Files & Streams

To store data, we are using variables within our programs. But the data what we stored in these variables will be lost either when a variable goes out of its scope or when the program is terminated, because they are stored in primary memory which is volatile in nature. And it is very difficult to handle large volumes of data using variables.

To overcome this, the data should be stored in secondary memory devices like hard disk in the form of files. The data stored in the form of files is called **persistent data**.

A file is a named collection of related information that is recorded on secondary storage. From user's perspective, a file is the smallest allotment of logical secondary storage that is data can't be written to secondary storage unless they are within a file.

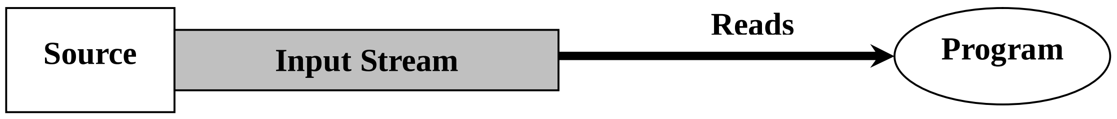
Commonly files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric or binary. In general, a file is a sequence of bits, bytes, line or records. A text file is a sequence of characters organized into lines. A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

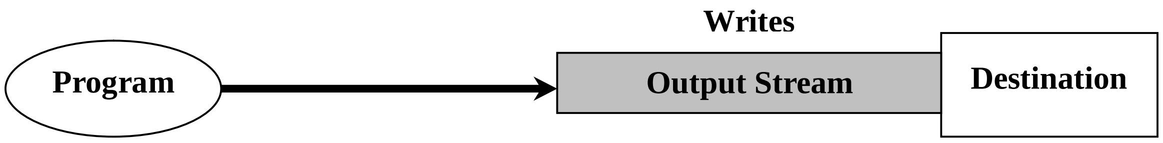
Thus, a **file** is a collection of related **records**. A record is composed of several **fields** and a field is a group of **characters**. Storing and managing data using files is known as **file processing**. In java reading and writing of data in a file can be done at the level of bytes or characters. Java provides capabilities to read and write class objects directly. The process of reading and writing objects is known as **Object Serialization**.

Streams

A stream is an **ordered sequence of data** which is flowing between source and destination. In Java, a stream is an object oriented interface between the program and the input / output devices. Input refers to the flow of data into a program and output means the flow of data out of a program.

Java streams are classified into two basic types: Input Stream and Output Stream. **Input Stream** extracts (**reads**) data from the source and sends it to the program. Similarly, **Output Stream** takes data from the program and sends (**writes**) it to the destination. The source or destination may be either a file or memory or even a device.





Stream Classes

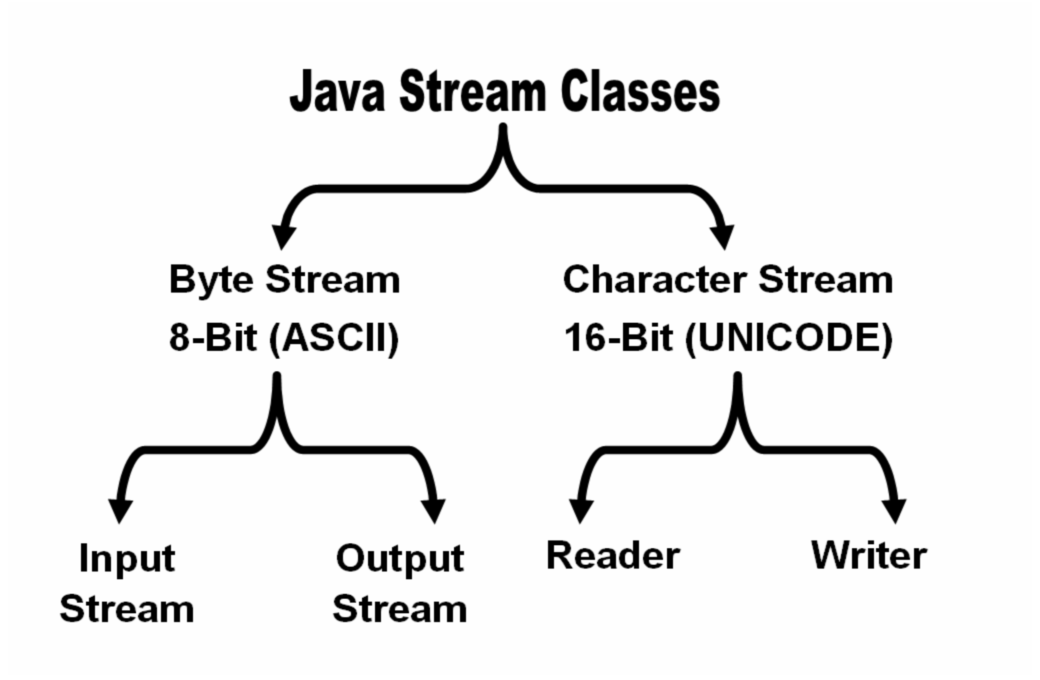
The **java.io** package contains many number of stream classes that provide capabilities for processing all types of data. These classes are categorized into 2 groups based on the data type on which they operate. They are **Byte Stream** Classes and **Character Stream** Classes.

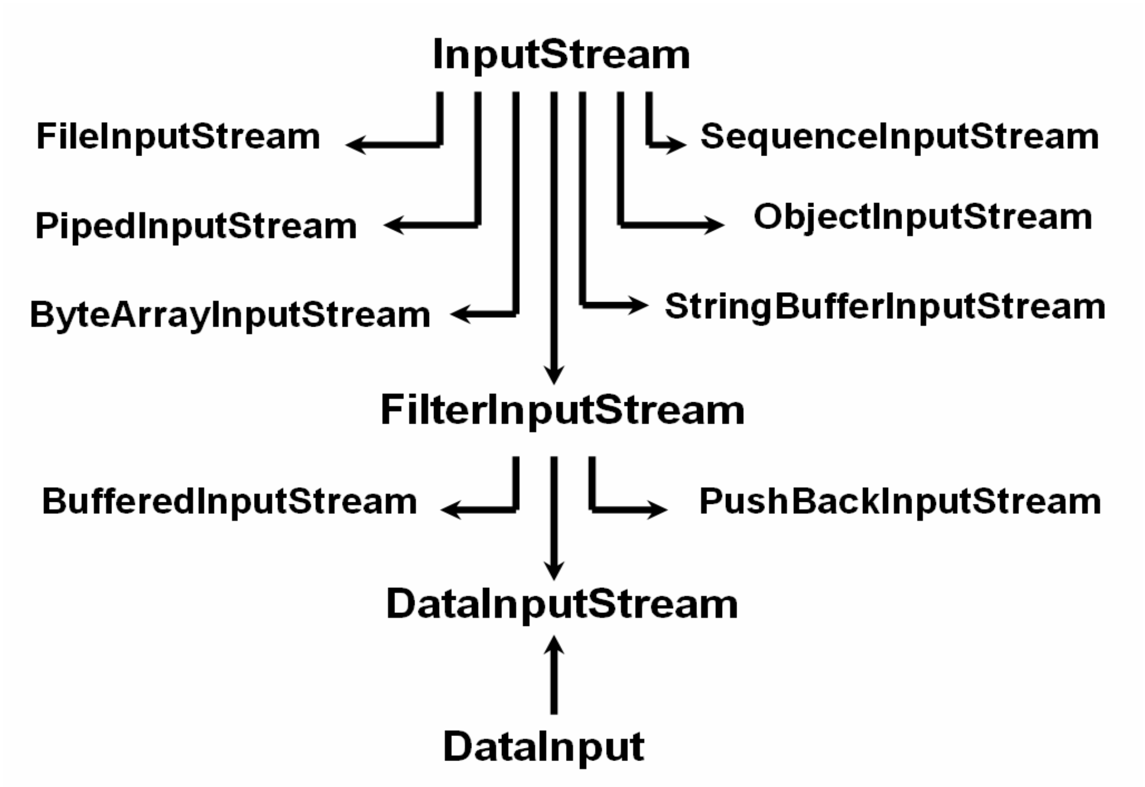
→ Byte Stream classes provide support for handling I/O operations on **bytes** (Binary Files).

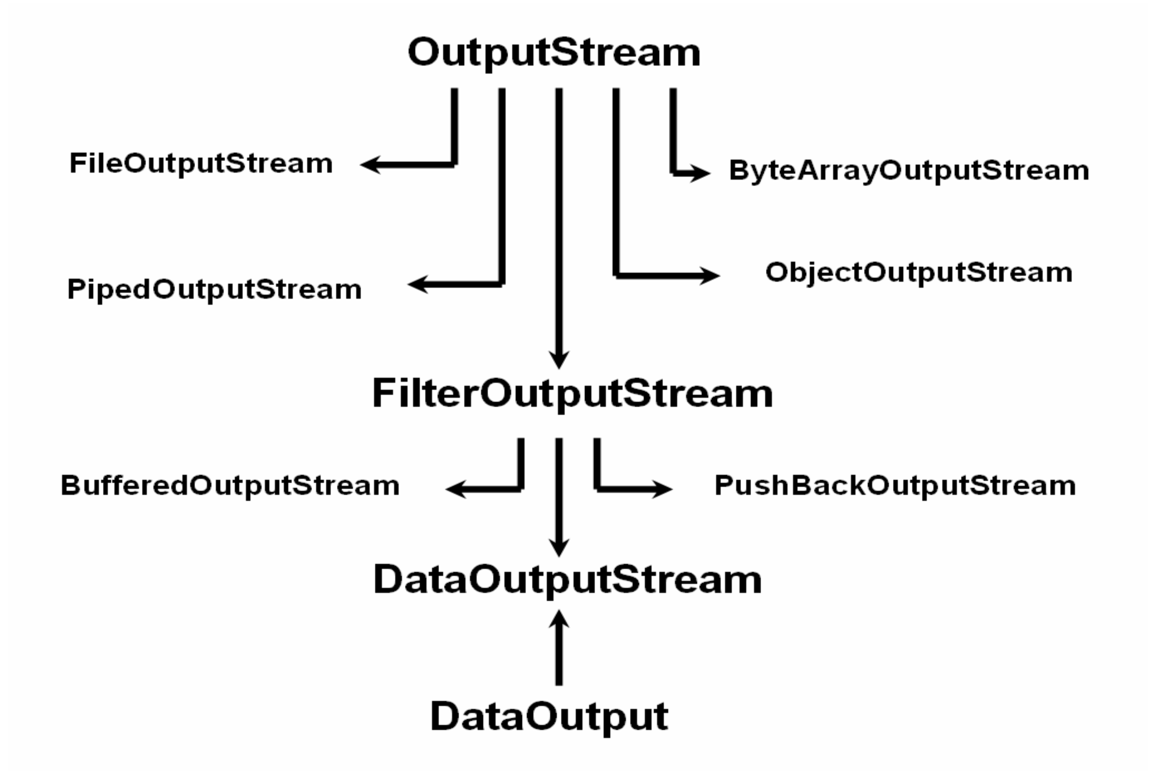
→ Character Stream classes provide support for managing I/O operations on **characters** (Text Files).

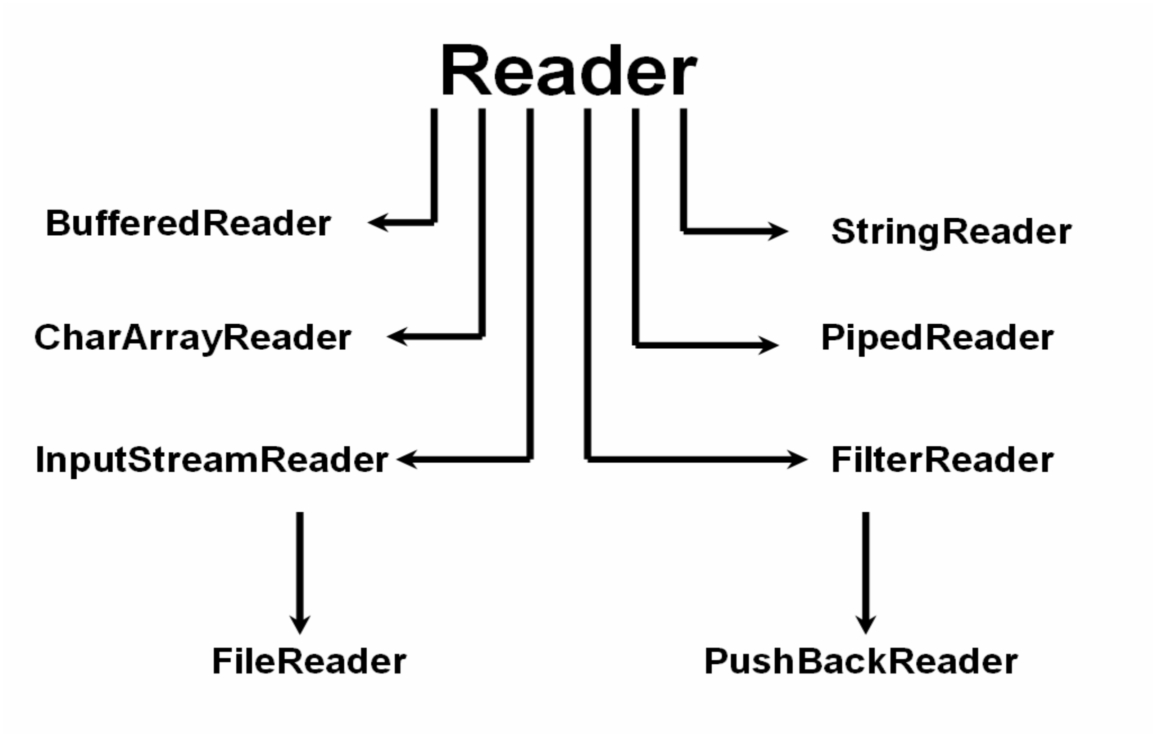
While executing a program, Java creates three stream objects that are associated with devices. They are:

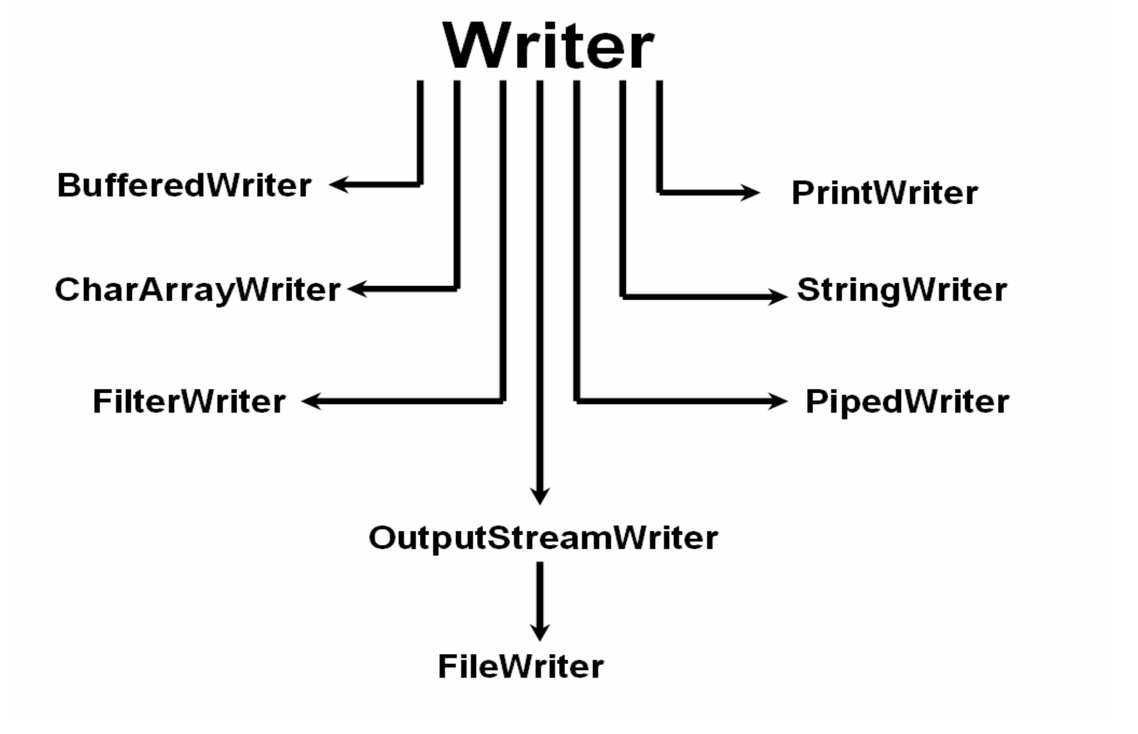
→	**System.in**	Standard input stream object	(Key Board)
→	**System.out**	Standard output stream object	(Monitor)
→	**System.err**	Standard error stream object	(Monitor)


Java Stream Classes
Byte Stream
8-Bit (ASCII)
Character Stream
16-Bit (UNICODE)
Input Stream
Output Stream
Reader
Writer


InputStream
FileInputStream
PipedInputStream
ByteArrayInputStream
SequenceInputStream
ObjectInputStream
StringBufferInputStream
FilterInputStream
BufferedInputStream
PushBackInputStream
DataInputStream
DataInput


OutputStream
FileOutputStream
ByteArrayOutputStream
PipedOutputStream
ObjectOutputStream
FilterOutputStream
BufferedOutputStream
PushBackOutputStream
DataOutputStream
DataOutput


Reader
BufferedReader
StringReader
CharArrayReader
PipedReader
InputStreamReader
FilterReader
FileReader
PushBackReader



List of tasks and Classes implementing them

Task	Byte Stream Class	Character Stream Class
Performing input operations	InputStream	Reader
Reading from files	FileInputStream	FileReader
Reading from a string	StringBufferInputStream	StringReader
Reading from primitive types	DataInputStream	None
Performing output operations	OutputStream	Writer
Writing to a file	FileOutputStream	FileWriter
Printing values & Objects	PrintStream	PrintWriter
Writing to a string	None	StringWriter
Writing primitive types	DataOutputStream	None

The File Class:

The **java.io** package includes a class named **File** which provides support for creating files and directories. This class contains several constructors and methods for supporting the file operations like creating, opening, closing, deleting, renaming, getting the file name, etc ...

Method	**Description**
boolean canRead()	Returns true if the file is readable
boolean canWrite()	Returns true if the file is writable
boolean exists()	Returns true if it exists
boolean isFile()	Returns true if it is a file
boolean isDirectory()	Returns true if it is a directory
boolean isAbsolute()	Returns true if the path is absolute
String getAbsolutePath()	Returns a string with the absolute path of the file or directory
String getName()	Returns a string with the name
String getPath()	Returns a string with the path
String getParent()	Returns a string with the parent directory
String[ ] list()	Returns an array of strings representing the list files and folders in the directory
long length()	Returns the length of the file in bytes.
long lastModified()	Returns the time at which it was last modified
boolean renameTo(File dest)	Renames the file object to dest.

Constructors of File Class:

- **File(String name)**
- **File(String path, String name)**
- **File(File directory, String name)**
- **File(URI url)**

Program to display the list of files & directories in a given directory

```
import java.io.File;

public class DOSDir
{
      public static void main(String args[]) throws Exception
      {
           File f1=new File(args[0]);
          String a[]=f1.list();
          int x=a.length;
```

```
            System.out.printf("\nNo of directories and files under %s are :
                                                          %5d\n",args[0],x);

           for(int i=0;i<x;i++)
           {
                  String fname=a[i];
                  File f2=new File(f1,fname);

                  if(f2.isDirectory())
                  System.out.printf("\n%5d. %-45s is a Directory",i+1,fname);

                  if(f2.isFile())
                  System.out.printf("\n%5d. %-45s is a File",i+1,fname);
           }
      }
}
```

Sample Output:

```
>java DOSDir c:\jsdk2.0
No of directories and files under c:\jsdk2.0 are :  7
   1. DeIsL1.isu            is a File
   2. README                is a File
   3. src                   is a Directory
   4. doc                   is a Directory
   5. examples              is a Directory
   6. bin                   is a Directory
   7. lib                   is a Directory
```

SEQUENTIAL ACCESS FILES:

As the name tells that the records in a sequential file are stored sequentially in the logical sequence of their primary key or record key values. The disadvantages of Sequential files are as follows.

- **Updating** a record requires the creation of a new file. To maintain file sequence, the records of the original file are copied to the new file to the point where we need modification. Then changes are made to the record and copied into the new file. Following this, the remaining records of the original file are copied to the new file.

- **Adding** a record necessitates the shifting of records from the appropriate point to the end of the file to create space for the new record.

- **Deleting** a record requires a compression of the file space.

But, these files are suitable for the situations where we need to access each and every record of the file sequentially. It is not possible to read and write a sequential file at the same time. In a program, if it is necessary to read the file again, we should first close that file and **reopen** it. Searching for a record in these files is a time consuming job, if the file size is very large.

BUFFERED STREAMS:

In unbuffered I/O streams, each read or write request is handled directly by the underlying Operating System. This makes a program much less efficient, since each such request triggers disk access, network activity which is relatively expensive.

Buffered Streams can be used to reduce this overhead. These streams can read data from or write data to a memory area known as **buffer**. When the buffer is full, the data will be written to the disk. Similarly when the buffer is empty, the data will be taken from the disk.

There are four buffered stream classes available in java. They are:

- BufferedInputStream
- BufferedOutputStream

} **Byte Stream Classes**

- BufferedReader
- BufferedWriter

} **Character Stream Classes**

Program to Simulate COPY command of DOS with the help of Streams

```
import java.io.*;

public class DOSCopy
{
        public static void main(String args[])
        {
              File src=new File(args[0]);
              File trgt=new File(args[1]);
              int k;
              try
              {
                 FileOutputStream fos=new FileOutputStream(trgt);
                 FileInputStream fis=new FileInputStream(src);
                 while((k=fis.read())!=-1)
                 {
                      fos.write((char)k);
                 }
```

```
                System.out.printf("\n\tContents of %s are copied to
                                                  %s",args[0],args[1]);
                System.out.printf("\n\tTo view the contents execute the
                                  following command at DOS Prompt");
                System.out.printf("\n\tTYPE %s\n\n",args[1]);
                fos.close();
                fis.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
}
```

<u>Output:</u>

```
>java  DOSCopy  abc  doop

        Contents of abc are copied to doop
        To view the contents execute the following command at DOS Prompt
        TYPE doop
```

RANDOM ACCESS FILES:

In these files, the record key value is mapped directly to the **storage location** so that we can access the records directly. Random Access files are sometimes called **Direct Access files**, because we can access the individual records of the file directly without searching through the entire records of the file. These are suitable for **instant-access** applications, such as Banking Systems, Reservation Systems and other Transaction Processing Systems, in which a particular record of information must be located immediately. In these files, the **record length** of the file should be **fixed** where as in Sequential Access file the record length may be variable.

The **java.io** package contains a class named **RandomAccessFile**, which performs all the operations on random access files. An object of this class maintains a file pointer, which indicates the current location from which data will be read or to which data will be written.

We can perform both read and write operations simultaneously on a RandomAccessFile, if we open it in **"rw"** mode.

Methods of the RandomAccessFile

RandomAccessFile (String f-name, String mode)

RandomAccessFile (File f-name, String mode)

long getFilePointer()	→ returns the current position of file pointer.
void seek(long)	→ jumps the cursor to the required byte.
int skipBytes(int)	→ moves the file pointer forward to the specified number of bytes

Program on Random Access Files

```
import java.io.*;
import java.lang.*;

public class StudRAF
{
     RandomAccessFile rf;

     String name;
     int rno,m1,m2,m3,total;

     public static final int recSize=60;
public StudRAF() throws FileNotFoundException, IOException
{
      rf=new RandomAccessFile("stud.txt","rw");
}

public StudRAF(String s) throws FileNotFoundException,
                                              IOException
{
      rf=new RandomAccessFile(s,"rw");
}

public void createBlankRecords(int n) throws IOException
{
      rf.seek(rf.length());
      for(int i=0;i<n;i++)
      {
            rf.writeInt(0);
            for(int j=0;j<20;j++)
                  rf.writeChar('\n');
            rf.writeInt(0);
            rf.writeInt(0);
            rf.writeInt(0);
            rf.writeInt(0);
      }
}

public void writeToRAF() throws IOException
{
      String resp;
      DataInputStream dis=new DataInputStream(System.in);
      do
      {
            System.out.println("\nEnter Rno,Name,M1,M2,M3:");
            rno=Integer.parseInt(dis.readLine());
            if(rno*60<=rf.length())
            {
                  name=dis.readLine();
                  m1=Integer.parseInt(dis.readLine());
                  m2=Integer.parseInt(dis.readLine());
                  m3=Integer.parseInt(dis.readLine());
                  total=m1+m2+m3;
                  rf.seek((rno-1)*60);
                  rf.writeInt(rno);
                  char ch;
                  for(int i=0;i<20;i++)
                  {
                        ch='\n';
                        if(i<name.length())
                              ch=name.charAt(i);
                        rf.writeChar(ch);
                  }
```

```
                rf.writeInt(m1);
                rf.writeInt(m2);
                rf.writeInt(m3);
                rf.writeInt(total);
            }
            else
            {
                System.out.println("\nThere is not enough blank
                                    space to insert this record");
                System.out.println("Create blanks records, then
                                                  proceed. ");
            }

            System.out.printf("\n\tDo U want 2 Add one more record
                                                      y / n:");
            resp=dis.readLine();
        }
        while(resp.equalsIgnoreCase("y"));
}

public void readFromRAF() throws IOException
{
        String resp;
        DataInputStream dis=new DataInputStream(System.in);

        boolean isExists=true;
        do
        {
            System.out.println("Which Record U want : ");
            int recno=Integer.parseInt(dis.readLine());
            try
            {
                rf.seek((recno-1)*60);

                rno=rf.readInt();
                if(rno!=0)
                {
                    StringBuffer sb=new StringBuffer(20);
                    int i=0;
                    boolean proceed=true;
                    while(proceed && i<20)
                    {
                        char ch=rf.readChar();
                        i++;
                        if(ch=='\n')
                            proceed=false;
                        else
                            sb.append(ch);
                    }
                    name=sb.toString();
                    rf.skipBytes((20-i)*2);
```

```
m1=rf.readInt();
m2=rf.readInt();
m3=rf.readInt();
total=rf.readInt();
System.out.printf("\n%-20s: %d","Roll Number",rno);
System.out.printf("\n%-20s: %s","Name",name);
System.out.printf("\n%-20s: %d","SUB-1",m1);
System.out.printf("\n%-20s: %d","SUB-2",m2);
System.out.printf("\n%-20s: %d","SUB-3",m3);
System.out.printf("\n%-20s: %d\n","Toatl Marks",total);

System.out.println("\nU are at "+rf.getFilePointer()+" byte");
}
else
{
      rf.skipBytes(56);
      System.out.println("\nNo Such Record");
}
}
catch(EOFException e)
{
      System.out.println("No such Record");
}
System.out.printf("\n\tDo U want 2 search y / n :");
resp=dis.readLine();
}
while(resp.equalsIgnoreCase("y"));
}

public void readRAFSequentially() throws IOException
{
      rf.seek(0);
      boolean hasRecords=true;
      do
      {
            try
            {
                  rno=rf.readInt();
                  if(rno!=0)
                  {
                        StringBuffer sb=new StringBuffer(20);
                        int i=0;
                        boolean proceed=true;
                        while(proceed && i<20)
                        {
                              char ch=rf.readChar();
                              i++;
                              if(ch=='\n')
                                    proceed=false;
                              else
                                    sb.append(ch);
                        }
```

```
        name=sb.toString();
        rf.skipBytes((20-i)*2);
        m1=rf.readInt();
        m2=rf.readInt();
        m3=rf.readInt();
        total=rf.readInt();
        System.out.printf("\n%-20s: %d","Roll Number",rno);
        System.out.printf("\n%-20s: %s","Name",name);
        System.out.printf("\n%-20s: %d","SUB-1",m1);
        System.out.printf("\n%-20s: %d","SUB-2",m2);
        System.out.printf("\n%-20s: %d","SUB-3",m3);
        System.out.printf("\n%-20s: %d\n","Toatl Marks",total);
}
else
        rf.skipBytes(56);
}
catch(EOFException e)
{
        hasRecords=false;
        System.out.println("\n End - of - File ");
}
}
while(hasRecords);
}

public void closeRAF() throws Exception
{
        rf.close();
}

public void dispFileSize() throws IOException
{
        long fileSize=rf.length();
        float fileSizeinKB=(float)(fileSize/1024.0);
        long noOfRecords=rf.length()/(long)(recSize);
        System.out.print("\nThe size of the file in KBs :
                                             "+fileSizeinKB);
        System.out.print("\nAnd it can hold upto "+noOfRecords+"
                                                       Records");
}
}
```

```
import java.io.*;
class DemoStudRAF
{
      public static void main(String args[]) throws IOException,
                                                          Exception
      {
      StudRAF sraf=new StudRAF(args[0]);
      DataInputStream dis=new DataInputStream(System.in);
      int choice;

      do
      {
      System.out.println("\n\n1.Create New BlankRecords\n2.Write
       \n3.Search for a Record\n4.Read Sequentially\n5.Dispaly
                                      File Size\n0.Quit\n");
      System.out.print("Enter U r Choice : ");
      choice=Integer.parseInt(dis.readLine());
      switch(choice)
      {
      case 0:
            sraf.closeRAF();
            System.exit(1);
      case 1:
      System.out.print("\nHow many records u want 2 create : ");
      int rec=Integer.parseInt(dis.readLine());
      sraf.createBlankRecords(rec);
      break;
      case 2:
            sraf.writeToRAF();
            break;
      case 3:
            sraf.readFromRAF();
            break;
      case 4:
            sraf.readRAFSequentially();
            break;
      case 5:
            sraf.dispFileSize();
            break;
      default:
            System.out.println("Enter a valid Choice");
      }
}
while(true);
}
}
```

OBJECT SERIALIZATION:

Java provides capabilities to read and write class objects directly. The process of reading and writing objects is known as **Object Serialization**. To achieve Object Serialization, the class should implement the **interface Serializable** of **java.io** package. The classes **ObjectInputStream** and **ObjectOutputStream** of java.io package provide methods to read one object from or write one object to a sequential file respectively.

```
import java.io.Serializable;
public class Employee implements Serializable
{
      private int empno;
      private String ename;
      private double salary;

      public Employee()
      {
            this(0,"",0.0);
      }
      public Employee(int eno, String enm, double sal)
      {
            empno=eno;
            ename=enm;
            salary=sal;
      }
      public void display()
      {
            System.out.printf("\n%-20s : %d\n","Emp No",empno);
            System.out.printf("%-20s : %s\n","Emp Name",ename);
            System.out.printf("%-20s : %8.2f\n","Salary",salary);
      }
}


import java.io.*;
import java.util.Scanner;
import java.lang.ClassNotFoundException;
public class OIPOPS
{
      public void writeToFile(String fname) throws
      FileNotFoundException, IOException, ClassNotFoundException
      {
            ObjectOutputStream oos=new ObjectOutputStream(
                                    new FileOutputStream(fname));
            Scanner input=new Scanner(System.in);
            int eno;
            String enm;
            double sal;
            System.out.println("Enter EmpNo, EmpName, Salary");
            System.out.println("Press F6 to Quit");
            do
            {
                  try
                  {
                        eno=input.nextInt();
                        enm=input.next();
                        sal=input.nextDouble();
                        Employee empRec=new Employee(eno,enm,sal);
                        oos.writeObject(empRec);
                  }

                  catch(Exception e)
                  {
                        input.nextLine();
                  }
            }
            while(input.hasNext());
            oos.close();
      }


      public void readFromFile(String fname) throws
      FileNotFoundException, IOException, ClassNotFoundException
      {
            ObjectInputStream ois=new ObjectInputStream(
                                    new FileInputStream(fname));
            Employee empRec;
            try
            {
                  while(true)
                  {
                        empRec=(Employee)ois.readObject();
                        empRec.display();
                  }
            }
            catch(EOFException e)
            {
                  ois.close();
                  return;
            }
      }
}


import java.io.*;
public class DemoObjectStreams
{
      public static void main(String a[])
      {
            OIPOPS object=new OIPOPS();
            DataInputStream dis=new DataInputStream(System.in);
            System.out.println("\nMenu");
```

```
          System.out.println("======");
          System.out.println("1.Add Records\n2.Read Records");
          try
          {
               System.out.print("Enter Ur Choice :");
               int choice=Integer.parseInt(dis.readLine());
               switch(choice)
               {
                    case 1:
                         object.writeToFile(a[0]);
                         break;

                    case 2:
                         object.readFromFile(a[0]);
                         break;
                    default:
                         System.out.println("Not a valid
                                                   option");
               }
          }
          catch(Exception e)
          {
               return;
          }
     }
}
```