Unit-3: Streams

File Handling

(Byte Stream, Character Stream)

File Class

- The File class from the java.io package, allows us to work with files.
- To use the File class, create an object of the class, and specify the filename or directory name:
- Example:-
- import java.io.File;   // Import the File class
- File myObj = new File("filename.txt"); // Specify the filename

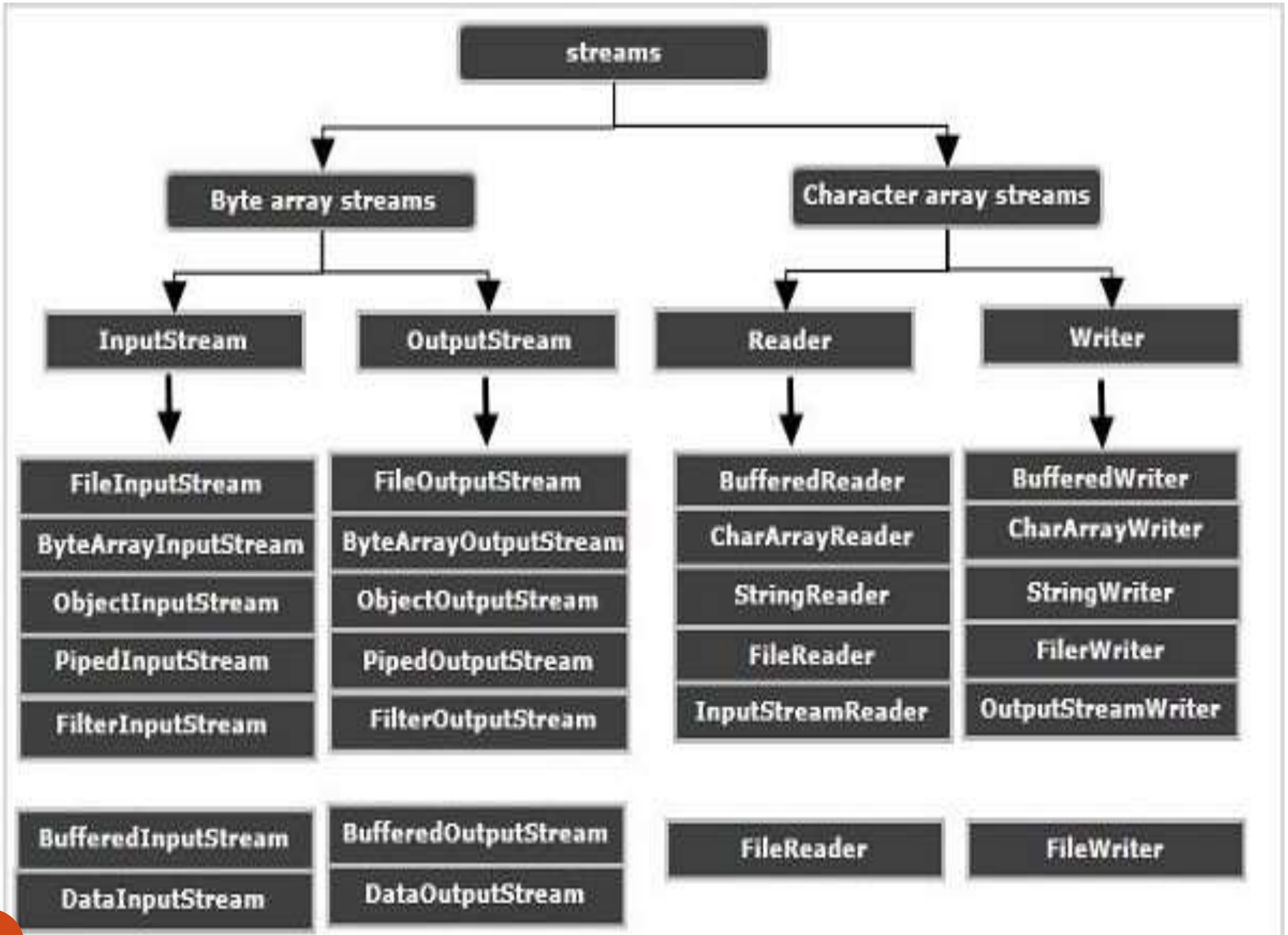
- The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

```
//file class with it's method
mport java.io.*;
public class fileCreation
{    public static void main(String
args[])
 {   try
{
 File obj=new File("E:\\h1.txt");
 if(obj.exists())
//obj.createNewFile()
{
 System.out.println("file
name:"+obj.getName());
System.out.println("file length:"
+obj.length());
System.out.println("file writable:"
+obj.canWrite());
   System.out.println("file readble: "
+obj.canRead());
 System.out.println(
obj.getAbsolutePath());
//obj.delete();
            }
  else
    {   System.out.println("already
exist");
    }}
   catch(Exception e)
            {    System.out.println(e);  }
     }}
```

Stream

- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- In general, a Stream will be an input stream or, an output stream.
- **InputStream** – This is used to read data from a source.
- **OutputStream** – This is used to write data to a destination.
- Based on the data they handle there are two types of streams :
- **Byte Streams** – These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
- **Character Streams** – These handle data in 16 bit Unicode. Using these you can read and write text data only.


streams
Byte array streams
Character array streams
InputStream
OutputStream
Reader
Writer
FileInputStream
ByteArrayInputStream
ObjectInputStream
PipedInputStream
FilterInputStream
FileOutputStream
ByteArrayOutputStream
ObjectOutputStream
PipedOutputStream
FilterOutputStream
BufferedReader
CharArrayReader
StringReader
FileReader
InputStreamReader
BufferedWriter
CharArrayWriter
StringWriter
FilerWriter
OutputStreamWriter
BufferedInputStream
DataInputStream
BufferedOutputStream
DataOutputStream
FileReader
FileWriter

Byte Stream Classes in Java

- Byte Stream classes are used to read bytes from the input stream and write bytes to the output stream. In other words, we can say that Byte Stream classes read/write the data of 8-bits. We can store video, audio, characters, etc., by using Byte Stream classes. These classes are part of the java.io package.

- The Byte Stream classes are divided into two types of classes, i.e., InputStream and OutputStream. These classes are abstract and the super classes of all the Input/Output stream classes.

InputStream

- InputStream is an abstract class that defines java's model of streaming byte input. it implements the closable interface. most of the methods in this class will throw an IO exception on error condition. Here, are some common methods used in class:

Methods	**Descriptions**
int available()	Returns the number of bytes of input currently available foe reading.
Void close()	Close the input source. further read attempts will generate an Io exception.
int read()	Returns an integer representation of the next available bytes of input. -1 returned when the end of the files is encountered.
Void reset()	Resets the input pointer to the previously set mark.

OutputStream

- OutputStream is an abstract class that defines streaming byte output. It implements the closable and flushable interfaces. most of the methods in this class returns void and throw an IOException in the case of errors. Here, are some common methods used in class:

Methods	**Descriptions**
Void write(int b)	Write a single bytes to an output stream. note that the parameter in an int , which allows you to call write() with expression without having to cast them back to byte.
Void close()	closes the output stream. Further write attempts will generate an IOException.
Void write(byte buff[])	Write a complete array of bytes to an output stream. .
Void flush()	Finalize the output state so that any buffers are cleared .that is, it flushes the output buffers.

FileInputStream

- The FileInputStream class creates an InputStream that you can use to read bytes from a file. its two most common constructors are shown here:
- FileInputStream(String filepath)
- FileInputStream(File fileobj)

Either can throw a FileNotFoundException. Here, filepath is the full path name of a file, and fileobj is a file object that describes the file.

- FileInputStream f=new FileInputStream("fileinput.java");
- File f0=new File("fileinput.java");

FileInputStream f1=new FileInputStream(f0);

//program to demostrate FileInputStream

```
import java.io.*;
class fileread
{
  public static void main(String[] args)  throws IOException
  {
     int size;
           FileInputStream f=new FileInputStream("c:\\User\\new.txt");
           System.out.println("total available bytes:" +(size=f.available()));
           for(int i=0;i<size;i++)
           {
                 System.out.print((char) f.read()); }
            f.close();
      } }
```

Output:

```
total available bytes:15
 welcome to java
```

FileOutputStream

- FileOutputStream creates an OutputStream that you can use to write bytes to a file. Its most commonly used constructors are shown here:
- FileOutputStream(String filepath)
- FileOutputStream(File fileobj)
- FileOutputStream(String filepath, boolean append)
- FileOutputStream(File fileobj, boolean append)

- They can throw a FileNotFoundException.here, filepath is the full path name of a file, and fileobj is a File object that describe the file. if append is true, the file is opened in append mode.

- Creation of a FileOutputStream is not dependent on the file already existing. FileOutputStream will create the file before opening it for output when you create the object.

//program to demostrate FileOutputStream

```
import java.io.*;
class filewrite
{
  public static void main(String[] args)  throws IOException
   {
     FileOutputStream fout=new
    FileOutputStream("c:\\Users\\Admin\\Desktop\\new.txt");
    String s="Welcome to java”;
     byte b[]=s.getBytes();                    //converting string into bytes
      fout.write(b);
      fout.close();
      System.out.println("success...");
}}
```

```
Output:  new.txt
          welcome to java
```

DataOutputStream and DataInputStream

- DataOutputStream and DataInputStream enables you to write or read primitive data to or from a stream. they implement the Dataoutput & DataInput interfaces, respectively.
- These interfaces define methods that convert primitive values to or from a sequence of bytes. These stream make it easy to store binary data, such as integers or float values in a file.

- DataOutputStream extends FilterOutputStream, which extends OutputStream.

  DataOutputStream(OutputStream out)
- DataInputStream is the complement if DataOutputStream .it extend FilterInputStream,which extends InputStream,and it implements the DataInput interfaces.

  DataInputStream(InputStream in)

```
//program for datainputstream &
    dataoutputstream
import java.io.*;
class filenew
{
  public static void main(String[] args)
    throws IOException
  {
    FileOutputStream fout = new
    FileOutputStream("test.txt");
    DataOutputStream out=new
    DataOutputStream(fout);
    out.writeDouble(9.2);
    out.writeInt(25);
    out.writeBoolean(true);
    out.writeUTF("hello");
    out.close();
    System.out.println("written");

    FileInputStream fin=new
    FileInputStream("test.txt");
    DataInputStream in=new
    DataInputStream(fin);
    double d=in.readDouble();
    int i=in.readInt();
    boolean b=in.readBoolean();
    System.out.println(in.readUTF());
    System.out.println("values:" +d+ " "
    +i+ " " +b);
    in.close();
  }
}
```

```
//program to demonstrate ByteArrayInputStream
import java.io.*;
 class ByteArrayEx
 {
    public static void main(String [] args) throws IOException
    {
       byte b[]={35,36,37,38};
      ByteArrayInputStream in=new ByteArrayInputStream(b);
     BufferedInputStream f=new BufferedInputStream(in);
      int c=0;
      while((c=f.read())!=-1)
    {
     char ch=(char) c ;
    System.out.println("ASCII value of Character is:" + c + ", Special character is:
   " + ch);
    }}}
```

Character stream

- While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters.

❑ Reader

- Reader is an abstract class that defines java's model of streaming character input. it implements the closable() and readable() interfaces.

❑ Writer

- Writer is an abstract class that define streaming character output. it implements the closable, flushable and Appendable interfaces.

Filereader

- The FileReader class creates a Reader that you can use to read the contents of a file. its two most commonly constructor are shown here:
- FileReaderd(String filepath)
- FileReader(File fileobj)
- Either can throw a FileNotFoundException.

```
//program to demonstrate a FileReader
import java.io.*;
class filenew
{     public static void main(String[] args)  throws IOException
   {
    FileReader fr=new FileReader("new.txt");
    String s;
    while((s= fr.readLine())!= null)
    {      System.out.print(s);
    }
}
}
```

Output:

```
Hello to java
```

FileWriter

- FileWriter creates a Writes that you can use to write to a file. its most commonly used constructor are shown here:
  - FileWriter(String filepath)
  - FileWriter(String filepath,boolean appened)
  - FileWriter(File fileobj)
  - FileWriter(File fileobj, boolean appened)

- They can throw IOException.
- Creation of FileWriter is not dependent on the file already existing. FileWriter create the file before opening it for output when you create object. in the case where you attempt to open a read-only file, an IOException will be thrown.

```
//program for filewriter class
import java.io.*;
class filenew
{
   public static void main(String[] args)
     throws IOException
 {
     String s="hello to java." +"  "+
    "welcome in file handling.";
char buffer[]=new char[s.length()];
 s.getChars (0,s.length(),buffer,0);
    FileWriter fr=new
    FileWriter("file2.txt");
for(int i=0;i<buffer.length;i+=2)
    {
      fr.write(buffer[i]);
   }
   fr.close();
   FileWriter f1=new
   FileWriter("file4.txt");
   f1.write(buffer);
   f1.close();
   }}
```

output:

```
 file2.txt
  hlot aawloei iehnln
 file4.txt
  hello to java. welcome in file handling.
```

BufferedReader

- BufferedReader improves performance by buffering input. it has two constructor:
- BufferedReader(Reader inputStream)
- BufferedReader(Reader inputStream,int bufsize)

BufferedWriter

- BufferedWriter is a writer that buffers output. Using a BufferedWriter can increase performance by reducing the number of times data is actually written to the output stream. A bufferedWriter has two constructor:
- Bufferedwriter(Writer outputstream)
- Bufferedwriter(Writer outputstream,int buffsize)

```
//program to demonstrate bufferreader
import java.io.*;
class filewrite
{     public static void main(String[] args)  throws IOException
   {     FileReader f=new FileReader(args[0]);
         BufferedReader br=new BufferedReader(f);
        int i;
        while((i=br.read())!=-1)
        {
            System.out.println((char)i);
        }
     }}
```

Output: java filewrite new.txt

Welcome to java.

```
//program to demonstrate bufferedWriter
import java.io.*;
class filechar
{
  public static void main(String[] args)  throws IOException
  {
         String str="this is to be written in file4.txt";
          char buffer[]={'a','b','c','d','e'};
          FileWriter f=new FileWriter("file4.txt");
          BufferedWriter br=new BufferedWriter(f);
          br.write(str);
          br.write(buffer,1,3);
          for(int i=10;i>=1;i--)
          {     br.write(i + " ");    }
          br.close();
  }}
```

Output: file4.txt

```
this is to be written in file4.txt bcd10 9 8 7 6 5 4 3 2 1
```

Random Access File

- This class is used for reading and writing to random access file. A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

constructor	Description
RandomAccessFile(File file,String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name,String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

```
 // program to access Random file access
import java.io.*;
public class filewrite
{
  public static void main(String[] args)
  {    try
    {
          double d = 1.5;
           float f = 14.56f;
           RandomAccessFile file = new RandomAccessFile("data.txt", "rw");
           file.writeUTF("PrepBytes is an Ed-Tech Company");
           file.seek(0); // File Pointer at index position - 0
           System.out.println("Use of read() method : " + file.readUTF());
            file.seek(0);
            byte[] b = { 1, 2, 3 };
            System.out.println("Use of read(byte[] b) : " + file.read(b));
            System.out.println("Use of readBoolean() : "    + file.readBoolean());
```

Conti…

```
     file.seek(0);
     file.writeDouble(d);
     file.seek(0);
     System.out.println("Use of readDouble() : " + file.readDouble());
     file.seek(0);
     file.writeFloat(f);
     file.seek(0);
      System.out.println("Use of readFloat() : " + file.readFloat());
     file.close();
    }
    catch (IOException e)
    {     System.out.println(e);
     }}}
```

- **NOTE: Seeking in a RandomAccessFile**
- To read or write at a specific location in a RandomAccessFile you must first position the file pointer at (AKA *seek*) the position to read or write. This is done using the seek() method. Here is an example of seeking to a specific position in a Java RandomAccessFile:

Stream Tokenizer class

- The Stream Tokenizer class helps in identifying the patterns in the input stream. It is responsible for breaking up the InputStream into tokens, which are delimited by a set of characters.

- The best use of this class is to identify the number of words or lines within a file. A stream is tokenized by creating a StreamTokenizer with a Reader object as its source and then setting parameters for the screen.

- Using a Java StreamTokenizer you can move through the tokens in the underlying Reader. You do so by calling the nextToken() method of the StreamTokenizer inside a loop. After each call to nextToken() the StreamTokenizer has several fields you can read to see what kind of token was read, it's value etc.

- <u>These fields are:</u>
- Ttype : The type of token read (word, number, end of line)

  Types of Token Types

- TT_WORD: A word is scanned. The string field value contains the word that is scanned.
- TT_NUMBER: A number is scanned. The double field Value contains the value of the number. Only decimal floating numbers are recognized.
- TT_EOL: An end-of-line is found.
- TT_EOF: The end-of-file is reached.

- sval : The string value of the token, if the token was a string (word)
- nval : The number value of the token, if the token was a number.

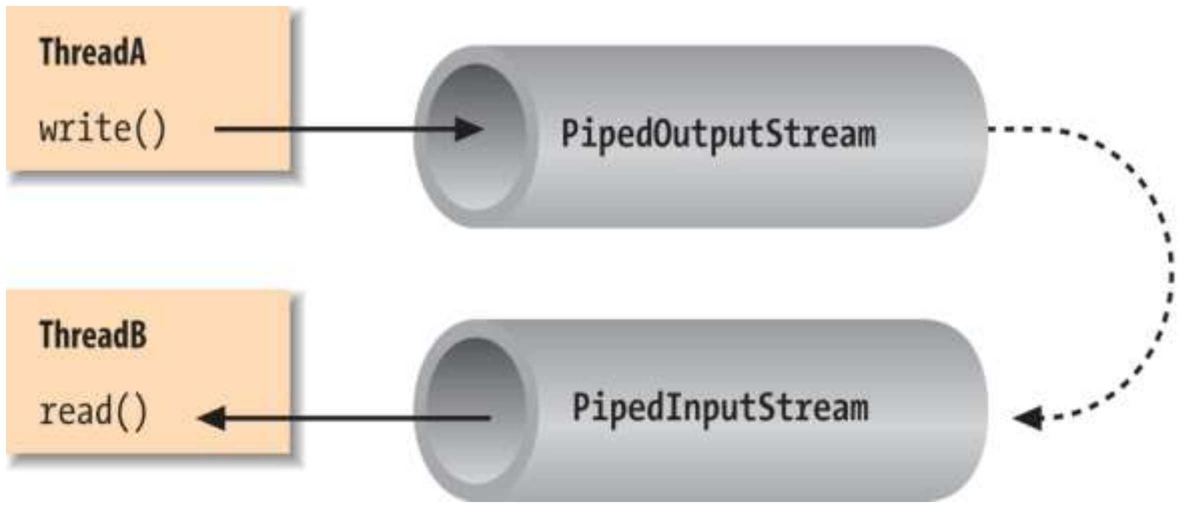
```
//program for stream tokenizer
import java.io.*;
public class wordcounter
{
    public static void main(String args[])throws IOException
{
            FileReader fr=new FileReader("arr.java");
            StreamTokenizer input=new StreamTokenizer(fr);
             int tok;
             int count=0;
            while((tok=input.nextToken())!=StreamTokenizer.TT_EOF)
            if(tok==StreamTokenizer.TT_WORD)
            {
            System.out.println("Word Found: "+input.sval);
            count++;
            }
            System.out.println("Found "+count+ " in file" );
 }}
```

```
import java.io.*;
public class wordcounter
{       public static void main(String args[])throws IOException
{
 StreamTokenizer input = new StreamTokenizer(=  new StringReader("Mary had
1 little lamb..." +"\n"+ "hello moto”));
 while(input.nextToken() != StreamTokenizer.TT_EOF)
{
      if(input.ttype == StreamTokenizer.TT_WORD)
     {
       System.out.println(input.sval);
     }
    else if(input.ttype == StreamTokenizer.TT_NUMBER)
    {
       System.out.println(input.nval);
     }
    else if(input.ttype == StreamTokenizer.TT_EOL)
    {       System.out.println();
}}}}
```

Java Pipe

- A pipe connects an input stream and an output stream.
- A piped I/O is based on the producer-consumer pattern, where the producer produces data and the consumer consumes the data.
- In a piped I/O, we create two streams representing two ends of the pipe. A PipedOutputStream object represents one end and a PipedInputStream object represents the other end. We connect the two ends using the connect() method on the either object.
- We can also connect them by passing one object to the constructor when we create another object.
- <u>Method-1</u>
- PipedInputStream pis = new PipedInputStream();
  PipedOutputStream pos = new PipedOutputStream();
  pis.connect(pos,20); /* Connect the two ends */

- Method-2
- PipedInputStream pis = new PipedInputStream(); PipedOutputStream pos = new PipedOutputStream(pis);
- We can produce and consume data after we connect the two ends of the pipe.
- A piped stream has a buffer with a fixed capacity to store data between the time it is written to and read from the pipe.



```
import java.io.*;
class encount
{
public static void main(String
    args[])throws IOException {
  PipedOutputStream pout=new
    PipedOutputStream();
  PipedInputStream pin=new
    PipedInputStream();
 pout.connect(pin);//connecting the
    streams
//creating one thread t1 which writes
    the data
Thread t1=new Thread()  {
public void run()  {
for(int i=1;i<=6;i++){
Try {   pout.write(i);
     Thread.sleep(1000);
}
catch(Exception e)
{System.out.println(e);}
} } };
//creating another thread t2 which reads
    the data
Thread t2=new Thread(){
public void run(){
try{
for(int i=1;i<=4;i++)
System.out.println(pin.read());
}
catch(Exception e)
{   System.out.println(e);  }
}};
//starting both threads
t1.start();
t2.start();
}}
```

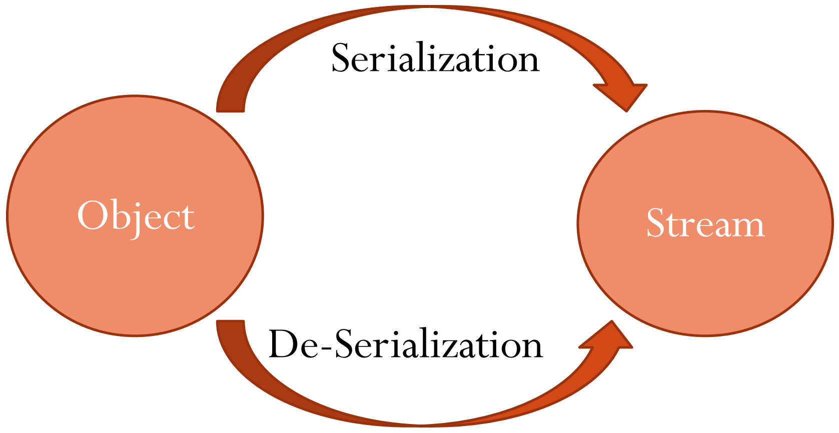
```
import java.io.*;
public class pipex
{  public static void main(String[] args) throws
    Exception
  {
   PipedInputStream pis = new
    PipedInputStream();
   PipedOutputStream pos = new
    PipedOutputStream();
   pos.connect(pis);
   Runnable producer = () -> produceData(pos);
   Runnable consumer = () -> consumeData(pis);
   new Thread(producer).start();
   new Thread(consumer).start();
  }
 public static void
    produceData(PipedOutputStream pos) {
   try {
    for (int i = 1; i <=5; i++) {
     pos.write((byte) i);
      pos.flush();
      System.out.println("Writing: " + i);
     Thread.sleep(500);
    }
    pos.close();
    } catch (Exception e) {
    e.printStackTrace();
  }}
  public static void
   consumeData(PipedInputStream pis) {
    try {
    int num = -1;
    while ((num = pis.read()) != -1) {
      System.out.println("Reading: " + num);
     }
      pis.close();
    } catch (Exception e) {
    e.printStackTrace();
  }}}
```

Serialization and De-serialization in Java

- Serialization in Java is a mechanism of writing the state of an object into a byte-stream. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.
- The reverse operation of serialization is called de-serialization where byte-stream is converted into an object. The serialization and de-serialization process is platform-independent, it means you can serialize an object on one platform and de-serialize it on a different platform.
- For serializing the object, we call the writeObject() method of ObjectOutputStream class, and for deserialization we call the readObject() method of ObjectInputStream class.
- We must have to implement the Serializable interface for serializing the object.

java.io.Serializable interface

- Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability.
  The Cloneable and Remote are also marker interfaces.
- The Serializable interface must be implemented by the class whose object needs to be persisted.
- <u>Advantages : </u>It is mainly used to travel object's state on the network (that is known as marshalling).



ObjectInputStream

- The Java ObjectInputStream class enables you to read Java objects from an InputStream instead of just raw bytes. You wrap an InputStream in a ObjectInputStream and then you can read objects from it. Of course the bytes read must represent a valid, serialized Java object. Otherwise reading objects will fail.
- Constructor: public ObjectInputStream(InputStream in)
- Method:

Method	**Desciption**
public final Object readObject() throws IOException, ClassNotFoundException{}	It reads an object from the input stream.
2) public void close() throws IOException {}	It closes ObjectInputStream.

ObjectOutputStream

- The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.
- Constructor: public ObjectOutputStream(OutputStream out)
- Method:

Method	**Description**
public final void writeObject(Object obj) throws IOException {}	It writes the specified object to the ObjectOutputStream
public void flush() throws IOException {}	It flushes the current output stream.
public void close() throws IOException {}	It closes the current output stream.

```
import java.io.*;
class Student implements Serializable
{
  int id;
  String name;
  Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
class Objectex
{
 public static void main(String args[]){
  try{
Student s1 =new Student(211,"ravi");
  //Creating stream and writing the
    object
  FileOutputStream fout=new
  FileOutputStream("f.txt");
ObjectOutputStream out=new
  ObjectOutputStream(fout);
out.writeObject(s1);
out.close();
System.out.println("success");
ObjectInputStream oi =
          new ObjectInputStream(new
  FileInputStream("f.txt"));
Student e= (Student) oi.readObject();
oi.close();
System.out.print(e.id );
 System.out.println(e.name);
 }
catch(Exception e) {
   System.out.println(e);}
 } }
```