MODULE III

Syllabus

- Input/Output:
  - Stream classes (byte stream and character stream classes)
  - reading console input.
  - Files

- Input refers to flow of data into a program and output means flow of data out of a program.
- Input to a program may come from various sources like keyboard, mouse, memory, disk, network.
- Output of a program may go to several destinations like screen, printer, memory, disk, network.
- Data that is transferred is treated as a sequence of bytes or characters.
- Java uses concept of streams to represent ordered sequence of data.

IO Basics : Streams

- Java programs perform I/O through streams.
- A stream presents a uniform, easy-to-use, object-oriented interface between the program and the i/o devices. It is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.
- Java implements streams within **java.io** package.

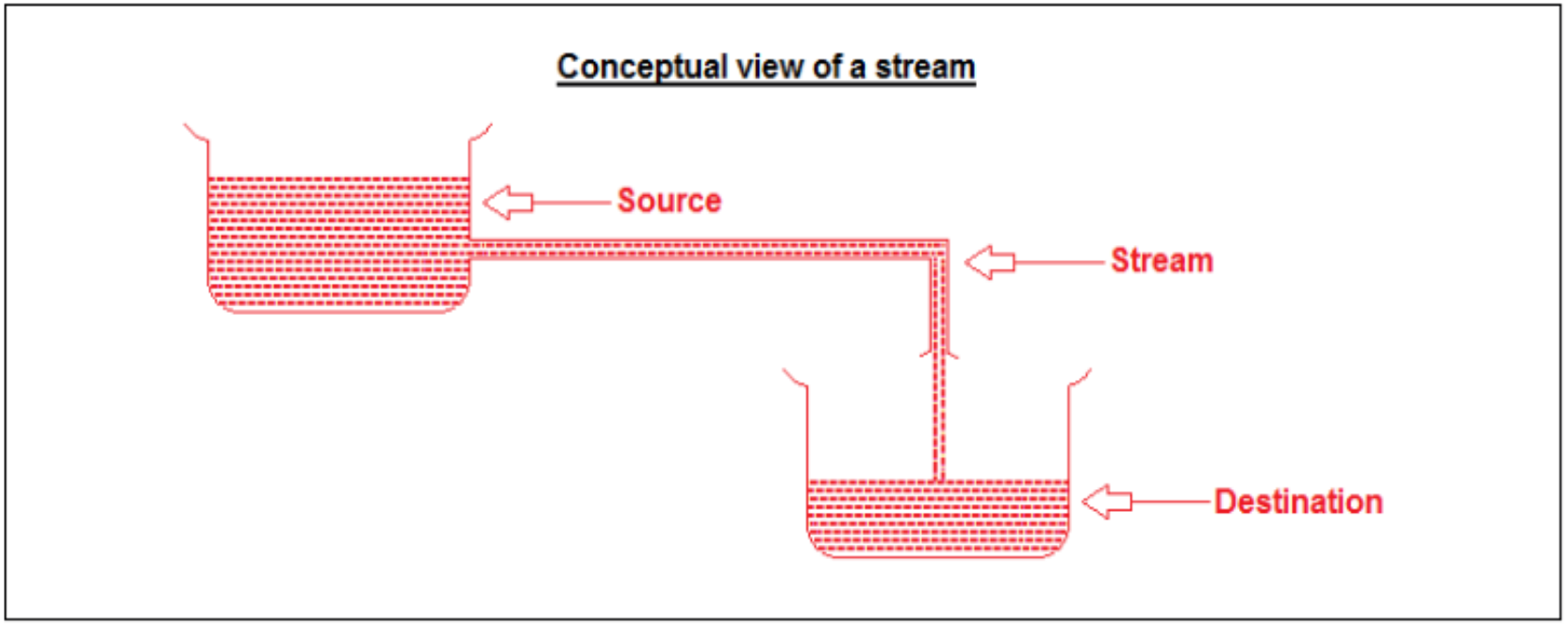
- Java streams are classified into two basic types:
  - Input stream
  - Output stream
- An input stream extracts(or reads) data from source(file) and sends it to the program. Similarly an output stream takes data from the program and sends it to the destination(file).

- Java.io package contains a large number of stream classes that provide capabilities for processing all types of data.

- Stream classes are categorized into two types based on the data on which they operate. They are:
  - Byte Stream : Used for reading or writing binary data.
  - CharacterStream : Used for handling input and output of characters.

    They use unicode.

- At the lowest level, all I/O is byte-oriented.

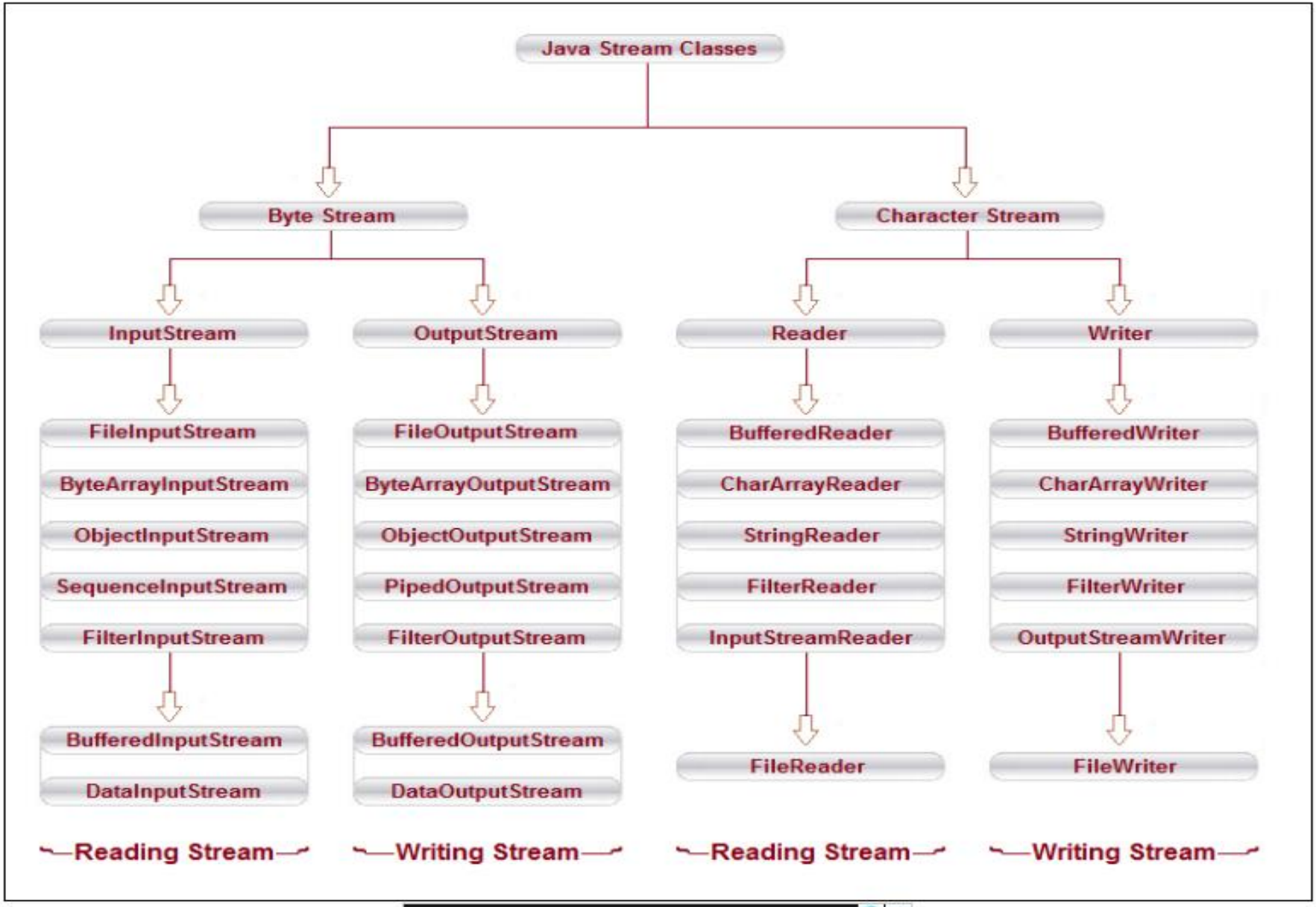


Types of Streams

The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

- **Byte stream classes**
- **Character stream classes**

Java Stream Classes Hierarchy



Byte Streams Classes

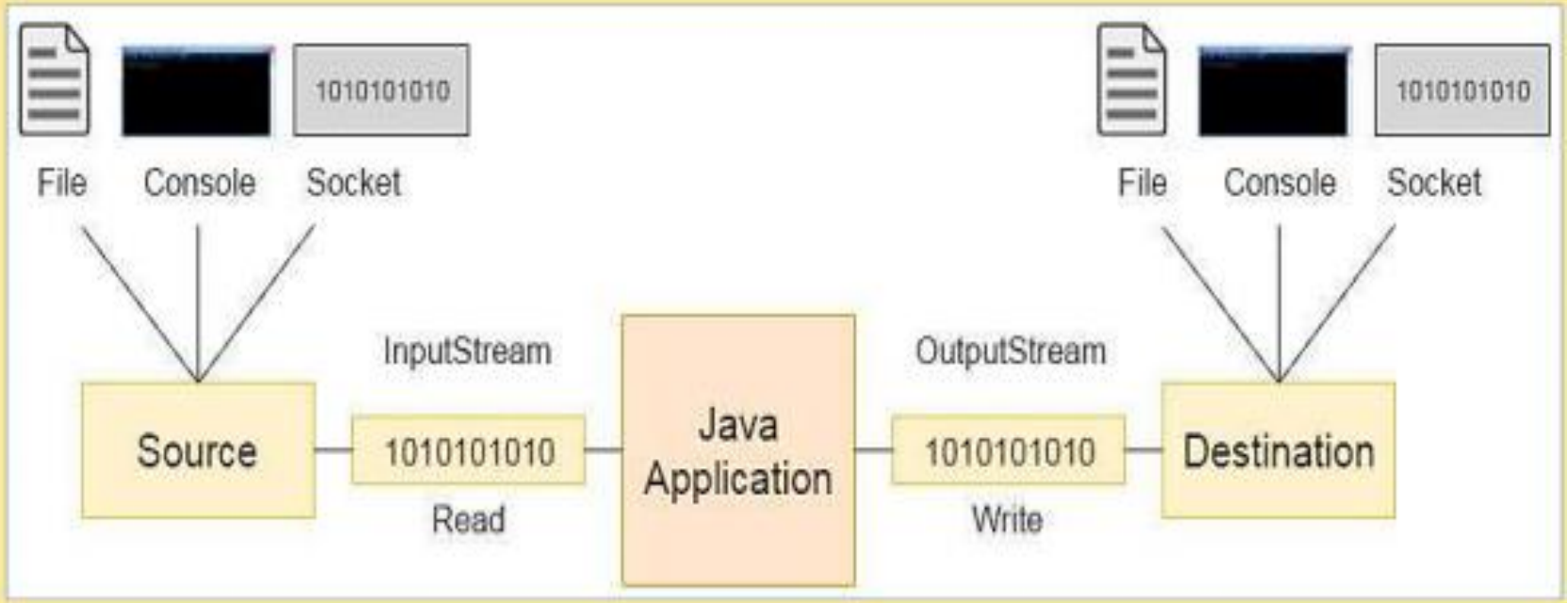
- Two abstract classes for Byte streams :
  - InputStream
  - OutputStream

- These classes define several key methods that the other stream classes implement.
- Eg: **read( )** and **write( )**, which, respectively, read and write bytes of data. Both methods are declared as abstract. They are overridden by derived stream classes.

- Each of these abstract classes has several concrete subclasses, that handle various devices, such as disk files, network connections, memory buffers etc.

- Java.io package has to be imported to use stream classes.

❖The working of Java OutputStream and InputStream



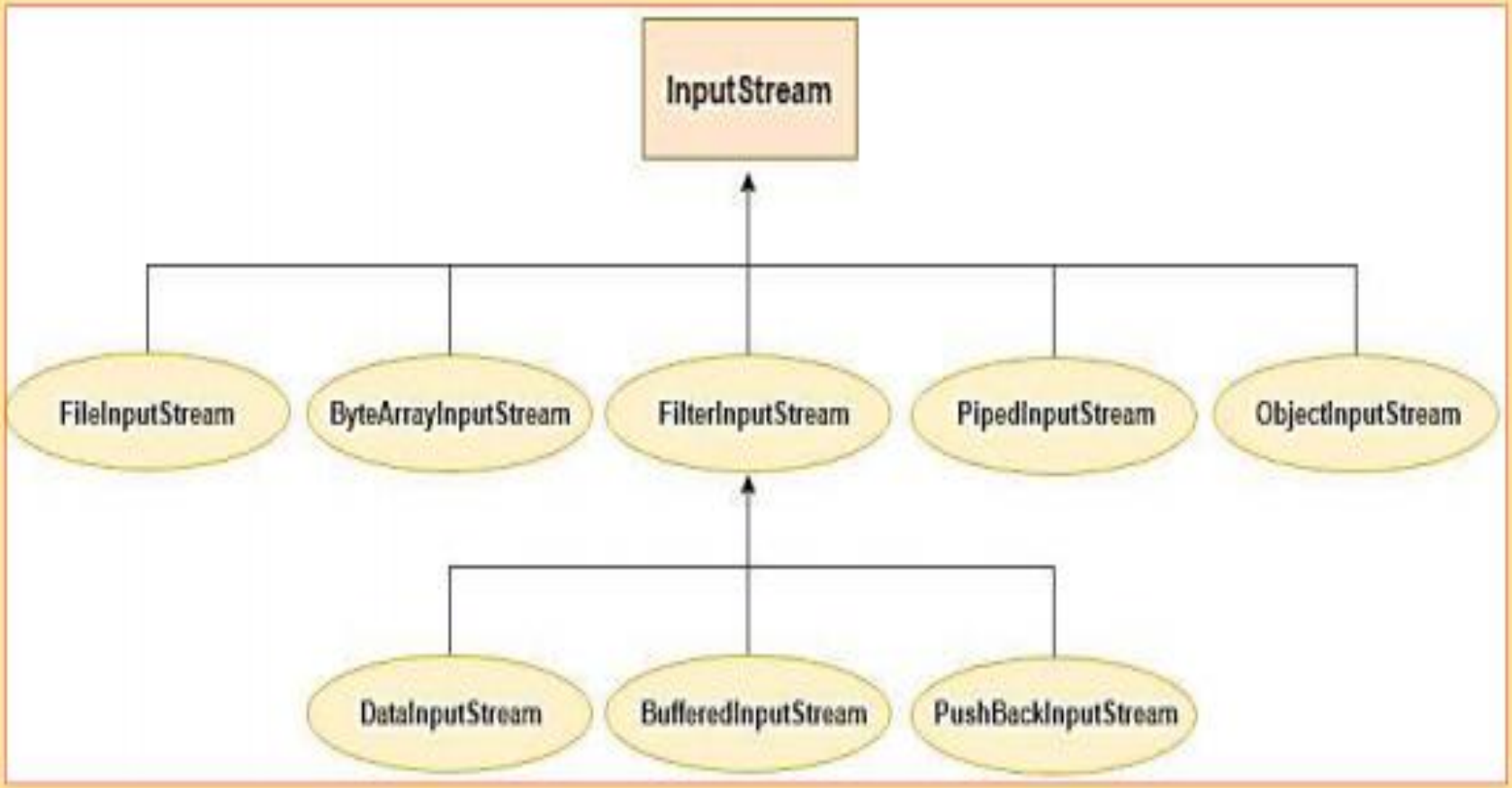
InputStream class

- InputStream class is an abstract class.
- It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

❖InputStream Hierarchy



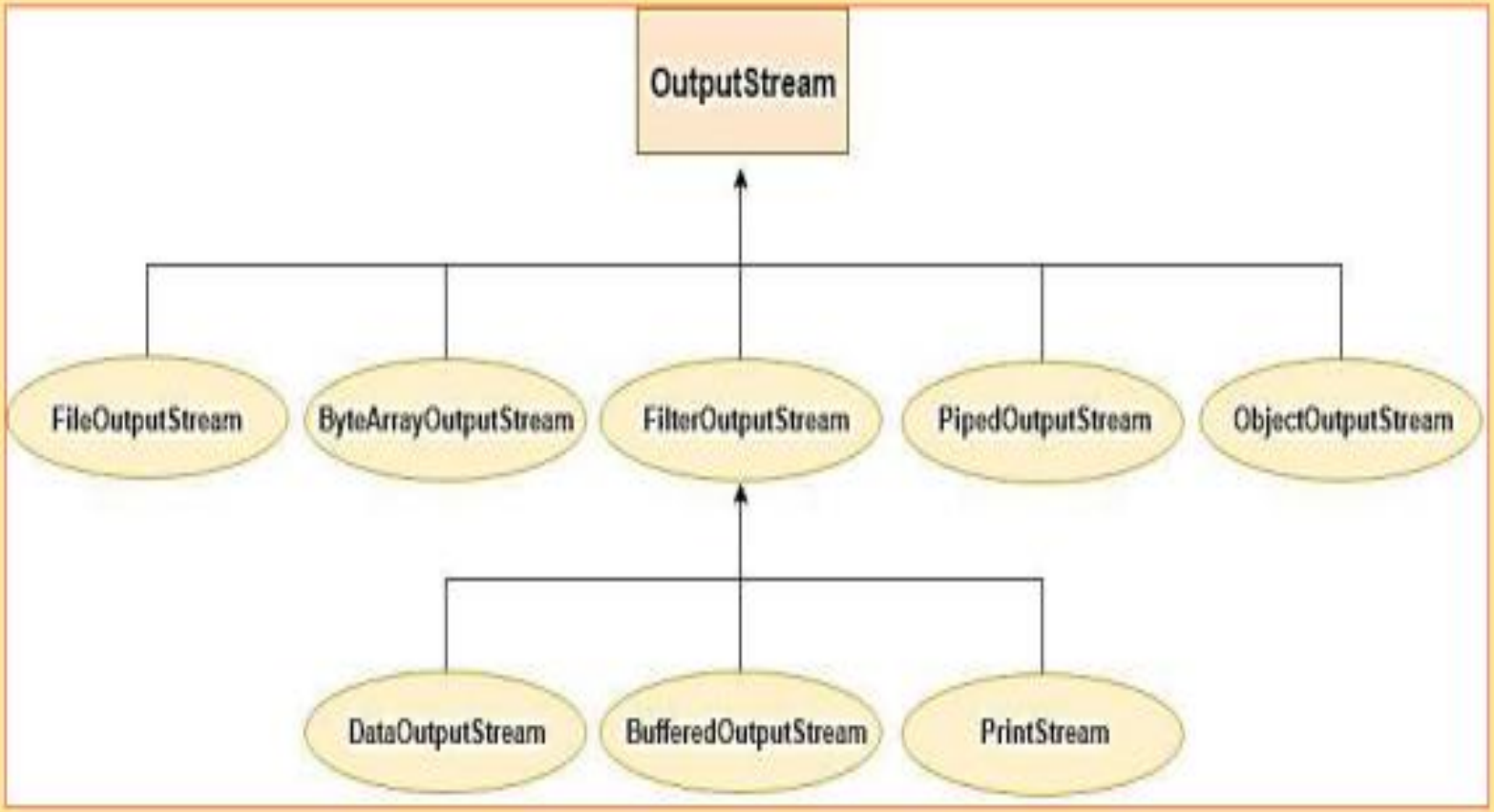
OutputStream class

- It is the superclass of all classes representing an output stream of bytes.
- It is an abstract class.
- An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

❖OutputStream Hierarchy



Byte Stream Classes	
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
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

BufferedInputStream Class

- The BufferedInputStream class of the java.io package is used with other input streams to read the data (in bytes) more efficiently.
- It extends the InputStream abstract class.

Working of BufferedInputStream

- The BufferedInputStream maintains an internal buffer of 8192 bytes.
- During the read operation in BufferedInputStream, a chunk of bytes is read from the disk and stored in the internal buffer. And from the internal buffer bytes are read individually.
- Hence, the number of communication to the disk is reduced. This is why reading bytes is faster using the BufferedInputStream.

BufferedInputStream

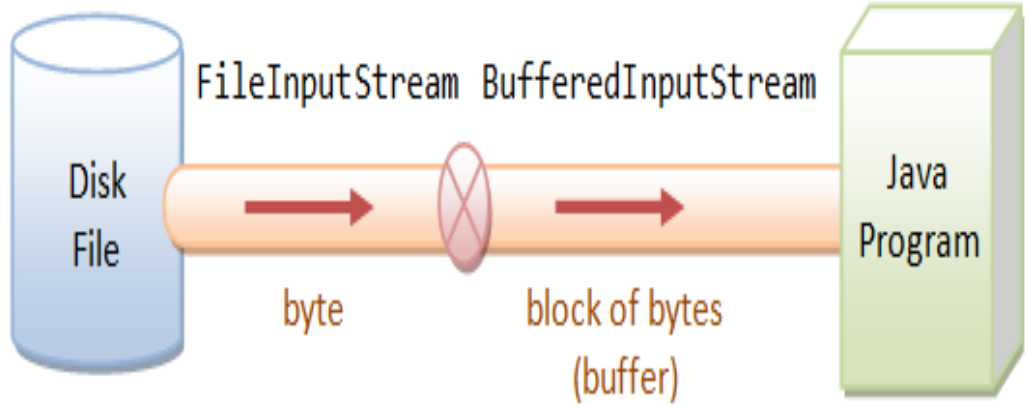
- In order to create a BufferedInputStream, we must import the java.io.BufferedInputStream package first. Once we import the package here is how we can create the input stream.

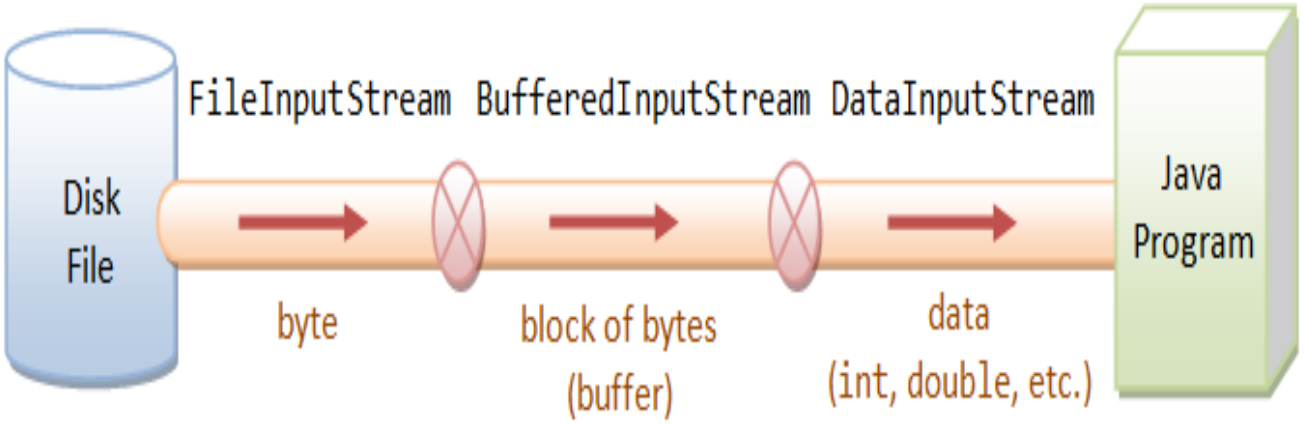
Example

```
//Creates a FileInputStream
FileInputStream file = new FileInputStream(String path);
// Creates a BufferedInputStream
BufferedInputStream buffer = new BufferInputStream(file);
```

In the above example, we have created a BufferdInputStream named buffer with the FileInputStream named file.

- Here, the internal buffer has the default size of 8192 bytes. However, we can specify the size of the internal buffer as well.
- // Creates a BufferedInputStream with specified size internal buffer
  BufferedInputStream buffer = new BufferInputStream(file, int size);
- The buffer will help to read bytes from the files more quickly.


FileInputStream
BufferedInputStream
Disk
File
Java
Program
byte
block of bytes
(buffer)

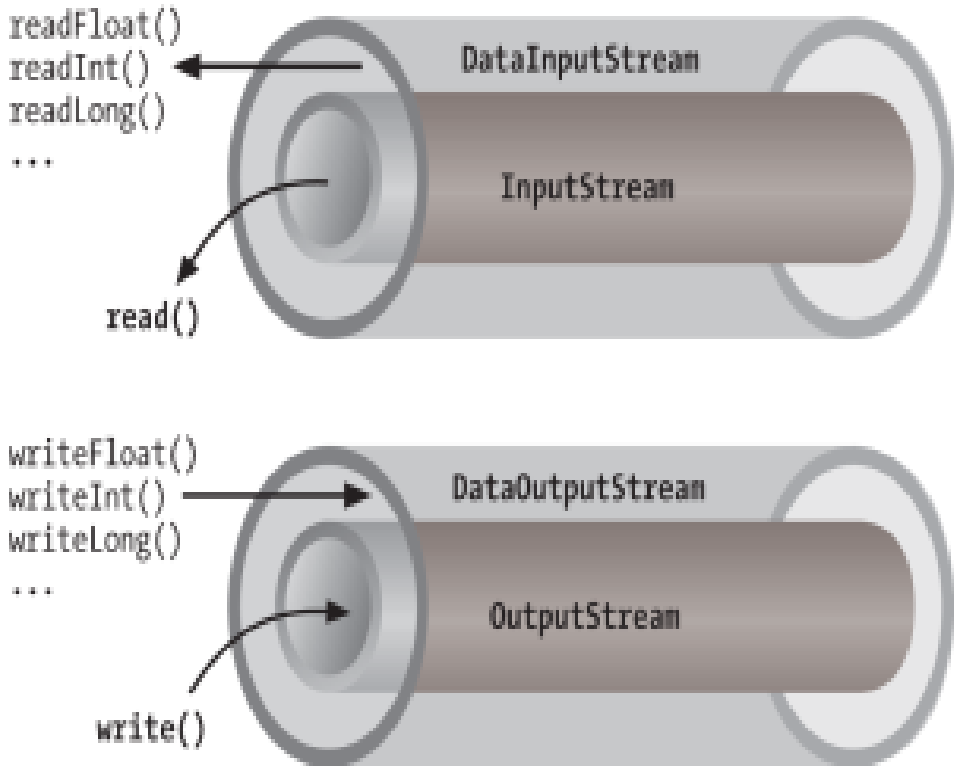

FileInputStream
BufferedInputStream
DataInputStream
Disk
File
Java
Program
byte
block of bytes
(buffer)
data
(int, double, etc.)

- **Example using DataInputStream class of Byte Stream Classes**

➢Java **DataInputStream** class allows an application to read primitive data from the input stream in a machine-independent way.

```
import java.io.*;
class inout
{
  public static void main(String args[])throws IOException
  {          int roll;
             String name;
             DataInputStream d=new DataInputStream(System.in);
             System.out.println("Enter Roll No");
             roll=Integer.parseInt(d.readLine());
             System.out.println("Enter name");
             name=d.readLine();
             System.out.println("ROLL : "+roll);
             System.out.println("NAME : "+name);
  } //readLine() is a function in DataInputStream
} //class and return type is String
```

```
C:\Users\cinitamary\Desktop>java inout
Enter Roll No
12
Enter name
cini
ROLL : 12
NAME : cini
```



Data streams in java

- **Example using ByteArrayInputStream class**

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte array as input stream.

Java ByteArrayInputStream class contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

```
/* This example demonstrates how to read one byte from the input stream at a time. */
import java.io.*;
public class ReadByteStreams
{       public static void main(String args[])throws IOException
        {       String str = args[0];
                byte b[] = str.getBytes();
                ByteArrayInputStream bais = new ByteArrayInputStream(b);
                int r;
                while ((r = bais.read()) != -1)
                {    System.out.print((char) r + " ");    }
        }
}
```

```
C:\Users\cinitamary\Desktop>java ReadByteStreams hello
h e l l o
```

Predefined Streams

- **java.lang** package defines a class called **System** which is automatically imported.
- **System** contains three predefined **public** and **static** stream variables: **in**, **out**, and **err**
  - public static **InputStream in** − It is the "standard" input stream.
  - public static **PrintStream out** − It is the "standard" output stream.
  - public static **PrintStream err** − It is the "standard" error output stream.
  - **System.in** refers to standard input stream, which is the keyboard by default. It enables the program to read data from keyboard.
  - **System.out** refers to the standard output stream. By default, this is the console.
  - **System.err** refers to the standard error stream, which also is the console by default.
- **System.in** is an object of type **InputStream.**
- **System.out** and **System.err** are objects of type **PrintStream**

Example

- The code to print output and an error message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

- The code to get input from console.

```
int i=System.in.read(); //returns ASCII code of 1st character
```

Character Stream Classes

- Two abstract classes for Character streams :
  - **Reader**
  - **Writer**

- These abstract classes handle Unicode character streams.

- These classes define several key methods that the other stream classes implement.

- Eg: **read()** and **write()**, are methods which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Character Stream Classes	
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 refers to stream between threads for inter thread communication
PipedWriter	Output pipe refers to stream between threads for inter thread communication
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

Reading Console Inputs

- In Java, there are three different ways for reading input from the user in the command line environment(console).
  - **Using Buffered Reader Class**
  - **Using Scanner Class**
  - **Using Console Class**

1.Using Buffered Reader Class

- The following line of code creates a **BufferedReader** that is connected to the keyboard:
- BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

BufferedReader- buffers the input charater stream

InputStreamReader - converts bytes to char

System.in - console input byte stream

- Now **br** is a character-based stream that is linked to the console through **System.in**.

```
import java.io.*;
class Test
{
   public static void main(String args[]) throws IOException
   {
      //Enter data using BufferReader
       BufferedReader reader = new BufferedReader(new
                                                    InputStreamReader(System.in));
  // Reading data using readLine
        String name = reader.readLine();
  // Printing the read line
        System.out.println(name);    }
}
```

This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader. We can read input from the user in the command line.

- **Advantage** - The input is buffered for efficient reading
- **Drawback** - The wrapping code is hard to remember.

2. Using Scanner Class

- This is probably the most preferred method to take input.

- The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line.

Advantages: • Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.

- Regular expressions can be used to find tokens.

Drawback:

- The reading methods are not synchronized

- The Scanner class is used is found in the java.util package.

Example

```
import java.util.Scanner;
class GetInputFromUser {
 public static void main(String args[]) {
        // Using Scanner for Getting Input from User
        Scanner in = new Scanner(System.in);
          String s = in.nextLine();
        System.out.println("You entered string "+s);
          int a = in.nextInt();
        System.out.println("You entered integer "+a);
            } }
```

3. Using Console Class

- The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

Advantages:

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

Drawback:

- Does not work in non-interactive environment (such as in an IDE).

```
// Java program to demonstrate working of System.console()
// Note that this program does not work on IDEs as
//System.console() may require console.
```

Example

```
public class Sample  {
   public static void main(String[] args)   {
       System.out.println("Enter name");
      String name = System.console().readLine();
      System.out.println(name);
     System.out.println("Enter password");
    char pass[]=System.console().readPassword();
     System.out.println(pass);
   }    } // Console class supplies no constructors. Instead, a
   //Console object is obtained by calling System.console( )
```

Comparing Byte Stream classes and Character Stream classes

Character-stream class	*Description*	*Corresponding byte class*
`Reader`	Abstract class for character-input streams	`InputStream`
`BufferedReader`	Buffers input, parses lines	`BufferedInputStream`
`LineNumberReader`	Keeps track of line numbers	`LineNumberInputStream`
`CharArrayReader`	Reads from a character array	`ByteArrayInputStream`
`InputStreamReader`	Translates a byte stream into a character stream	*(none)*
`FileReader`	Translates bytes from a file into a character stream	`FileInputStream`
`FilterReader`	Abstract class for filtered character input	`FilterInputStream`
`PushbackReader`	Allows characters to be pushed back	`PushbackInputStream`
`PipedReader`	Reads from a `PipedWriter`	`PipedInputStream`
`StringReader`	Reads from a `String`	`StringBufferInputStream`
`Writer`	Abstract class for character-output streams	`OutputStream`
`BufferedWriter`	Buffers output, uses platform's line separator	`BufferedOutputStream`
`CharArrayWriter`	Writes to a character array	`ByteArrayOutputStream`
`FilterWriter`	Abstract class for filtered character output	`FilterOutputStream`
`OutputStreamWriter`	Translates a character stream into a byte stream	*(none)*
`FileWriter`	Translates a character stream into a byte file	`FileOutputStream`
`PrintWriter`	Prints values and objects to a `Writer`	`PrintStream`
`PipedWriter`	Writes to a `PipedReader`	`PipedOutputStream`
`StringWriter`	Writes to a `String`	*(none)*

Reading a Character

- **read( )**

  int read( ) throws IOException

- Reads a character from the input stream and returns an integer value.

- It returns –1 when the end of the stream is encountered.

```
import java.io.*;
class BRRead
{       public static void main(String args[]) throws IOException
        {       char c;
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                System.out.println("Enter characters, 'q' to quit.");
                do
                {
                        c = (char) br.read();
                        System.out.println(c);
                } while(c != 'q');
        }
}
```

```
Output:
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

Reading a String

- **readLine( )**

  String readLine( ) throws IOException

```
import java.io.*;
class BRReadLines
{     public static void main(String args[]) throws IOException
      {    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            String str;
            System.out.println("Enter lines of text.");
            System.out.println("Enter 'stop' to quit.");
            do
            {
                  str = br.readLine();
                  System.out.println(str);
            } while(!str.equals("stop"));
      }
}
```

```
Enter lines of text.
Enter 'stop' to quit.
sdf
sdf
asd
asd
stop
stop
```

Q) Write a program to read 100 lines of text or until stop is entered and store it in an array. Display the array.

```
//A tiny Text Editor
import java.io.*;
class TinyEdit
{    public static void main(String args[]) throws IOException
     {   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         String str[] = new String[100];
         System.out.println("Enter lines of text.");
         System.out.println("Enter 'stop' to quit.");
         for(int i=0; i<100; i++)
         {
              str[i] = br.readLine();
              if(str[i].equals("stop")) break;
         }
         System.out.println("\nHere is your array:");
         for(int i=0; i<100; i++)
         {
              if(str[i].equals("stop")) break;
                   System.out.println(str[i]);
         }
     }
}
```

Writing Console Outputs

- In Java there are three different ways for writing to the console
  - PrintStream class
  - PrintWriter class
  - Console class
- Console output is obtained with **print()** and **println()** methods defined by **PrintStream** class(which is the type of object referenced by **System.out**)
- **PrintStream** also implements **write( )**.
- **write( )** can be used to write to the console.

  void write(int *byteval*)
- This method writes to the stream the byte specified by *byteval*.
- *byteval* is declared as an integer, only the low-order eight bits are written.

```
class WriteDemo
{     public static void main(String args[]) {
            int b;
            b = 'A';
            System.out.write(b);
            System.out.write('\n');   }
}
```

PrintWriter Class

- The recommended method of writing to the console is through a **PrintWriter** stream.
- It is used to print the formatted representation of objects to the text-output stream.
- **Class declaration**

  public class PrintWriter extends Writer
- **PrintWriter** is the character-based classes.

Methods of PrintWriter class

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.

PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

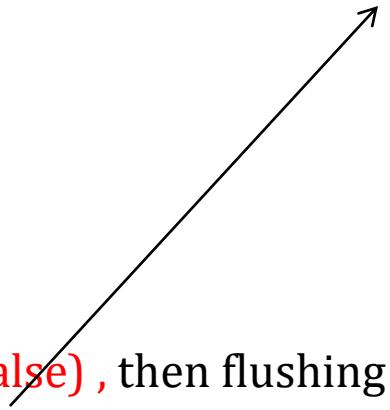
PrintWriter(OutputStream *outputStream*, boolean *flushOnNewline*) – This is one of the **constructor** of PrintWriter class.

- *flushOnNewline* controls whether Java flushes the output stream every time a **println( )** method is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

Example

PrintWriter pw = new PrintWriter(System.out, true);

```
import java.io.*;
public class PrintWriterDemo
{
	public static void main(String args[])
	{
		PrintWriter  pw = new PrintWriter(System.out, true);
		pw.println("This is a string");
		int i = -7;
		pw.println(i);
		double d = 4.5e-7;
		pw.println(d);
	}
} //if  new PrintWriter(System.out, false) , then flushing is not automatic
```



Output:
This is a string
-7
4.5E-7

We can also use the **Console** class to write output to the console, for example, using the **printf()** method with a String argument:
Example: console.printf(progLanguauge + " is very interesting!");

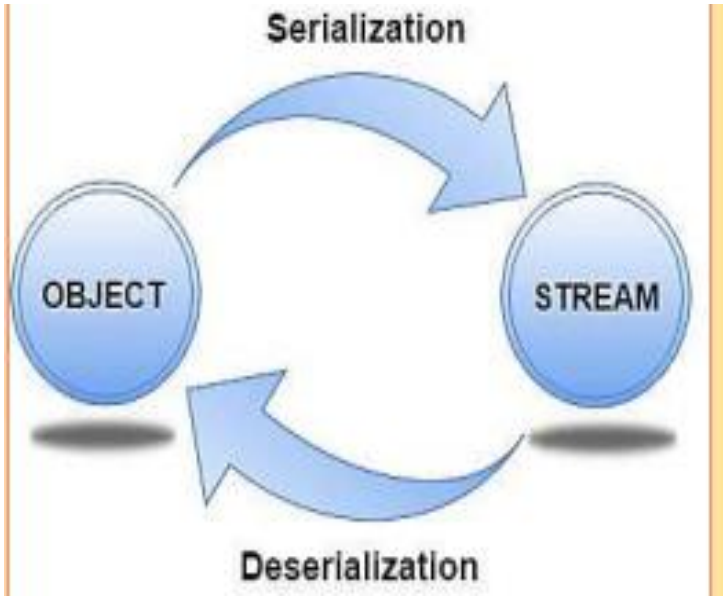
SERIALIZATION

- Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
- After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.
- For serializing the object, we call the writeObject() method of ObjectOutputStream, and for deserialization we call the readObject() method of ObjectInputStream class.
- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.
- Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

- We must have to implement the **Serializable interface** for serializing the object.

Advantages of Java Serialization

- It is mainly used to travel object's state on the network (which is known as marshaling).



ObjectOutputStream class

- 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

1) public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

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

ObjectInputStream class

- An **ObjectInputStream** class deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

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

Example of Java Serialization

In this example, we are going to serialize the object of **Student** class. The **writeObject()** method of **ObjectOutputStream** class provides the functionality to serialize the object. We are saving the state of the object in the file named **f.txt**.

Serialize.java

```
import java.io.*;
class Student implements Serializable{
String name;
int y;
Student(int roll,String s){
this.y=roll;
this.name=s;}
}
class Serialize{
public static void main(String args[]) throws IOException{
    Student s1=new Student(111,"Anu");
    FileOutputStream fout=new FileOutputStream("f.txt");
    ObjectOutputStream out=new ObjectOutputStream(fout);
    out.writeObject(s1);
    out.flush();
    out.close();
    System.out.println("success");
    }
}                                          //o/p  success
```

DeSerialize.java

```
import java.io.*;
class Student implements Serializable{
String name;
int y;
Student(int roll,String s){
this.y=roll;
this.name=s;}
}
class DeSerialize{
public static void main(String args[]) throws IOException
    ,ClassNotFoundException  {
    Student s1=null;
    FileInputStream fout=new FileInputStream("f.txt");
    ObjectInputStream out=new ObjectInputStream(fout);
    s1=(Student)out.readObject();
    System.out.println(s1.name);
    System.out. println(s1.y);
}
    }
```

o/p Anu
111

WORKING WITH FILES

- File handling is an important part of any application.
- Java has several methods for creating, reading, updating, and deleting files.
- 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

Create a File

- To create a file in Java, you can use the createNewFile() method.
- This method returns a boolean value: true if the file was successfully created, and false if the file already exists.

Example

```
import java.io.*;
public class CreateFile{
public static void main(String args[])throws IOException{
   File ob=new File("File1.txt");
   ob.createNewFile();
   System.out.println("created " + ob.getName());
   }      }  //o/p  created File1.txt
```

Write to a File using FileWriter class

- In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above.
  - **void write(String text)It is used to write the string into FileWriter.**
- Note that when we are done writing to the file, we should close it with the close() method:

Example

```
import java.io.*;
public class WriteFile{
public static void main(String args[])throws IOException{
   FileWriter wr=new FileWriter("ob.txt");
   wr.write("hai hello");    //writes hai hello to the file ob.txt
   wr.close();
   }     }
```

Write to a File using FileOutputStream class

- To write to a file, defined by **FileOutputStream**.

  void write(int byteval) throws IOException

- Writes the byte specified by *byteval* to the file. If an error occurs during writing, an IOException is thrown.

Example

```
import java.io.*;
public class WriteFileOutputStream {
    public static void main(String[] args) throws IOException{
            FileOutputStream out=new FileOutputStream("ob.txt");
            String str="Hello World";
                    byte[] strToBytes = str.getBytes();;
                    out.write(strToBytes);
                    out.close();
            }
}
```

Read from a File using FileReader class

- In the following example, we use the FileReader class together with its read() method to read data from the file.
- **int read()It is used to return a character in ASCII form. It returns -1 at the end of file.**

Example

```
import java.io.*;
public class ReadFileReader {
    public static void main(String args[])throws IOException{
        FileReader fr=new FileReader("ob.txt");
        int i;
        while((i=fr.read())!=-1)
        System.out.print((char)i);
        fr.close();
    }
}
```

Read from a File using FileInputStream class

- To read from a file, defined by **FileInputStream**

  int read( ) throws IOException

- It reads a single byte from the file and returns the byte as an integer value. **read( )** returns –1 when the end of the file is encountered.

Example

```
import java.io.*;
public class ReadFileInputStream {
    public static void main(String args[]) throws IOException{
        FileInputStream fin=new FileInputStream("ob.txt");
        int i=0;
        while((i=fin.read())!=-1){
         System.out.print((char)i);}
        fin.close();
      }
     }
```

Read from a File using Scanner class

Example

```
import java.io.*;
import java.util.*;
public class ReadFileScanner {

    public static void main(String[] args) {
            try {
                    Scanner scanner = new Scanner(new File("ob.txt"));
                    while (scanner.hasNextLine()) {
                            System.out.println(scanner.nextLine());
                    }
                    scanner.close();
            } catch (FileNotFoundException e) {
                    e.printStackTrace();
            }
    }
}
```

❖Get File Information

```
import java.io.File;  // Import the File class

public class GetFileInfo {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.exists()) {
      System.out.println("File name: " + myObj.getName());
      System.out.println("Absolute path: " + myObj.getAbsolutePath());
      System.out.println("Writeable: " + myObj.canWrite());
      System.out.println("Readable " + myObj.canRead());
      System.out.println("File size in bytes " + myObj.length());
    } else {
      System.out.println("The file does not exist.");
    }
  }
}
```

The output will be:

```
File name: filename.txt
Absolute path: C:\Users
Writeable: true
Readable: true
File size in bytes: 0
```

❖Delete a File

➢To delete a file in Java, use the delete() method:

```
import java.io.File;  // Import the File class

public class DeleteFile {
  public static void main(String[] args) {
    File myObj = new File("filename.txt");
    if (myObj.delete()) {
      System.out.println("Deleted the file: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the file.");
    }
  }
}
```

The output will be:

```
Deleted the file: filename.txt
```

❖Delete a Folder

```
import java.io.File;

public class DeleteFolder {
  public static void main(String[] args) {
    File myObj = new File("C:\\Users\\MyName\\Test");
    if (myObj.delete()) {
      System.out.println("Deleted the folder: " + myObj.getName());
    } else {
      System.out.println("Failed to delete the folder.");
    }
  }
}
```

The output will be

Deleted the folder: Test

Example :

Write a program that uses read( ) to input from a file specified as command line argument. Display the contents of the text file.

```
import java.io.*;
class ShowFile
 {    public static void main(String args[]) throws IOException
{
int i;
FileInputStream fin;

     fin = new FileInputStream(args[0]);


do
{     i = fin.read();
if(i != -1)    System.out.print((char) i);
} while(i != -1);
fin.close();
}
}                    // java ShowFile ob.txt
```

Q1)Write a program to copy contents from one file to another file using command line arguments.

```
import java.io.*;
class CopyFile
{       public static void main(String args[]) throws IOException
{       int i;
            FileInputStream fin;
            FileOutputStream fout;
            fin = new FileInputStream(args[0]);
        fout = new FileOutputStream(args[1]);
        do
            {
            i = fin.read();
            if(i != -1)
            fout.write(i);
            } while(i != -1);
            fin.close();
            fout.close();    // java CopyFile ob1.txt   ob2.txt
}     }
```

TUTORIAL

Q1) Write a program to create a file that could store details of three students. Details include rollno, name and address and are provided through keyboard.

Q2) Write a Java program that accepts N integers through console and sort them in ascending order.

Q3) Write a java program that accepts integers from a file and display their sum.

Q4) Write a program to count the number of words in a text file.

Q5) Write a program to read product code and name from keyboard and write it to a file