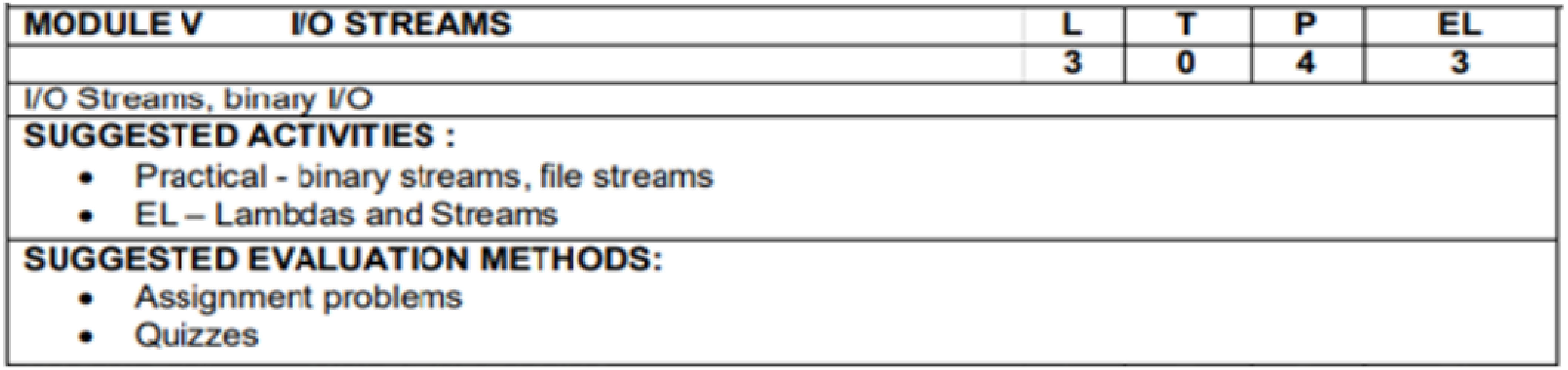
CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Module V



MODULE V I/O STREAMS	L	T	P	EL
	3	0	4	3

I/O Streams, binary I/O

SUGGESTED ACTIVITIES :

- Practical - binary streams, file streams
- EL – Lambdas and Streams

SUGGESTED EVALUATION METHODS:

- Assignment problems
- Quizzes

Unicode

- Unicode is a character encoding
  - What : assigns a number to every character (and symbols)
  - Why : every computer prints the same character (and symbols).
- Java uses a Unicode
  - What :16 bit Unicode UTF-16
    - UTF-Unicode Transformation unit
  - Why : to unify the world language characters. i.e 2^{16} - 65,536 characters
  - Example : java code can contain Tamil /Chinese character(or symbol) as class name(or variable name) and string literal.

Introduction

- **I/O**
  - Java Programs read inputs from source(eg., File) and write outputs to destination(eg. File)
  - Source or destination can also be a console, file, network, memory buffer, or another program.
- In Java standard inputs and outputs are handled by *streams.*
- A ***stream*** is a sequence of data.
  - **Input Stream :** to read data from a source, one item at a time
  - ***output stream:*** to write data to a destination, one item at time
- Stream I/O operations involve three steps:
  - *Open* an input/output stream associated with a physical device
    - (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
  - *Read* from the opened input stream until "end-of-stream" encountered
  - *Write* to the opened output stream (and optionally flush the buffered output).
  - *Close* the input/output stream.

I/O Streams

- Byte Streams
  - handle I/O of raw binary data.
- Character Streams
  - handle I/O of character data, automatic translation to and from the local character set.
- Buffered Streams
  - optimize input and output by reducing the number of calls to the native API.
- Data Streams
  - handle binary I/O of primitive data type and String values.
- Object Streams
  - handle binary I/O of objects.
- Scanning and Formatting
  - allows a program to read and write formatted text.
- I/O from the Command Line
  - describes the Standard Streams and the Console object.

Byte streams

- Byte streams are used to read/write *raw bytes* serially from/to an external device.
- Byte streams perform input and output on 8 bits.
  - Example: the text file uses **unicode encoding** to represent character in two bytes, the byte stream will read one byte at a time.
- Byte streams should only be used for the most primitive I/O.
- read/write stream Class
  - java.io.InputStream
  - java.io.OutputStream

1Byte=8Bits
1Char=16Bits(2Bytes)

Input Stream

In Java, the throws keyword is used in a method signature to indicate that the method might throw one or more exceptions. It signals that the method does not handle these exceptions directly, so they need to be caught and handled by the method that calls it, or by a higher-level method.

- Input Stream
  - **To Read from an InputStream require read() method**

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

- The read() method:
  - returns the int in the range of 0 to 255 –>bytes that read
  - returns -1 -> end of stream
  - throws an IOException if it encounters an I/O error.

```
public int read(byte[] bytes, int offset, int length) throws IOException

bytes=> destination byte
Offset=> start location of the destination bytes array
Length => length of the byte to be read


public int read(byte[] bytes) throws IOException
```

OutputStream

- Output Stream
  - **To write** bytes r**equire write() method**

```
public void write(int unsignedByte) throws IOException
```

  - To write a block of byte-array :

```
public void write(byte[] bytes, int offset, int length) throws IOException

example:write(bytes, 0, bytes.length)
public void write(byte[] bytes) throws IOException
```

Character streams

- Why?
  - Easy to write programs
  - Easy to internationalize i,.e that are not dependent upon a specific character encoding.
  - Efficient than byte streams.
    - In java Char Data type is of two-byte, the only unsigned type in Java.
    - To read a single character require single read operations whereas byte stream require two read operations (char use 2 byte).
- read/write character stream class
  - java.io.Reader
  - java.io.Writer

Reader

An abstract method in Java is a method that is declared without an implementation (i.e., without a method body). It only provides the method signature and must be implemented by subclasses that inherit from the abstract class where the abstract method is declared.

Key Points:
No implementation: An abstract method only has a method signature (its name, return type, and parameters) but no body.
Defined in an abstract class: Abstract methods can only exist within an abstract class.
Must be implemented by subclasses: Any concrete (non-abstract) subclass of an abstract class must provide an implementation for all of its abstract methods, or it must itself be declared abstract.

- superclass Reader operates on char.
- read() method
  - abstract method read() to read one character from the input source.
  - read() returns the character as an int between 0 to 65535
    - (a char in Java can be treated as an unsigned 16-bit integer);
  - or -1 if end-of-stream is detected;
  - or throws an IOException if I/O error occurs.

```
public int read() throws IOException
public int read(char[] chars, int offset, int length) throws IOException
public int read(char[] chars) throws IOException
```

Writer

- superclass `Writer` operates on `char`.
- `write()` method
  - It declares an `abstract method write()` to write one character into destination.
  - or throws an `IOException` if I/O error occurs.

```
public void abstract void write(int aChar) throws IOException

public void write(char[] chars, int offset, int length) throws IOException


public void write(char[] chars) throws IOException
```

Character-stream class	Description	Byte-stream class
Reader	input streams 8-bit bytes 16-bit Unicode	InputStream
BufferedReader	Buffers input, parses lines	BufferedInputStream
CharArrayReader		ByteArrayInputStream
InputStreamReader	Translates byte stream into character stream (UTF-8/UTF-16 …)	-
FileReader		FileInputStream
Writer	output streams	OutputStream
BufferedWriter		BufferedOutputStream
CharArrayWriter	Writes to a character array	ByteArrayOutputStream
OutputStreamWriter		-
FileWriter	Translates character stream into a byte File	FileOutputStream
PrintWriter		PrintStream

//FileInputStream is a subclass of InputStream.

//Using byte stream classes!!!

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {          //Here we are just copying bytes from one file to another!!
        public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
```

```
try {
   in = new FileInputStream("SOURCE.txt");
  out =new FileOutputStream("DESTINATION.txt");
  int c;
  while ((c = in.read()) != -1)
  {  out.write(c); }        out.write((char)c);        //this is to copy charachters
  }                                                   from one file to another
finally {
  if (in != null) {
     in.close(); }
  if (out != null)
   out.close(); }
}


}
}
```

OR

```
try (FileInputStream in = new
FileInputStream("SOURCE.TXT");
FileOutputStream out = new
FileOutputStream("DESTINATION.TXT"))
{
int c;
while ((c = in.read()) != -1)
{ out.write(c); }
    }
catch(IOException e){ }
}
}
```

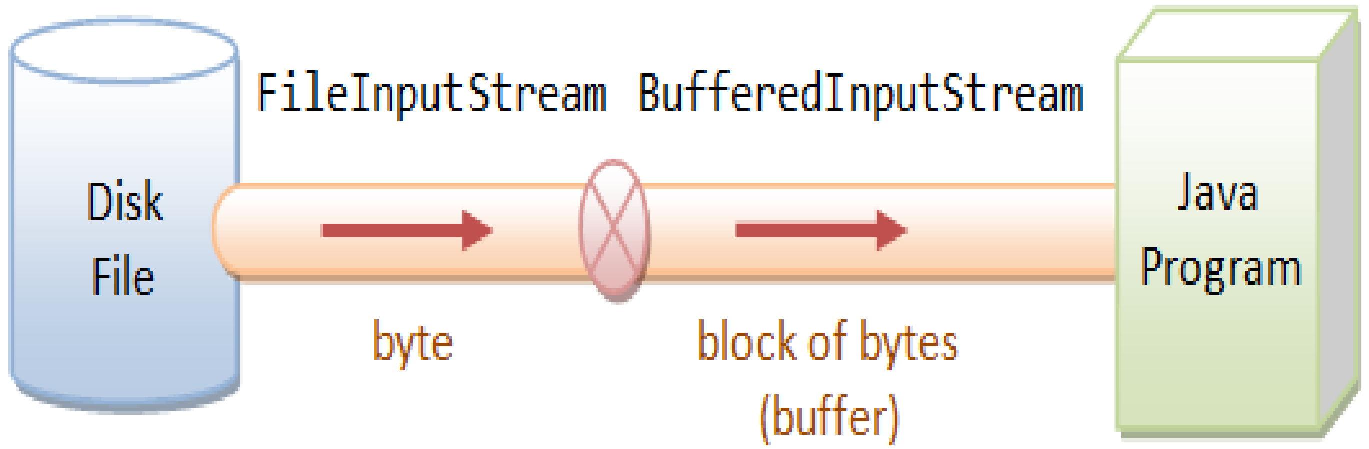
In your code, in becomes null if it is explicitly assigned null before opening the file. However, in this specific example, in is not null because it has been initialized with a FileInputStream object when you execute:

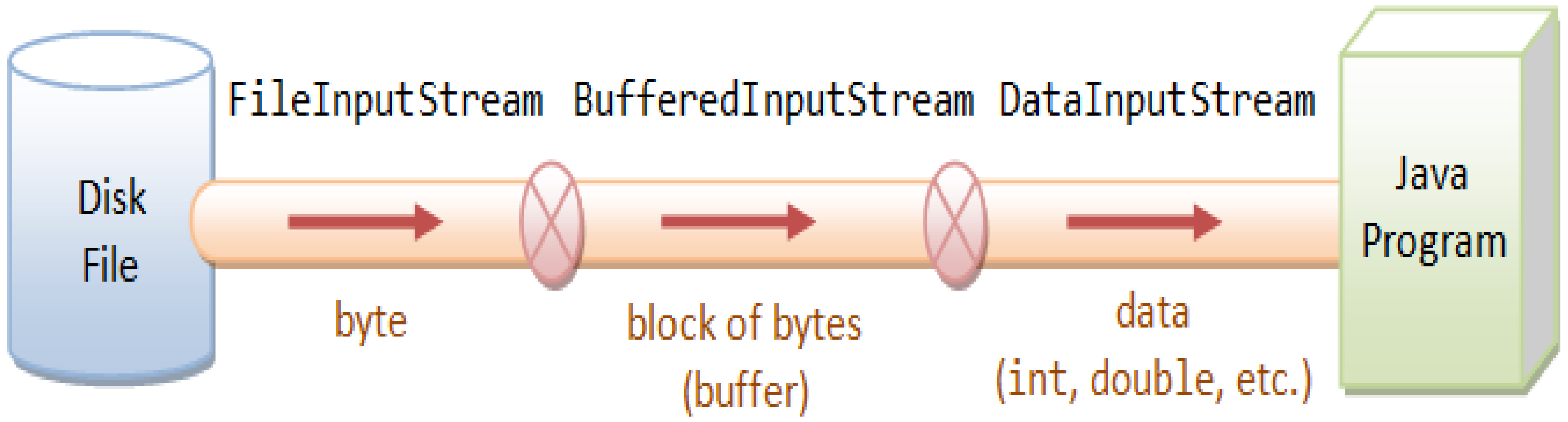
java
in = new FileInputStream("SOURCE.txt");

So, unless something goes wrong when creating the FileInputStream (like the file doesn't exist or can't be accessed), in will not be null.

```
import java.io.FileReader;                    //Using Character stream classes!!
import java.io.FileWriter;
import java.io.IOException;
public class CopyCharacters {
        public static void main(String[] args) throws IOException {
                FileReader inputStream = null;
                FileWriter outputStream = null;
                try {
                        inputStream = new FileReader(“SOURCE.txt");
                        outputStream = new FileWriter(“DESTINATION.txt”);
                        int c;
                        while ((c = inputStream.read()) != -1) {
                                outputStream.write(c); }
                    }
                finally {
                        if (inputStream != null) {
                                inputStream.close(); }
                        if (outputStream != null) {
                                outputStream.close(); }
                         }

 } }
```





Ref :https://www.ntu.edu.sg

//Byte Stream class

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBuffer{
    public static void main(String[] args) throws IOException {
  try (BufferedInputStream in = new BufferedInputStream(new FileInputStream("Source.txt"));
   BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("Dest.txt")))
   {
    int c;
   while ((c = in.read()) != -1)
    { out.write(c); }
      }
  catch(IOException e) { }
  }
  }
```

```
import java.io.FileInputStream;                //Character stream class
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBuffer{
    public static void main(String[] args) throws IOException {
 try (BufferedReader in = new BufferedReader(new FileReader("Source.txt"));
  BufferedWriter out = new BufferedWriter(new FileWriter("Dest.txt")))

  {
   String s;
   while ((s = in.readLine()) != null) {

   out.write(l);
     }
  catch(IOException e) { }
 }
 }
```

BufferedReader and BufferedWriter perform buffered I/O, instead of character-by-character.
BufferedReader provides a new method readLine(), which reads a line and returns a String (without the line delimiter).

Lines could be delimited by "\n"

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBuffer{
    public static void main(String[] args) throws IOException {
  try (BufferedReader in = new BufferedReader(new FileReader("Source.txt"));
   PrintWriter out = new PrintWriter(new FileWriter("Dest.txt")))

   {
    String s;
    while ((s = in.readLine()) != null) {

    out.println(l);
     }
   catch(IOException e) { }
  }
  }
```

In Java, the flush() method is typically used with output streams, such as FileOutputStream or BufferedWriter, but not with FileReader. Here's a detailed explanation:

flush() in Output Streams
The flush() method is used to force any buffered output to be written to the underlying stream (or file). It ensures that all data currently stored in the buffer is physically written to the file or output destination.

1. writer.flush():
The flush() method forces any data in the buffer (that hasn't yet been written) to be sent to the underlying file or stream. It ensures that all buffered data is physically written out.
If you call flush() without closing the writer, the buffer is cleared, and the data is written, but the stream remains open for further writing.

2. writer.close():
The close() method does two things:
It flushes any remaining data in the buffer to the file or output stream.
It releases system resources associated with the writer, such as the file handle. Once the stream is closed, you cannot write to it anymore.

Calling close() implicitly calls flush() as part of its operation, so if you call close(), any buffered data will be written out, even if you don't call flush() beforehand.

autoflush : PrintWriter object flushes the buffer on every invocation of println or format.
To flush a stream manually, invoke its flush method. The flush method is valid on any output stream.

```
import java.io.*;

class read{
 public static void main(String[] args) throws Exception{
// Read text file with specified unicode.
   FileInputStream fr=new FileInputStream("F:/java/eee.txt");
   InputStreamReader isr=new InputStreamReader(fr,"UTF-8");
   BufferedReader br=new BufferedReader(isr);
    String s;
   while ((s = in.readLine()) != null) {
         System.out.println(s);
}
}
}
```

```
public InputStreamReader(InputStream in) // Use default Unicode/charset
public InputStreamReader(InputStream in, String charsetName)throws UnsupportedEncodingException
public InputStreamReader(InputStream in, Charset cs)
```

InputStreamReader/OutputStreamWriter wraps in BufferedReader/BufferedWriter to read/write in multiple bytes.