**I/O operation**

Java performs input/output through streams. A stream is an abstraction that either produces or consumes information.

The input object reads a stream of data from the file, and an output object writes a stream of data to the file.

An input object is also called an input stream and an output object is called an output stream.

**Binary I/O vs. Text I/O**

Text I/O requires encoding or decoding.

Binary I/O does not require encoding or decoding.

**Design of Java IO classes:**

1. Object

a. InputStream

i. FileInputStream

ii. FilterInputStream

-> DataInputStream

-> BufferedInputStream

iii. ObjectInputStream

b. OutputStream

i. FileOutputStream

ii. FilterOutputStream

-> DataOutputStream

-> BufferedOutputStream

iii. ObjectOutputStream

FileInputStream / FileOutputStream is used to read / write the bytes from / to the files.

**Constructor of FileInputStream**

1. public FileInputStream(File f) throws FileNotFoundException

2. public FileInputStream(String f) throws FileNotFoundException

FileInputStream class throws this exception in these three cases:

1. File name does not exists

2. Passed file name is a directory rather than a regular file.

3. If the file does not have reading permissions.

**A Sample program to read bytes from a file**

import java.io.\*;

class FileInputStreamDemo

{

public static void main(String[] args)

{

try

{

File file = new File("File Input Demo.txt");

PrintWriter input = new PrintWriter(file);

input.print("Ashu Akash Singh");

input.close();

FileInputStream fin = new FileInputStream("File Input Demo.txt");

int i;

while( ( i = fin.read() )!= -1)

{

System.out.print((char)i);

}

System.out.println();

}

catch (FileNotFoundException e)

{}

catch(IOException e)

{}

}

}

**Constructor of FileOutputStream class**

1. public FileOutputStream(File f) throws FileNotFoundException

2. public FileOutputStream(String f) throws FileNotFoundException

3. public FileOutputStream(File f, boolean append) throws FileNotFoundException

4. public FileOutputStream(String s, boolean append) throws FileNotFoundException

Unlike FileInputStream class, FileOutputStream class does not throw FileNotFoundException if the file does not found, instead it will create an empty file with the given name. Then it writes the bytes of data to the file.

**F‎ileOutputStream class throws FileNotFoundException in the following 3 cases:**

1. File can't be created.

2. The given file is a directory rather than a regular file.

3. The passed is read only, writing permission is not available.

**A Sample program to write bytes data to the file**

import java.io.\*;

class FileOutputStreamDemo

{

public static void main(String[] args)

{

try

{

FileOutputStream fout = new FileOutputStream("File Output Demo.txt");

fout.write('a');

fout.write('b');

fout.close();

FileInputStream fin = new FileInputStream("File Output Demo.txt");

int i;

while ( ( i = fin.read() )!= -1 )

{

System.out.println((char)i);

}

fin.close();

}

catch (FileNotFoundException e)

{

e.printStackTrace();

}

catch (IOException e)

{

e.printStackTrace();

}

}

}

**Note:**

i. read() returns the ASCII integer data.

ii. write() writes ASCII character data.

**A Sample program of File-Copy**

import java.io.\*;

import java.util.Scanner;

public class FileCopy

{

public static void copyFile(String src, String dest) throws FileNotFoundException, IOException

{

FileInputStream fis = new FileInputStream(src);

FileOutputStream fos = new FileOutputStream(dest);

int i;

while ( ( i = fis.read())!= -1)

{

fos.write(i);

}

}

}

public class FileCopyMain

{

public static void main(String[] args)

{

try

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter source file name: ");

String src = scn.nextLine();

System.out.print("Enter target file name: ");

String dest = scn.nextLine();

FileCopy.copyFile(src, dest);

}

catch(FileNotFoundException fnfe)

{

System.out.println("Error: Given files are not found");

}

catch(IOException ioe)

{

System.out.println("Error: Reading or Writing Failed"); ioe.printStackTrace();

}

}

}

**Limitation of FileInputStream and FileOutputStream**

FIS and FOS allow to read and write data in the form of primitive data type of byte only. It is not possible to read and write data in the form of primitive data or objects.

To overcome from above problem we will use FilterInputStream and FilterOutputStream.

**FilterInputStream and FilterOutputStream**

Filter classes are used to improve the performance of the InputStream and OutputStream.

The filter connected to the InputStream is called FilterInputStream.

The filter connected to the OutputStream is called FilterOutputStream.

Filter cannot be connected to the source or the destination directly, instead they can be connected to the other InputStream or OutputStream.

So, all FilterInputStream classes contain constructor to take other InputStream and OutputStream object as argument, which use its basic source of data to read or write data.

**Classes of FilterInputStream**

1. DataInputStream

2. BufferedInputStream

**Classes of FilterOutputStream**

1. DataOutputStream

2. BufferedOutputStream

**DataInputStream and DataOutputStream**

DIS and DOS is used to read or write streams of data in the form of primitive data from or to the stream.

DIS reads bytes from the stream and converts them into appropriate primitive types values.

DOS converts primitive type values into bytes and output the bytes to the stream.

DIS extends from FilterInputStream class and DataInput inteface.

DOS extends from FilterOutputStream class and DataOutput interface.

DataInput and DataOutput interface contains functions to be used by DIS and DOS.

**DataInput inteface Methods (All methods throws IOException)**

1. public byte readByte()

2. public short readShort()

3. public int readInt()

4. public long readLong()

5. public float readFloat()

6. public double readDouble()

7. public char readChar()

8. public boolean readBoolean()

9. public String readUTF()

10. public String readLine()

**DataOutput interface Methods (All methods throws IOException)**

1. public void writeByte(byte b)

2. public void writeShort(short s)

3. public void writeInt(int a)

4. public void writeLong(long l)

5. public void readFloat(float f)

6. public void writeDouble(double d)

7. public void writeChar(char ch)

8. public void writeBoolean(boolean b)

9. public void writeUTF(String s)

10. public void writeBytes(String s)

**Note: readXXX methods must be called in the same order in which writeXXX methods are called, otherwise wrong value will be returned or EOFException will be raised.**

**DIS and DOS are created using the following constructors:**

1. public DataInputStream(InputStream in)

2. public DataOutputStream(OutputStream out)

**A Sample DataOutputStreamDemo program**

import java.io.\*;

class DataOutputStreamDemo

{

public static void main(String[] args)

{

try

{

FileOutputStream fos = new FileOutputStream("Data.txt");

DataOutputStream dos = new DataOutputStream(fos);

dos.writeInt(97);

dos.writeFloat(45.5F);

dos.writeChar('a');

dos.writeBoolean(true);

fos.close();

}

catch (Exception e)

{}

}

}

After compilation and execution of the program, check the size of the file. It will be of 11 bytes (int + float + char + boolean i.e. 4 + 4 + 2 + 1) bytes.

**A Sample DataInputStream program**

import java.io.\*;

class DataInputStreamDemo

{

public static void main(String[] args)

{

try

{

FileInputStream fis = new FileInputStream("Data.txt");

DataInputStream dis = new DataInputStream(fis);

int a = dis.readInt();

float f = dis.readFloat();

char ch = dis.readChar();

boolean b = dis.readBoolean();

System.out.println(a);

System.out.println(f);

System.out.println(ch);

System.out.println(b);

}

catch (Exception e)

{}

}

}

Reading and Writing data using DIS and DOS is a machine independent fashion. It enables to write data on one machine and reading it on another machine.

**BufferedInputStream and BufferedOutputStream**

BIS and BOS is used to speed the input/output of data to/from the file.

**Limitation of DIS and DOS**

The limitation of DIS and DOS is that we can only read and write primitive data types. Using DIS and DOS we cannot read or write objects from persistence media.

To overcome from this problem, ObjectInputStream and ObjectOutputStream classes will be used.

**ObjectInputStream and ObjectOutputStream**

These classes are used to store object's state permanently in files or to send to remote computers via network.

ObjectInputStream extends from InputStream class and ObjectInput and ObjectStreamConstants interface.

ObjectOutputStream extends from OutputStream class and ObjectOutput and ObjectStreamConstants interface.

ObjectInput and ObjectOutput interfaces are the subinterfaces of DataInput and DataOutput interface.

**ObjectInputStream class Method**

public Object readObject() throws IOException + all methods of DataInput inteface

**ObjectOutputStream class Method**

public void writeObject(Object obj) throws IOException + all methods of DataOutput interface.

Note: To write or send object to external world it must of type java.io.Serializable interface. It means object class must be a subclass of java.io.Serializable interface, else writeObject() throws an error java.io.NotSerializableException.

**Constructors to create ObjectStream classes**

1. public ObjectInputStream(InputStream in)

2. public ObjectOutputStream(OutputStream out)

**Serialization**

Serializable is a marker interface. It is a process of converting object into streams of bytes and sending them to the underlying OutputStream.

Serialization is performed by using writeObject() method of the ObjectOutputStream.

java.io.Serializable interface does not contain any method, it just provides permission or identity to the JVM to serialize object.

**De - serialization**

Deserialization is the process of converting stream of bytes into object, by using readObject() method of ObjectInputStream class.

**How can readObject() method load class from serialized object?**

By class name from the serialized file.

**What is the functionality of writeObject() and readObject()?**

In Serialization writeObject() stores object in the file with the following information:

1. Class name

2. Current .classname serial number

3. Non - transient variables and their data-types.

4. Those variables current modified values.

In Deserialization, readObject() creates object with the state object in the file.

1. readObject() first read the classname from the serializable file and loads that class using Class.forName(). While loading, static variables will get the memory with its originally assigned values and Static Blocks are executed.

2. Then it compares the serialVersionUID number of the loaded class and serialVersionUID number from the serialized file.

a. If the serial number did not match, then it terminates deserialization process and throws exception java.io.InvalidClassException.

b. If the serial number matches, then it creates object with the current loaded class without using new keyword and constructor.

3. Then readObject() populates the value

a. from the serialized file of non transient variables.

b. default values of the transient variables.

c. value of non - static variable those are not defined in the serialized file.

**serialVersionUID**

Serialization runtime associates each serializable class a version number called serialVersionUID, which is used during deserialization, to verify that the sender and receiver of the serialized object have the loaded class for that object that are compatible with respect to serialization. If the receicer has a loaded class for the object that has different serialVersionUID corresponding to that of sender's class, then deserialization raise unexpected InvalidClassException.

A serializable class can declare its own serialVersionUID explicitly by using a field named "serialVersionUID" which must be of static, final and of type long.

**If an object is written to an ObjectStream more than once, will it store multiple copies of it?**

No, it will not. When an object is written, for the first time a serial number is created for that object. The JVM writes the complete contents of that object along with the serial number. After the first time, only serial number is stored, if the same object is written again. It is so because while deserialization, readObject() matches the serial number of loaded object to the saved object, if they will match then only it will deserialized.

**Serialization with IS-A relation**

If super class is deriving from Serializable interface, then all its subclasses will be serialized also. There is no need to serialize subclasses explicitly. But if subclass will be serialized, then superclass will not be serialized.

**A Sample program of Serialization with IS-A relation**

import java.io.\*;

class A implements Serializable

{

int x;

A()

{

x = 10;

}

}

class B extends A

{

int y;

B()

{

y = 20;

}

public String toString()

{

return "x: " + x + "\n" +

"y: " + y;

}

}

class SerializationISADemo

{

public static void main(String[] args) throws Exception

{

B objB = new B();

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("ISADemo.ser"));

objB.x = 30;

objB.y = 40;

oos.writeObject(objB);

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("ISADemo.ser"));

System.out.println(ois.readObject());

}

}

Output:

x: 30

y: 40

**Is it compulsory to serialize superclass in order to serialize subclass?**

No (see below example)

import java.io.\*;

class A

{

int x;

A()

{

x = 10;

}

}

class B extends A implements Serializable

{

int y;

B()

{

y = 20;

}

public String toString()

{

return "x: " + x + "\n" +

"y: " + y;

}

}

class SerializationISADemo

{

public static void main(String[] args) throws Exception

{

B objB = new B();

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("ISADemo.ser"));

objB.x = 30;

objB.y = 40;

oos.writeObject(objB);

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("ISADemo.ser"));

System.out.println(ois.readObject());

}

}

Output:

x: 10

y: 40

**Serialization with HAS-A relation**

**To serialize an object, will its internal object must also be serialized?**

Yes, else object will not serialized.

class Address

{

String city = "Sasaram";

}

class Student implements java.io.Serializable

{

String name = "Ashu";

Address add = new Address();

}

The above program will raise an exception java.io.NotSerializableException, because Address is not a subclass of Serializable interface.

**Solution to the above problem**

Custom Serialization

Any serializable object has the option of customizing its own serialization, if it implements the following pair of methods:

1. private void writeObject(ObjectOutputStream oos)

2. private void readObject(ObjectInputStream ois)

Inside private void writeObject(ObjectOutputStream oos) method, we must call oos.defaultWriteObject() for normal serialization of the current object.

Inside private void readObject(ObjectInputStream ois) method, we must call ois.defaultReadObject() for normal deserialization of the current object.

**A Sample program to illustrate Serialization with HAS-A relation**

import java.io.\*;

class Address

{

String city;

int pinCode;

Address(String city, int pinCode)

{

this.city = city;

this.pinCode = pinCode;

}

}

class Student implements Serializable

{

String name;

int rollNo;

transient Address add;

Student(String name, int rollNo, Address add)

{

this.name = name;

this.rollNo = rollNo;

this.add = add;

}

private void writeObject(ObjectOutputStream oos)

{

try

{

oos.defaultWriteObject();

oos.writeUTF(add.city);

oos.writeInt(add.pinCode);

}

catch (Exception e)

{

e.printStackTrace();

}

}

private void readObject(ObjectInputStream ois)

{

try

{

ois.defaultReadObject();

String city = ois.readUTF();

int pinCode = ois.readInt();

add = new Address(city, pinCode);

}

catch (Exception e)

{

e.printStackTrace();

}

}

public String toString()

{

return "Name: " + name + "\n" +

"Roll No.: " + rollNo + "\n" +

"City: " + add.city + "\n" +

"Pin Code: " + add.pinCode;

}

}

class SerializeHASADemo

{

public static void main(String args[])

{

try

{

Address add = new Address("Sasaram", 821115);

Student stu = new Student("Ashu", 101, add);

final long serialVersionUID = 41L;

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("SerializeHASADemo.ser"));

oos.writeObject(stu);

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("SerializeHASADemo.ser"));

System.out.println(ois.readObject());

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

**What is a marker interface?**

It is an interface which is used to mark the object is of specific type to perform special operation. Marker interface does not have any methods and is an empty interface.