Unit 7

PAGE No.
DATE / /

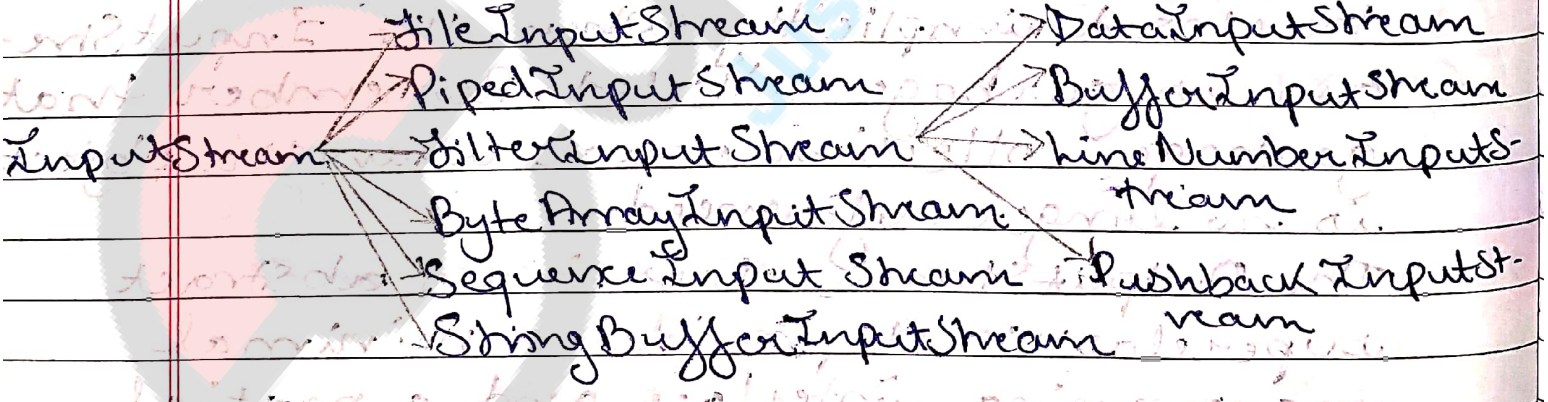
Q) Input streams and Output streams

The java io package contains two classes, InputStream and OutputStream, from which most of the other classes in the package derive.

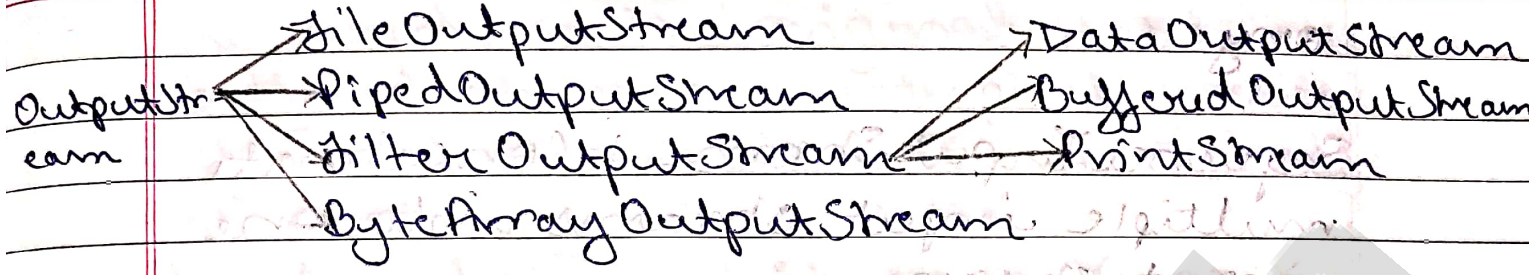
The InputStream class is an abstract superclass that provides a minimal programming interface and a partial implementation of input streams. The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading and resetting the current position within the stream. An input stream is automatically opened when you create it. You can explicitly close a stream with the close method or let it be closed implicitly when the InputStream is garbage collected. Remember that garbage collection occurs when the object is no longer referenced.

The OutputStream class is an abstract superclass that provides a minimal programming interface and a partial implementation of output streams. OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when you create it. You can explicitly close an output stream

with the close method, or let it be closed implicitly when the OutputStream is garbage collected, which occurs when the object is no longer referenced.
The java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions for example, FileInputStream and FileOutputStream are input and output streams that operate on files on the native file system.
The first of the following two figures shows the class hierarchy for the input stream classes in the java.io package. The second figure shows the class hierarchy for the output stream classes in the java.io package.



As you can see from the diagram, InputStream inherits from the Object class; six classes inherit directly from InputStream. One of InputStream's descendants, FilterInputStream, is itself an abstract class with four children.



As you can see OutputStream inherits from the Object class; four classes inherit directly from OutputStream. One of OutputStream's descendants, FilterOutputStream, is itself an abstract class with three descendants.

The following is an overview of the non-abstract classes that subclass directly from InputStream and OutputStream:

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

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

ByteArrayInputStream and ByteArrayOutputStream:- Read data from or write data to a byte array in mem-

ory.

SequenceInputStream :- Concatenate multiple input streams into one input stream.

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

Filtered Streams

FilterInputStream and FilterOutputStream are subclasses of InputStream and OutputStream, respectively and are both themselves abstract classes. These classes define the interface for filtered streams which process data as its being read or written. For example, the filtered streams BufferedInputStream and BufferedOutputStream buffer data while reading and writing to speed it up.

DataInputStream and DataOutputStream :- Read or write primitive Java data types in a machine-independent format.

BufferedInputStream and BufferedOutputStream :- Buffer data while reading or writing, thereby reducing the number of accesses

required on the original data source. Buffered streams are typically more efficient than similar non-buffered streams.

LineNumberInputStream :- Keeps track of line numbers while reading.

PushbackInputStream :- An input stream with a one-byte pushback buffer. Sometimes when reading data from a stream it is useful to peek at the next character in the stream in order to decide what to do next. If you peek at a character in the stream, you'll need to put it back so that it can be read again and processed normally.

PrintStream :- An output stream with convenient printing methods.

In addition to the stream classes, java.io contains these other classes:

File :- Represents a file on the native file system. You can create a file object for a file on the native file system and then query the object for information about that file (such as its full pathname).

FileDescriptor :- Represents a file-handle (or descriptor) to an open file or an open socket. You will not typically use this class.

PAGE No.
DATE / /

RandomAccessFile :-
Represents a random access file.

StreamTokenizer :-
Breaks the contents of a stream into tokens. Tokens are the smallest unit recognized by a text-parsing algorithm (such as words, symbols and so on). A StreamTokenizer object can be used to parse any text file. For example, you could use it to parse a Java source file into variable names, operators and so on, or an HTML file into HTML tags, words and such.

java.io defines three interfaces:-
DataInput and DataOutput :- These two interfaces describe streams that can read and write primitive Java types in machine-independent format. DataInputStream, DataOutputStream and RandomAccessFile implement these interfaces.

FilenameFilter :- This The list method in the File class uses a FilenameFilter to determine which files in a directory to list. The FilenameFilter accepts or rejects files based on their names. You could use FilenameFilter to implement simple regular expression style file search patterns

PAGE No.
DATE / /

Working with Random Access Files talks about how to use random access files. It also provides a special section that shows you how to write filters for objects that implement the DataInput and DataOutput interfaces. Filters implemented in this fashion are more flexible than regular filtered streams because they can be used on random access files and on some sequential files.

9) FileInputStream and FileOutputStream :-

FileInputStream:-

FileInputStream stream is used for reading data from the files.

Commonly used constructors of FileInputStream:

1) FileInputStream(File file):- Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.

2) FileInputStream(String name): Creates a FileInputStream by opening a connection to an actual file, the file named by the path name in the file system.

Example:

FileInputStreamExample.java

```
import java.io.FileInputStream;
class IOTest{
  public void readFile(){
    try{
      FileInputStream fis= new FileInputSt-
ream("F:\\New folder\\data1.txt");
  int i;
while((i=fis.read())!=-1){
  System.out.print((char)i);
  }
  }catch(Exception e){
    e.printStackTrack();
  }
  }
}
public class FileInputStreamExample{
public static void main(String args[]){
  IOTest obj=new IOTest();
  obj.readFile();
  }
}
```

Output:-

```
Hello World
```

FileOutputStream:

FileOutputStream is used to create a file and write data into it. It will create a file, if it does not exist.

Commonly used constructors of FileOutputStream:-

1) FileOutputStream(File file)
Creates a file output stream to write to the file represented by the specified File object.

2) FileOutputStream(String name) :-
Creates a file output stream to write to the file with the specified name.

Example:
FileOutputStreamExample.java

```
import java.io.FileOutputStream;
class IOTest{
   String str= "Hello www.codesjava.com";
   public void writeFile(){
     try{
      FileOutputStream fos= new FileOutput-
Stream("F:\\New folder\\data2.txt");
  byte b[]=str.getBytes();
   fos.write(b);
   fos.flush();
   fos.close();
System.out.println("Contents written
successfully.");
  }catch(Exception e){
     e.printStackTrace();
   }
  }
}
public class FileOutputStreamExample{
 public static void main(String args[]){
```

```
IOTest obj = new IOTest();
 obj.writeFile();
 }
}
```

Output:
Contents written successfully.

Example: Reading the data from one file and writing it into another file.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
class IOTest{
 public void readFile(){
  try{
  FileInputStream fis = new FileInputStream
  ("F:\\New folder\\data1.txt");
  FileOutputStream fos = new FileOutputStream
  ("F:\\New folder\\data7.txt");
  int i;
  while((i=fis.read())!=-1){
      fos.write(i);
  }
  System.out.println("Content written
  successfully.");
  }catch(Exception e){
      e.printStackTrace();
  }
 }
}
public class ReadWriteExample1{
```

```
public static void main(String args[]){
 IOTest obj = new IOTest();
  obj.readFile();
 }
}
```

Output:
Contents written succesfully

Read and copy with FileInputStream and FileOutputStream

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;

public class FileInput_OutputExample{
public static void main(String [] args) throws Exception{
InputStream is = new FileInputStream("in.txt");
OutputStream os = new FileOutputStream("out.txt");
int c;
while((c = is.read()) != -1){
 System.out.print((char) c);
  os.write(c);
 }
is.close();
 os.close();
}
}
```

Q] Binary and Character Streams

A stream is a sequence of data that is available over time. A source generates data as a stream. Destination consumes or read data are available as a stream. In other words, a stream explains the flow of data which allows reading or writing. Two ways of performing operations on streams in Java are by using Byte Stream and character stream.

BYTE STREAM	CHARACTER STREAM
A mechanism that performs input and output of 8 bit bytes.	A mechanism in Java that performs input and output operations of 16 bit Unicode.
Performs input and output operations of 8 bit bytes.	Performs input and output operations of 16 bit Unicode.
Common classes are FileInputStream and FileOutputStream.	Common classes are FileReader and FileWriter.

Byte Stream in Java:-

Byte streams in Java help to perform input and output operations of 8-bit bytes. In other words, it processes data byte by byte. The most frequently used

classes for Byte stream operations are FileInputStream and FileOutputStream. The FileInputStream helps to read from the source while FileOutputStream helps to write to the destination.

Eg:-

```
public class Program{
 public static void main(String args[])
 throws IOException{
  FileInputStream in=null;
  FileOutputStream out=null;
  try{
  in=new FileInputStream("input.txt");
   out=new FileOutputStream("output.txt");
  int c;
 while((c=in.read())!=-1){
  out.write(c);
  }
 }finally{
  if(in!=null){
   in.close();
  }
 if(out!=null){
  out.close();
  }
 }
}
```

According to the above program, there are two objects of FileInputStream and FileOutputStream. The while loop

reads data in the input.txt file and writes them in the new file output.txt until reaching the end of the file. The finally block will close the files. Finally, the output.txt file will also have the same content as the input.txt file. Usually, it is possible to use Byte Stream with any file type.

Character Stream in Java :-

Character Stream in java helps to perform input and output for 16 bit Unicode. The most common classes for character streaming in Java are FileReader and FileWriter. Internally FileReader uses FileInputStream. Similarly, the FileWrite uses FileOutputStream.

```
import java.io.*;
public class Program {
public static void main (String args[])
throws IOException {
  FileReader in = null;
  FileWriter out = null;
try {
  in = new FileReader("input.txt");
  out = new FileWriter("output.txt");
  int c;
while((c = in.read()) != -1) {
  out.write(c);
}
```

```
} finally {
  if(in!=null){
  in.close();
  }
  if(out!=null){
    out.close();
  }
}
}
```

According to the above program, there are two objects of FileReader and FileWriter. The while loop reads the Unicode characters in the input.txt file and writes them to the new file called output.txt until reaching the end of the file. The finally block will close the files. In the end, the output.txt file will be also have the same content as the input.txt file. The FileReader reads two bytes at a time while FileWriter writes two bytes at a time.

6) Buffered Reader/Writer

The java.io.BufferedReader and java.io.BufferedWriter classes are the character based equivalents of the byte oriented BufferedInputStream and BufferedOutputStream classes.

When a program reads from a BufferedReader, text is taken from buffer

(a block of memory) rather than directly from the underlying input stream until the buffer is empty. At this point, a read request to the underlying character stream extracts a new block of data, which refills the buffer. When a program writes to a BufferedWriter, the text is placed in the buffer. The text is moved to the underlying output stream or other target only when the buffer fills up or when the writer is explicitly flushed, which can make writes much faster than would otherwise. In order to create BufferedReader, one of the following two constructors can be used.

BufferedReader(Reader in)
BufferedReader(Reader in, int bufferSize)

The first argument in is Reader object which is the underlying character input stream from which data will be read. If the buffer size is not set, the default size of 8192 characters is used. Similarly to create BufferedWriter, one of the following two constructors are used.

BufferedWriter(Writer out)
BufferedWriter(Writer out, int bufferSize)

The first argument out is a Writer

object which is the underlying character output stream to which buffered data is written. If buffer size is not set, the default size of 8192 characters is used. The BufferedReader and BufferedWriter classes have the usual methods associated with Reader and Writer classes like read(), write(), close() etc. In addition, they also provide the following methods:

- String readline():- The readline() method of the BufferedReader class reads a single line of text and returns it as a string. The return string is null when the operation attempts to read past the end of the file.
- void newline():- The newline() method of the BufferedWriter class sends the preferred end of line character (or characters) for the platform being used to run the program.

Now let us consider a program that reads a file and converts its contents to uppercase and write them to a new file.

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

```
{
BufferedReader br=new BufferedReader
    (new FileReader("java.txt"));
BufferedWriter bw= new BufferedWriter(
  new BufferedFileWriter ("java.txt"));
  int ch;
  while ((ch=br.read())!=-1)
    {
    if(Character.isLowerCase((char)ch))
  bw.write(Character.toUpperCase((char)
    ch));
    else
     bw.write((char) ch));
      }
      br.close();
      bw.close();
    }
    catch(Exception e){e.printStackTrace
      ();}
      }
    }
```

Output:-

Welcome Java World.

WELCOME JAVA WORLD

Q) Object serialization and Deserialization

Serialization in java is a mechanism of writing the state of an object into a byte stream and deserialization is the process of converting a stream of bytes back into a copy of the

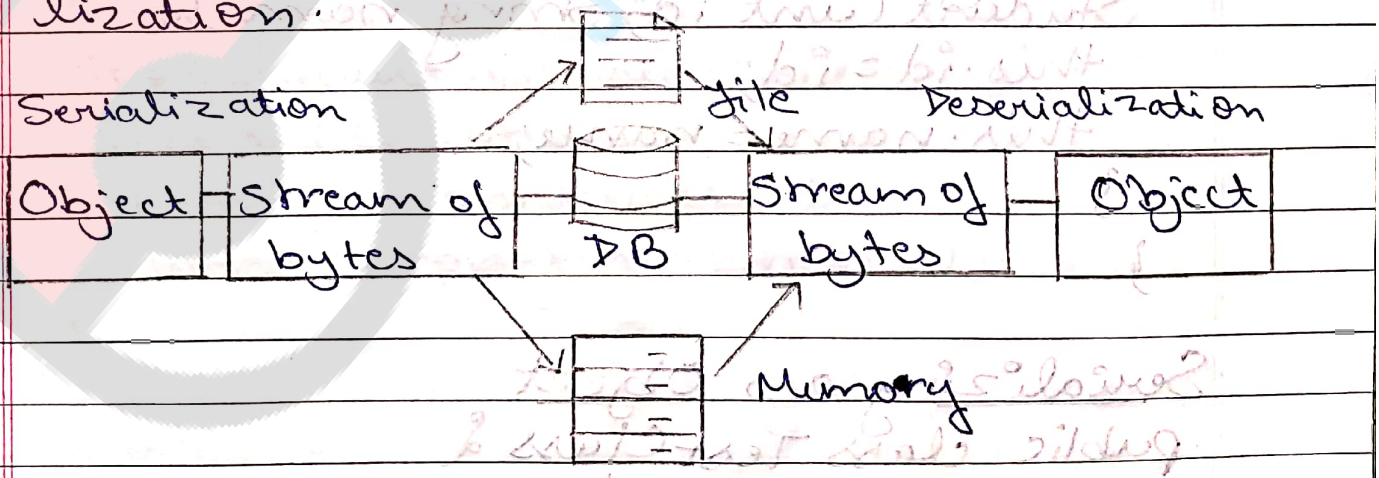
original object A Java object is serializable if its class or any of its superclasses implements either the java.io.Serializable interface or its subinterface, java.io.Externalizable.

Why?

This serialization and deserialization helps us in many scenarios like gaming, session state management etc.

Lets understand this by examining computer games.

When we stop or pause a running computer game, it usually starts from the state where it was left off when we play it again. This is possible through the process of Serialization; which is saving the current object state as byte stream into file or database. Then to restore the object's state when we access the game again is called Deserialization.



Marker Interface

The Serializable interface is a "marker" interface. This means that it has no

methods or fields, but simply "marks" a class as being able to be serialized. When the Java Virtual Machine (JVM) encounters a class that is "marked" as Serializable during a serialization process, the Virtual Machine will assume that it is safe to write to the stream. These all happens somewhat automatically for a programmer. Following are the well-known Marker Interfaces.

- rmi.Remote
- io.Serializable
- lang.Cloneable

Example

```
import java.io.*;
class Student implements Serializable
 {
  int id;
  String name;
  Student(int id, String name){
   this.id = id;
   this.name = name;
  }
 }
```

Serializing an Object

```
public class TestClass {
public static void main(String[]args){
 try {
Student st = new Student(101, "John");
FileOutputStream fos= new FileOutputStr.
```

```
eam("student.info");
ObjectOutputStream oos = new ObjectOutputS-
tream(fos);
oos.writeObject(st);
oos.close();
fos.close();
} catch(Exception e) {
System.out.println(e);
  }
  }
}
```

Deserialization of Object

```
public class TestClass{
public static void main(String[]args){
Student st=null;
try{
FileInputStream fis=new FileInputStream
  ("student.info");
ObjectInputStream ois=new ObjectInputSt-
ream(fis);
  st=(Student)ois.readObject();
  }catch(Exception e){
  System.out.println(st.id);
  System.out.println(st.name);
  }
}
```

Serialization and Variables :-

Instance Variables :-

These variables are serialized, so during deserialization we will get back

to the serialized state.

Static Variables:-
These variables are not serialized, so during deserialization static variable value will be loaded from the class. But, any static variable that is provided a value during class initialization is serialized. However in usual cases, where you would provide the value to a static variable at the main class at run-time would not be serialized.

Transient Variables:
Transient variables are not serialized, so during deserialization those variables will be initialized with corresponding default values.

Super class variables:-
If super class also implemented Serializable interface then those variables will be serialized, otherwise it wont serialize the super class variables. While deserializing, Java Virtual Machine (JVM) will run default constructor in super class and populates the default values. Same thing will happen for all super classes.