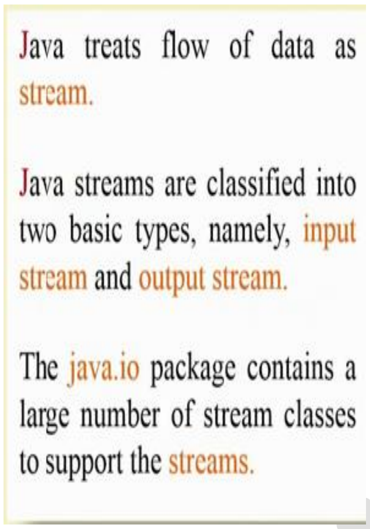
UNIT II Part E

Java I/O

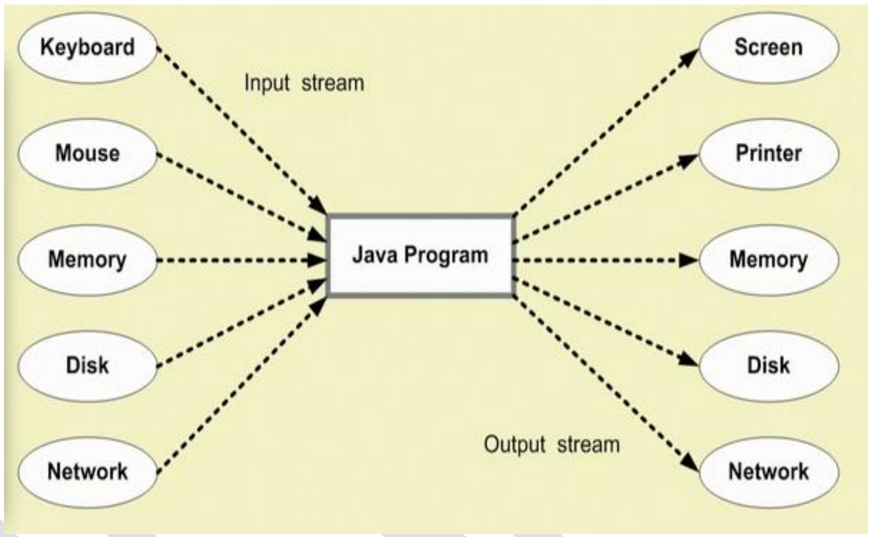
Java I/O (Input and Output) is used to process the input and produce the output.
Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

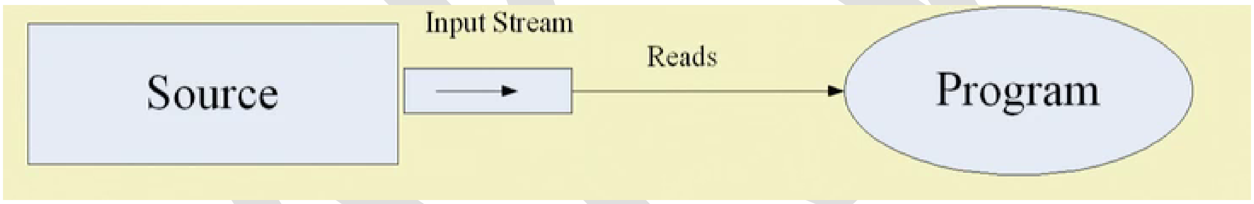
Streams in Java:


Java treats flow of data as stream.

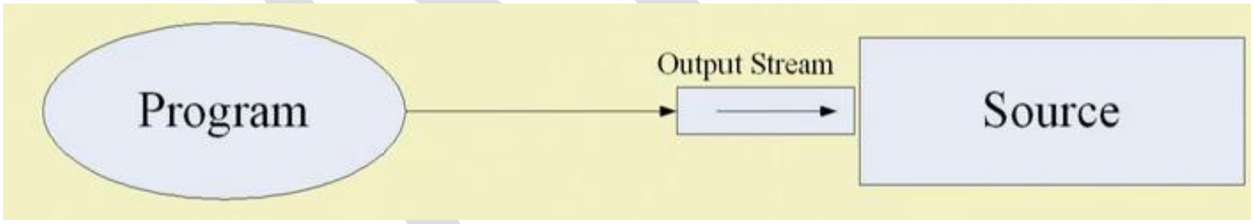
Java streams are classified into two basic types, namely, input stream and output stream.

The java.io package contains a large number of stream classes to support the streams.

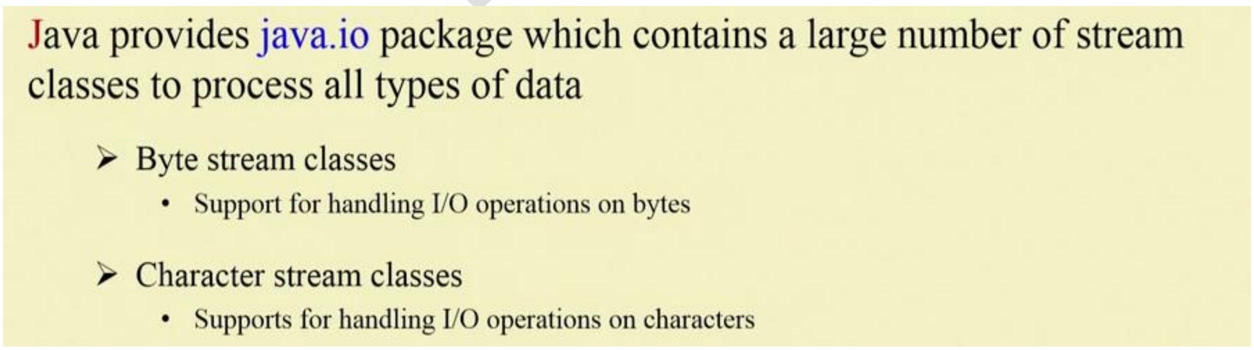




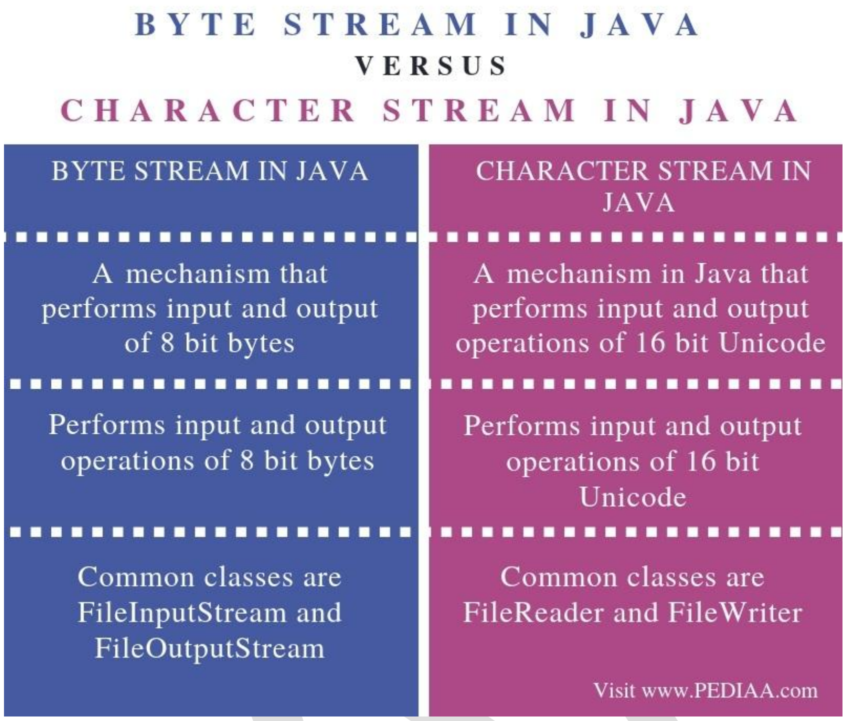
Reading Data into program



Writing Data to a Destination


Java provides java.io package which contains a large number of stream classes to process all types of data

- Byte stream classes
    - Support for handling I/O operations on bytes
- Character stream classes
    - Supports for handling I/O operations on characters


BYTE STREAM IN JAVA
VERSUS
CHARACTER STREAM IN JAVA

BYTE STREAM IN JAVA	CHARACTER STREAM IN JAVA
A mechanism that performs input and output of 8 bit bytes	A mechanism in Java that performs input and output operations of 16 bit Unicode
Performs input and output operations of 8 bit bytes	Performs input and output operations of 16 bit Unicode
Common classes are FileInputStream and FileOutputStream	Common classes are FileReader and FileWriter

Visit www.PEDIAA.com

<u>Byte Stream</u>

Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class InputOutputStreamExample
{
        public static void main(String args[]) throws IOException
        {
                //Creating FileInputStream object
                File file = new File("myFile.txt");
                FileInputStream fis = new FileInputStream(file);
                byte bytes[] = new byte[(int) file.length()];
                //Reading data from the file
                fis.read(bytes);
                //Writing data to another file
                File out = new File("D:/File/CopyOfmyFile.txt");
                FileOutputStream outputStream = new FileOutputStream(out);
                //Writing data to the file
                outputStream.write(bytes);
                outputStream.flush();
                System.out.println("Data successfully written in the specified file");
        }
}
```

Character Stream:

In Java, characters are stored using Unicode conventions. Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source and write to destination.

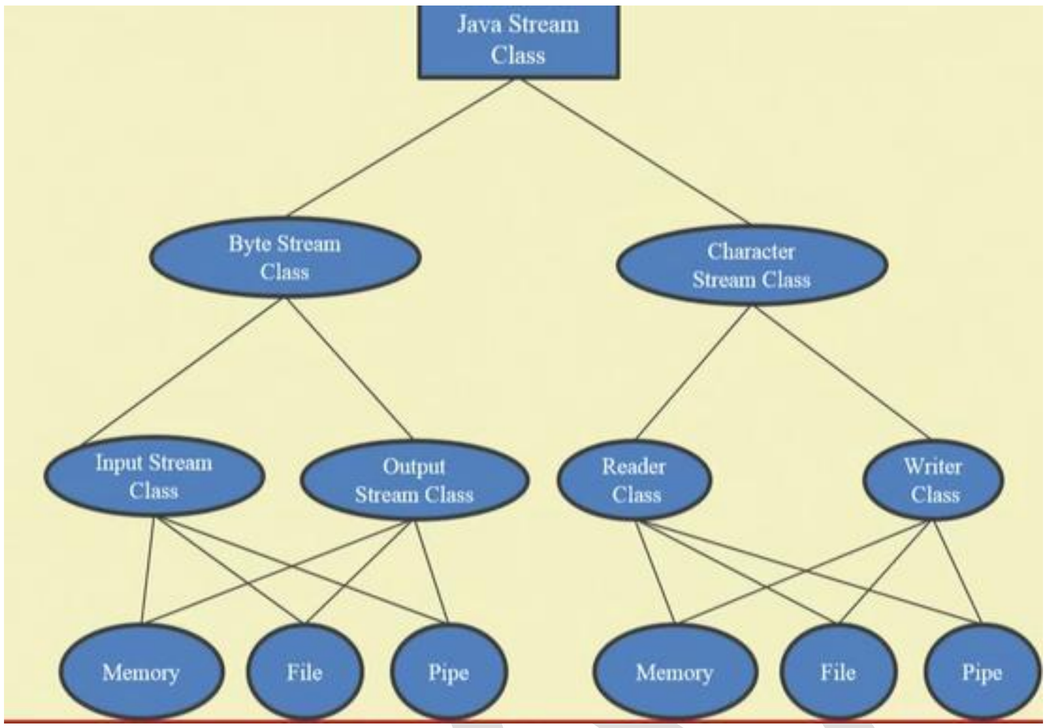
```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class ReaderWriterStreamExample
{
        public static void main(String args[]) throws IOException
        {
                //Creating FileReader object
                File file = new File("myFile.txt");
                FileReader reader = new FileReader(file);
                char chars[] = new char[(int) file.length()];
                //Reading data from the file
                reader.read(chars);
                //Writing data to another file
                File out = new File("D:/File/CopyOfmyFile.txt");
                FileWriter writer = new FileWriter(out);
                //Writing data to the file
                writer.write(chars);
                writer.flush();
                System.out.println("Data successfully written in the specified file");
        }
}
```

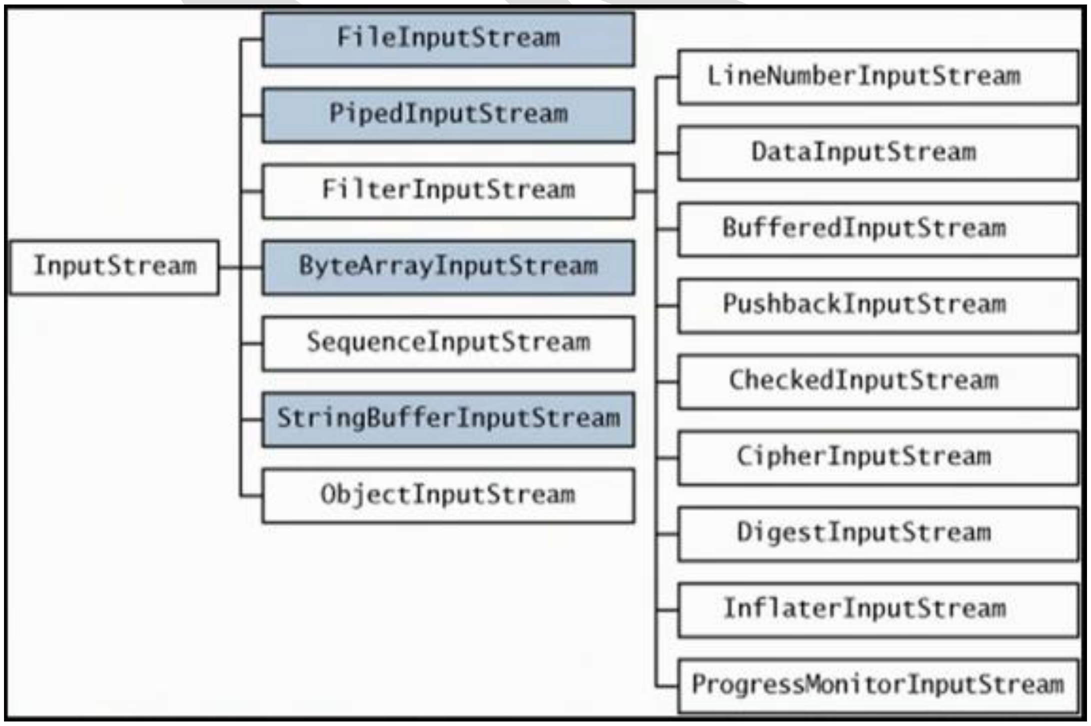
When to use Character Stream over Byte Stream?

- In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

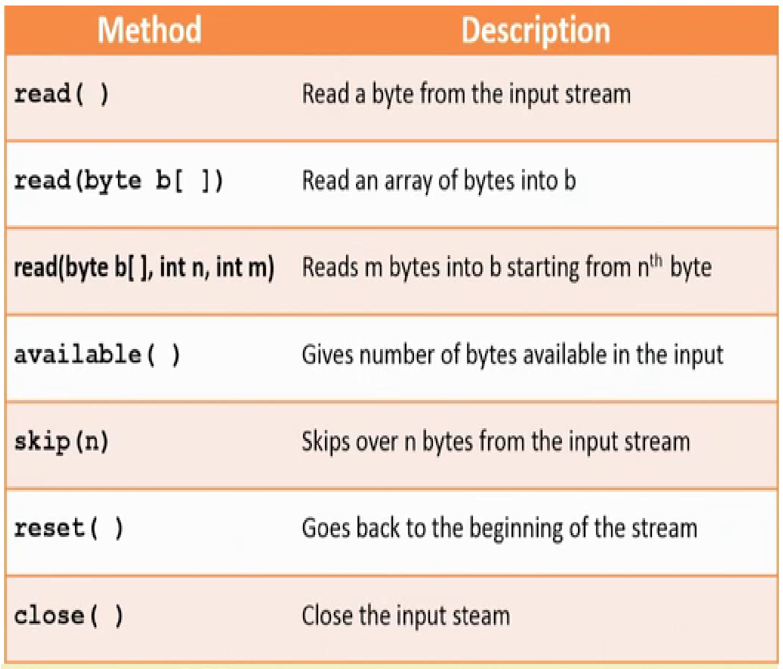
When to use Byte Stream over Character Stream?

- Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

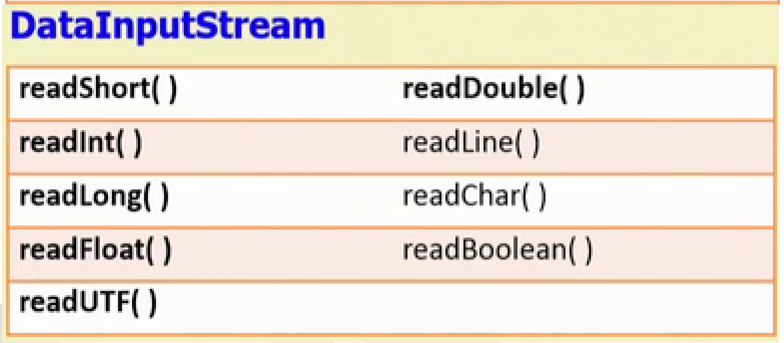




Input Stream class is used to read 8 bit bytes and support a number of input related methods



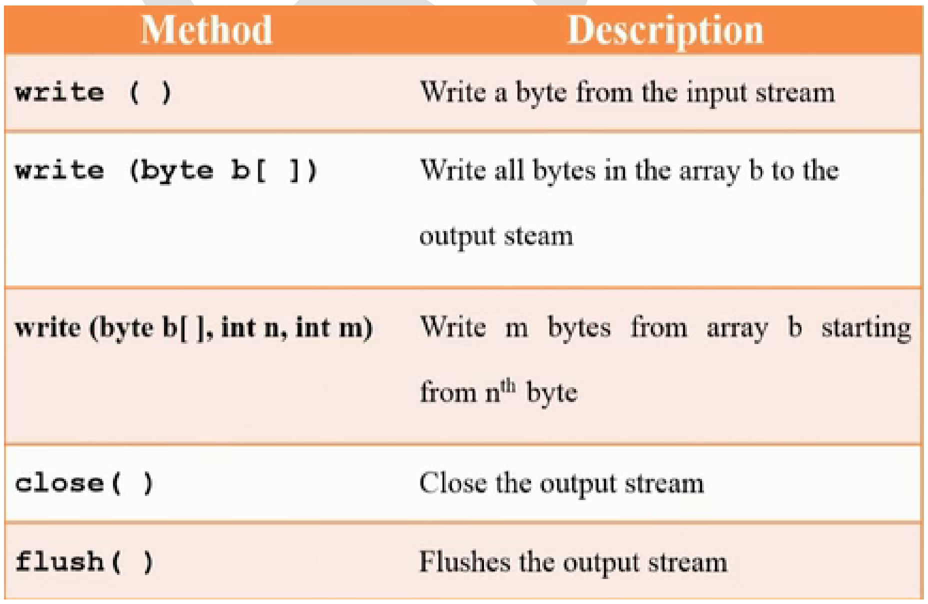
Method	Description
read( )	Read a byte from the input stream
read(byte b[ ])	Read an array of bytes into b
read(byte b[ ], int n, int m)	Reads m bytes into b starting from nth byte
available( )	Gives number of bytes available in the input
skip(n)	Skips over n bytes from the input stream
reset( )	Goes back to the beginning of the stream
close( )	Close the input steam



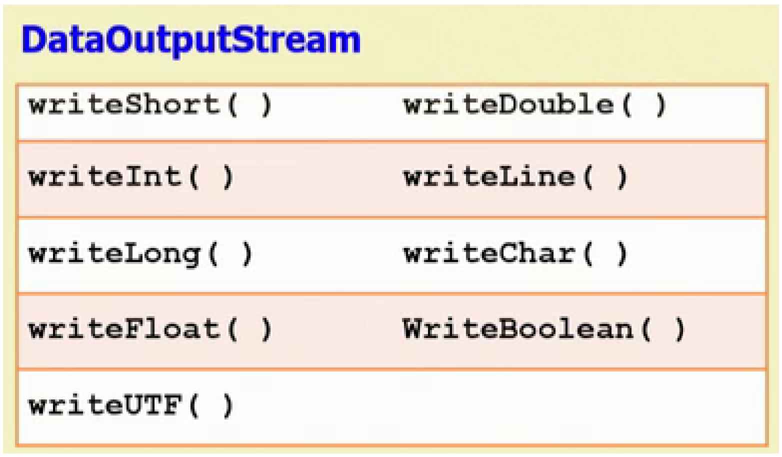
DataInputStream

readShort( )	readDouble( )
readInt( )	readLine( )
readLong( )	readChar( )
readFloat( )	readBoolean( )
readUTF( )	

Output stream classes is used to write 8 Bit bytes and support a number of input related methods



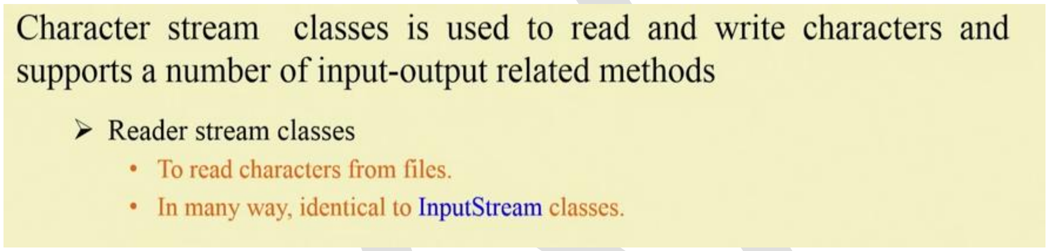
Method	Description
write ( )	Write a byte from the input stream
write (byte b[ ])	Write all bytes in the array b to the output steam
write (byte b[ ], int n, int m)	Write m bytes from array b starting from nth byte
close( )	Close the output stream
flush( )	Flushes the output stream



DataOutputStream

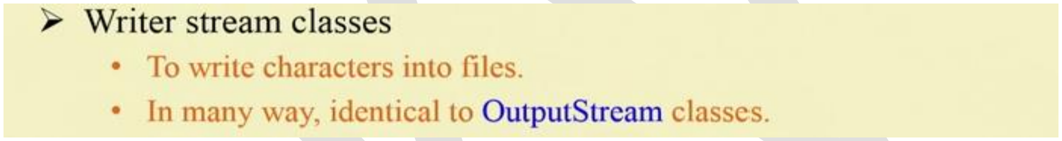
writeShort( )	writeDouble( )
writeInt( )	writeLine( )
writeLong( )	writeChar( )
writeFloat( )	WriteBoolean( )
writeUTF( )	

<u>Character Stream classes</u>



Character stream classes is used to read and write characters and supports a number of input-output related methods

- Reader stream classes
  - To read characters from files.
  - In many way, identical to InputStream classes.



- Writer stream classes
  - To write characters into files.
  - In many way, identical to OutputStream classes.

<u>Reading data from keyboard using InputStreamReader class and BufferedReader class</u>

InputStreamReader class can be used to read data from keyboard.It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

BufferedReader class can be used to read data line by line by readLine() method.

```
import java.io.*;
class InputBufferedReaderExample
{
        public static void main(String args[])throws Exception
        {
                InputStreamReader r=new InputStreamReader(System.in);
                BufferedReader br=new BufferedReader(r);
                System.out.println("Enter your name");
                String name=br.readLine();
                System.out.println("Welcome "+name);
        }
}
```

<u>Java Console Class</u>

The Java Console class is be used to get input from console. It provides methods to read texts and passwords. If you read password using Console class, it will not be displayed to the user.

```
import java.io.Console;
class ReadStringTest
{
   public static void main(String args[])
         {
                     Console c=System.console();
                     System.out.println("Enter your name: ");
                     String n=c.readLine();
                     System.out.println("Welcome "+n);
   }
}
```

Java PrintWriter class

Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

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.

```
import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample
{
         public static void main(String[] args) throws Exception
         {
                //Data to write on Console using PrintWriter
                PrintWriter writer = new PrintWriter(System.out);
                writer.write("Javatpoint provides tutorials of all technology.");
                writer.flush();
                writer.close();
                //Data to write in File using PrintWriter
                PrintWriter writer1 =null;
                writer1 = new PrintWriter(new File("C:/java19/IO/testout.txt"));
                writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");
                writer1.flush();
                writer1.close();
        }
}
```

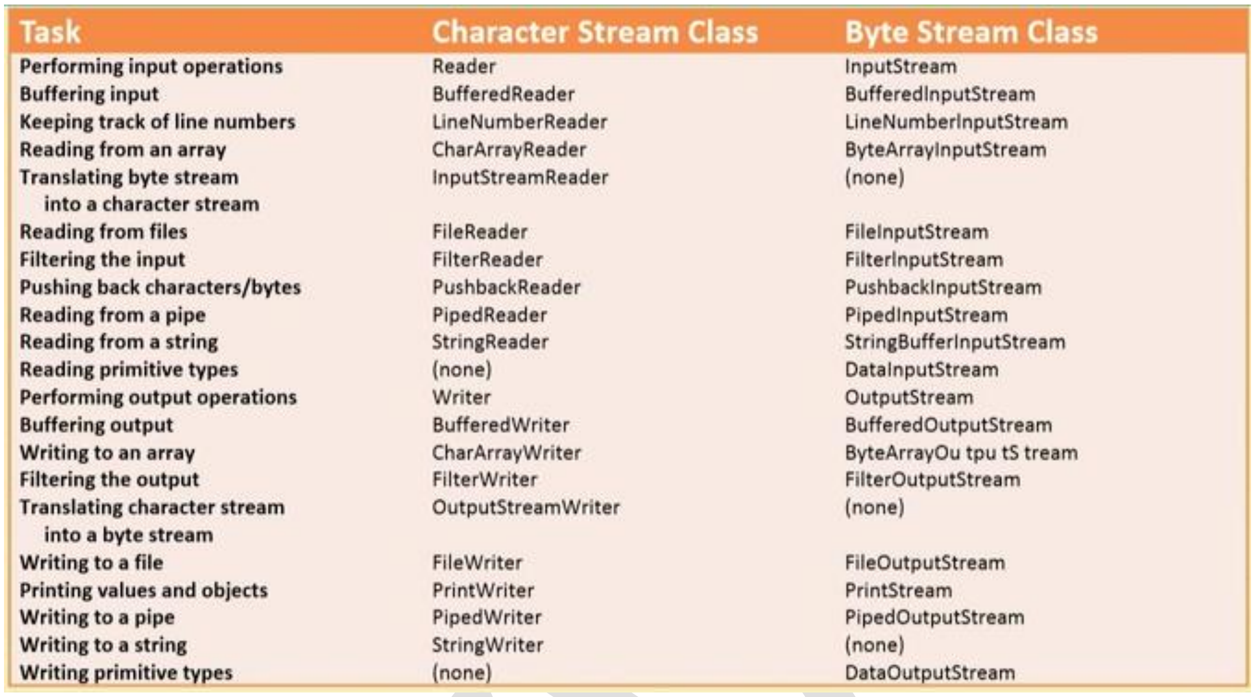
Closeable and Flushable Interface in JAVA

Closeable interface (of Closeable Flushable Java)

The **Closeable** interface includes only one abstract method, **close()**. When close() method is called, the system resources held by the stream object are released and can be used by other part of the program (avoids memory leaks). Many stream classes implement this interface and overrides the close() method. Any class that implements this interface can use close() method to close the stream handle. Also if the super class implements this interface, the sub class can use this method. For example, the **InputStream** implements this method and its subclass **FileInputStream** can use **close()** method. For that matter, all the streams can use close() method as the super classes of all streams, InputStream, OutputStream, Reader and Writer, implement Closeable interface.

Flushable interface (of Closeable Flushable Java)

The **Flushable** interface includes only one method – **flush()**. Many destination streams implement this interface and overrides the flush() method. When this method is called, the data held in the buffers is flushed out to the destination file to write.



Task	Character Stream Class	Byte Stream Class
Performing input operations	Reader	InputStream
Buffering input	BufferedReader	BufferedInputStream
Keeping track of line numbers	LineNumberReader	LineNumberInputStream
Reading from an array	CharArrayReader	ByteArrayInputStream
Translating byte stream into a character stream	InputStreamReader	(none)
Reading from files	FileReader	FileInputStream
Filtering the input	FilterReader	FilterInputStream
Pushing back characters/bytes	PushbackReader	PushbackInputStream
Reading from a pipe	PipedReader	PipedInputStream
Reading from a string	StringReader	StringBufferInputStream
Reading primitive types	(none)	DataInputStream
Performing output operations	Writer	OutputStream
Buffering output	BufferedWriter	BufferedOutputStream
Writing to an array	CharArrayWriter	ByteArrayOu tpu tS tream
Filtering the output	FilterWriter	FilterOutputStream
Translating character stream into a byte stream	OutputStreamWriter	(none)
Writing to a file	FileWriter	FileOutputStream
Printing values and objects	PrintWriter	PrintStream
Writing to a pipe	PipedWriter	PipedOutputStream
Writing to a string	StringWriter	(none)
Writing primitive types	(none)	DataOutputStream

Stream Tokenizer

Java.io.StreamTokenizer class parses input stream into “tokens”.It allows to read one token at a time. Stream Tokenizer can recognize numbers, quoted strings, and various comment styles. To use StreamTokenizer we need to understand some static fields of it.

nval : if current token is number, **nval gives that number**.
sval : If current **token is word**, it gives the character of that word.
TT_EOF : This is the point that represents that **end of file** has been read.
TT_EOL : This represents that **end of line has been read**.
TT_NUMBER : This represents that a **number has been read**.
TT_WORD : This represents that **word token has been read**.
ttype : This contains the **type of the token which has been read**.

```
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;
public class StreamTokenizerExample
{
        public static void main(String args[]) throws IOException
        {
                FileReader fileReader = new FileReader("D:/File/file.txt");
                StreamTokenizer st = new StreamTokenizer(fileReader);
                while(st.nextToken() != StreamTokenizer.TT_EOF)
                {
                        if(st.ttype == StreamTokenizer.TT_NUMBER)
                        {
                                System.out.println("Number: "+st.nval);
                        }
                        else if(st.ttype == StreamTokenizer.TT_WORD)
                        {
                                System.out.println("Word: "+st.sval);
                        }
                        else if(st.ttype == StreamTokenizer.TT_EOL)
                        {
                                System.out.println("--End of Line--");
                        }
                }
        }
}
```

Serialization and Deserialization in Java

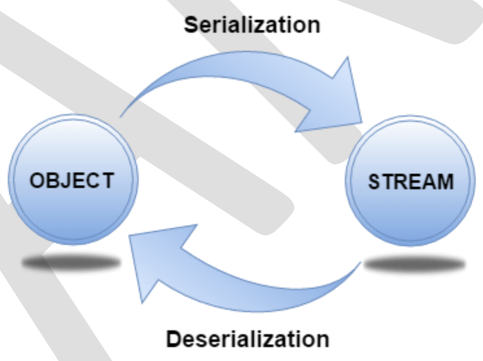
Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*. The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

For serializing the object, we call the **writeObject()** method *ObjectOutputStream*, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

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).



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.

It must be implemented by the class whose object you want to persist. The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

```
import java.io.Serializable;
public class Student implements Serializable
{
        int id;
        String name;
        public Student(int id, String name)
        {
                this.id = id;
                this.name = name;
        }
}
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

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.

```
import java.io.*;
class SerializationExample
{
        public static void main(String args[])
        {
                try
                {
                        //Creating the object
                        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.flush();
                        //closing the stream
                        out.close();
                        System.out.println("success");
                }
                catch(Exception e)
                {
                        System.out.println(e);
                }
        }
}
```

Example of Java Deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

```
import java.io.*;
class DeserializationExample
{
   public static void main(String args[])
         {
                  try
                  {
                           //Creating stream to read the object
                           ObjectInputStream in=new ObjectInputStream(new
FileInputStream("f.txt"));
                           Student s=(Student)in.readObject();
                           //printing the data of the serialized object
                           System.out.println(s.id+" "+s.name);
                           //closing the stream
                           in.close();
                  }
                  catch(Exception e)
                  {
                           System.out.println(e);
                  }
   }
}
```