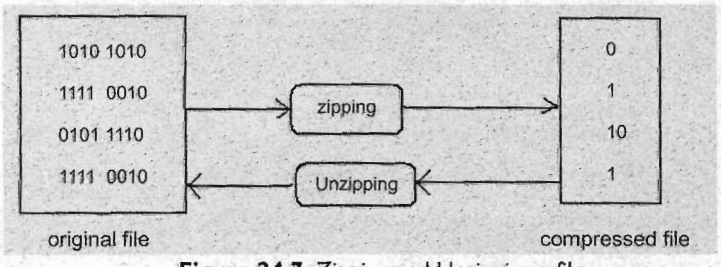

Figure 24.7 Zipping and Unzipping a file

In Java, classes are provided to zip and unzip the file contents. We can use `DeflaterOutput` class for zipping a file content and `InflaterInputStream` class for unzipping the file co These classes are found in `java.util.zip` package.

Zipping a File using DeflaterOutputStream

Let us now learn how to compress data in a file, say 'file1' by following these steps:

- Attach the input file 'file1' to `FileInputStream` for reading data.
- Take the output file 'file2' and attach it to `FileOutputStream`. This will help to write data into 'file2'.
- Attach `FileOutputStream` to `DeflaterOutputStream` for compressing the data.
- Now, read data from `FileInputStream` and write it into `DeflaterOutputStream`. It will compress the data and send it to `FileOutputStream` which stores the compressed data into the output file. These steps are shown in Figure 24.8 and implemented in Program 7.

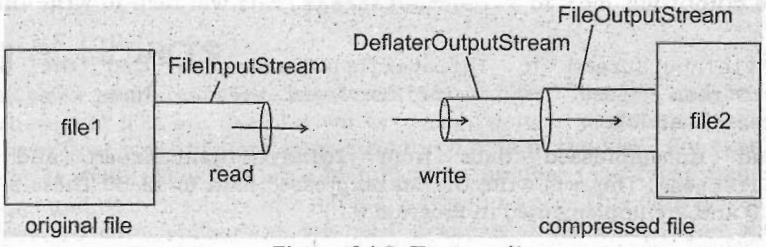

Figure 24.8 Zipping a file

Program 7: Write a program to compress the data contained in a file using `DeflaterOutputStream`.

Note

Before running this program, create 'file1' with some data.

```
//Compressing a file using a DeflaterOutputStream
import java.io.*;
import java.util.zip.*;
class Zip
{
      public static void main(String args[ ]) throws Exception
      {
            //attach the original file:file1 to
            FileInputStream for reading data
            FileInputStream fis = new FileInputStream("file1");

            //attach compressed file:file2 to FileOutputStream
            FileOutputStream fos = new FileOutputStream("file2");

            //attach FileOutputStream to DeflaterOutputStream
            DeflaterOutputStream dos = new DeflaterOutputStream(fos);

            //read data from FileInputStream and write it into
            DeflaterOutputStream
            int data;
            while((data = fis.read()) != -1)
            dos.write(data);

            //close the files
            fis.close();
            dos.close();
      }
}
```

```
            iis.close();
        }
    }
```

```
Output:
C:\> javac UnZip.java
C:\> java UnZip
```

In this program, we take the compressed file 'file2' from where data is read and uncompressed by `InflaterInputStream`. The output of this program is 'file3' which contains the original uncompressed data.

Serialization of Objects

So far, we wrote some programs where we stored only text into the files and retrieved same text from the files. These text files are useful when we do not want to perform any calculations on the data. What happens if we want to store some structured data in the files? For example, we want to store some employee details like employee identification number (`int` type), name (`String` type), salary (`float` type) and date of joining the job (`Date` type) in a file. This data is well structured and got different types. To store such data, we need to create a class `Employee` with the instance variables `id`, `name`, `sal`, `doj` as shown here:

```
class Employee implements Serializable
{
    //instance var
    private int id;
    private String name;
    private float sal;
    private Date doj;
}
```

Then create an object to this class and store actual data into that object. Later, this object should be stored into a file using `ObjectOutputStream`. Please observe that the `Serializable` interface should be implemented by the class whose objects are to be stored into the file. This is the reason why `Employee` class implements `Serializable` interface.

To store the `Employee` class objects into a file, follow these steps:

- First, attach `objfile` to `FileOutputStream`. This helps to write data into `objfile`.

```
FileOutputStream fos = new FileOutputStream("objfile");
```

- Then, attach `FileOutputStream` to `ObjectOutputStream`.

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

- Now, `ObjectOutputStream` can write objects using `writeObject()` method to `FileOutputStream`, which stores them into the `objfile`.

Storing objects into a file like this is called 'serialization'. The reverse process where objects can be retrieved back from a file is called 'de-serialization'.

Important Interview Question

What is serialization?

Serialization is the process of storing object contents into a file. The class whose objects are stored in the file should implement 'Serializable' interface of java.io package.

Serializable interface is an empty interface without any members in it. It does not contain methods also. Such an interface is called 'marking interface' or 'tagging interface'. Marking inter is useful to mark the objects of a class for a special purpose. For example, 'Serializable' inter marks the class objects as 'serializable' so that they can be written into a file. If Serializ interface is not implemented by the class, then writing that class objects into a file will lea NotSerializableException. However, any static and transient variables of the class canno serialized. Suppose, we declare variables in the class as:

```
static int x = 15;
transient String str= "mypassword";
```

Now, these variables cannot be written into the file. Such variables are useful when the program wants to restrain from storing some sensitive data into the file.

Important Interview Question

Which type of variables cannot be serialized?

static and transient variables cannot be serialized.

Once the objects are stored into a file, they can be later retrieved and used as and when needed This is called de-serialization.

What is de-serialization?

De-serialization is a process of reading back the objects from a file.

To read Employee class objects from objfile, follow these steps:

- Attach objfile to FileInputStream. This helps to read objects from objfile.

```
FileInputStream fis = new FileInputStream("objfile");
```

- Attach FileInputStream to ObjectInputStream. This ObjectInputStream gets the ob from FileInputStream.

```
ObjectInputStream ois = new ObjectInputStream(fis);
```

- Now, read objects from ObjectInputStream using readObject() method as:

```
Employee e = (Employee) ois.readObject();
```

The serialization process is shown in Programs 9 and 10 while Program 11 is showing the serialization.

Program 9: Write a program to create Employee class whose objects are to be stored into a file.

```
//Employee class
import java.io.*;
import java.util.Date;
class Employee implements Serializable
{
     //instance var
     private int id;
     private String name;
     private float sal;
     private Date doj;
```

```
        //initialize the var
        Employee(int i, String n, float s, Date d)
        {
                id = i;
                name = n;
                sal = s;
                doj = d;
        }

        //to display employee details
        void display()
        {
                System.out.println(id+"\t"+name+"\t"+sal+"\t"+doj);
        }

        //to accept data from keyboard and store into Employee object
        static Employee getData() throws IOException
        {

                //to accept data from keyboard
                BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));

                //accept employee id number, name and salary
                System.out.print("Enter emp id: ");
                int id= Integer.parseInt(br.readLine());

                System.out.print("Enter name: ");
                String name = br.readLine();

                System.out.print("Enter salary: ");
                float sal = Float.parseFloat(br.readLine());

                //take current system date and time as for joining
                Date d = new Date();

                //create Employee object with the accepted data
                Employee e= new Employee(id, name, sal, d);

                //return the Employee object
                return e;
        }
}
```

Output:

```
C\>javac Employee.java
```

Program 10: Write a program to show serialization of objects.

```
//ObjectOutputStream is used to store objects to a file
import java.io.*;
import java.util.*;

class StoreObj
{
        public static void main(String args[ ]) throws Exception
        {
                //to accept data from keyboard
                BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));

                //to store objects into objfile
                FileOutputStream fos = new FileOutputStream("objfile");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
                //ask how many objects to store
                System.out.print("How many objects? ");
                int n =Integer.parseInt(br.readLine());

                //store n objects into objfile
                for(int i=0; i<n; i++)
                {
                        //create Employee object with data from keyboard
                        Employee e1= Employee.getData();

                        //store Employee object into ObjectOutputStream
                        oos.writeObject(e1);
                }

                //close the objfile
                oos.close();
        }
}
```

Output:

```
C:\> javac StoreObj.java
C:\> java StoreObj
How many objects? 3
Enter emp id: 10
Enter name: Suresh
Enter salary: 9800.50
Enter emp id: 11
Enter name: Rajani
Enter salary: 5000.75
Enter emp id: 12
Enter name: Swapna
Enter salary: 3000.00
```

Program 11: Write a program showing de-serialization of objects.

```
//ObjectInputStream is used to read objects from a file
import java.io.*;
class GetObj
{
        public static void main(String args[ ]) throws Exception
        {
                //to read objects from objfile
                FileInputStream fis = new FileInputStream("objfile");
                ObjectInputStream ois = new ObjectInputStream(fis);

                //read objects and display till a null object is read
                try{
                        Employee e;
                        while((e = (Employee) ois.readObject())!= null)
                        {
                                e.display();
                        }
                }catch(EOFException ee){
                        System.out.println("End of file reached");
                }
                finally{
                //close the objfile
                ois.close();
                }
        }
}
```

Output:

```
C:\> javac GetObj.java
C:\> java GetObj
10    Suresh  9800.5          Tue Sep 25 19:56:44 GMT+05:30 2007
11    Rajani          5000.75 Tue Sep 25 19:57:03 GMT+05:30 2007
12    Swapna  3000.0          Tue Sep 25 19:57:20 GMT+05:30 2007
End of file reached
```

In this program, we read `Employee` class objects from `objfile` using `readObject()` method of `ObjectInputStream` class.

Counting Number of Characters in a File

Let us write a program to count number of characters, words, and lines of a text file. To count the number of characters, we take a counter `char_count` that is incremented whenever a character is read using `read()` method. To count non-space characters, we can use the logic:

```
if(ch != ' ')  ++char_count;
```

To count the number of words, we take a counter `word_count` that is incremented whenever a space is found, since words are separated by spaces. But, we should not count repeated spaces between words. For this purpose, we use:

```
if(!prev && ch == ' ')  ++word_count;
if(ch == ' ')  prev=true; else prev= false;
```

where, `prev` is a boolean variable that becomes `true` if there is a space encountered previously.

To count the number of lines, we take a counter `line_count` that is incremented whenever a `\n` is found. This `\n` is released when Enter button is pressed at the end of a line.

```
if(ch == '\n')  ++line_count;
```

Pressing Enter button releases two characters at the end of each line. They are `\r` and `\n`. These characters are counted as 2 characters by our program. So these excess characters (2 per a line) should be deducted from the character count. Similarly, when three words are there in a line, they get 2 spaces between. Since, we count only spaces to judge the number of words, we should add the number of lines to the `word_count` to get correct number of words. This logic is used in Program 12.

Program 12: Write a program which accepts a filename from command line argument and displays the number of characters, words, and lines in the file.

```
//Counting no. of chars in a text file
import java.io.*;
class Count
{
      public static void main(String args[ ])
      throws IOException
      {
            //vars
            int ch;
            boolean prev= true;

            //counters
            int char_count=0;
            int word_count=0;
```

```
                int line_count=0;

                //attach the file: args[0] to FileInputStream to read data
                FileInputStream fin = new FileInputStream(args[0]);

                //read characters from the file till the end
                while((ch= fin.read()) != -1)
                {
                        if(ch != ' ') ++char_count;
                        if(!prev && ch == ' ') ++word_count;
                        //dont count if previous char is space
                        if(ch == ' ') prev=true; else prev= false;
                        if(ch == '\n') ++line_count;

                }

                //display the count of characters, words and lines
                char_count -= line_count*2;
                word_count += line_count;
                System.out.println("No. of chars= "+ char_count);
                System.out.println("No. of words= "+ word_count);
                System.out.println("No. of lines= "+ line_count);

                //close the file
                fin.close();

        }
}
```

Output:

```
C:\> javac Count.java
C:\> java Count myfile
No. of chars= 44
No. of words= 12
No. of lines= 6
```

File Copy

Sometimes we need to copy the entire data of a text file into another text file. Streams are useful in this case. To understand how to use streams for copying a file content to another file, we can use the following logic:

- For reading data from the input file, attach it to `FileInputStream`.
- For writing data into the output file, which is to be created, attach it to `FileOutputStream`.
- Now, read data from `FileInputStream` and write into `FileOutputStream`. This means, the data is read from the input file and send to output file.

These steps are implemented in Program 13. Please note that this program can copy not only text files, but also image (`.gif` or `.jpg`) files.

Program 13: Write a program to read the contents of the input file and write them into an output file.

Important Interview Question

The input file need to be already available.

```
//Copying a file contents as another file.
import java.io.*;
class CopyFile
```

```
{
        public static void main(String args[ ])
        throws IOException
        {

                //take a var
                int ch;

                //for reading data from args[0]
                FileInputStream fin = new FileInputStream(args[0]);

                //for writing data into args[1]
                FileOutputStream fout = new FileOutputStream(args[1]);

                //read from FileInputStream and write into FileOutputStream
                while((ch= fin.read()) != -1)
                fout.write(ch);

                //close the files
                fin.close();
                fout.close();

                System.out.println("1 file copied");
        }
}
```

Output:

```
C:\> javac CopyFile.java
C:\> java CopyFile  car.gif  car11.gif
1 file copied
```

The output file is created by this program. The names of both the input file and output file are passed from command line arguments.

File Class

File class of `java.io` package provides some methods to know the properties of a file or a directory. First of all, we should create the File class object by passing the filename or directory name to it.

```
File obj = new File(filename);
File obj = new File(directoryname);
File obj = new File("path", filename);
File obj= new File("path", directoryname);
```

Remember, when we pass a filename or directory name, it need not necessarily exist on our computer. We can also judge whether it really exists on our computer system or not using File class methods.

File Class Methods

`File class` includes the following methods:

- `boolean isFile()`: This method returns true if the File object contains a filename, otherwise false.
- `boolean isDirectory()`: This method returns true if the File objects contains a directory name.
- `boolean canRead()`: This method returns true if the File object contains a file which is readable.

- `boolean canWrite()`: This method returns true if the file is writeable.
- `boolean can Execute()`: This method returns true if the file is executable.
- `boolean exists()`: This method returns true when the File object contains a file or directory which physically exists in the computer.
- `String getParent()`: This method returns the name of the parent directory of a file or directory.
- `String getPath()`: This method gives the name of directory path of a file or directory.
- `String getAbsolutePath()`: This method gives the absolute directory path of a file or directory location. Absolute path is mentioned starting from the root directory.
- `long length()`: This method returns a number that represents the size of the file in bytes.
- `boolean delete()`: This method deletes the file or directory whose name is in File object.
- `boolean createNewFile()`: This method automatically creates a new, empty file indicated by File object, if and only if a file with this name does not yet exist.
- `boolean mkdir()`: This method creates the directory whose name is given in File object.
- `boolean renameTo(File newname)`: This method changes the name of the file as newname.
- `String[ ] list()`: This method returns an array of strings naming the files and directories in the directory.

Let us write a program that accepts a file or directory name from command line arguments. Then the program checks if that file or directory physically exists or not and it displays the properties of that file or directory. This is shown in Program 14.

Program 14: Write a program that accepts a file or directory name from command line arguments. Pass the filename or directory name at command line to this program. This program will display its properties.

```
//Displaying file properties
import java.io.*;
class FileProp
{
      public static void main(String args[ ])
      {
            //accept file name or directory name through command line args
            String fname = args[0];

            //pass the filename or directory name to File object
            File f = new File(fname);

            //apply File class methods on File object
            System.out.println("File name: "+ f.getName());
            System.out.println("Path: "+ f.getPath());
            System.out.println("Absolute path: "+ f.getAbsolutePath());
            System.out.println("Parent: "+ f.getParent());
            System.out.println("Exists: "+ f.exists());
            if(f.exists())
            {
                  System.out.println("Is writeable: "+ f.canWrite());
                  System.out.println("Is readable: "+ f.canRead());
                  System.out.println("Is a directory: "+ f.isDirectory());
                  System.out.println("File size in bytes: "+ f.length());
            }
      }
}
```

Output:

```
C:\> javac FileProp.java
```

```
C:\> java FileProp myfile.txt
File name: myfile.txt
Path: myfile.txt
Absolute path: D:\rnr\myfile.txt
Parent: null
Exists: true
Is writeable: true
Is readable: true
Is a directory: false
File size in bytes: 61
```

We write another program, where we want to accept a directory name from the keyboard and then display all the contents of the directory. For this purpose, `list()` method can be used as:

```
String arr[ ] = f.list();
```

In the preceding statement, the `list()` method causes all the directory entries copied into the array `arr[ ]`. Then we pass these array elements arr[i] to File object and test them to know if they represent a file or directory.

```
File f1 = new File(arr[i]);
if(f1.isFile()) System.out.println(": is a file");
if(f1.isDirectory()) System.out.println(": is a directory");
```

This logic is implemented in Program 15.

Program 15: Write a program to accept a directory name and display its contents into an array.

```
//Display the contents of a directory.
import java.io.*;
class Contents
{
     public static void main(String args[ ]) throws IOException
     {
          // enter the path and dirname from keyboard
          BufferedReader br = new BufferedReader(new
          InputStreamReader(System.in));

          System.out.print("Enter dirpath: ");
          String dirpath = br.readLine();
          System.out.print("Enter dirname: ");
          String dname = br.readLine();

          //create File object with dirpath and dname
          File f = new File(dirpath, dname);

          //if directory exists, then
          if(f.exists())
          {
               //get the contents into arr[ ]
               //now arr[i] represents either a file or sub directory
               String arr[ ] = f.list();

               //find no.of entries in the directory
               int n = arr.length;

               //display the entries
               for(int i=0; i<n; i++)
               {
                    System.out.print(arr[i]);

                    //create File object with the entry and test
                    //if it is a file or directory
                    File f1 = new File(arr[i]);
```

```
                                    if(f1.isFile()) System.out.println(": is a file");
                                    if(f1.isDirectory()) System.out.println(": is a
                                    directory");
                            }

                            System.out.println("No of entries in this directory: "+ n);
                    }
                    else System.out.println("Directory does not exist");
            }
    }
```

Output:

```
C:\> javac Contents.java
C:\> java Contents
Enter dirpath: c:\
Enter dirname: rnr
DIAMONDS.GIF: is a file
MyMessage.java: is a file
Myclass.class: is a file
textfile: is a file
twist.gif: is a file
same: is a directory
pool: is a directory
pack: is a directory
oracle: is a directory
misc: is a directory
jdbc: is a directory
advjava: is a directory
Thumbs.db: is a file
Search.class: is a file
mypack: is a directory
App.java: is a file
Palindrome.java: is a file
One.class: is a file
Table.java: is a file
Table.class: is a file
:
:

No of entries in this directory: 320
```

In this program, we accept the directory name and get its contents into an array `arr[ ]`. Then we pass these array elements arr[i] to File object and test them each to know if it is a file or directory.

Conclusion

Streams are needed to move data from one place to another. Input streams help to receive data coming from another place, whereas output streams help to send data to some other place. Byte streams handle data in the form of individual bytes and Character streams are useful to receive and send characters. Using streams, it is possible to store data permanently in the form of files on a secondary storage medium. If the programmer wants to store structured data, he should store the data first in a class object and then the object contents should be written to a file. This is called serialization. However, these techniques are not suitable for handling large volumes of data. For this purpose, the programmer should go for a database like Oracle or Sybase where the data is stored in the form of tables, and then a Java program can be constructed to retrieve and use the data of the database.