Module V-Part II

Java I/O and Streams

JAVA I/O

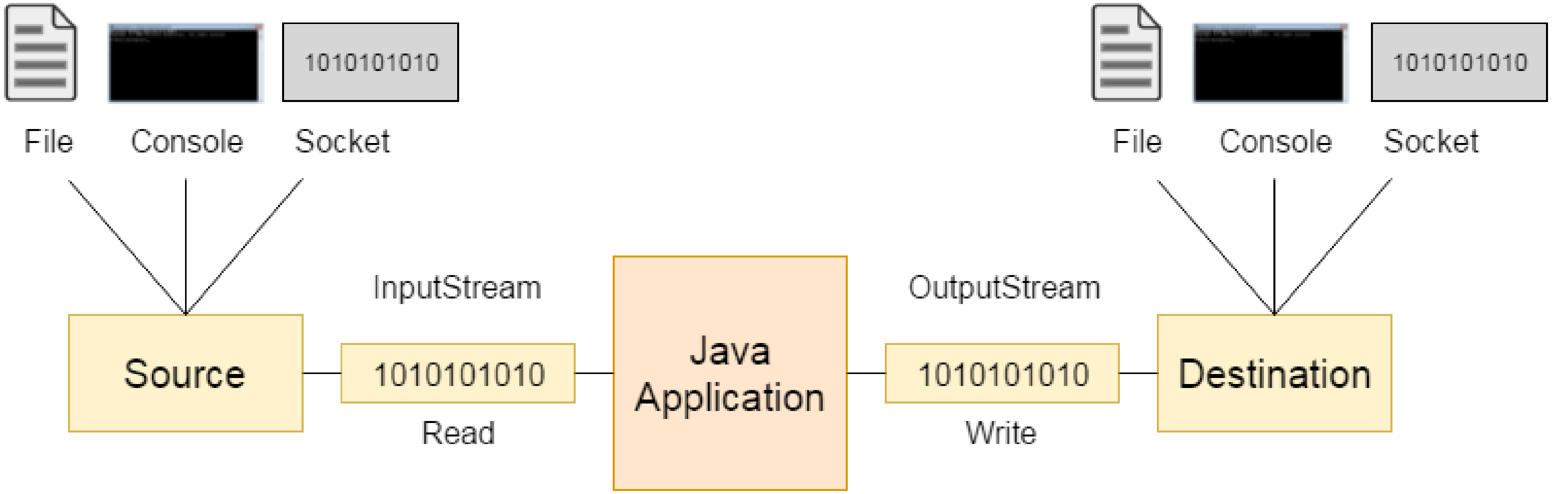
- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a stream to make I/O operation fast.
- The java.io package contains all the classes required for input and output operations.

Stream

- A stream is a sequence of data.
- In Java, a stream is composed of bytes.
- It's called a stream because it is like a stream of water that continues to flow.
- 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

OutputStream vs InputStream

- OutputStream
  - Java application uses an output stream to write data to a **destination**; it may be a file, an array, peripheral device or socket.
- InputStream
  - Java application uses an input stream to read data from a **source**; it may be a file, an array, peripheral device or socket.



Java InputStream and OutputStream

Character Stream vs Byte Stream

- Java defines two types of streams. They are,
- **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.
- **Byte Stream**
  - 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.

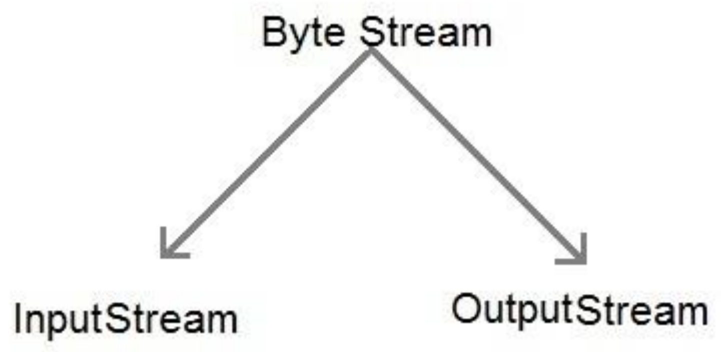
- **Note**: Names of character streams typically end with Reader/Writer and names of byte streams end with InputStream/OutputStream

When to use Character Stream /byte stream

- **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.
- A **byte stream** is suitable for processing raw data like binary files.

Java Byte Stream Classes

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

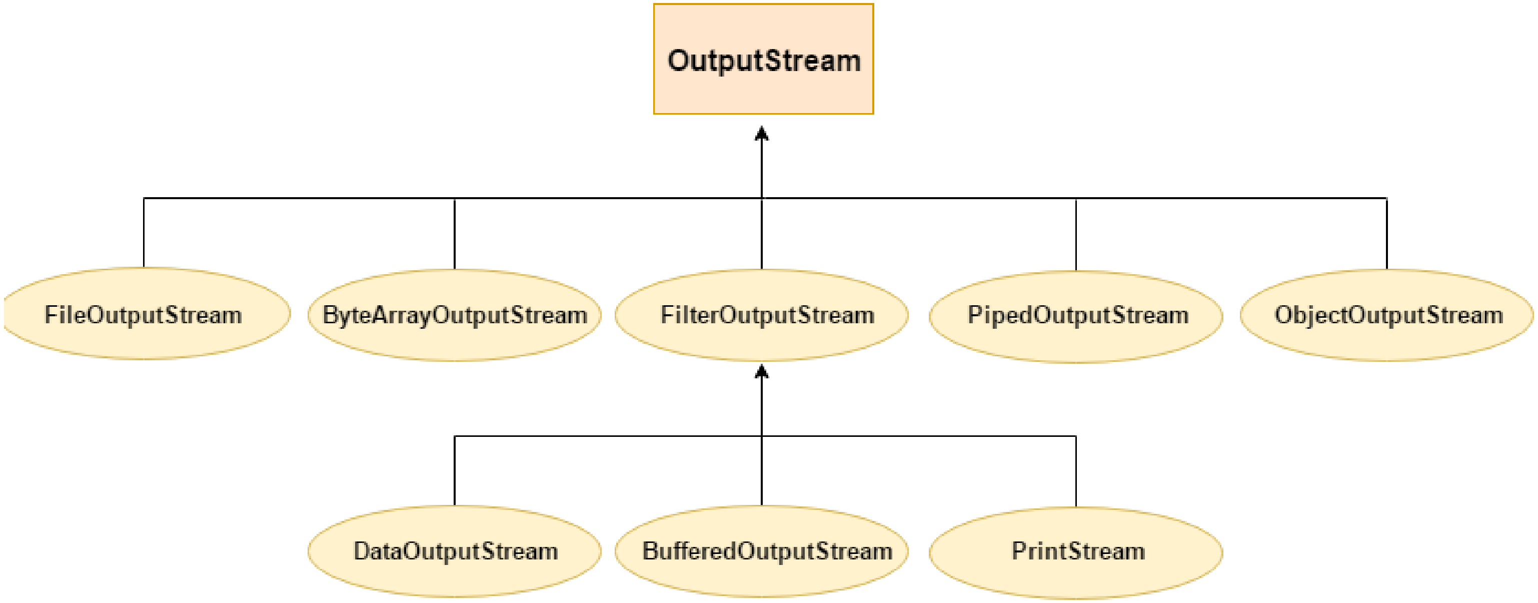


- These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

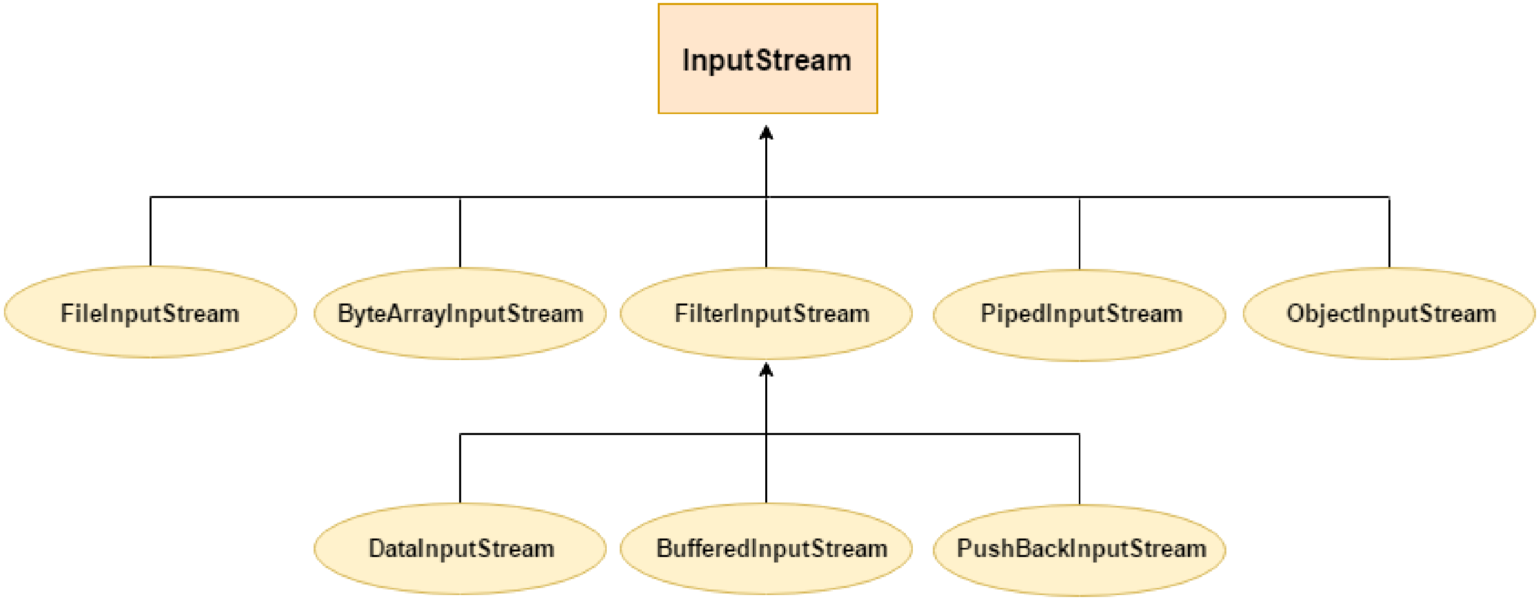
OutputStream Hierarchy



Useful methods of OutputStream Class

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.

InputStream Hierarchy

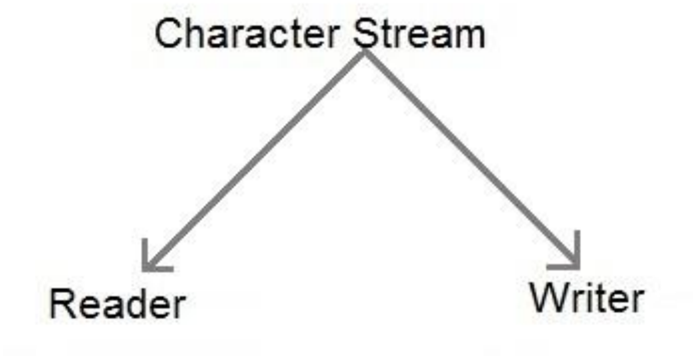


Useful methods of InputStream

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.

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.

Some important Charcter stream classes

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() metho d.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

File handling

JAVA.IO.FILE CLASS IN JAVA

- The File class is Java’s representation of a file or directory path name.
- Because file and directory names have different formats on different platforms, a simple string is not adequate to name them.
- The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either **absolute or relative**.
- **Relative or Absolute?**
  - An absolute path always contains the root element and the complete directory list required to locate the file.
  - For example, C:/home/sally/statusReport is an absolute path.
  - All of the information needed to locate the file is contained in the path string.
  - A relative path needs to be combined with another path in order to access a file.
  - For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.

- First of all, we should create the File class object by passing the filename or directory name to it.
- A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing.
- These restrictions are collectively known as access permissions.
- **Instances of the File class are immutable**; that is, once created, the abstract pathname represented by a File object will never change.

Constructors of File class

- **File(File parent, String child) :**
  - Creates a new File instance from a parent abstract pathname and a child pathname string.
- **File(String pathname) :**
  - Creates a new File instance by converting the given pathname string into an abstract pathname.
- **File(String parent, String child) :**
  - Creates a new File instance from a parent pathname string and a child pathname string.
- **File(URI uri) :**
  - Creates a new File instance by converting the given file: URI into an abstract pathname.

Important methods of file class

Modifier and Type	Method	Description
static File	createTempFile(String prefix, String suffix)	It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name.
boolean	createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean	canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean	isAbsolute()	It tests whether this abstract pathname is absolute.

boolean	isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean	isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String	getName()	It returns the name of the file or directory denoted by this abstract pathname.
String	getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path	toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI	toURI()	It constructs a file: URI that represents this abstract pathname.
File[]	listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long	getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[]	list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
boolean	mkdir()	It creates the directory named by this abstract pathname.

Creating a file

```
import java.io.File;
import java.io.IOException;

public class FileDemo
{
        public static void main(String[] args)
        {
        try
        {
        File Ob = new File("FileDemo.txt");
        if (Ob.createNewFile())
                {
        System.out.println("******File created******");
                System.out.println("Name of the file = " + Ob.getName());
                }
                else
                {
        System.out.println("File already exists.");
                }
        }
        catch (IOException e)
        {
        e.printStackTrace();
        }
        }
}
```

OUTPUT

```
******File created******
Name of the file = FileDemo.txt
```

[Note: New text file will be created in the current directory]

Writing a File- using FileWriter

- FileWriter is meant for writing streams of characters.
- FileWriter creates the output file , if it is not present already.
- **Constructors:**
  - **FileWriter(File file)** – Constructs a FileWriter object given a File object.
  - **FileWriter (File file, boolean append)** – constructs a FileWriter object given a File object.
  - **FileWriter (FileDescriptor fd)** – constructs a FileWriter object associated with a file descriptor.
  - **FileWriter (String fileName)** – constructs a FileWriter object given a file name.
  - **FileWriter (String fileName, Boolean append)** – Constructs a FileWriter object given a file name with a Boolean indicating whether or not to append the data written.

- **Methods:**
  - **public void write (int c) throws IOException –** Writes a single character.
  - **public void write (char [] stir) throws IOException –** Writes an array of characters.
  - **public void write(String str)throws IOException –** Writes a string.
  - **public void write(String str,int off,int len)throws IOException –** Writes a portion of a string. Here off is offset from which to start writing characters and len is number of character to write.
  - **public void flush() throws IOException** flushes the stream
  - **public void close() throws IOException** flushes the stream first and then closes the writer.

Example- FileWriter

```
import java.io.FileWriter;
import java.io.IOException;

public class FileDemo
{
 public static void main(String[] args)
{
  try
{
FileWriter obj = new FileWriter("FileDemo.txt");
obj.write("Welcome");
obj.close();
System.out.println("File is Updated.");
         }
   catch (IOException e)
         {
e.printStackTrace();
         }
 }
}
```

Output

```
File is Updated
```

[File “FileDemo.txt” now contains the string Welcome]

Writing a File-using FileOutputStream

- The **FileOutputStream** is a byte output stream class that provides methods for writing bytes to a file.
- We can create an instance of this class by supplying a File or a path name, and/or specify to overwrite or append to an existing file, using the following constructors:
  - FileOutputStream(File file)
  - FileOutputStream(File file, boolean append): if append is true, then the bytes will be written to the end of an existing file rather than the beginning.
  - FileOutputStream(String name)
  - FileOutputStream(String name, boolean append)

- And the following list describes the key methods implemented by **FileOutputStream** class:
  - **void write(int)**: writes the specified byte to the output stream.
  - **void write(byte[])**: writes the specified array of bytes to the output stream.
  - **void write(byte[], int offset, int length)** : writes length bytes from the specified byte array starting at offset to the output stream.
  - **void close()** : Closes this file output stream and releases any system resources associated with the stream.
- Almost methods throw IOException so remember to handle or declare to throw it in your code.

Example-FileOutputStream

```
import java.io.FileOutputStream;
public class Example {
  public static void main(String args[]){
       try{
         FileOutputStream fout=new
FileOutputStream("C:\\Users\\Rose\\Desktop\\Online\\a.txt");
         String s=“File Content";
         byte b[]=s.getBytes();//converting string into byte array
         fout.write(b);
         fout.close();
         System.out.println(“Success...");
        }catch(Exception e){System.out.println(e);}
    }
}
```

Output

- Success…

Reading from File- using FileReader

- FileReader is meant for reading streams of characters.
- **Constructors:**
  - **FileReader(File file) –** Creates a FileReader , given the File to read from
  - **FileReader(FileDescripter fd)** – Creates a new FileReader , given the FileDescripter to read from
  - **FileReader(String fileName)** – Creates a new FileReader , given the name of the file to read from

- **Methods:**
  - **public int read () throws IOException –**
    - Reads a single character.
  - **public int read(char[] cbuff) throws IOException –**
    - Reads characters into an array.
  - **public abstract int read(char[] buff, int off, int len) throws IOException –**
    - Reads characters into a portion of an array.
      Parameters:
      buf – Destination buffer
      off – Offset at which to start storing characters
      len – Maximum number of characters to read
  - **public void close() throws IOException**
    - closes the reader.
  - **public long skip(long n) throws IOException –**
    - Skips characters.
    - Parameters:
      n – The number of characters to skip

Example: FileReader

```
import java.io.*;
public class ReadFile
{
 public static void main(String[] args) throws Exception
 {
  // pass the path to the file as a parameter
  FileReader fr =
   new FileReader("C:\\Users\\Rose\\Desktop\\Online\\a.txt");

  int i;
  while ((i=fr.read()) != -1)
   System.out.print((char) i);
 }
}
```

Output

File Content

USING BUFFEREDREADER:

- This method reads text from a character-input stream.
- It does buffering for efficient reading of characters, arrays, and lines.
- The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.
- In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.
- It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders.

Example-BufferedReader and FileReader

```
import java.io.*;
public class ReadFile
{
  public static void main(String[] args)throws Exception
  {
  // We need to provide file path as the parameter:
  // double backquote is to avoid compiler interpret words
  // like \test as \t (ie. as a escape sequence)
  File file = new File("C:\\Users\\Rose\\Desktop\\Online\\a.txt");

  BufferedReader br = new BufferedReader(new FileReader(file));

  String st;
  while ((st = br.readLine()) != null)
    System.out.println(st);
  }
}
```

Output

File Content

Reading from file-Scanner class

- A simple text scanner which can parse primitive types and strings using regular expressions.
- A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace.
- The resulting tokens may then be converted into values of different types using the various next methods.

Example-Reading file using scanner class

```
import java.io.*;
import java.util.Scanner;
public class FileDemo
{
  public static void main(String[] args)
                {
                try
                {
                                File Obj = new File("FileDemo.txt");
                                Scanner obj1 = new Scanner(Obj);
                                while (obj1.hasNextLine())
                                {
                String obj2 = obj1.nextLine();
                System.out.println(obj2);
                                }
                                obj1.close();
                }
                catch (FileNotFoundException e)
                {
                                e.printStackTrace();
                }
                }
}
```

Output

```
Welcome
```

Reading from file-FileInputStream class

- The **FileInputStream** is a byte input stream class that provides methods for reading bytes from a file.
- We can create an instance of this class by supplying a File or a path name, using these two constructors:
  - FileInputStream(File file)
  - FileInputStream(String name)

- The following list describes the key methods implemented by **FileInputStream** class:
  - **int available()**: returns an estimate of the number of remaining bytes that can be read.
  - **int read()**: reads one byte of data, returns the byte as an integer value. Return -1 if the end of the file is reached.
  - **int read(byte[])**: reads a chunk of bytes to the specified byte array, up to the size of the array. This method returns -1 if there's no more data or the end of the file is reached.
  - **int read(byte[], int offset, int length)**: reads up to length bytes of data from the input stream.
  - **long skip(long n)**: skips over and discards n bytes of data from the input stream. This method returns the actual number of bytes skipped.
  - **void close()**: Closes this file input stream and releases any system resources associated with the stream.

Copying a file

```
import java.io.*;
public class FileCopyDemo
{
    public static void main(String[] args)
    {
                FileInputStream a = null;
                FileOutputStream b = null;
                try
                {
                    File obj_in =new File("FileDemo.txt");
                    File obj_out =new File("FileDemo1.txt");

                    a = new FileInputStream(obj_in);
                    b = new FileOutputStream(obj_out);
                byte[] buffer = new byte[1024];
                int length;
                    while ((length = a.read(buffer)) > 0)
                                    {
                                    b.write(buffer, 0, length);
                                    }
                a.close();
                b.close();
                System.out.println("File copied successfully!!");
                }
catch(IOException e)
                    {
                                        e.printStackTrace();
                     }
   }
}
```

- Output

```
File Copied Successfully
```

Serialization and Deserialization in Java

Serialization and Deserialization

- **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.

Serializable Interface

- We must have to implement the *Serializable* interface for serializing the object.
- Serializable is a marker interface (has no data member and method).
- 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.

WRITEOBJECT() AND READOBJECT(

- The ObjectOutputStream class contains **writeObject()** method for serializing an Object.
- Syntax:
  - public final void writeObject(Object obj) throws IOException
- The ObjectInputStream class contains **readObject()** method for deserializing an object.
- Syntax:
  - public final Object readObject() throws IOException, ClassNotFoundException

Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network.

Points to remember

1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.

2. Only non-static data members are saved via Serialization process.

3. Static data members and transient data members are not saved via Serialization process.So, if you don't want to save value of a non-static data member then make it **transient**.

4. Constructor of object is never called when an object is deserialized.

5. Associated objects must be implementing Serializable interface.

SerialVersionUID

- The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization.
- If the reciever has loaded a class for the object that has different UID than that of corresponding sender's class, the Deserialization will result in an **InvalidClassException**.
- A Serializable class can declare its own UID explicitly by declaring a field name.
- It must be static, final and of type long. i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;
- If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification.

Example-Serialization and DeSerialization

```
import java.io.*;

class Demo implements java.io.Serializable
{
    public int a;
    public String b;

    // Default constructor
    public Demo(int a, String b)
    {
        this.a = a;
        this.b = b;
    }

}
```

```
class Test
{
    public static void main(String[] args)
    {
        Demo object = new Demo(1, “Hello");
        String filename = "file.ser";
        // Serialization
        try
        {
            //Saving of object in a file
            FileOutputStream file = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(file);
            // Method for serialization of object
            out.writeObject(object);
            out.close();
            file.close();
            System.out.println("Object has been serialized");
        }
        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }
Demo object1 = null;
        // Deserialization
        try
        {
            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);
            // Method for deserialization of object
            object1 = (Demo)in.readObject();
            in.close();
            file.close();
            System.out.println("Object has been deserialized ");
            System.out.println("a = " + object1.a);
            System.out.println("b = " + object1.b);
        }
        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }
        catch(ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException is caught");
        }
    }
}
```

Output

```
Object has been serialized
Object has been deserialized
a = 1
b = Hello
```

EXAMPLE2

```
import java.io.*;

class Emp implements Serializable {
private static final long serialversionUID =
                                 129348938L;
    transient int a;
    static int b;
    String name;
    int age;

    // Default constructor
public Emp(String name, int age, int a, int b)
    {
        this.name = name;
        this.age = age;
        this.a = a;
        this.b = b;
    }

}
```

```
public class SerialExample {
public static void printdata(Emp object1)
  {
      System.out.println("name = " + object1.name);
      System.out.println("age = " + object1.age);
      System.out.println("a = " + object1.a);
      System.out.println("b = " + object1.b);
  }
public static void main(String[] args)
  {
      Emp object = new Emp("ab", 20, 2, 1000);
      String filename = "shubham.txt";
      // Serialization
      try {
        // Saving of object in a file
        FileOutputStream file = new FileOutputStream
                               (filename);
        ObjectOutputStream out = new ObjectOutputStream
                               (file);
        // Method for serialization of object
        out.writeObject(object);
        out.close();
        file.close();
        System.out.println("Object has been serialized\n"
                  + "Data before Deserialization.");
        printdata(object);
        // value of static variable changed
        object.b = 2000;
      }
```

```
 catch (IOException ex) {
      System.out.println("IOException is caught");
    }
object = null;
    // Deserialization
    try {
      // Reading the object from a file
      FileInputStream file = new FileInputStream
                    (filename);
      ObjectInputStream in = new ObjectInputStream
                    (file);
      // Method for deserialization of object
      object = (Emp)in.readObject();
      in.close();
      file.close();
      System.out.println("Object has been deserialized\n"
                + "Data after Deserialization.");
      printdata(object);
         }
    catch (IOException ex) {
      System.out.println("IOException is caught");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("ClassNotFoundException" +
                " is caught");
    }
  }
}
```

OUTPUT

```
Object has been serialized
Data before Deserialization.
name = ab
age = 20
a = 2
b = 1000
Object has been deserialized
Data after Deserialization.
name = ab
age = 20
a = 0
b = 2000
```