I/O streams:

A stream is an abstraction that either produces or consumes information. A stream is attached to a physical device by the Java IO system. There are 2 types of streams.

1. byte stream
2. character stream

byte stream:

Byte stream handles input and output data in the form of bytes. byte streams are defined using 2 class hierarchy. At the top of the hierarchy are 2 abstract classes

1. input stream
2. output stream.

→ these abstract classes have several concrete subclasses that handles the various devices such as disc files, network connections and memory buffers.

→ The abstract classes input stream and output stream have several important methods, two among them are read() and write() which are used to read and write bytes of data.

character stream:

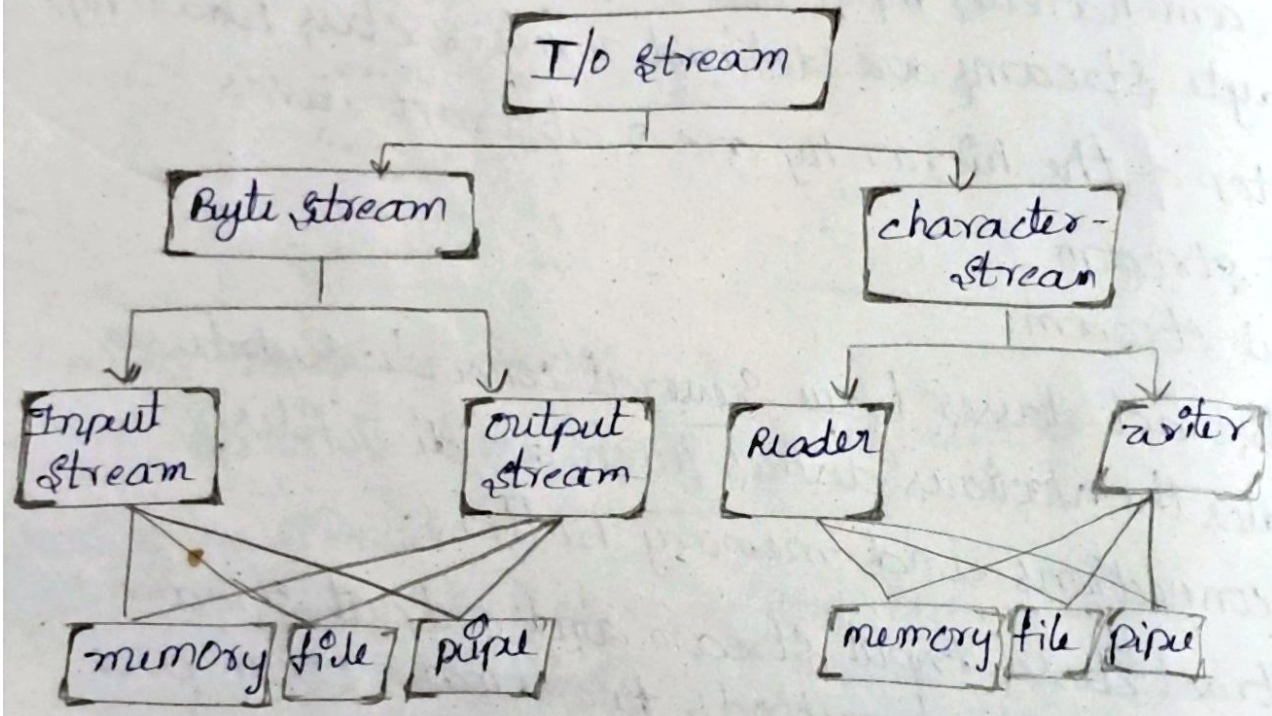
→ character stream is used to handle input and output in the form of characters. they use unicode and can be internationalized

→ character streams are defined using two abstract classes

and the top of the hierarchy are two abstract classes reader and writer.

- these abstract classes handle unicode (provide all other languages) character streams. the abstract classes reader and writer specify several important methods that the concrete subclasses have to implement. two of them are: read and write. which read and write the characters of data.

(The pre-defined streams:)

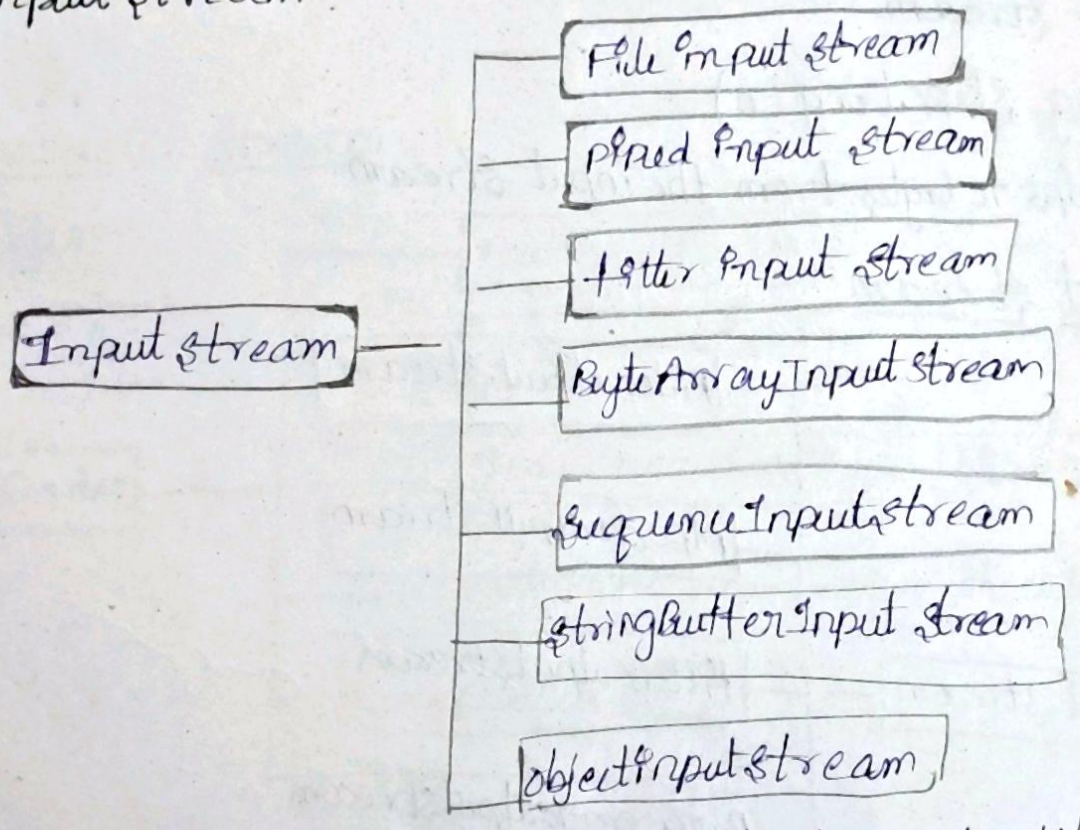


the pre-defined streams:

All the java programs automatically import the java.lang package. this package defines a class System which Encapsulates several aspects of the java runtime Environment. System class also contains pre-defined variables in, out, err.

Byte stream:-
These variables are stream variables and are declared as public, static and final by default.
System.in refers to the input stream which is by default the keyword.
System.out refers to the output stream which is by default the monitor
System.err refers to the Error stream which is also points to the monitor.
System.in is an object of the class input stream and System.out, System.err are objects of the class printstream.

Input stream:



Input stream classes are used to read 8 bit bytes and support input related operations like:
1. reading bytes 2. closing stream 3. marking positions in streams 4. skipping ahead in stream 5. finding the no. of bytes in streams.

Methods: (commonly used methods)

(i.) int read( )

Read a byte from the input stream.

(ii) int read(byte[ ] b)

Read an array of bytes into b

(iii) void close( )

closes the input stream

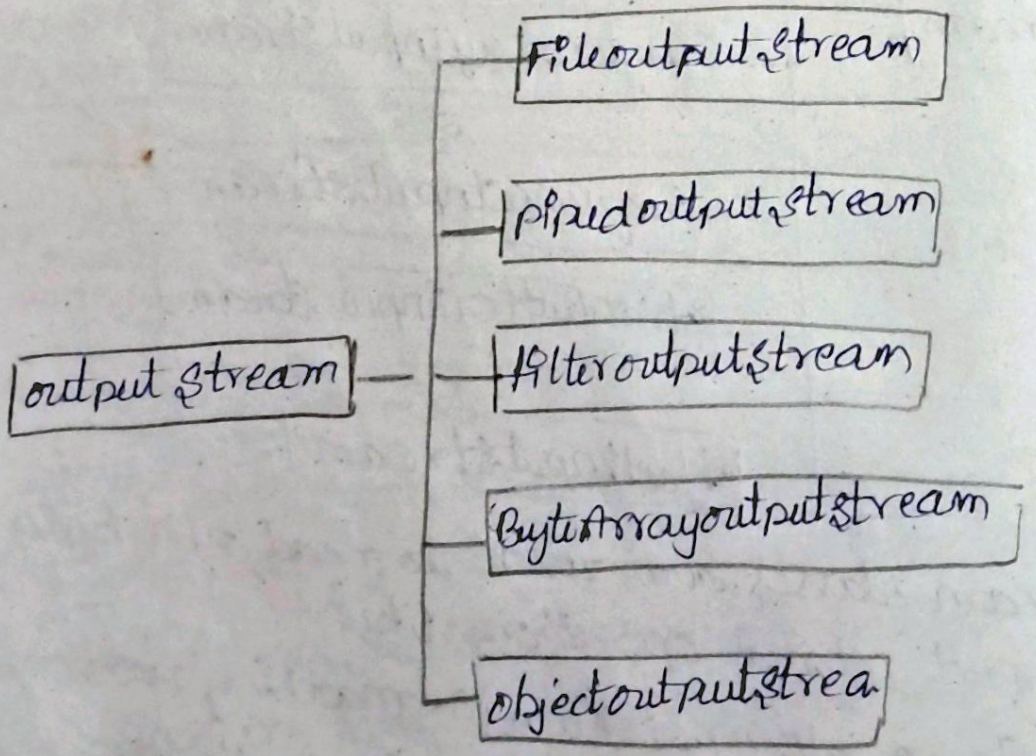
(iv) int available( )

counts the no of bytes available in the input stream.

(v) long skip(long n)

skips n bytes from the input stream.

output stream:



Commonly used Methods:

① void write(byte b)

writes a single byte to the output stream

② void write(byte b[ ])

writes all bytes in the array b[ ] to the output stream.

③ void close( )

closes an output stream.

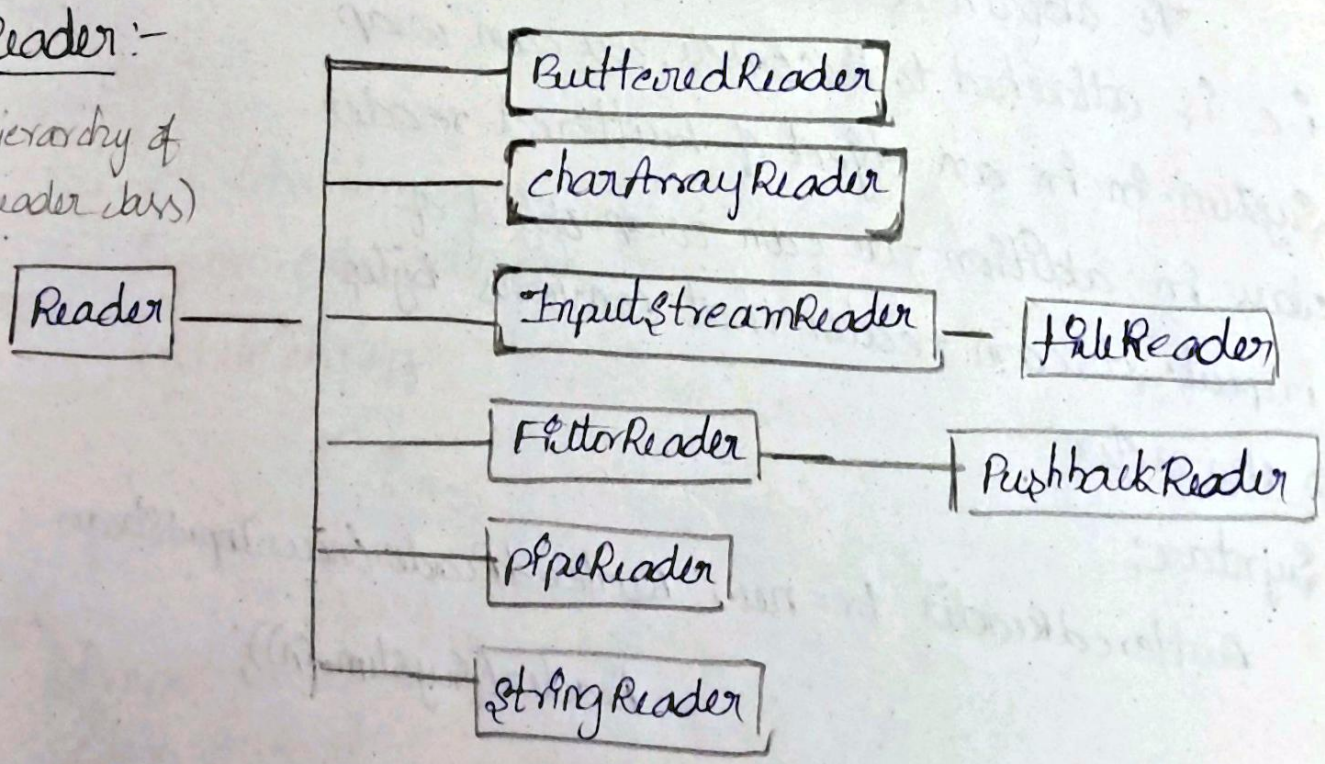
④ void flush( )

flushes the output stream

Character Stream:-

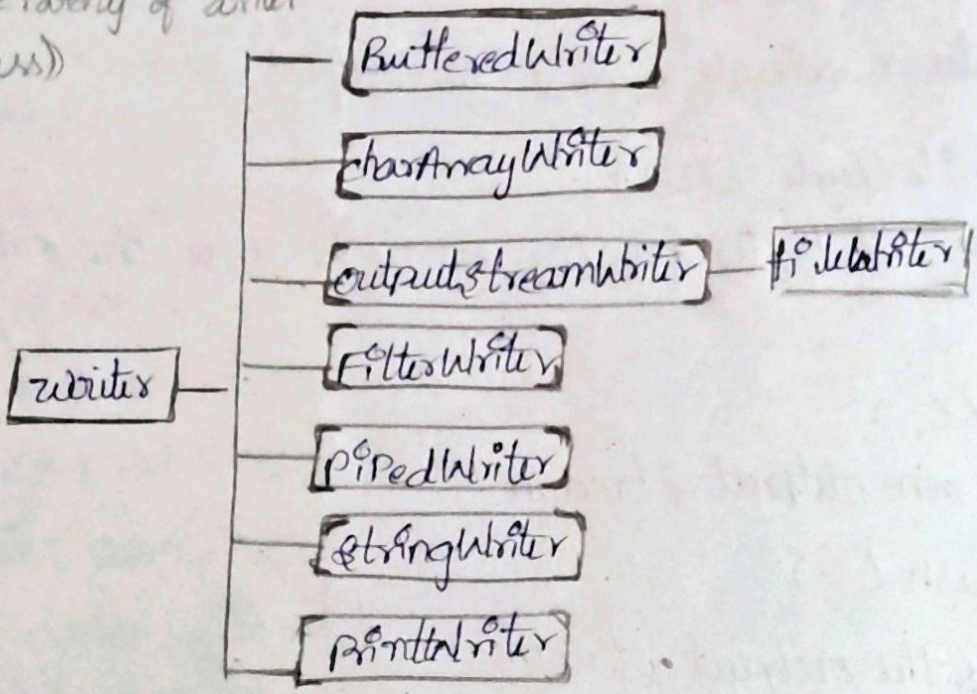
Reader:-

(Hierarchy of reader class)



writer:

(hierarchy of writer-class))



Reading input from console:

To obtain a character based stream i.e is attached to the console, we can wrap System.in in an object of buffered reader-class in addition we can wrap object of input stream reader that converts bytes to character.

Syntax:

```
BufferedReader br=new BufferedReader(new InputStream
                                     Reader(System.in));
```

Reading character:

To read a character from the object of bufferedReader class, we can use the read() method.

Syntax:

```
int read() throws IOException
```

Each time read() is invoked, it reads a character from the input stream and returns it as an integer. It returns (-1) when it reaches End of the stream. (6)

Ex:-
```
import java.io.*;
class BRRead
{
 public static void main(String args[]) throws
                                   IOException-checked Exception
                                   ↓ handled by JVM
 {
  char ch;
  System.out.println("Enter a character and q to quit");
  BufferedReader br = new BufferedReader(new InputStream
                                   Reader(System.in));
  do {
   ch = (char)br.read();
   System.out.println(ch);
  } while(ch != 'q');
 }
}
```

Ex:-
```
import java.io.*;
class BRRead
{
 public static void main(String args[])
 {
  try
   {
    char ch;
```

```
System.out.println("Enter a character and 'q' to quit");
BufferedReader br=new BufferedReader(new InputStreamReader(
                                        System.in));

do
 {
 ch=(char)br.read();
 System.out.println(ch);
}while(ch!='q');
}catch(IOException e)
 {
 System.out.println(e);
 }
}
}
```

Reading Strings:

To Read a string from the input string we can use readline() of the bufferedReaderclass.

Syntax:

String readline() throws IOException

Ex:-

```
import java.io.*;
class BRReadline
{
 public static void main(String args[])
 {
 try
  {
  DataInputStream dis;String s;
```

```
System.out.println("Enter a string and 'quit' to quit");
BufferedReader br = new BufferedReader(new InputStreamReader
                                        System.in);
```

(2)

```
do
{
string = (str)br.readLine();
System.out.println(str);
        (! str.equals("quit"));
} while (str != quit);  (8) (! str.equalsIgnoreCase("quit"))
} catch (IOException e)
{
System.out.println(e);
  }
  }
}
```

-writing output to the console:

We can use the write() to write data to the console. write() is from the PrintStream class

Syntax:-

```
void write(int bytevalue)
```

Ex:

```
import java.io.*;
class WriteDemo
{
public static void main(String args[])
{
int i;
i = 'A';
System.out.write(i);
  }
}
```

Printwriter:

→ In order to write character based output to the console we can make use of printwriter class that uses the character stream.

Ex:

```
import java.io.*;
class PrintWriterDemo
{
 public static void main(String args[ ])
  {
   PrintWriter PW= new PrintWriter(System.out, true);
   PW.println("this is a string");
   int i=99;
   PW.println(i);
   float f=23.45;
   PW.println(f);
  }
}
```

true → enable auto flushing

false → disabled auto flushing

Reading and writing file:

Two of the most commonly used stream classes to file operations are ~~File output~~ FileInputStream and FileOutputStream which creates bytestream attached to files. To open a file simply create object for one of these classes passing the filename as the parameter to the constructor

FileInputStream(String filename) throws FileNotFoundException

FileOutputStream(String filename) throws FileNotFoundException

→ when you are done with a file, we have to close a file. this is done by invoking the close method which is implemented by both FileInputStream and FileOutputStream classes.

Syntax:
close( ) throws IO Exception

→ closing a file releases the system resources allocated to the file

→ To read from a file, we can use a version of the read( ) that is defined by the FileInputStream class

Syntax: int read( ) throws IOException

→ Each time read( ) is invoked it reads a single byte to the file and returns the byte value as an integer

→ Read( ) return(-1) when it reaches end of the file

Ex:

```
import java.io.*;
class ReadFile
{
Public static void main(String args[ ])
 {
  int i;
  FileInputStream fin=null;
```

```
if(args.length!=1)
{
System.out.println("usage: ReadFile filename");
return;
}
try
{
fin=new FileInputStream(args[0]);
do{
   i=fin.reader
  if(i!=-1)
System.out.println((char)i);
}while(i!=-1);
fin.close();
}catch(FileNotFoundExcepticn e)
{
System.out.println(e);
}catch(IOException e)
{
System.out.println(e);
}
}
}
```

Ex:

```
import java.io.*;
class FileReadDemo
{
 Public static void main(String args[])
 {
  int i;
  String fname;
  FileInputStream fp=null;
  try
  {
   BufferedReader br=new BufferedReader(new Input-
                    StreamReader(System.in));
   System.out.println("Enter a file name to read");
    fname=br.readline();
    fp=new FileInputStream(fname);
    do{
      i=fp.read();
      if(i!=-1)
      System.out.println((char)i);
    }while(i!=-1);
     fp.close();
 }catch(FileNotFoundException e)
 {  }
catch(IOException e)
{ }
}
}
```

Writing to a file:

```
import java.io.*;
class FileReadWriteDemo
{
Public static void main (String args[ ])
{
  int i;
  FileInputStream fin= null;
  FileOutputStream fout= null;
  if (args.length != 2)
  {
  System.out.println("usage: FileReadWriteDemo file1 file2");
return;
}
try{
  fin=new FileInputstream(args[0]);
  fout=new FileOutputstream(args[1]);
do{
   i=fin.read();
   if(i!=-1)
   fout.write(i);
   }while(i!=-1);
   fin.close();
   fout.close();
  }catch(FileNotFoundException e
{ }
catch(IOException e)
{ }
}
```

RandomAccessFile:

→ RandomAccessFile class supported by the java.io package allows users to create files that can be used for Reading and writing data with random access such files are known as RandomAccessFiles.

→ A file can be created and opened for random access by giving a mode string as a parameter to the constructor when we open the file.

→ The mode strings are 'r' - for reading only rw - for both reading and writing also an existing file can be obtained by using the rw mode. RandomAccessFile supports a filepointer that can be moved to random positions on the file before reading & writing. The file pointer can be moved using the Seek() of RandomAccessFile class

Syntax:

```
file = new RandomAccessFile("file1.txt","rw")
```

→ when the file is opened using the above format, the file pointer is positioned automatically at the beginning of the file

Ex:
```
import java.io.*;
class RandomIo
{
  public static void main(String args[])
  {
    RandomAccessFile f1=null;
```

```
try
{
H=new RandomAccessFile("file1.txt","rw");
H.writeChar('A');
H.writeInt(9.9);
H.writeDouble(23.5);
H.seek(1);
System.out.println(H.readChar());
System.out.println(H.readInt());
System.out:println(H.readDouble());
H.seek(2);→moves to second line of that file
System.out.println(H.readInt());
H.seek(H.length);→length());
H.writeBoolean(true);
H.seek(4);
System.out.println(H.readBoolean());
catch(IOException e)
{
System.out.println(e);
}
```

Output:- A
9.9
23.5
9.9
true

File Class:

The File class from the java.io package allows us to work with the file to use the File class create an object of the class and specify the File name & directory name

Syntax:

```
import java.io.*;
File f1=new File("file1.txt");
```

Methods:

1. canRead( )
   Test whether the file is readable or not
2. canWrite( )
   Tests whether or the file is writable or not
3. createNewFile( )
   creates an Empty file.
4. delete( )
   deletes a file.
5. exists( )
   checks whether a file is exists or not
6. getName( )
   returns the name of the file
7. getAbsolutePath( )
   returns the absolute path name of the file
8. length( )
   returns the size of the file.

Ex:

```
import java.io.*;
class FileDemo
{
  public static void main (String args[ ])
  {
    try
     {
    File f1=new File("data.txt");
    if(f1.createFile ( )")
    {
    System.out.println("File"+f1.getName ( )+"created");
    }
    Else{
     System.out.println(" File Already exists");
    }
   } catch(IOException e)
    {
     e.printStackTrace ( );
    }
  }
}
```

Object Serialization:

→ Serialization is the prows of converting an object into sequence of bytes that can be stored on to a disk (or) saved in database (or) transmit over a network using streams.

→ the reverse process of converting a stream of bytes into object is called deserialization.

→ In order to serialize an object, its corresponding class or any of its sub classes must implement the serializable interface or its sub interface Externalizable.

→ the entire process of serialization and deserialization are JVM independent means that an object can be serialized on one platform and deserialized on a different platform

→ classes objectoutputstream and objectInputStream contains methods to perform serialization and deserialization respectively

Example: serialization implementation

```
import java.io.*;
class Emp implements Serializable
{
  String name;
  String address;
}
class SerializationDemo
{
  public static void main(String args[])
  {
Emp e = new Emp();
e.name="Vijay";
e.address="Guntur";
try{
FileOutputStream fout=new FileOutputStream("data.txt");
ObjectOutputStream os=new ObjectOutputStream(fout);
os.writeObject(e);
os.close();
```

```
fout.close();
System.out.println("the serialized object is saved in data.txt");
}catch(IOException e)
{System.out.println(e);}
}
}
```

Serialization is the process of saving the state of an object to persistant storage say a file on disc

Deserialization is the process of restoring the state of an object from a persistant storage.

Deserialization implementation:

```
import java.io.*;
class DeSerializationDemo
{
 public static void main(String args[])
 {
  Emp emp=null;
  try{
  FileInputStream fin=new FileInputStream("data.txt");
  ObjectInputStream is=new ObjectInputStream(fin);
  emp=(Emp)is.readObject();
  is.close();
  fin.close();
  System.out.println("the name of employee is:"+temp.name);
  System.out.println("the address of employee is:"+temp.address);
  }catch(IOException e)
  {}
```

to implement deserialization first we need to implement serialization

Exploring NIO:

↓
New IO

① Starting with version 1.4 java has provided a second io system called NIO (New io)

② It supports a buffer oriented, channel based approach to io operations.

③ With the release of JDK7 (java development kit) the NIO system was greatly expanded, providing enhanced support for file handling and file system features

④ The nio system is built on two fundamental features, i.e., buffers and channels.

⑤ A buffer holds data a channel represents an open connection to an io device such as a file or a socket

Buffers:

Buffers are defined in the java.nio package. all buffers are subclasses of buffer class. Several methods defined by the buffer class are:

1. final int capacity(): returns the no of elements that invoking buffer is capable of holding

2. final Buffer clear():
clears the invoking buffer and returns a reference to the buffer.

3. abstract buffer duplicate():
returns a buffer i.e identical to the invoking buffer. Thus Both the buffers will contain and refer to the same elements.

4. abstract boolean isReadonly():
Returns true if the invoking buffer is readonly, false otherwise.

5. Final int limit():
returns the invoking buffer limit

channels:

① channels are defined in the package java.nio.channels

② A channel represents an open connection to an io source of ~~data~~ such relations

③ channels implement channels interface.

④ A method to obtain a channel is by invoking the method getchannel() on an object that supports channels

⑤ Several io classes that supports channels are FileInputStream, FileOutputStream, RandomAccessFile, Socket, ServerSocket.

⑥ the type of channel returned depends on the type of object that invokes the getChannel()

⑦ when object of classes FileInputStream, FileOutputStream RandomAccessFile is used to invoke getchannel() then File channel will be returned.

⑧ when an object of class Socket, ServerSocket invoke getChannel() then the channel returned is SocketChannel

⑨ Both the channels FileChannel and SocketChannel supports various read and write methods that Enables us to perform io operations to the channels