Unit 8:

I/O and Streams

Java.io Package in Java :

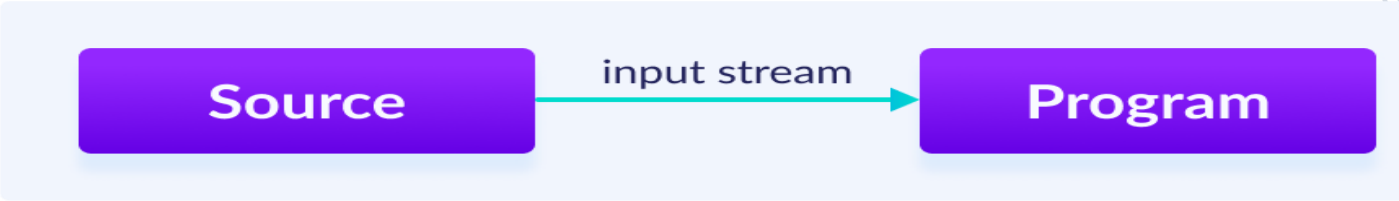
❖ This package provides for system input and output through data streams, serialization and the file system.

Streams:

❖ A stream can be defined as a sequence of data. There are two kinds of Streams :

1. **InPutStream** – The **InputStream** is used to read data from a source.
2. **OutPutStream** – The **OutputStream** is used for writing data to a destination.

Reading data from source



Writing data to destination

Types of Streams

It can be classified into:

Byte Stream :

- Byte stream is used to read and write a single byte (8 bits) of data.
- All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

Character Stream :

- Character stream is used to read and write a single character of data.
- All the character stream classes are derived from base abstract classes Reader and Writer.

Predefined Streams :

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) **System.out**: standard output stream
2) **System.in**: standard input stream
3) **System.err**: standard error stream

Byte Stream classes:

- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

- You can write byte-oriented as well as character-oriented data through FileOutputStream class.

Note : *for character-oriented data, it is preferred to use* ***FileWriter*** *than FileOutputStream****.***

- In Java, we can copy the contents of one file to another file. This can be done by the **FileInputStream** and **FileOutputStream** classes.

Character Stream classes:

- Though there are many classes related to character streams, but the most frequently used classes are, **FileReader** and **FileWriter**.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Java Files and Directories :

- The File class from the java.io package, allows us to work with files.
- The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.
- The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

- To use the File class, create an object of the class, and specify the filename or directory name:

```
File myObj = new File("filename.txt"); // Specify the filename
```

- A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

The File class has many useful methods for getting information about files. Below are some:

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

Reading Console Input :

- In Java, console input is accomplished by reading from **System.in**. To obtain a character- based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.

- Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast.

- InputStreamReader class can be used to read data from keyboard.

It performs two tasks:
1. connects to input stream of keyboard
2. converts the byte-oriented stream into character-oriented stream

Syntax:

```
InputStreamReader inputStreamReader = new InputStreamReader(System.in);
BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
```

Writing Console Output :

- In java, there are two methods to write console output. Using the 2 following methods, we can write output data to the console.

1. Using print() and println() methods
2. Using write() method

1. The PrintWriter Class :

- **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes it easier to internationalize your program.

Syntax:

PrintWriter(OutputStream ***outputStream***, boolean ***flushOnNewline***)

Eg.

```
PrintWriter printWriter = new PrintWriter(System.out, true);
printWriter.println("This is Texas College");
```

2. Using write() method :

- The write() method take integer as argument, and writes its ASCII equalent character on to the console.

Syntax:

write(int b); eg. System.out.write(26);

Difference between Scanner and BufferReader Class in Java

Sr. No.	Key	Scanner Class	BufferReader Class
1	Synchronous	Scanner is not syncronous in nature and should be used only in single threaded case.	BufferReader is syncronous in nature. During multithreading environment, BufferReader should be used.
2	Buffer Memory	Scanner has little buffer of 1 KB char buffer.	BufferReader has large buffer of 8KB byte Buffer as compared to Scanner.
3	Processing Speed	Scanner is bit slower as it need to parse data as well.	BufferReader is faster than Scanner as it only reads a character stream.
4	Methods	Scanner has methods like nextInt(), nextShort() etc.	BufferReader has methods like parseInt(), parseShort() etc.
5	Read Line	Scanner has method nextLine() to read a line.	BufferReader has method readLine() to read a line.

Reading and Writing Files :

- Java provides a number of classes and methods that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.

FileInputStream(String *fileName*) throws **FileNotFoundException** :

- The FileInputStream class of the java.io package can be used to read data (in bytes) from files.
- It extends the InputStream abstract class.

Syntax:

```
FileInputStream fileInputStream = new FileInputStream(String fileNamePath);
```


FileOutputStream(String *fileName*) throws **FileNotFoundException**

- The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files.
- It extends the OutputStream abstract class.

Syntax:

```
FileOutputStream output = new FileOutputStream(String path, boolean value);
```

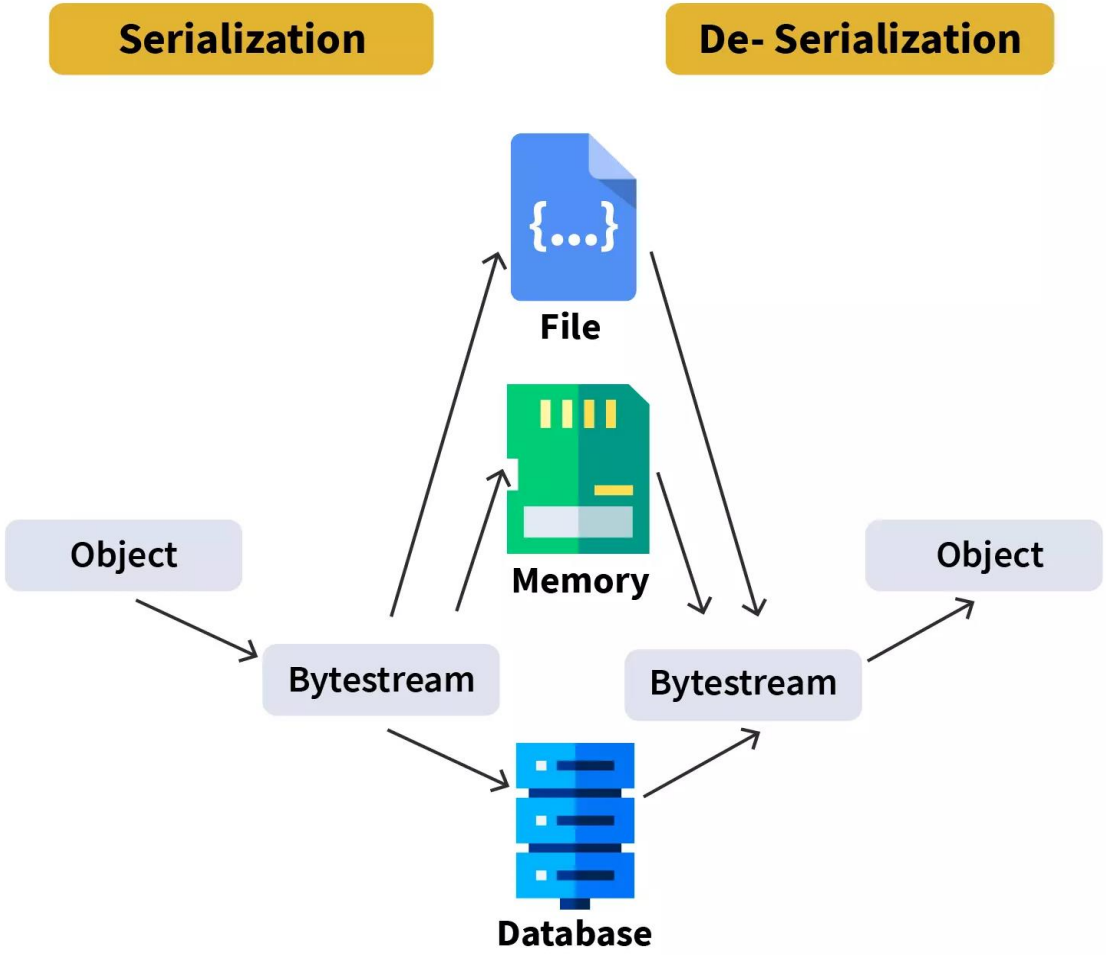
Serialization Interface:

- The Serializable interface is present in java.io package.
- Serialization is a mechanism of converting the state of an object into a byte stream.
- Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory.

Why Do We Need Serialization in Java?

- Serialization mechanism is usually used when there is a need to send your data (objects) over the network or to store in files. Now the hardware components like network infrastructure, hard disk etc understands only bytes and bits, not the java objects.

- So, Serialization is used in this case which translates Java object's state to byte-stream to send it over the network or save it in file.



How Do We Serialize An Object?

- Serialization in java can be implemented using java.io.Serializable interface.

- For serializing the object, we will be using the **writeObject**() method of **ObjectOutputStream** class.

```
FileOutputStream fileOutputStream = new FileOutputStream(”fileName");
ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
```

- For deserializing the object, we will be using the **readObject**() method of **ObjectInputStream** class.

```
FileInputStream fileInputStream = new FileInputStream(”fileName");
ObjectInputStream objectOutputStream = new ObjectInputStream(fileInputStream);
```


SerialVersionUID in Java

- The serialization process at runtime associates an id with each Serializable class which is known as SerialVersionUID. It is used to verify the sender and receiver of the serialized object. The sender and receiver must be the same. To verify it, SerialVersionUID is used. The sender and receiver must have the same SerialVersionUID, otherwise, InvalidClassException will be thrown when you deserialize the object.

- Syntax:

```
private static final long serialVersionUID = 1L;
```