Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, i/o devise, other program etc.

In general, a Stream will be an input stream or, an output stream.

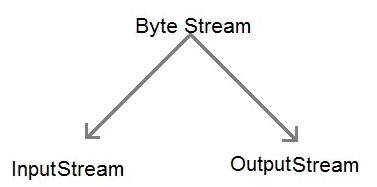
* **InputStream** − This is used to read data from a source.
* **OutputStream** − This is used to write data to a destination.

Based on the data they handle there are two types of streams −

* **Byte Streams** − These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
* **Character Streams** − These handle data in 16 bit Unicode. Using these you can read and write text data only.

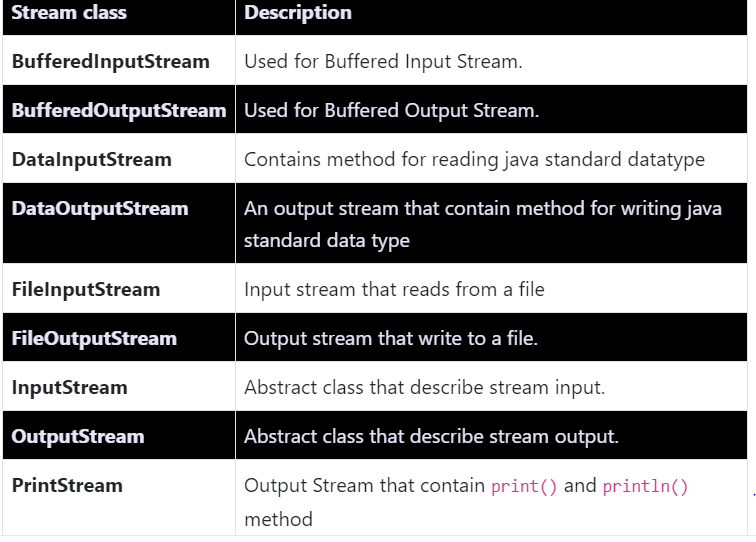
**Java Byte Stream Classes**

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

#### **Some important Byte stream classes.**

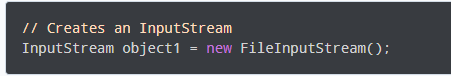


These classes define several key methods. Two most important are

1. read() : reads byte of data.
2. write() : Writes byte of data.

## Create an InputStream

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



Here, we have created an input stream using FileInputStream. It is because InputStream is an abstract class. Hence we cannot create an object of InputStream.

**Note**: We can also create an input stream from other subclasses of InputStream.

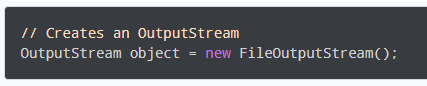
**Methods of InputStream**

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

* read() - reads one byte of data from the input stream
* read(byte[] array) - reads bytes from the stream and stores in the specified array
* available() - returns the number of bytes available in the input stream
* mark() - marks the position in the input stream up to which data has been read
* reset() - returns the control to the point in the stream where the mark was set
* markSupported() - checks if the mark() and reset() method is supported in the stream
* skips() - skips and discards the specified number of bytes from the input stream
* close() - closes the input stream

## Create an OutputStream

In order to create an OutputStream, we must import the java.io.OutputStream package first. Once we import the package, here is how we can create the output stream.



Here, we have created an object of output stream using FileOutputStream. It is because OutputStream is an abstract class, so we cannot create an object of OutputStream.

**Note**: We can also create the output stream from other subclasses of the OutputStream class.

**Methods of OutputStream**

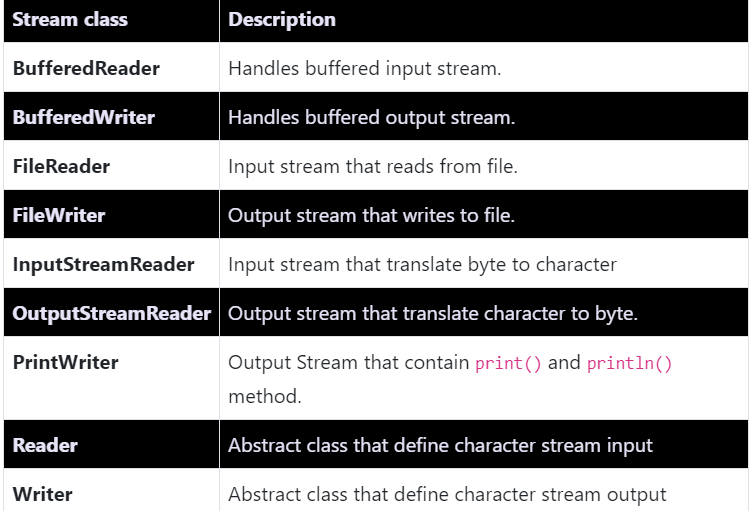
The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

* write() - writes the specified byte to the output stream
* write(byte[] array) - writes the bytes from the specified array to the output stream
* flush() - forces to write all data present in output stream to the destination
* close() - closes the output stream

## Java Character Stream Classes

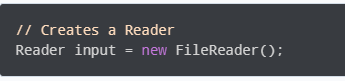
Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.

These two abstract classes have several concrete classes that handle unicode character.



## Create a Reader

In order to create a Reader, we must import the java.io.Reader package first. Once we import the package, here is how we can create the reader.



Here, we have created a reader using the FileReader class. It is because Reader is an abstract class. Hence we cannot create an object of Reader.

**Note**: We can also create readers from other subclasses of Reader.

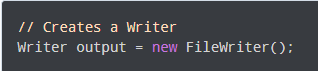
**Methods of Reader**

The Reader class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

* ready() - checks if the reader is ready to be read
* read(char[] array) - reads the characters from the stream and stores in the specified array
* read(char[] array, int start, int length) - reads the number of characters equal to length from the stream and stores in the specified array starting from the start
* mark() - marks the position in the stream up to which data has been read
* reset() - returns the control to the point in the stream where the mark is set
* skip() - discards the specified number of characters from the stream

## Create a Writer

In order to create a Writer, we must import the java.io.Writer package first. Once we import the package, here is how we can create the writer.



Here, we have created a writer named output using the FileWriter class. It is because the Writer is an abstract class. Hence we cannot create an object of Writer.

**Note**: We can also create writers from other subclasses of the Writer class.

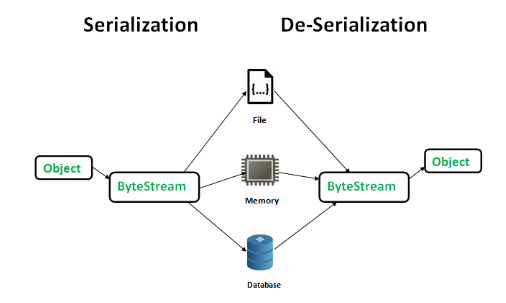
## Methods of Writer

The Writer class provides different methods that are implemented by its subclasses. Here are some of the methods:

* write(char[] array) - writes the characters from the specified array to the output stream
* write(String data) - writes the specified string to the writer
* append(char c) - inserts the specified character to the current writer
* flush() - forces to write all the data present in the writer to the corresponding destination
* close() - closes the writer

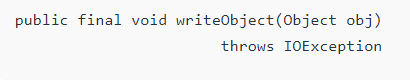
# **Serialization and Deserialization in Java**

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. This mechanism is used to persist the object.

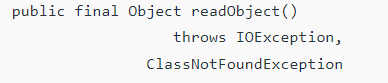


The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

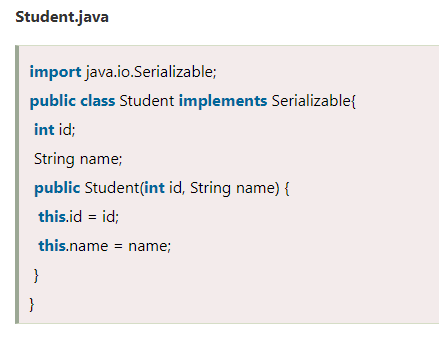
To make a Java object serializable we implement the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.



The ObjectInputStream class contains **readObject()** method for deserializing an object.



Let's see the example given below:



**Student** class implements Serializable interface. Now its objects can be converted into stream. The main class implementation of is showed in the next code.



### **Example of Java Deserialization**

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization.

