Introduction to File Input and Output

We have seen in this class how objects communicate with each other.

Program often need to communicate with the outside world. The means of communication are input (such as a keyboard) and output (such as the computer screen).

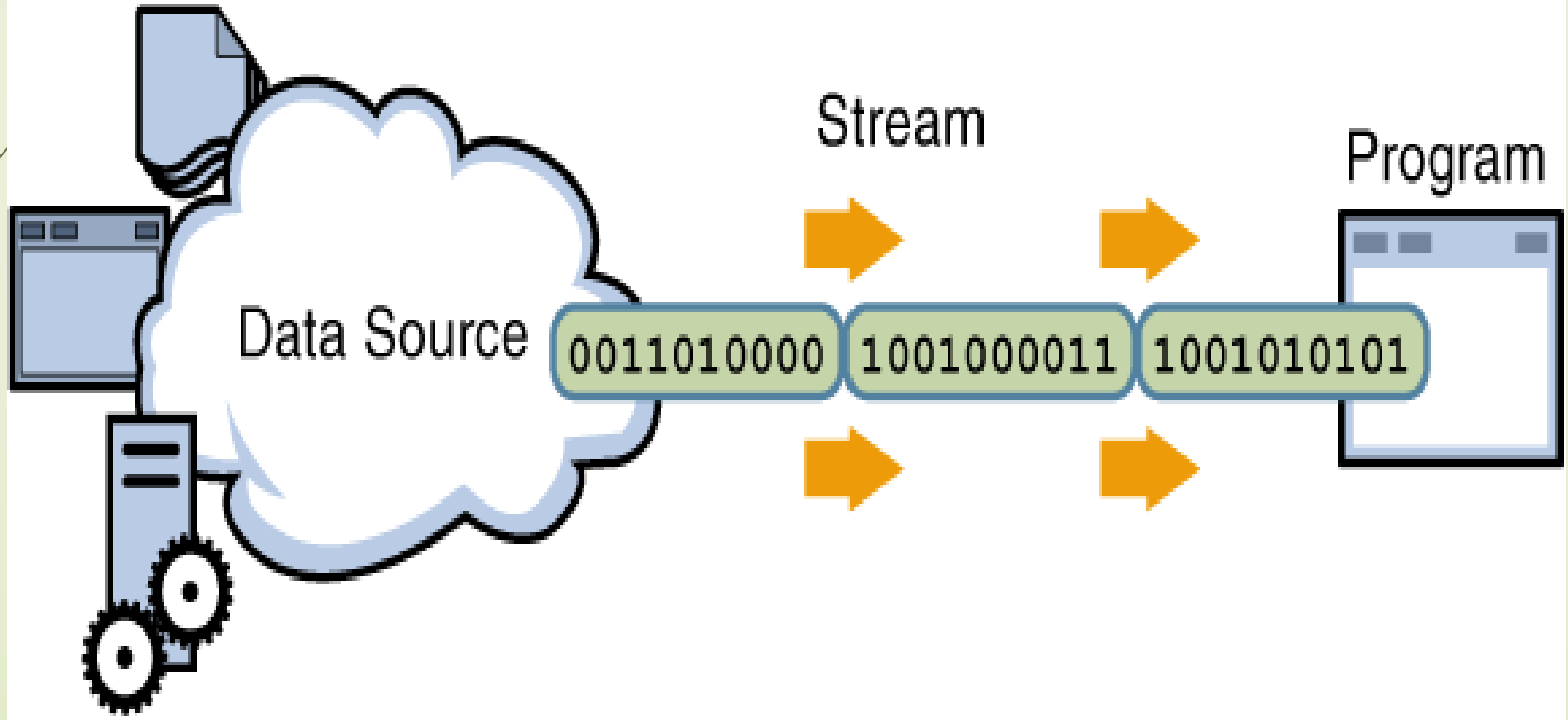
Programs can also communicate through stored data, such as files.

I/O Streams

- A **stream** is a communication channel that a program has with the outside world.
- It is used to transfer data items in succession.
- An **Input/Output (I/O)** Stream represents an input source or an output destination.
- A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.
- **Streams** support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- Some **streams** simply pass on data; others manipulate and transform the data in useful ways.
- No matter how they work internally, all streams present the same simple model to programs that use them.
- **A stream is a sequence of data.**

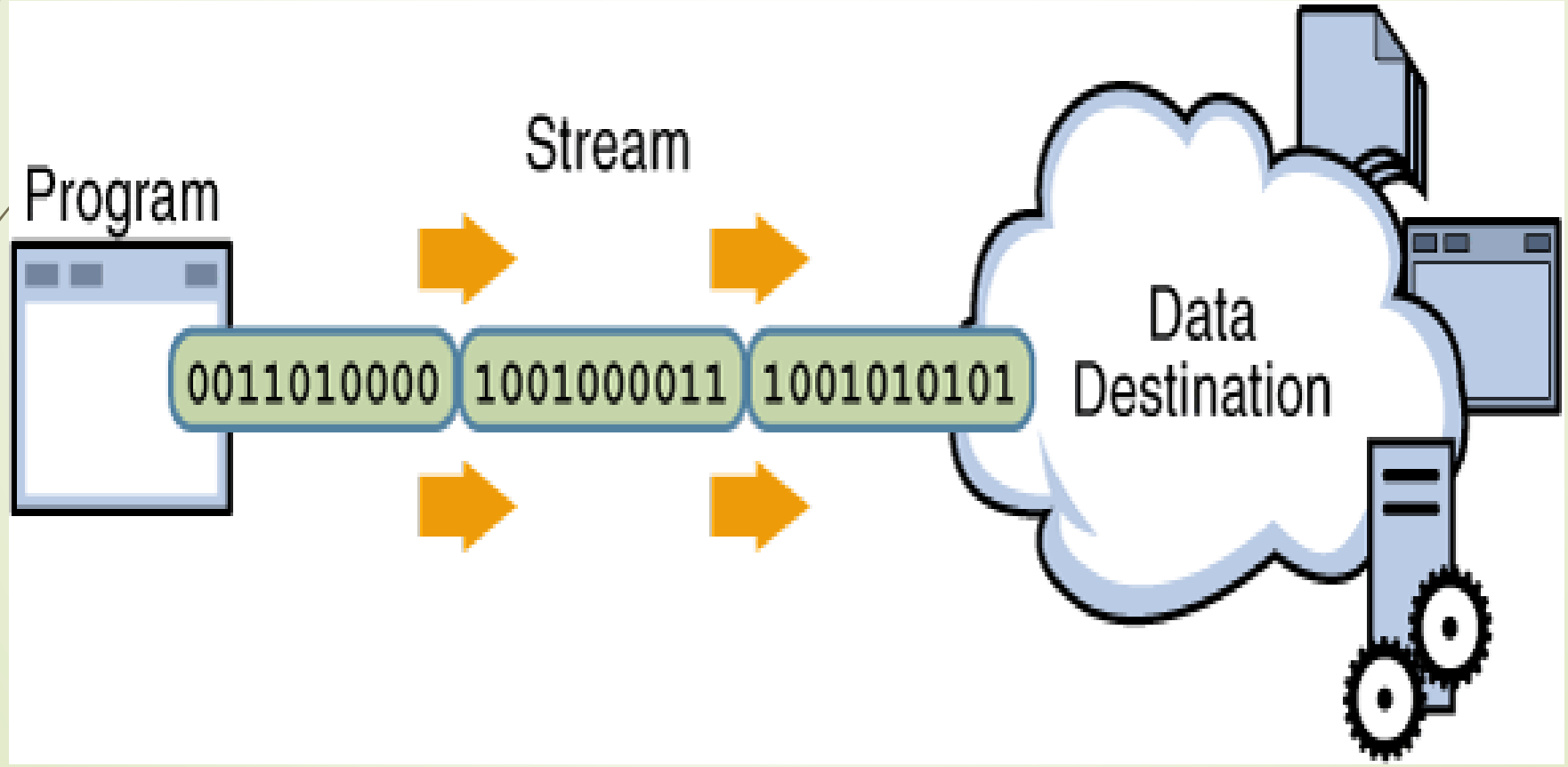
Reading information into a program.

A program uses an input stream to read data from a source, one item at a time.



Writing information from a program.

A program uses an output stream to write data to a destination, one item at time.



The Java IO API

- The **java.io** package contains many classes that your programs can use to read and write data.
- Most of the classes implement sequential access streams.
- The sequential access streams can be divided into two groups: those that read and write bytes and those that read and write Unicode characters.
- Each sequential access stream has a specialty, such as reading from or writing to a file, filtering data as its read or written, or serializing an object.

Types of Streams

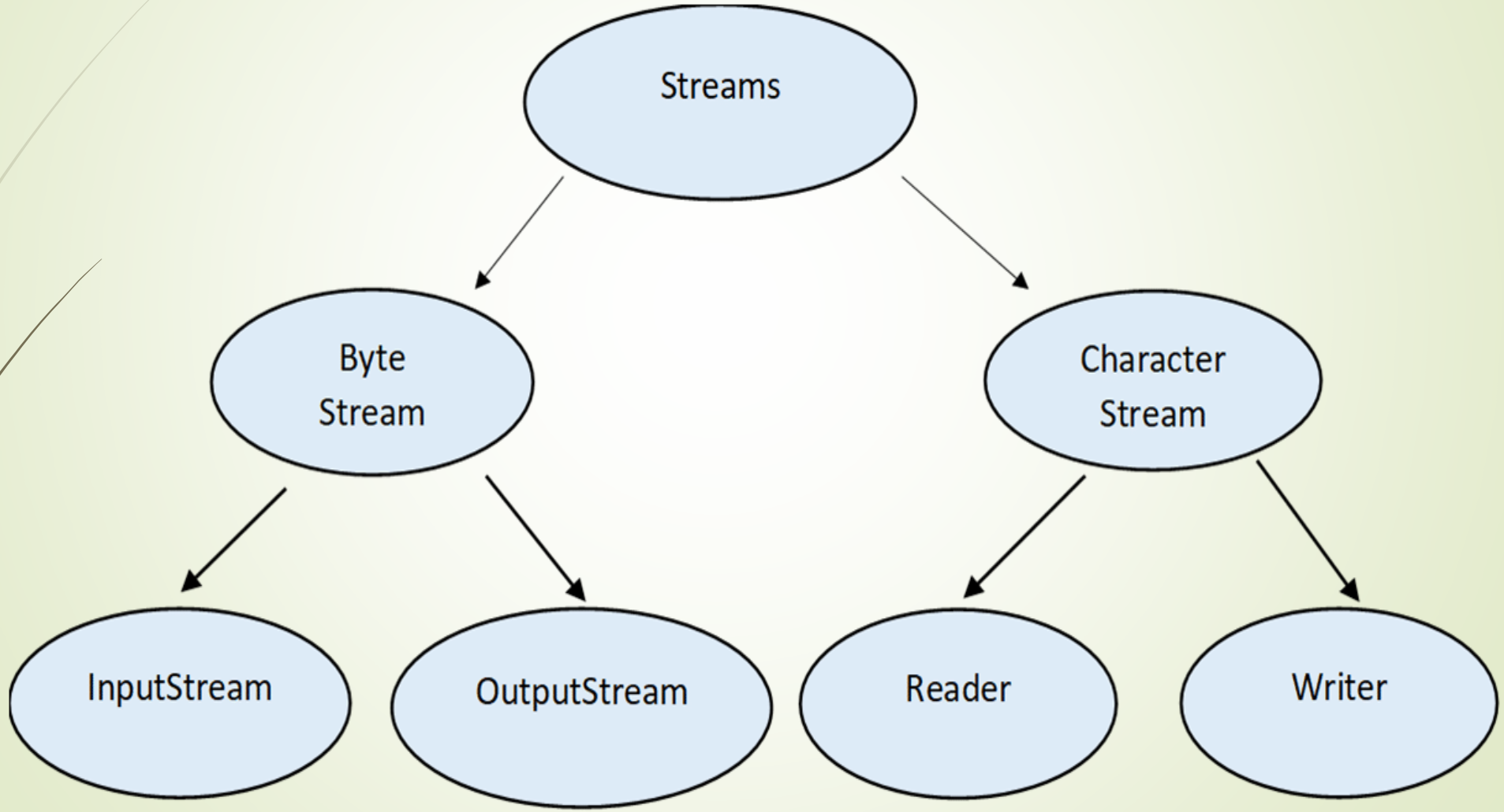
1. Byte Streams:

- Byte streams perform input and output of 8-bit bytes.
- They read and write data one byte at a time.
- Using byte streams is the lowest level of I/0, so if you are reading or writing character data the best approach is to use character streams.
- Other stream types are built on top of byte streams.

2. Character Streams:

- All character stream classes are descended from **Reader** and **Writer.**
- **Reader** and **Writer** classes that perform file **I/0**, **FileReader** and **FileWriter**.

Types of Streams

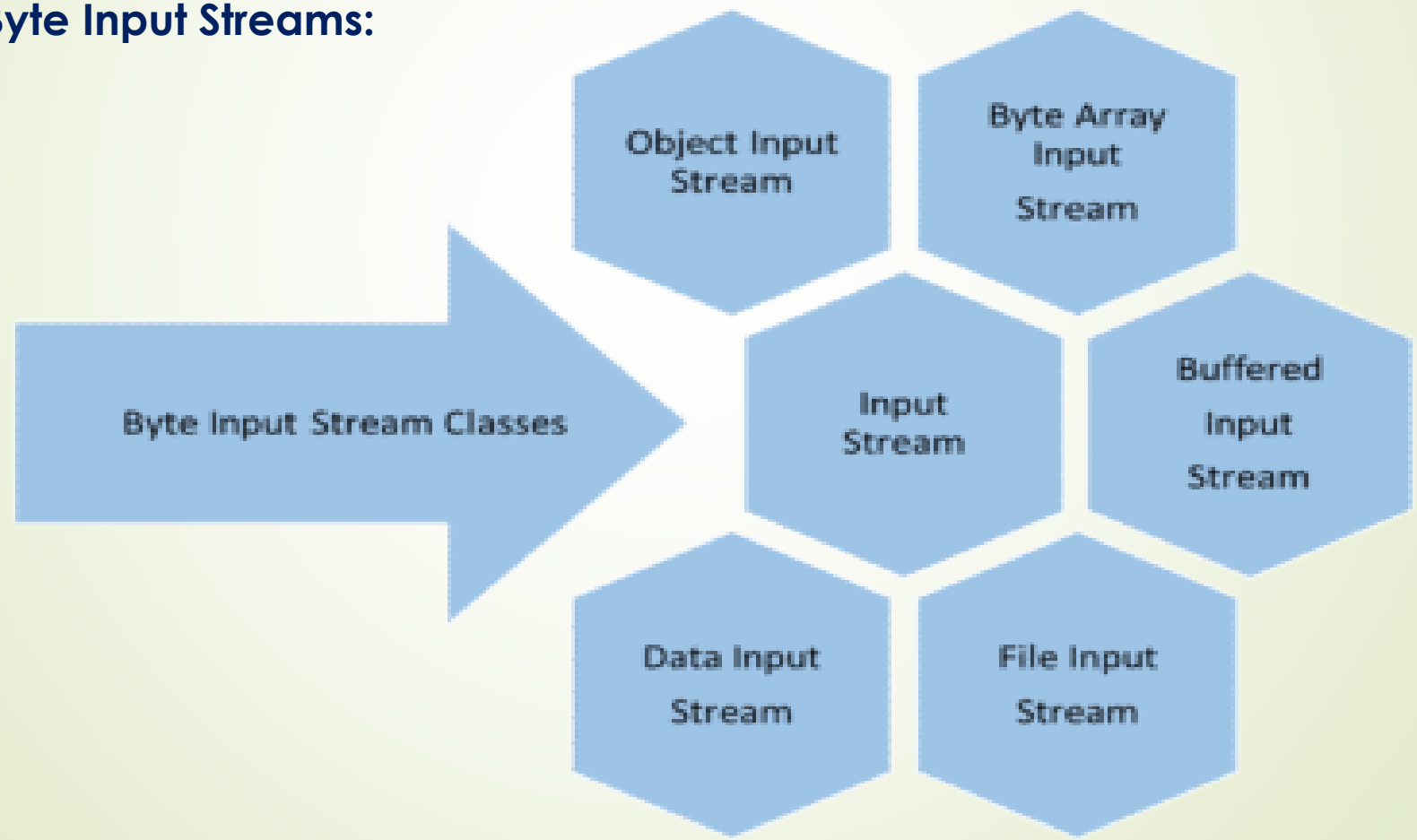


Byte Input Stream:

These are used to read byte data from various input devices.

InputStream is an abstract class and it is the super class of all the input byte streams.

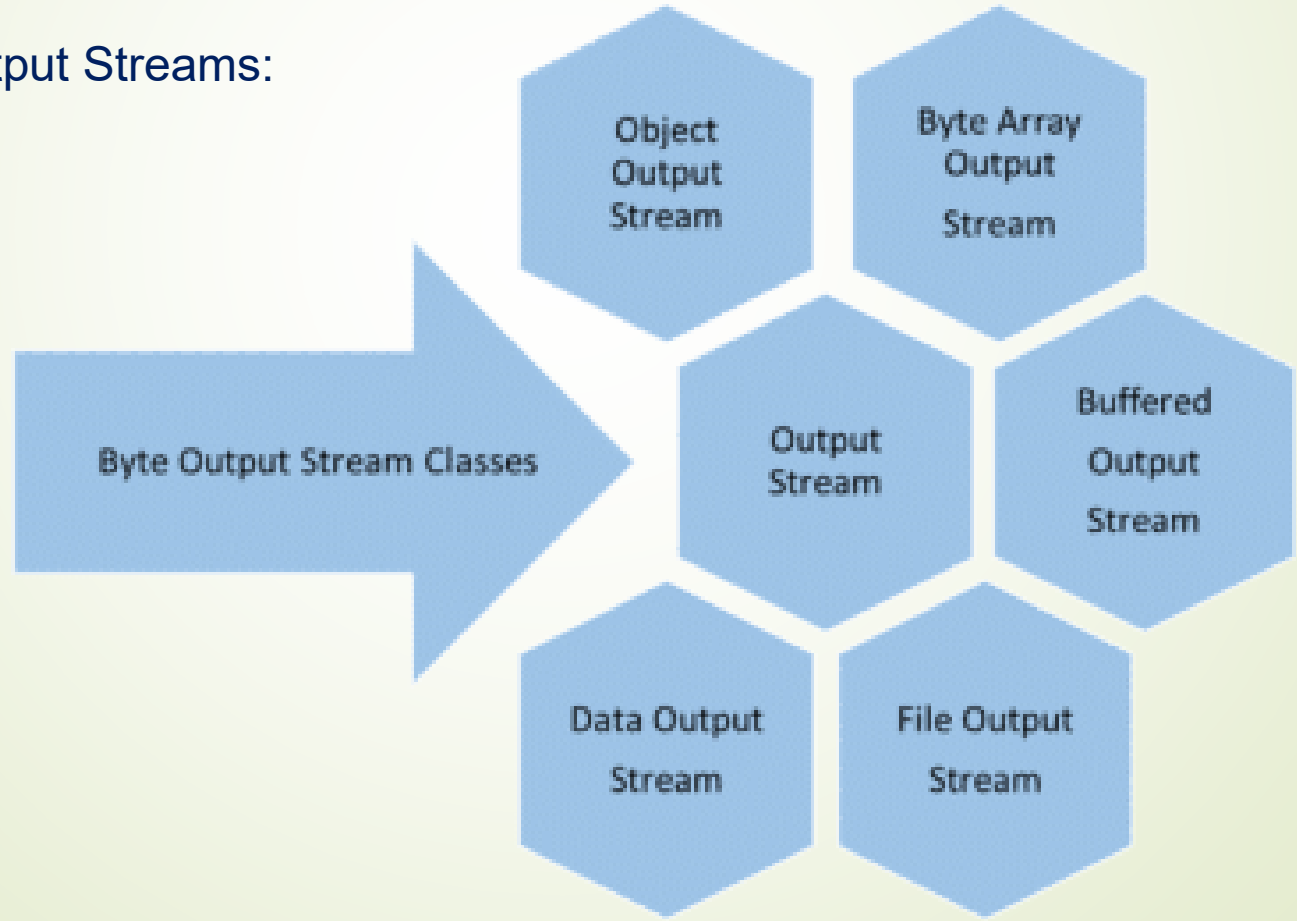
List of Byte Input Streams:



Byte Output Stream

- ✓ These are used to write byte data to various output devices.
- ✓ Output Stream is an abstract class and it is the superclass for all the output byte streams.

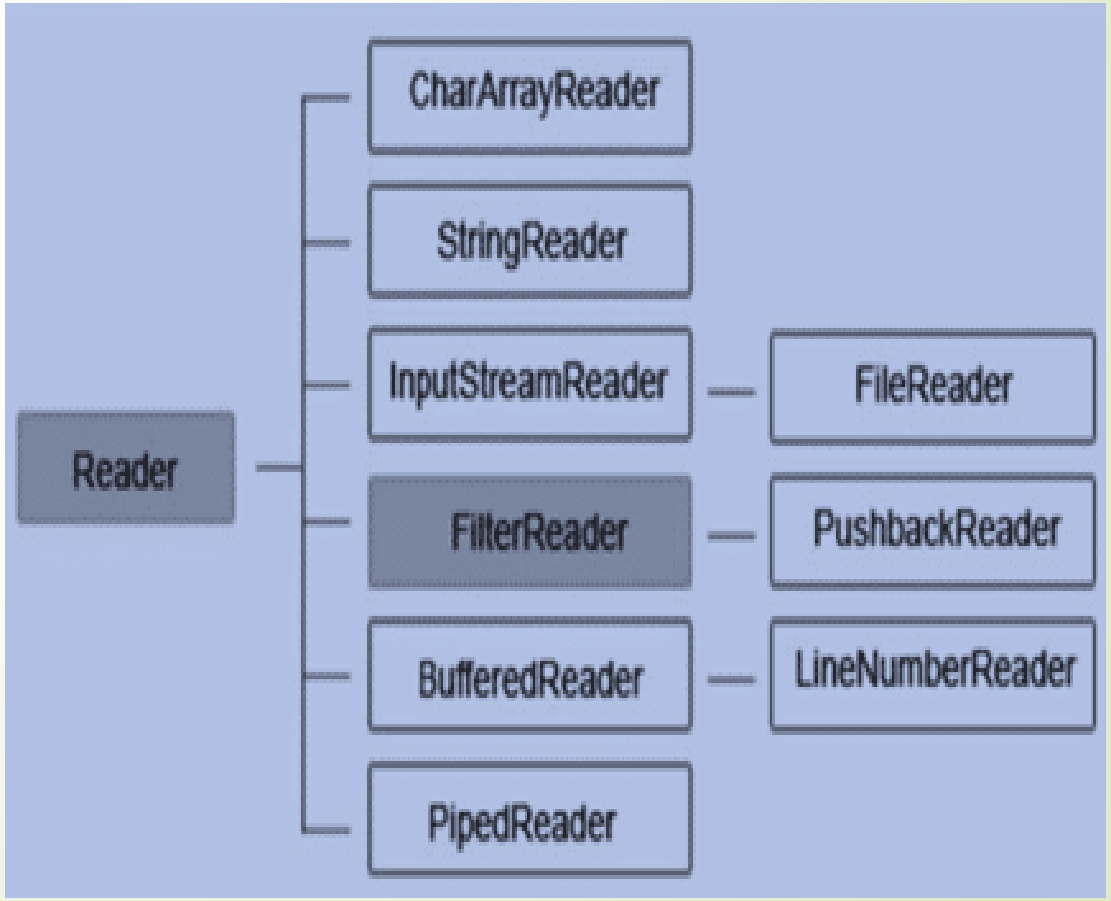
List of Byte Output Streams:



Character Input Stream:

- These are used to read char data from various input devices.
- Reader is an abstract class and is the super class for all the character input streams.

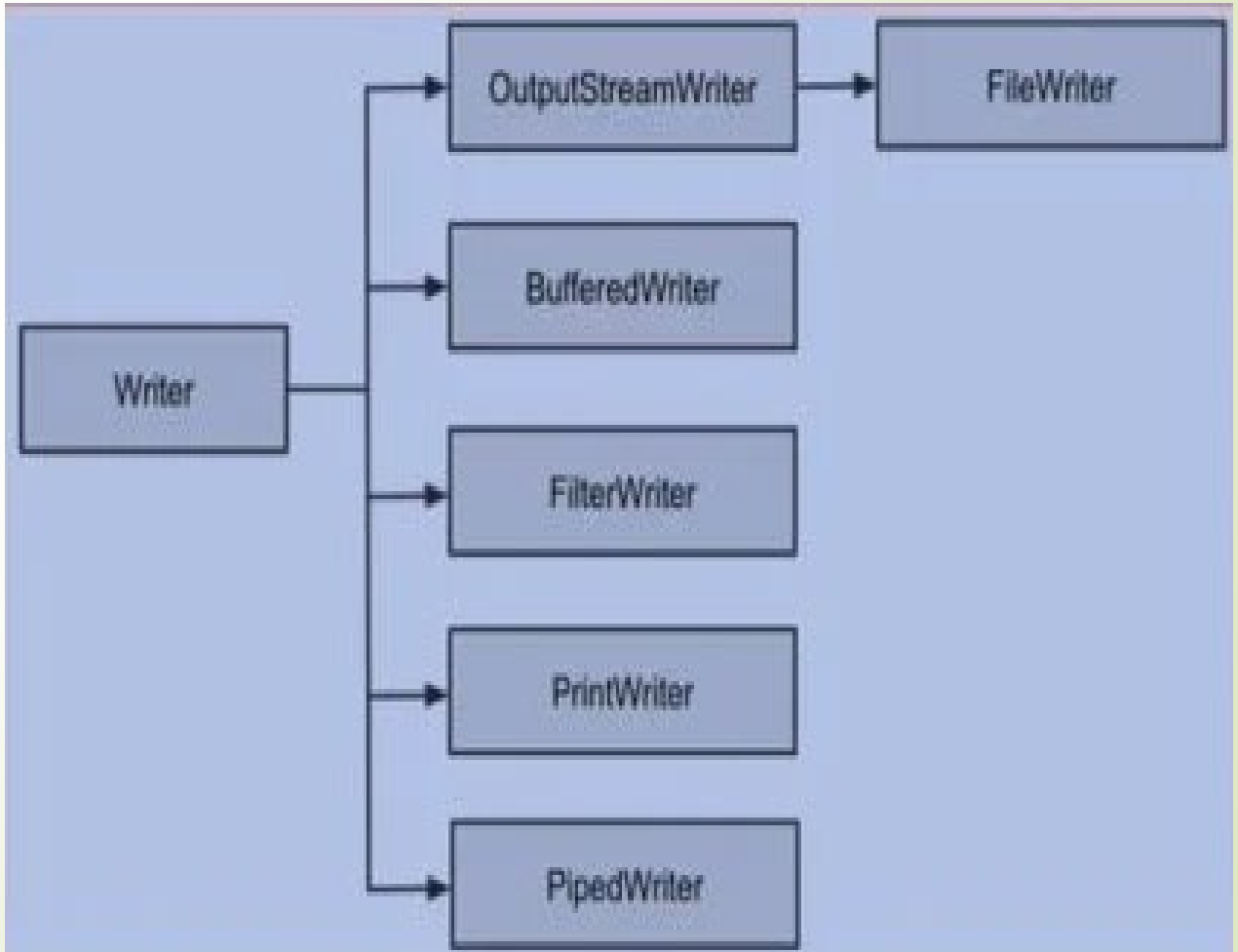
List of Character Input Streams:



Character Output Stream

- ✓ These are used to write char data to various output devices.
- ✓ Writer is an abstract class and is the super class of all the character output streams.

List of Character Output Stream:



Wait:

1. What does the InputStream class in Java represent?

   A. A sequence of characters.

   B. A sequence of data read from an input source.

   C. A sequence of bytes written to an output source.

   D. A sequence of graphical elements.

Java.IO Package

Agenda:

1. File
2. FileWriter
3. FileReader
4. BufferedWriter
5. BufferedReader
6. PrintWriter

File:

```
File f = new File("abc.txt");
```

- This line 1st checks whether **abc.txt** file is already available (or) not if it is already available then "f" simply refers that file.

- If it is not already available then it won't create any physical file just creates a java **File object** represents name of the file.

Example:

```
import java.io.*;
class  FileDemo {
      public static void main(String[ ] args) throws IOException      {
            File  f = new  File("cricket.txt");
            System.out.println(f.exists());      //false
            f.createNewFile();
            System.out.println(f.exists());      //true
            }
}
```

output : 1st run : false true	output : 2nd run : true true

A java File object can represent a directory also.

```
// Example:
import java.io.*;
class  FileDemo
{
      public static void main(String[ ] args) throws IOException
      {
            File f=new File("cricket123");
            System.out.println(f.exists());      //false
            f.mkdir();
            System.out.println(f.exists());      //true
      }
}
```

File class constructors:

Note: In UNIX everything is a file, java "file IO" is based on UNIX operating system hence in java also we can represent both files and directories by File object only.

1. File f=new File(String name);

Creates a java File object that represents name of the file or directory in current working directory.

2. File f=new File(String subdirname, String name);

Creates a File object that represents name of the file or directory present in specified sub directory.

3. File f=new File(File subdir, String name);

Ways to Create a File in Java

In Java, there are three ways to create a file. They are as follows:

1. **Using File.createNewFile() method**
2. **Using FileOutputStream class**
3. **By using File.createFile() method**

Question

1. Write code to create a file named with **demo.txt** in current working directory.

2. Write code to create a directory named with **CdacDelhi** in current working directory and create a file named with **abc.txt** in that directory.

3. Write code to create a file named with **demo.txt** present in **D:\PG-DAC** folder.

Solution of Question No. 1

```
import  java.io.*;
class  FileDemo
{
    public static void main(String[ ]  args) throws IOException
    {
        File f = new  File("demo.txt");
        f.createNewFile();
    }
}
```

Solution of Question No. 2

```
import  java.io.*;
class  FileDemo
{
    public static void main(String[ ]  args)  throws  IOException
    {
        File  f1 = new File("CdacDelhi");
        f1.mkdir();
        File  f2 = new  File("CdacDelhi", "abc.txt");
        f2.createNewFile();
    }
}
```

Solution of Question No. 3

```
import  java.io.*;
class  FileDemo
{
    public static void main(String[ ]  args)  throws  IOException
    {
        File  f = new File("D:\\PG-DAC", "demo.txt");
        f.createNewFile();
    }
}
```

Important Methods of File Class:

1.) boolean exists();

Returns **true** if the physical file or directory available.

2.) boolean createNewFile();

This method 1st checks whether the physical file is already available or not if it is already available then this method simply returns **false** without creating any physical file.

If this file is not already available then it will create a new file and returns **true**

3.) boolean mkdir();

This method 1st checks whether the directory is already available or not if it is already available then this method simply returns **false** without creating any directory.

If this directory is not already available then it will create a new directory and returns **true**

Important Methods of File Class:

4.) boolean isFile();

Returns true if the File object represents a physical file.

5.) boolean isDirectory();

Returns true if the File object represents a directory.

6.) String[] list();

It returns the names of all files and subdirectories present in the specified directory.

7.) long length();

Returns the no of characters present in the file.

8.) boolean delete();

To delete a file or directory.

Questions:

1. Write a program to display the names of all files and directories present in **D:\\Delhi**.

2. Write a program to display only **file** names.

3. Write a program to display only **directory** names

FileWriter:

By using **FileWriter** object we can write character data to the file.

Constructors:

FileWriter **fw** = new FileWriter(String **name**);

FileWriter **fw** = new FileWriter(File **f**);

The above 2 constructors meant for **overriding**.

FileWriter:

Instead of overriding if we want append operation then we should go for the following 2 constructors.

FileWriter **fw** = new FileWriter(String **name**, boolean **append**);

FileWriter **fw** = new FileWriter(File **f**, boolean **append**);

If the specified physical file is not already available then these constructors will create that file.

Methods of FileWriter Class

1.) write(int ch);

To write a single character to the file.

2.) write(char[ ] ch);

To write an array of characters to the file.

3.) write(String s);

To write a String to the file.

4.) flush();

To give the guarantee the total data include last character also written to the file.

5.) close();

To close the stream.

Methods of FileWriter Class

Note :

1. The main problem with **FileWriter** is we have to insert line separator manually, which is difficult to the programmer. **('\n')**

2. And even line separator varing from system to system.

FileReader Class

By using **FileReader** object we can read **character data** from the file.

Constructors:

FileReader **fr** = new FileReader(String **name**);

FileReader **fr** = new FileReader (File **f**);

Methods of FileReader Class

1.) int read();

It attempts to read next character from the file and return its Unicode value. If the next character is not available then we will get -1.

int **I** = **fr**.read();

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

As this method returns **unicode** value , while printing we have to perform type casting.

2.) int read(char[ ] ch);

It attempts to read enough characters from the file into char[ ] array and returns the no of characters copied from the file into char[ ] array.

File **f** = new File("**abc.txt**");

Char[ ] **ch** = new Char[(int)**f**.length()];

3. void Close()

Usage of FileWriter and FileReader is not recommended because :

1. While writing data by **FileWriter** compulsory we should insert line separator**(\n)** manually which is a bigger **headache** to the programmer.

2. While reading data by **FileReader** we have to read character by character instead of line by line which is not convenient to the programmer.

3. To overcome these limitations we should go for **BufferedWriter** and **BufferedReader** concepts.

BufferedWriter Class

By using **BufferedWriter** object we can write **character** data to the file.

Constructors:

BufferedWriter **bw** = new BufferedWriter(writer w**);**

BufferedWriter **bw** = new BufferedWriter(writer **w**, int **buffersize**);

Note: BufferedWriter never communicates directly with the file it should communicates via some **writer** object.

Which of the following declarations are valid?

1. BufferedWriter **bw** = new BufferedWriter("**cricket.txt**"); // (invalid)
2. BufferedWriter **bw** = new BufferedWriter (new File("**cricket.txt**")); // (invalid)
3. BufferedWriter **bw** = new BufferedWriter (new FileWriter("**cricket.txt**")); // (valid)

Methods of BufferedWriter Class

1. **write(int ch);**
2. **write(char[ ] ch);**
3. **write(String s);**
4. **flush();**
5. **close();**
6. **newline();**

   Inserting a new line character to the file.

Note : When ever we are closing **BufferedWriter** automatically underlying writer will be closed and we are not close **explicitly**.

Question:

When compared with **FileWriter** which of the following capability(facility) is available as method in **BufferedWriter**.

1. Writing data to the file.
2. Closing the writer.
3. Flush the writer.
4. Inserting newline character.

Ans : 4

BufferedReader Class

This is the most enhanced(better) Reader to read character data from the file.

Constructors:

BufferedReader **br** = new BufferedReader(Reader **r**);

BufferedReader **br** = new BufferedReader(Reader **r**, int **buffersize**);

Note: BufferedReader can not communicate directly with the **File** it should communicate via some **Reader object.**

The main advantage of **BufferedReader** over **FileReader** is we can read data **line by line** instead of **character by character.**

Methods BufferedReader Class

1. int read();
2. int read(char[ ] ch);
3. String readLine();

   It attempts to read next line and return it , from the File. if the next line is not available then this method returns null.

4. void close();

Wait:

1. What is the purpose of the **BufferedInputStream** class in Java?

A. To read data from an input source with buffering.

B. To write data to an output source with buffering.

C. To handle errors in the input stream.

D. To manage memory allocation for input streams.

PrintWriter Class

- This is the most enhanced Writer to write text data to the file.
- By using **FileWriter** and **BufferedWriter** we can write only character data to the File but by using **PrintWriter** we can write any type of data to the File.

Constructors:

1. PrintWriter **pw** = new PrintWriter(String **name**);
2. PrintWriter **pw** = new PrintWriter(File **f**);
3. PrintWriter **pw** = new PrintWriter(Writer **w**);

PrintWriter can communicate either directly to the **File** or via some **Writer** object also.

Methods of PrintWriter Class

1. write(int ch);
2. write (char[ ] ch);
3. write(String s);
4. flush();
5. close();
6. print(char ch);
7. print (int i);
8. print (double d);
9. print (boolean b);

Methods of PrintWriter Class

10. print (String s);
11. println(char ch);
12. println (int i);
13. println(double d);
14. println(boolean b);
15. println(String s);

Note:

Ques: What is the difference between **write(100)** and **print(100)** ?

Ans: In the case of **write(100)** the corresponding character **"d"** will be added to the File but in the case of **print(100)** "100" value will be added directly to the File.

Note 1:

The most enhanced Reader to read character data from the File is **BufferedReader**.

The most enhanced **Writer** to write character data to the File is **PrintWriter**.

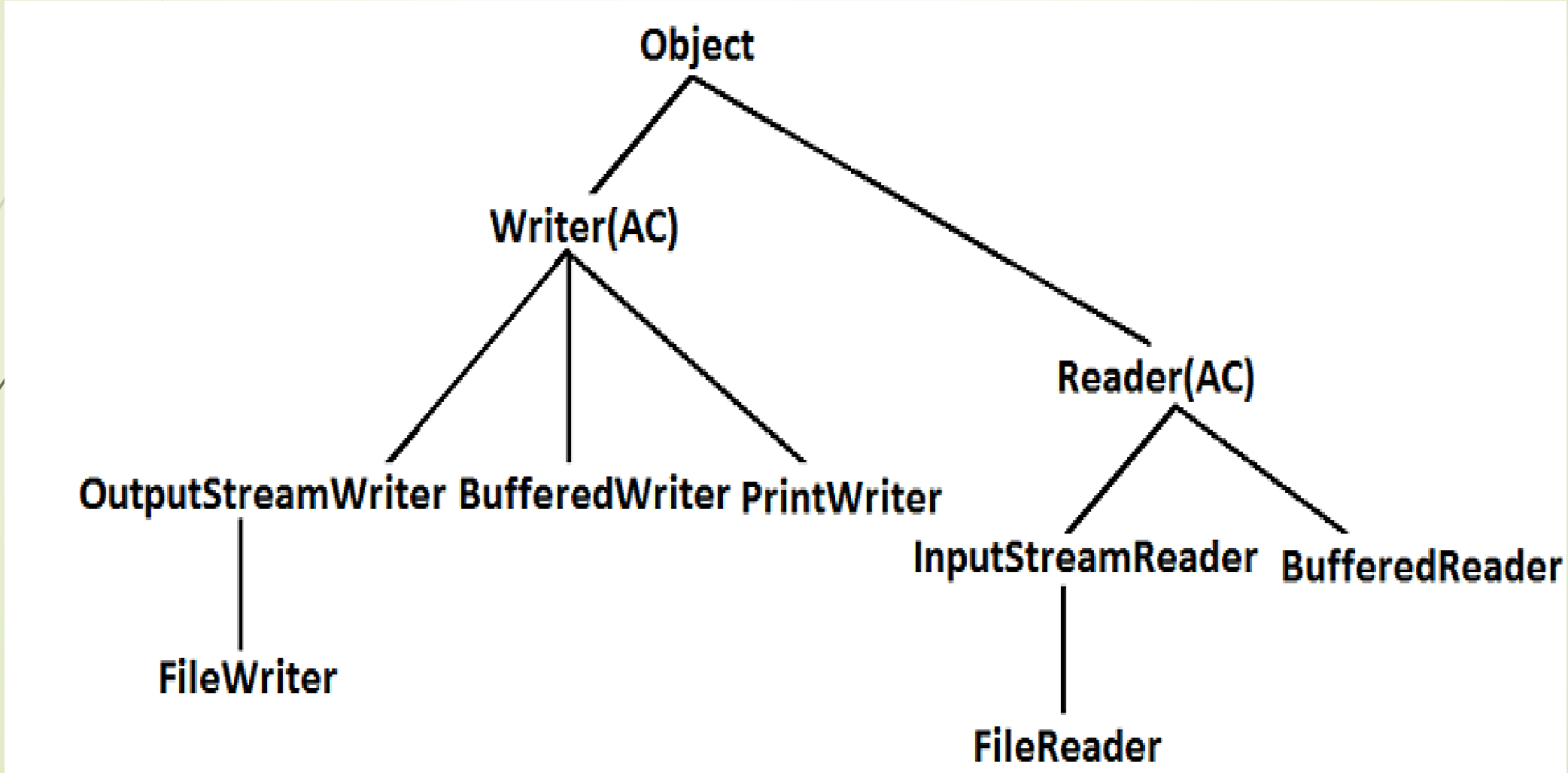
Note:

Note 2:

In general we can use **Readers** and **Writers** to handle character data. Where as we can use **InputStreams** and **OutputStreams** to handle binary data(like images, audio files, video files etc).

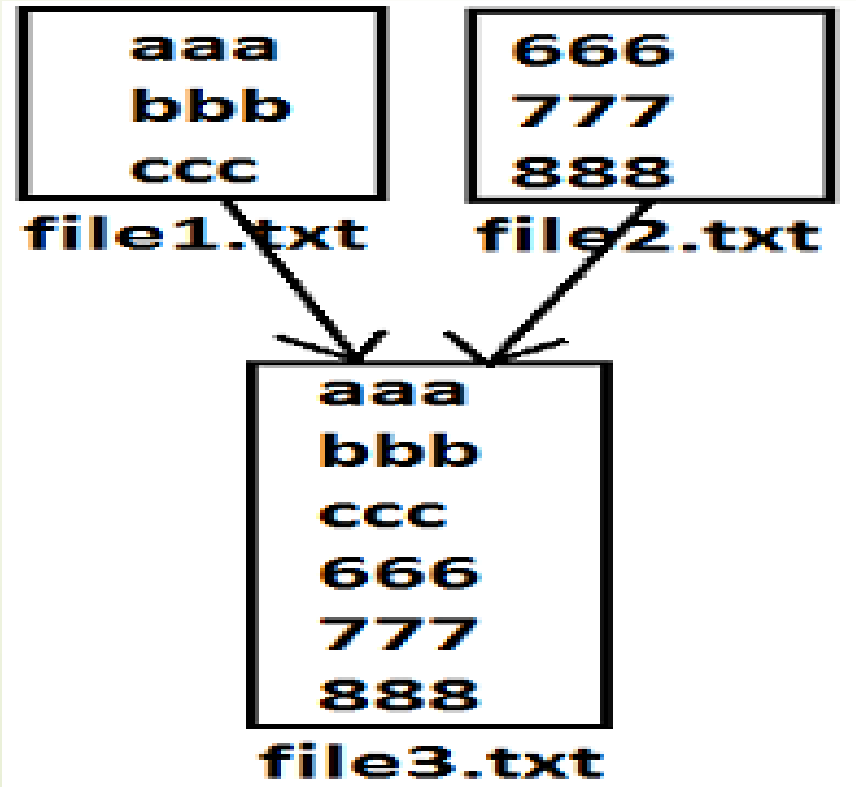
We can use **OutputStream** to write binary data to the File and we can use **InputStream** to read binary data from the File.

Diagram:


Object
Writer(AC)
Reader(AC)
OutputStreamWriter
BufferedWriter
PrintWriter
InputStreamReader
BufferedReader
FileWriter
FileReader

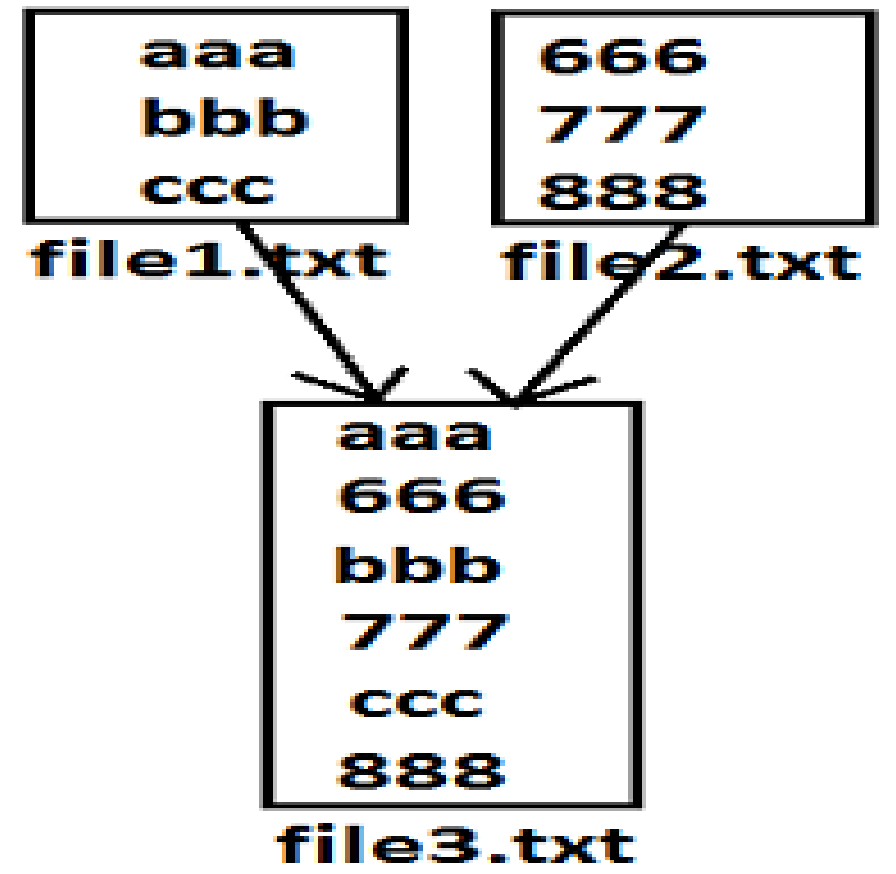
Practical:

Write a program to merge Two File contents into another file.



Practical:

Write a program to perform file merge operation where merging should be performed line by line alternatively.

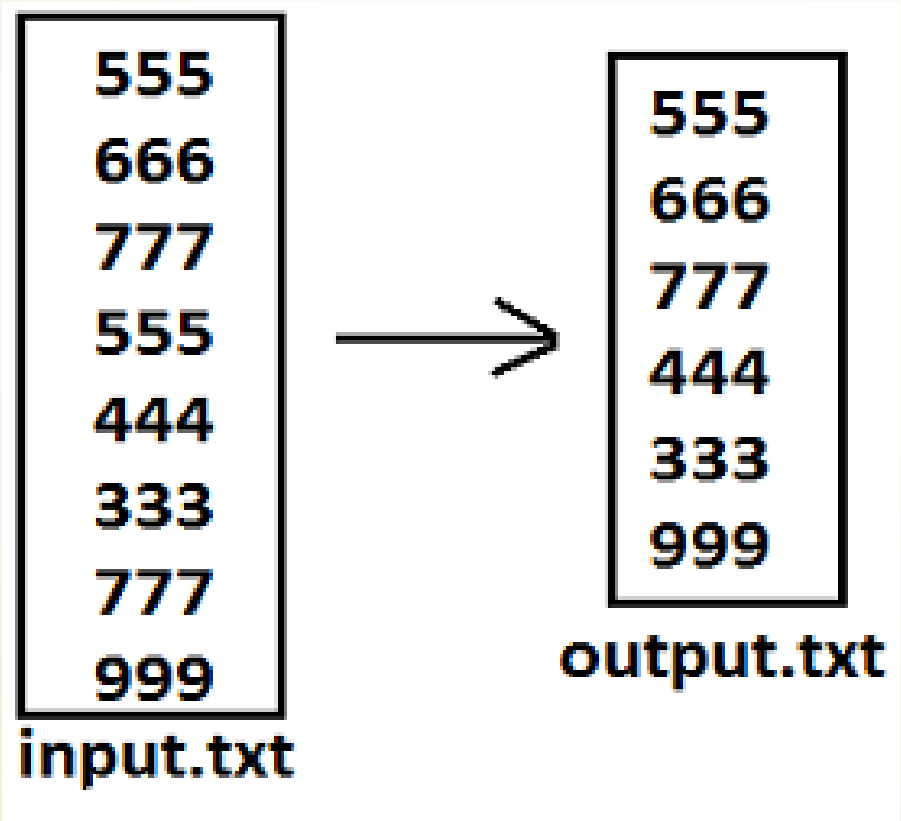


Practical:

Write a program to merge data from all files present in a folder into a new file.

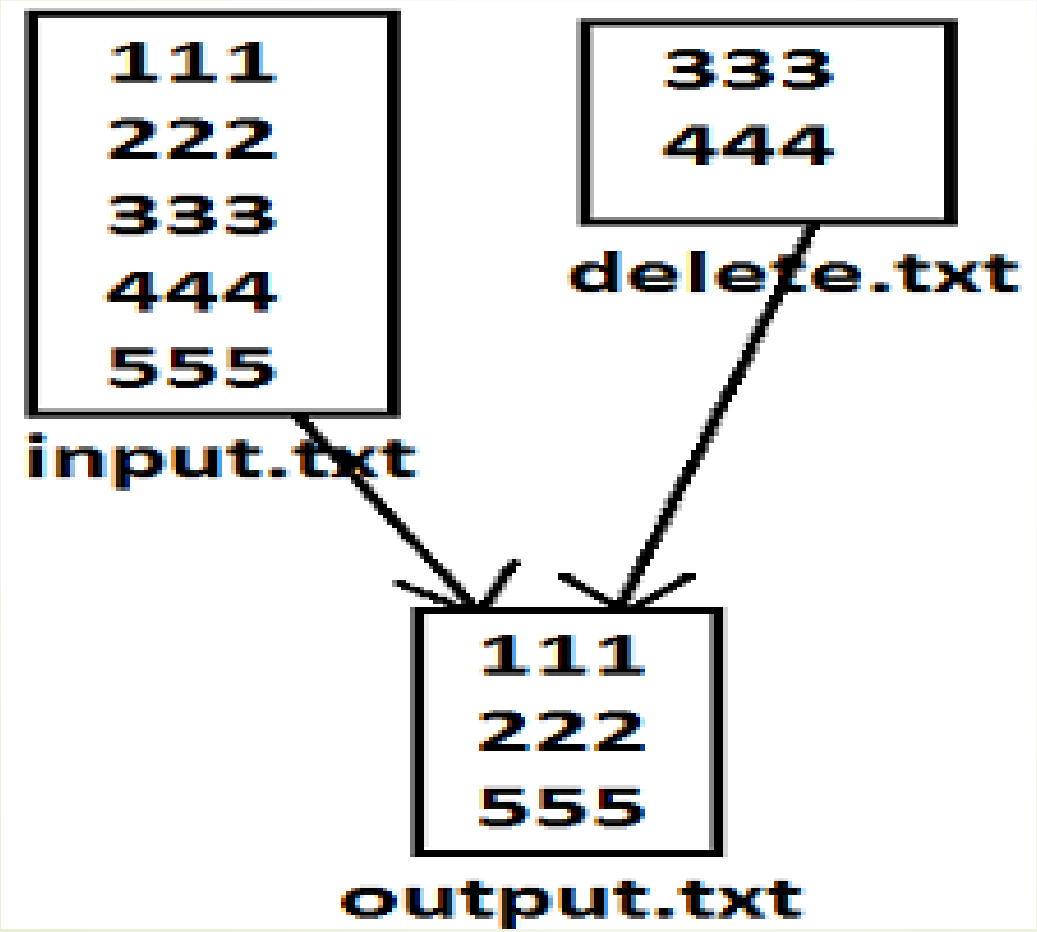
Practical:

Write a program to delete duplicate numbers from the file.



Practical:

Write a program to perform file extraction operation.



Serialization and Deserialization in Java

Serialization

What are Object Files?

The object file is a collection of different types of objects. To create the object files we have to take the help of streams like **ObjectOutputStream**, **ObjectInputStream**, etc. To write the object or read the object then the particular object must be **serialized**.

An object can be said to be **serialized** only when its class implements the **Serializable** interface. If we are reading or writing any object which is not serialized then JVM will throw a run time error or **Exception** saying **NotSerializableException**.

What is Serializable Interface in Java?

Serializable is an **interface** defined in the **java.io** package with **zero** methods.

If an interface is defined with zero methods(has no data member and methods) then it is called a **marker interface** or **tagged interface**.

The main job of a marked or tagged interface is providing instructions to JVM to perform a special task.

Example: **Cloneable**, **Serializable**, **EventListener**, etc.

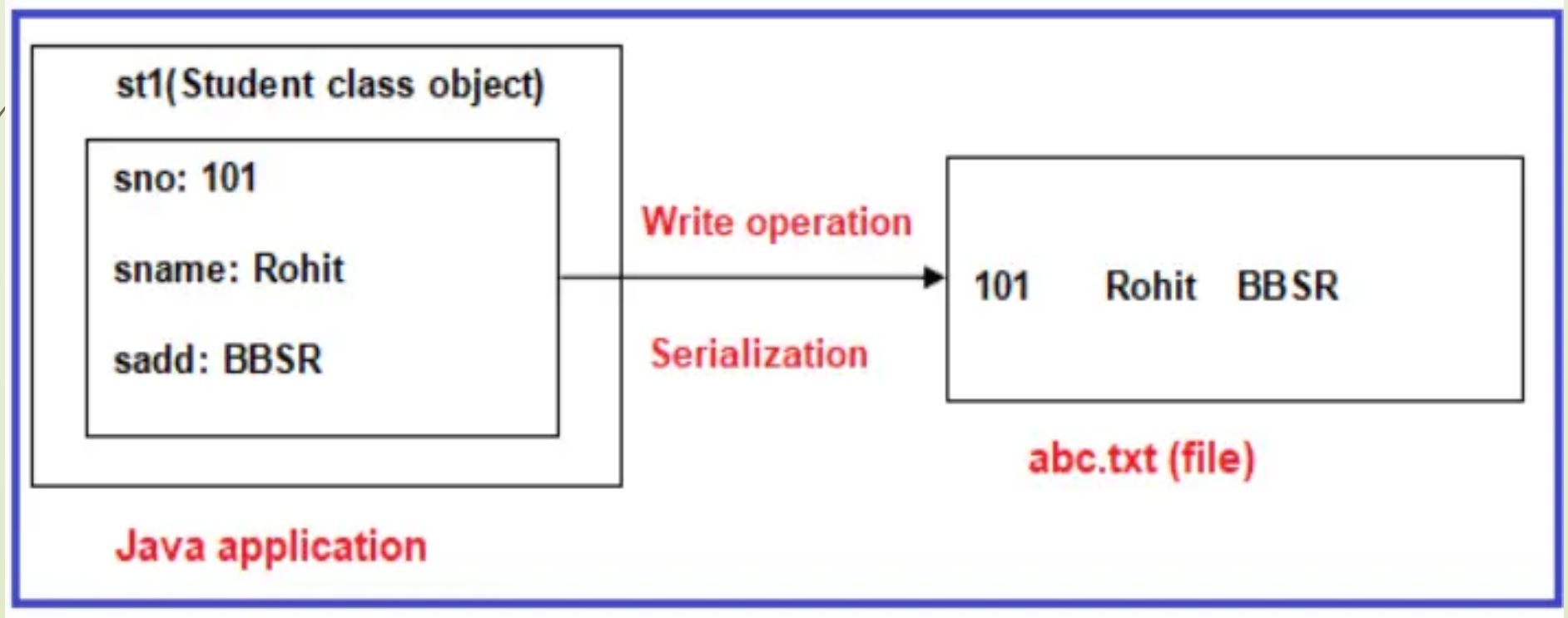
If we are declaring a class by implementing a serializable interface then we are giving an instruction to the JVM to allow us to read the object from files and write the object into object files.

Note: In small applications, the industry generally prefers **serialization** and **deserialization** persistence operations on the files.

What is Serialization in Java?

The process of capturing object data and writing that data into a file is called **serializable**. In other words, **Serialization** is a process of converting the object into a stream of bytes which is nothing but writing the object into an object file.

For better understanding please have a look at the following image.



What is Serialization in Java?

In serialization, the objects will not be written to a file but the data of that object will be written to a file. To perform this serialization we can use a class called **ObjectOutputStream**. It is mainly used in **Hibernate**, **RMI**, **JPA**, **EJB**, and **JMS(Java Message Service)** technologies.

Creation of **ObjectOutputStream:**

```
ObjectOutputStream oos = new ObjectOutputStream(OutputStream);
```

Advantages of Serialization

1. To **save/persist** the state of an object.
2. It is mainly used to travel the **object's state on the network** (which is known as **marshaling**).

What is Stream?

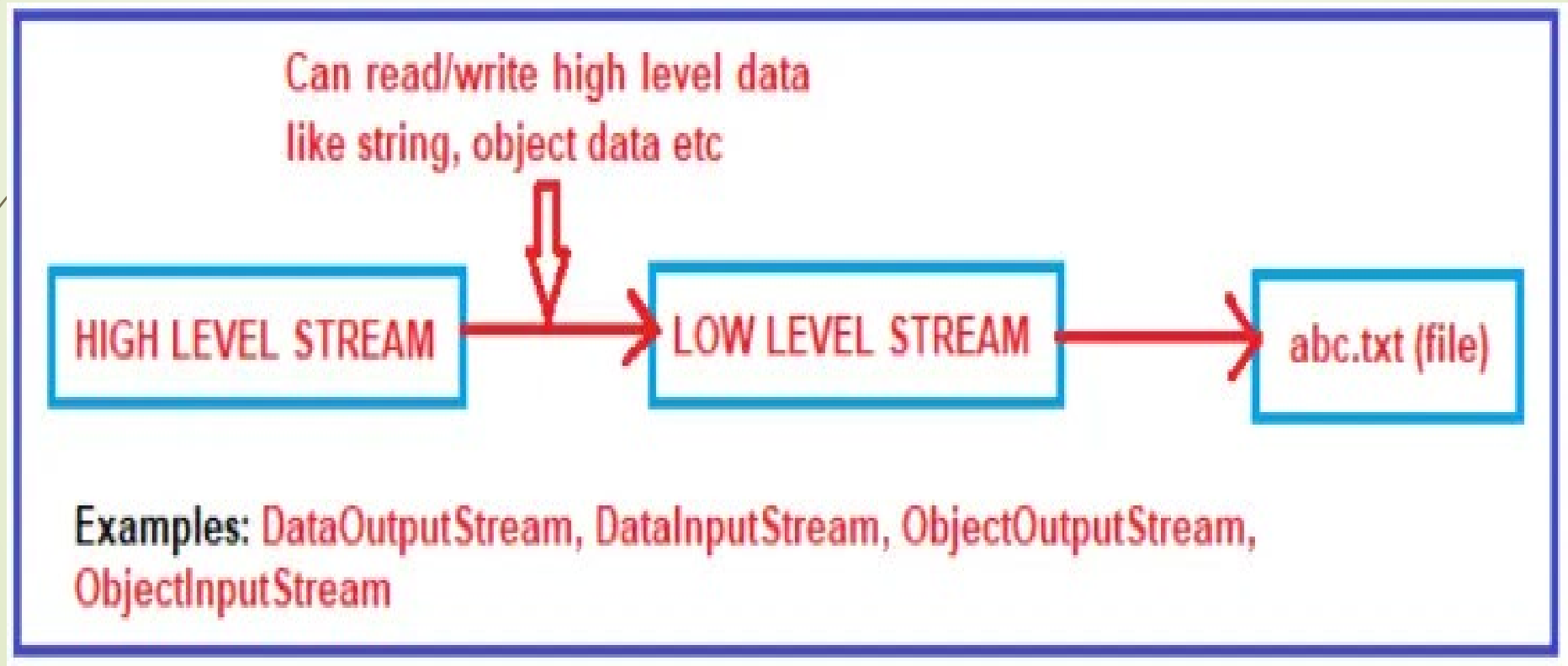
1. A **stream** is a continuous flow of data that represents the communication channel between the **application** and **destination resource** (file).
2. **Byte stream** can **read** and **write** bytes.
3. **Character** stream can read and write characters.

Deserialization

Deserialization is a process of converting a stream of bytes into an object which is nothing but reading the object from an object file. To perform this deserialization we use the class called **ObjectInputstream**.

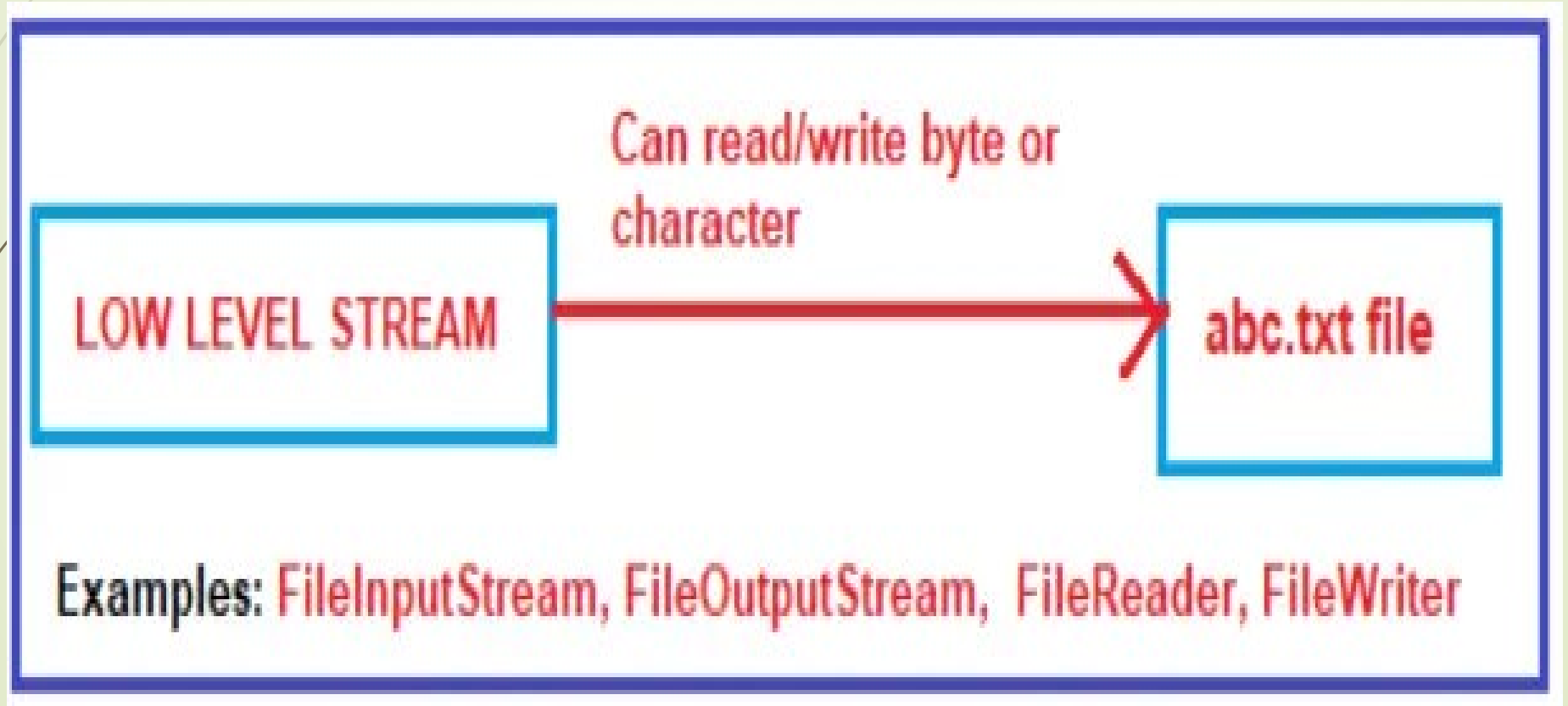
What is the difference between a **High-level stream** and a **Low-level stream**?

The **High-level Streams** interact with files through low-level streams and can work with data like string values and object values, etc.



What is the difference between a **High-level stream** and a **Low-level stream**?

The Low-level streams interact with files directly and can deal with low-level data like characters and bytes.



Points to remember:

1. Java objects become Serializable only when the class of that object implements the **java.io.Serializable marker interface.**
2. In small applications, the industry generally prefers **serialization** and **deserialization** persistence operations on the files.
3. There is no terminology called selective deserialization. The object data that is returned through the serialization process must be retrieved through the deserialization process.
4. The structure of the class must be the same while performing serialization and deserialization operations. Otherwise, the serialization process raises **java.io.InvalidClassException.**
5. In order to send Java objects over the network, they must be designed as serializable object.
6. All wrapper class objects and collection framework data structures are serializable by default.