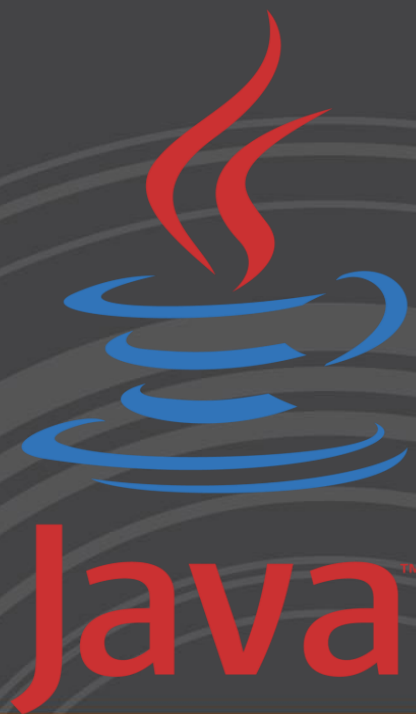


Object Oriented Programming



Java™



Unit 6



File Handling

Getting information about files and folders: File Class in JAVA File Class to Create Files and Directories: Create New File.

Overview of streams API in JAVA: I/O Streams in JAVA, Character Streams, Implement Buffered OutputStream, Reading from Console, Implement Scanner.

Serialization: Serialization, Storing objects via serialization Metadata and File attributes: Metadata and File attributes.

Basic File Attribute: Basic File Attribute File Visitor Interface: File Visitor Interface, Random Access File.



File Handling

Getting information about files and folders: File Class in JAVA
Create Files and Directories: Create New File.

File class in Java

- The **java.io.File** class is an abstract representation of file and directory pathnames.
- 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.
- A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

File a = new File("<FILE_PATH>");

defines an abstract file name for the file in directory
<FILE_PATH>. This is an absolute abstract file name.

File class in Java

Method	Type	Description
<code>canRead()</code>	Boolean	Tests whether the file is readable or not
<code>canWrite()</code>	Boolean	Tests whether the file is writable or not
<code>createNewFile()</code>	Boolean	Creates an empty file
<code>delete()</code>	Boolean	Deletes a file
<code>exists()</code>	Boolean	Tests whether the file exists
<code>getName()</code>	String	Returns the name of the file
<code>getAbsolutePath()</code>	String	Returns the absolute pathname of the file
<code>length()</code>	Long	Returns the size of the file in bytes
<code>list()</code>	String[]	Returns an array of the files in the directory
<code>mkdir()</code>	Boolean	Creates a directory

Example

```
package filehandling;
```

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

```
import java.util.Date;
```

```
public class Demo1_DirectoryList {
```

```
    public static void main(String args[]) {
```

```
        File file = new File("D:\\TEACHER"); //Displays all files & folders under mentioned  
        directory
```

```
        String[] fileList = file.list();
```

```
        for(String name:fileList){
```

```
            System.out.println(name);
```

```
        }
```

```
    }
```

```
}
```

Example

```
File f=new File("D:\\javaprogram\\DemoJ.txt");
if(f.exists())
{
    System.out.println("File exists");
    System.out.println(f.getAbsolutePath());
}
else
{
    System.out.println("Does not exists");
    if(f.createNewFile())
    {
        System.out.println("File created");
    }
}
```



Overview of streams API in JAVA: I/O Streams in JAVA, Character Streams, Implement Buffered OutputStream, Reading from Console, Implement Scanner.

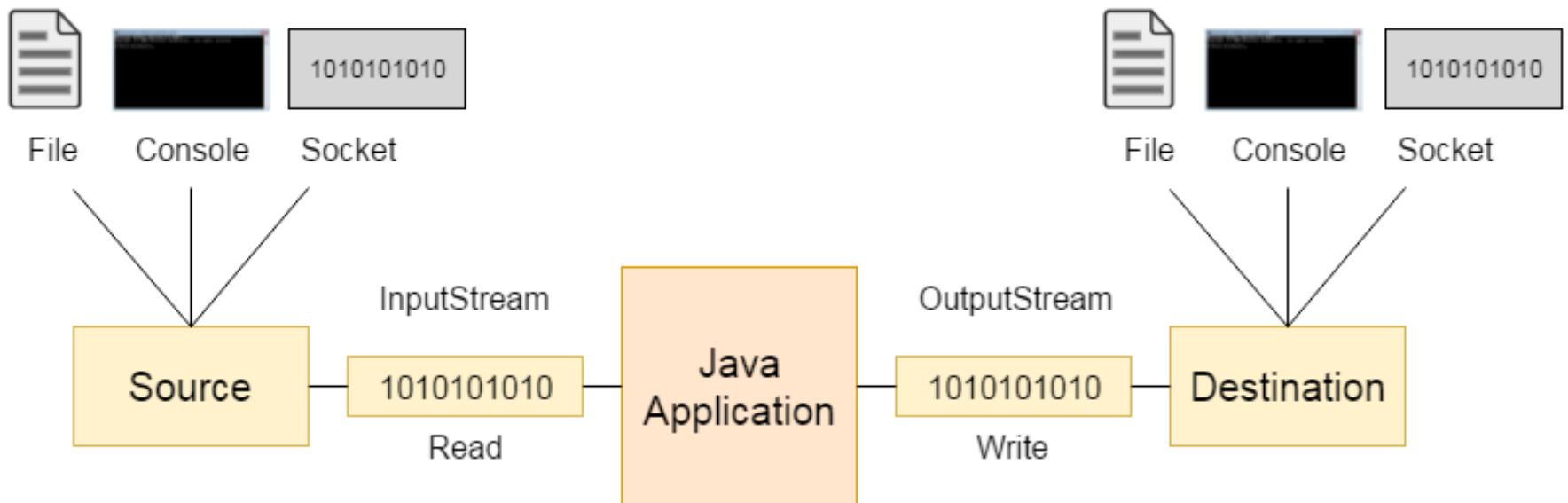
Streams

- A stream is a **sequence of data**. In Java a stream is **composed of bytes**. It is an intermediate between program and I/O devices(information).
 - Java encapsulates Stream under **java.io package**. Java defines two types of streams. They are:
 - **Byte Stream:** It provides a convenient means for handling input and output of byte. Used for reading binary data.
 - **Character Stream:** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and used for text.
 - In java, three streams are created for us automatically. All these streams are attached with console.
- 1) **System.out:** standard output stream. Is a PrintStream.
 - 2) **System.in:** standard input stream. Is an InputStream.
 - 3) **System.err:** standard error stream. Is a PrintStream.

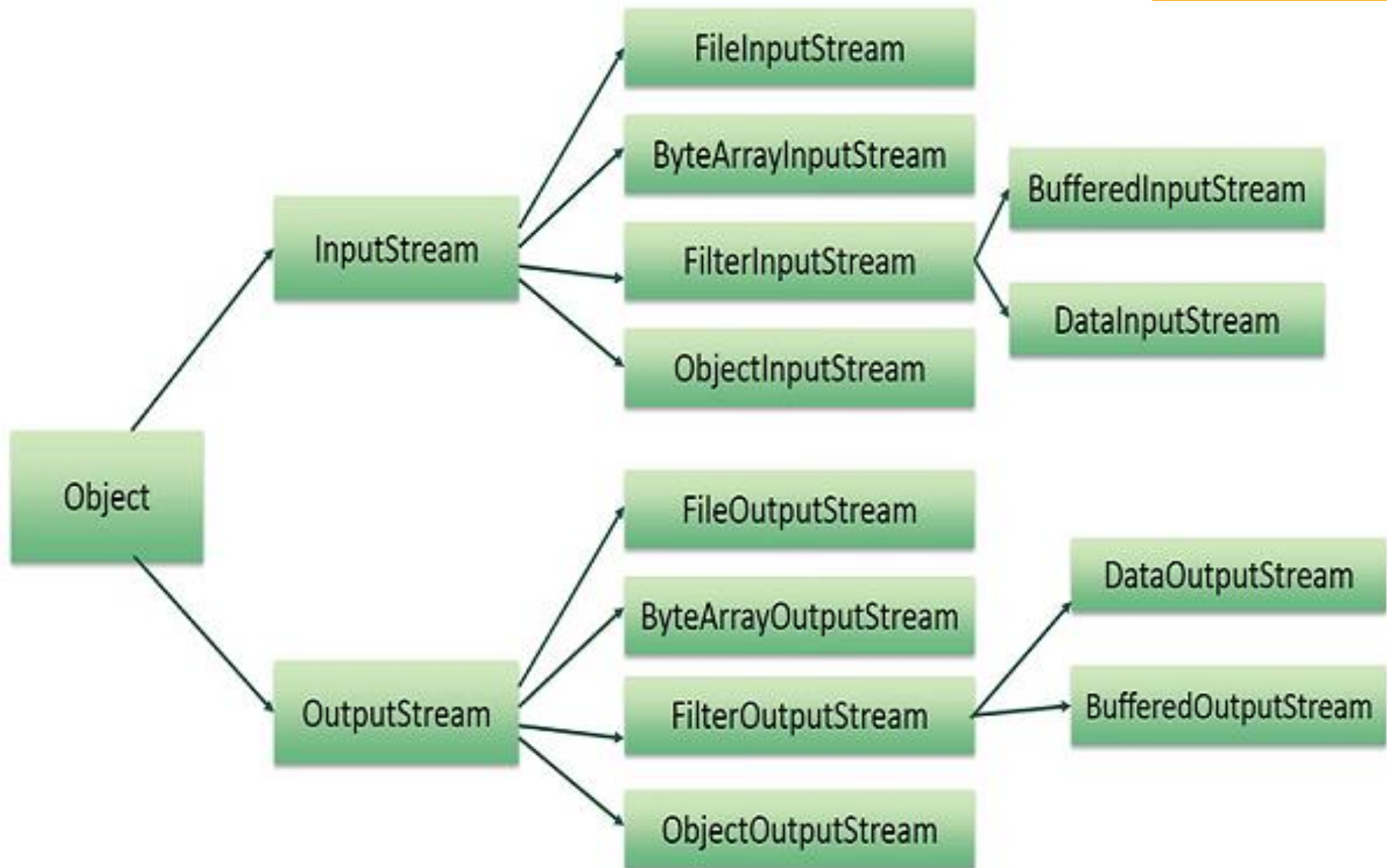
```
System.out.println("simple message")  
System.err.println("error message")
```

Byte Stream

- **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.
- These classes defines two important methods: **read()** and **write()** used to read and write the data in the form of bytes.



Byte Stream Classes

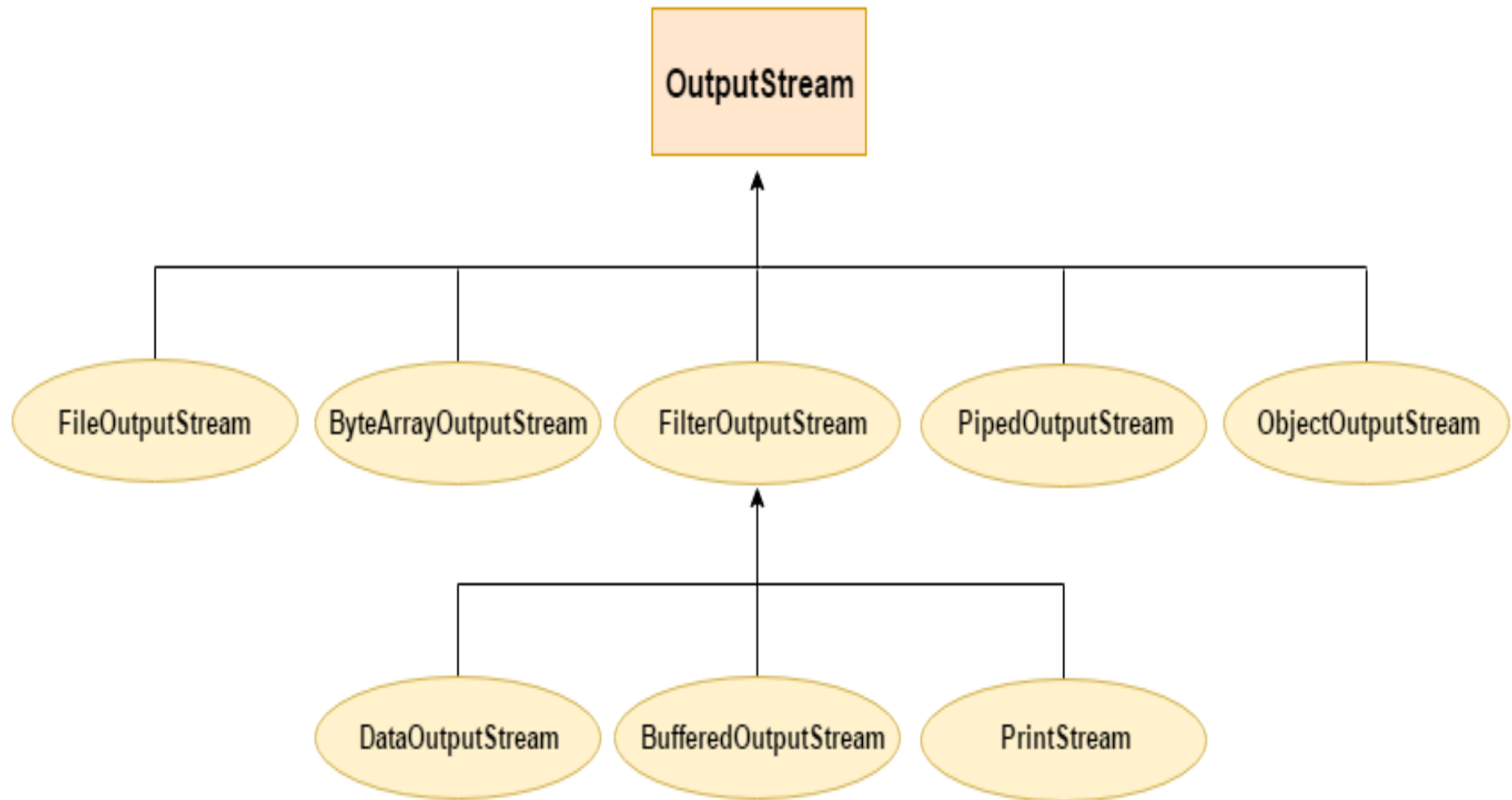


OutputStream class

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

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

OutputStream Hierarchy

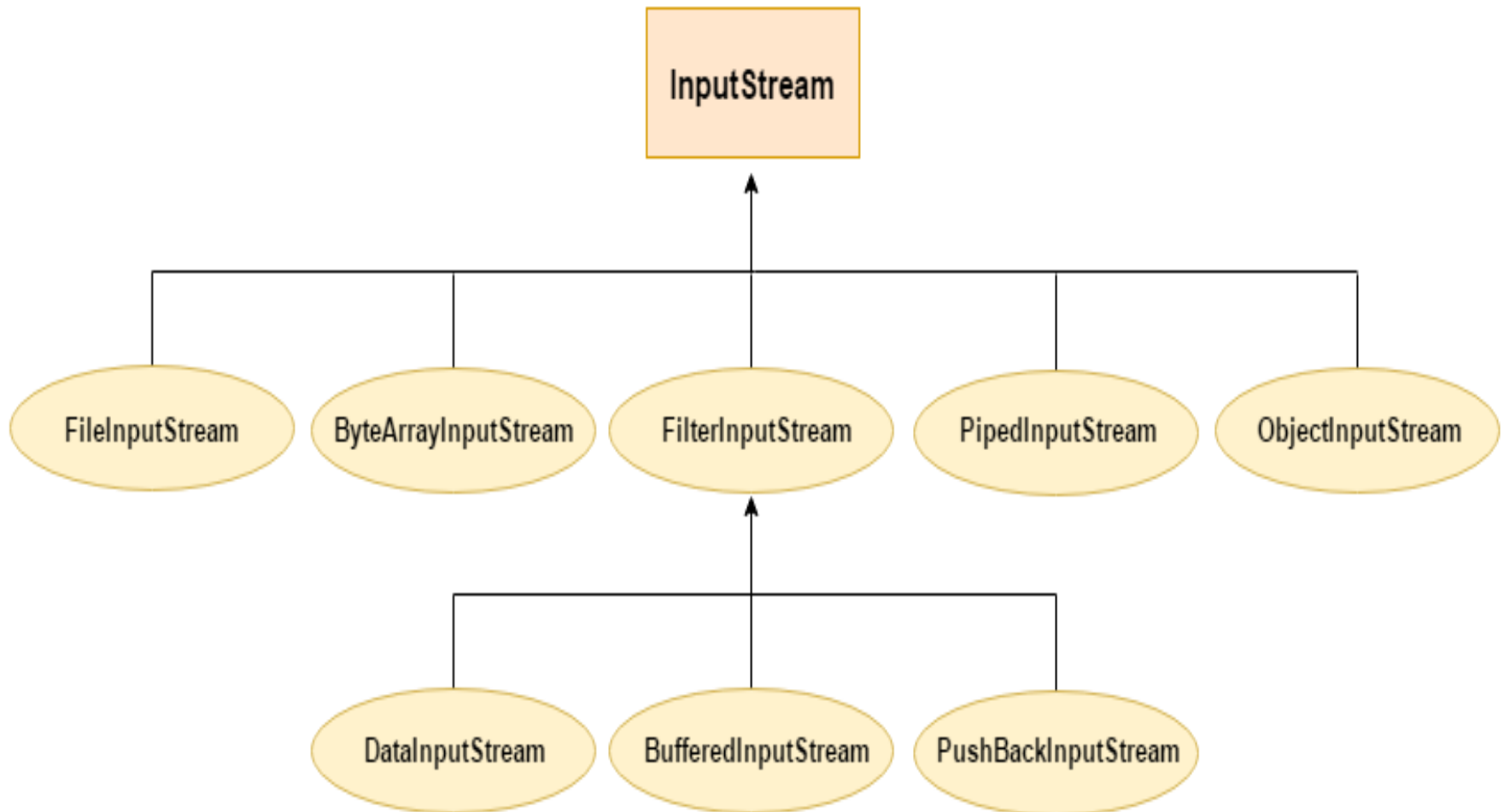


InputStream class

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

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 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



Reading Console Input

1. Using Buffered Reader Class

- This is the Java classical method to take input, introduced in JDK1.0.
- 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.

▪ Advantages

The input is buffered for efficient reading.

▪ Drawback:

The wrapping code is hard to remember.

```
// Java program to demonstrate BufferedReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //Enter data using BufferedReader
        InputStreamReader is=new InputStreamReader(System.in);
        BufferedReader reader = new BufferedReader(is);

        // Reading data using readLine
        String name = reader.readLine();

        // Printing the read line
        System.out.println(name);
    }
}
```


Reading Console Input

2. Using Scanner Class(JDK 1.5)

- 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**.

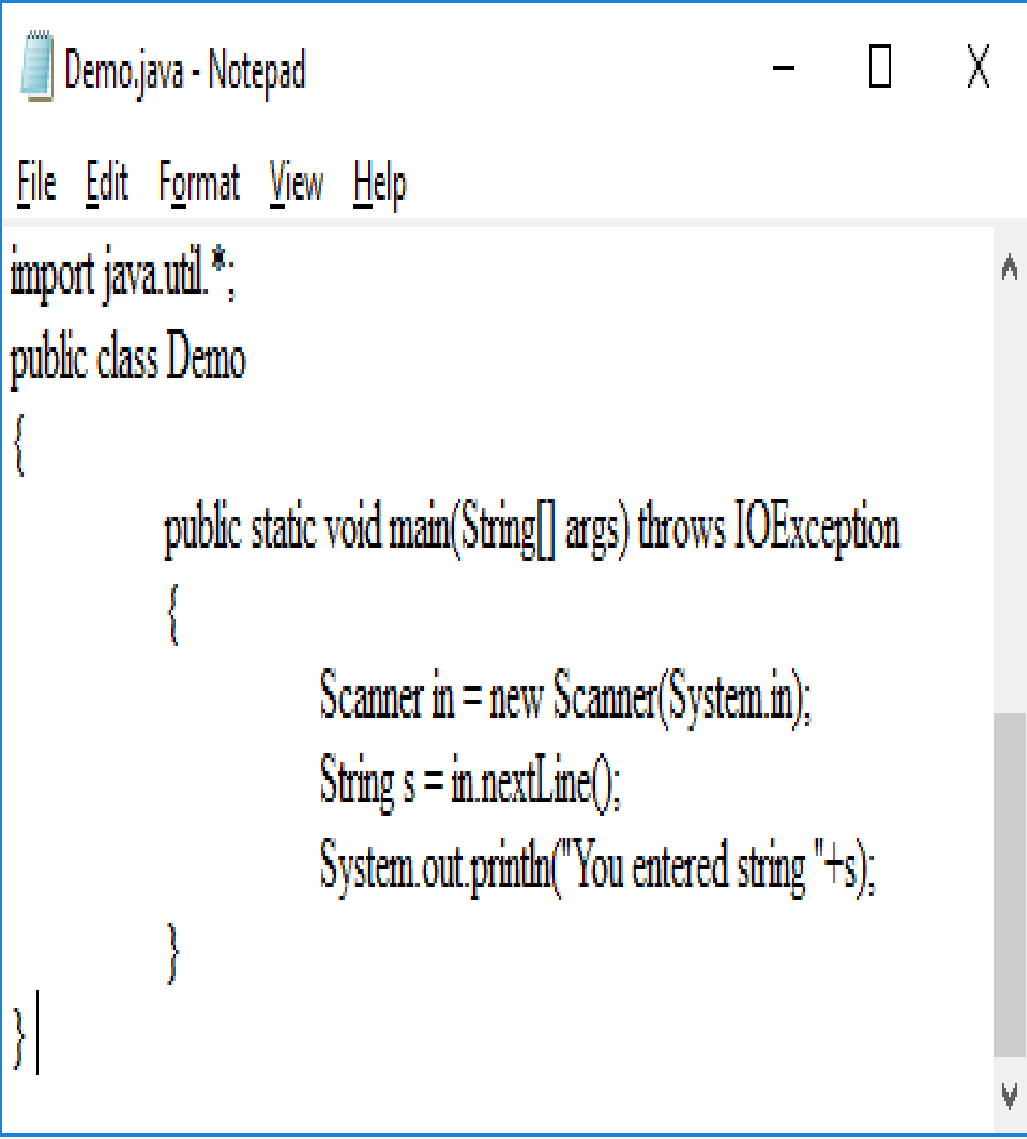
- **Advantages:**

- Convenient methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input.

- Regular expressions can be used to find tokens.

- **Drawback:**

The reading methods are non synchronized



```
import java.util.*;

public class Demo
{
    public static void main(String[] args) throws IOException
    {
        Scanner in = new Scanner(System.in);
        String s = in.nextLine();
        System.out.println("You entered string "+s);
    }
}
```

Reading Console Input

3. Using Console Class(JDK 1.6)

- It has been becoming a preferred way for reading user's input from the command line.
- In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like `System.out.printf()`).
- **Advantages:**
 - Reading password without echoing the entered characters.
 - Reading methods are synchronized.
 - Format string syntax can be used.

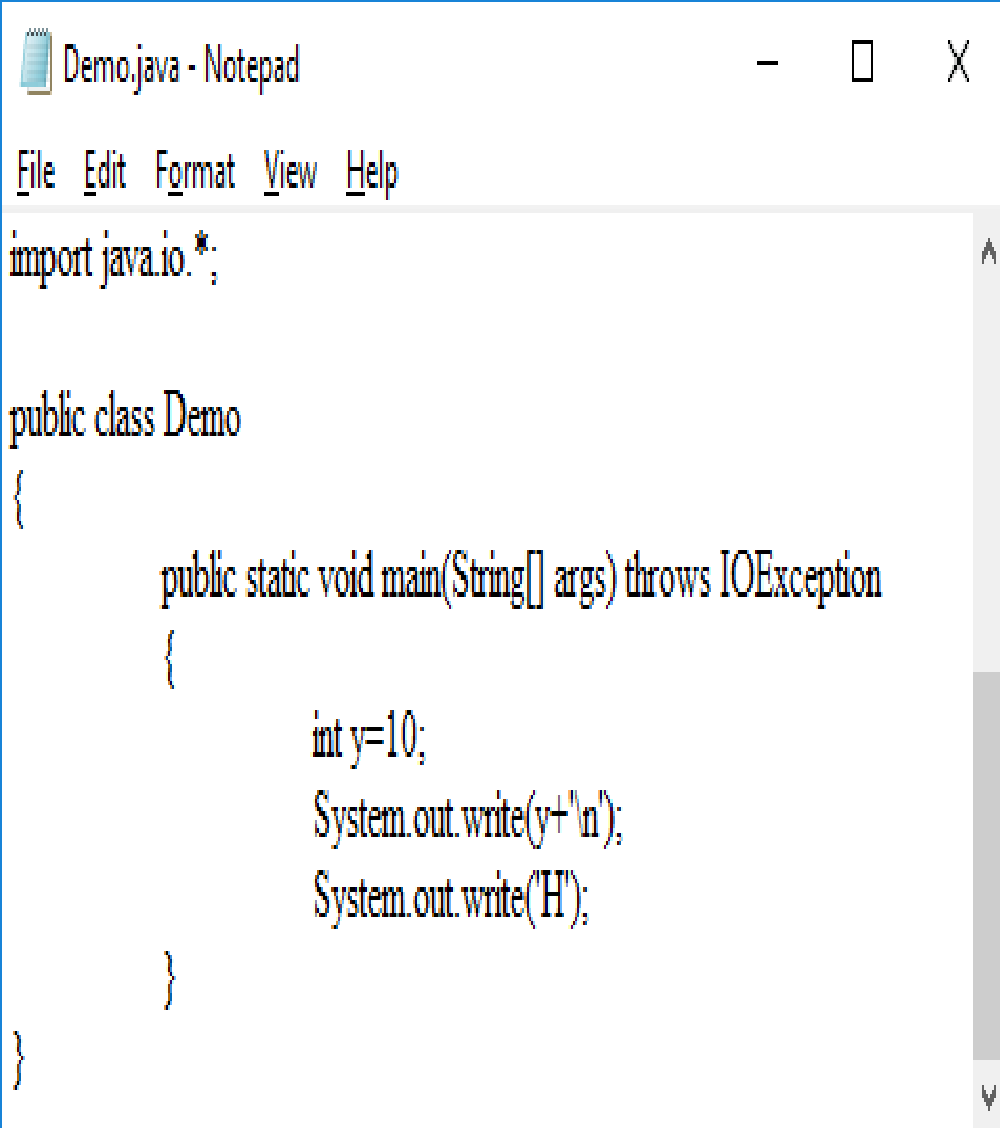
```
public class Sample
{
    public static void main(String[] args)
    {
        // Using Console to input data from user
        String name = System.console().readLine();

        System.out.println(name);
    }
}
```

Writing Console Output

1. Using write() method

- **PrintStream** is an output stream derived from **OutputStream**, it also implements the low-level method **write()**.
- Thus, **write()** can be used to write to the console.



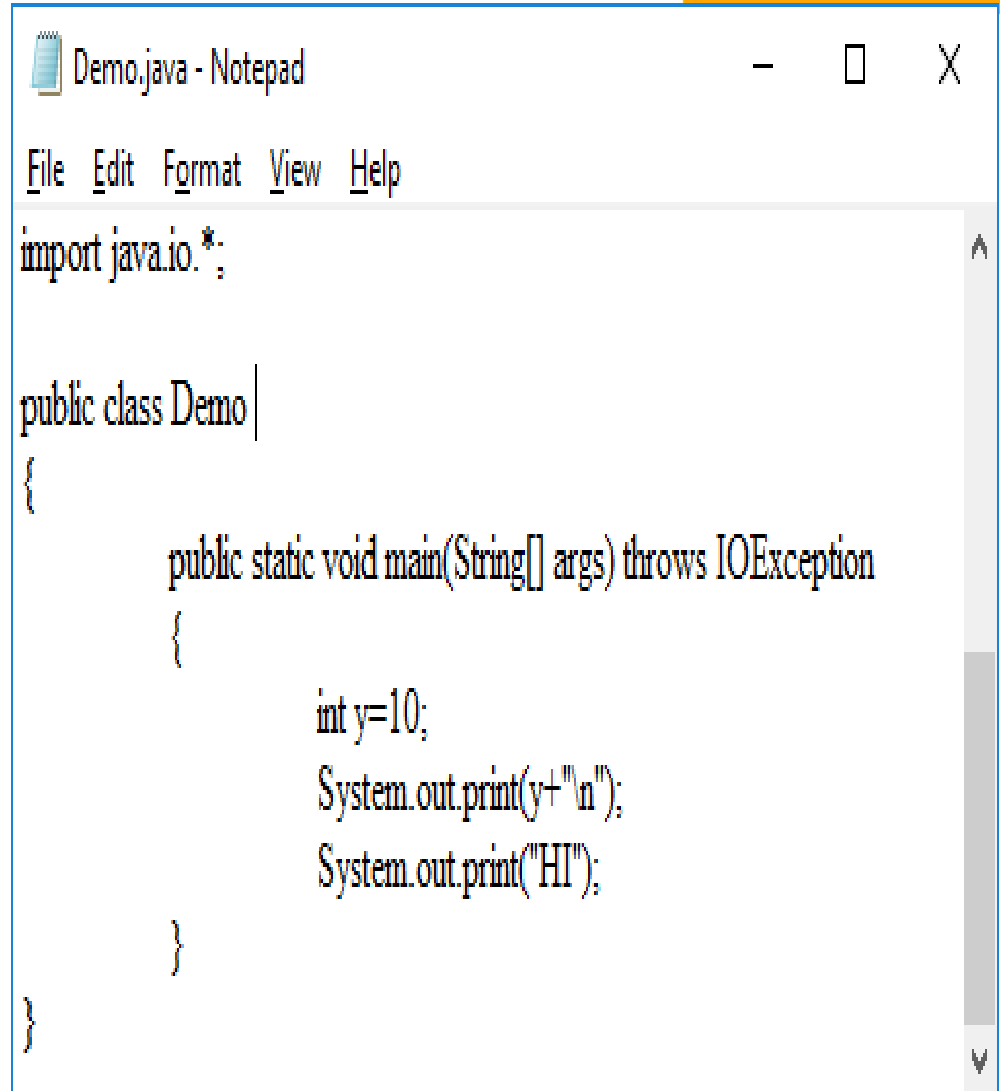
```
import java.io.*;

public class Demo
{
    public static void main(String[] args) throws IOException
    {
        int y=10;
        System.out.write(y+'n');
        System.out.write('H');
    }
}
```

Writing Console Output

2. Using print() method

- **print()** method belongs to **PrintWriter** class and can be used to write a string and a newline character to the console.



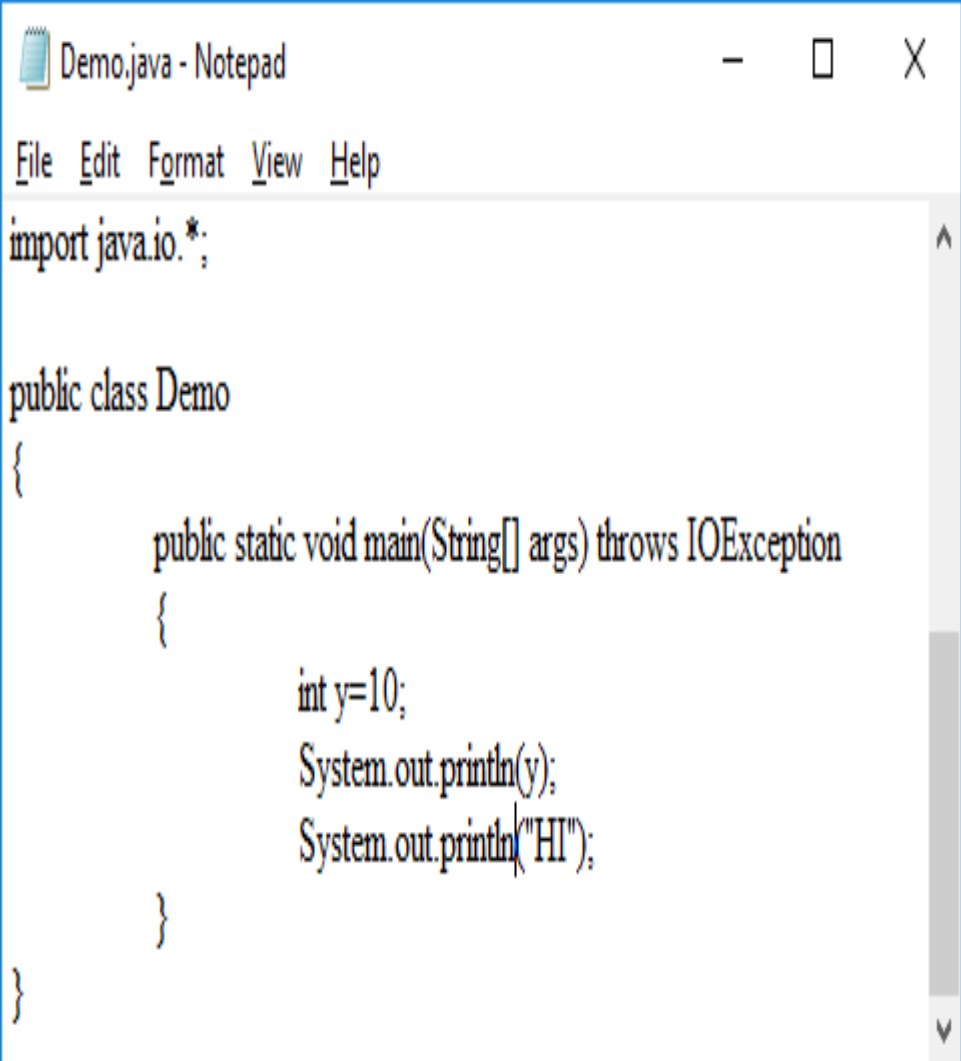
```
import java.io.*;

public class Demo
{
    public static void main(String[] args) throws IOException
    {
        int y=10;
        System.out.print(y+"\n");
        System.out.print("HI");
    }
}
```

Writing Console Output

3. Using println() method

- **println()** method belongs to **PrintWriter** class and can be used to directly write a separate line in console output .i.e. this method directly comes with newline character.



```
import java.io.*;

public class Demo
{
    public static void main(String[] args) throws IOException
    {
        int y=10;
        System.out.println(y);
        System.out.println("HI");
    }
}
```

Writing Files using Byte Stream Classes

- Java **FileOutputStream**(**java.io.FileOutputStream** class) is an output stream used for writing data to a file.
- If you have to write primitive values into a file, use **FileOutputStream** class. You can write byte-oriented as well as character-oriented data through **FileOutputStream** class.
- But, for character-oriented data, it is preferred to use **FileWriter** than **FileOutputStream**.
- The declaration for **java.io.FileOutputStream** class:

public class FileOutputStream extends OutputStream

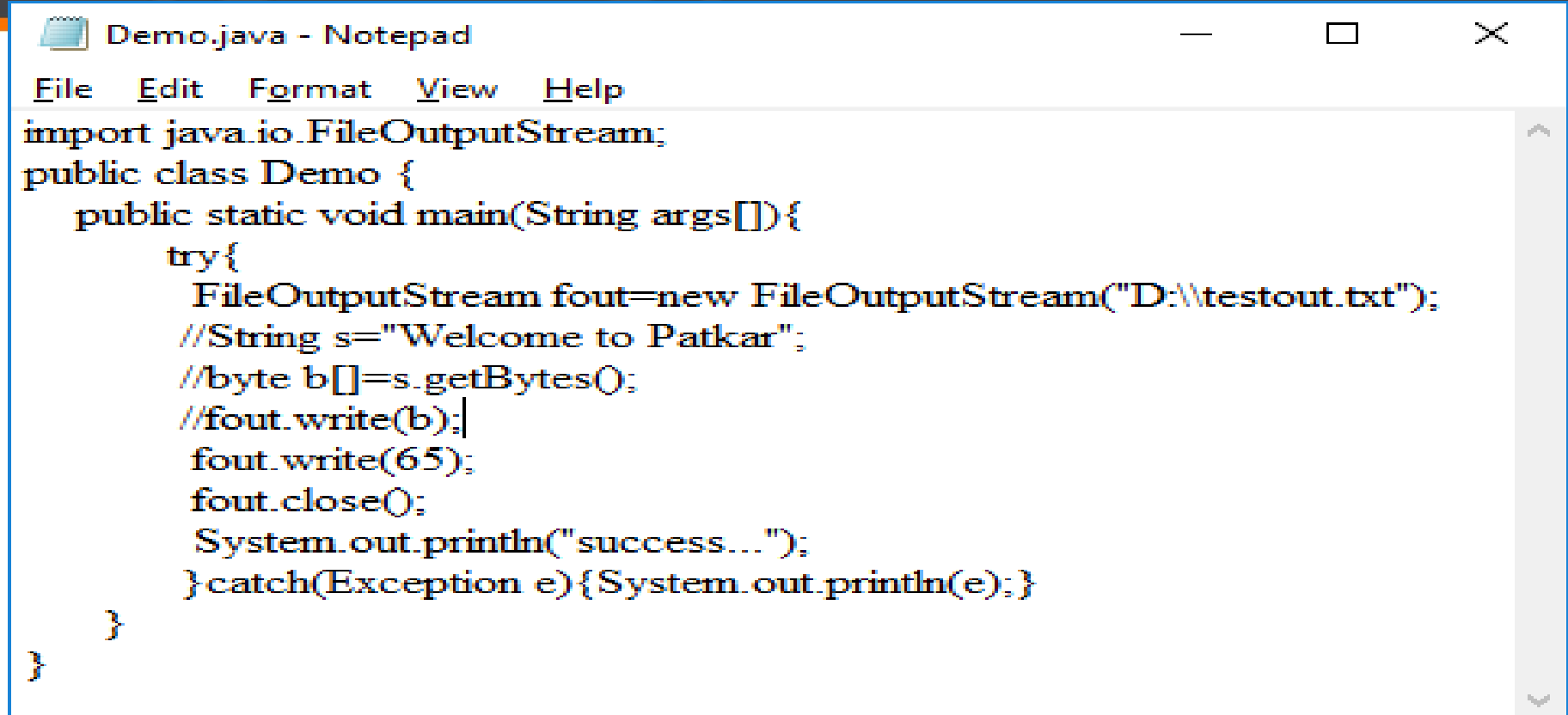
- **Constructors**
- **FileOutputStream(File f)**
- **FileOutputStream(File f, Boolean b)**
- **FileOutputStream(FileDescriptor obj)**
- **FileOutputStream(String path)**
- **FileOutputStream(Boolean append)**

Writing Files using Byte Stream Classes

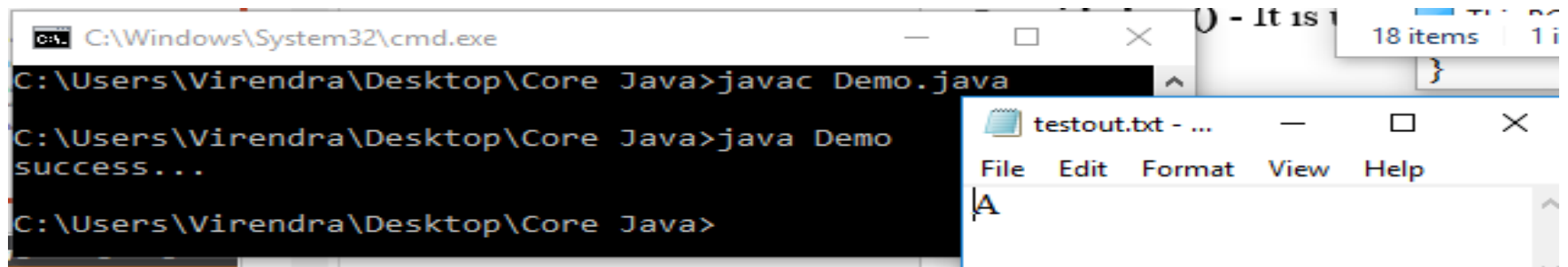
FileOutputStream class methods

- Method Description
- **protected void finalize()** - It is used to clean up the connection with the file output stream.
- **void write(byte[] ary)** - It is used to write ary.length bytes from the byte array to the file output stream.
- **void write(int b)** - It is used to write the specified byte to the file output stream.
- **FileDescriptor getFD()** - It is used to return the file descriptor associated with the stream.
- **void close()** - It is used to closes the file output stream.

Writing Files using Byte Stream Classes



```
import java.io.FileOutputStream;
public class Demo {
    public static void main(String args[]){
        try{
            FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
            //String s="Welcome to Patkar";
            //byte b[]=s.getBytes();
            //fout.write(b);
            fout.write(65);
            fout.close();
            System.out.println("success...");
        }catch(Exception e){System.out.println(e);}
    }
}
```



cmd C:\Windows\System32\cmd.exe

```
C:\Users\Virendra\Desktop\Core Java>javac Demo.java
C:\Users\Virendra\Desktop\Core Java>java Demo
success...
C:\Users\Virendra\Desktop\Core Java>
```

testout.txt - ...

File Edit Format View Help

A

Reading File using Byte Stream Classes

- Java **FileInputStream** class(**java.io.FileInputStream** class) obtains **input bytes from a file**.
- It is used for reading **byte-oriented data** (streams of raw bytes) such as image data, audio, video etc.
- You can also read character-stream data. But, for reading streams of characters, it is recommended to use **FileReader** class.
- The declaration for **java.io.FileInputStream** class:

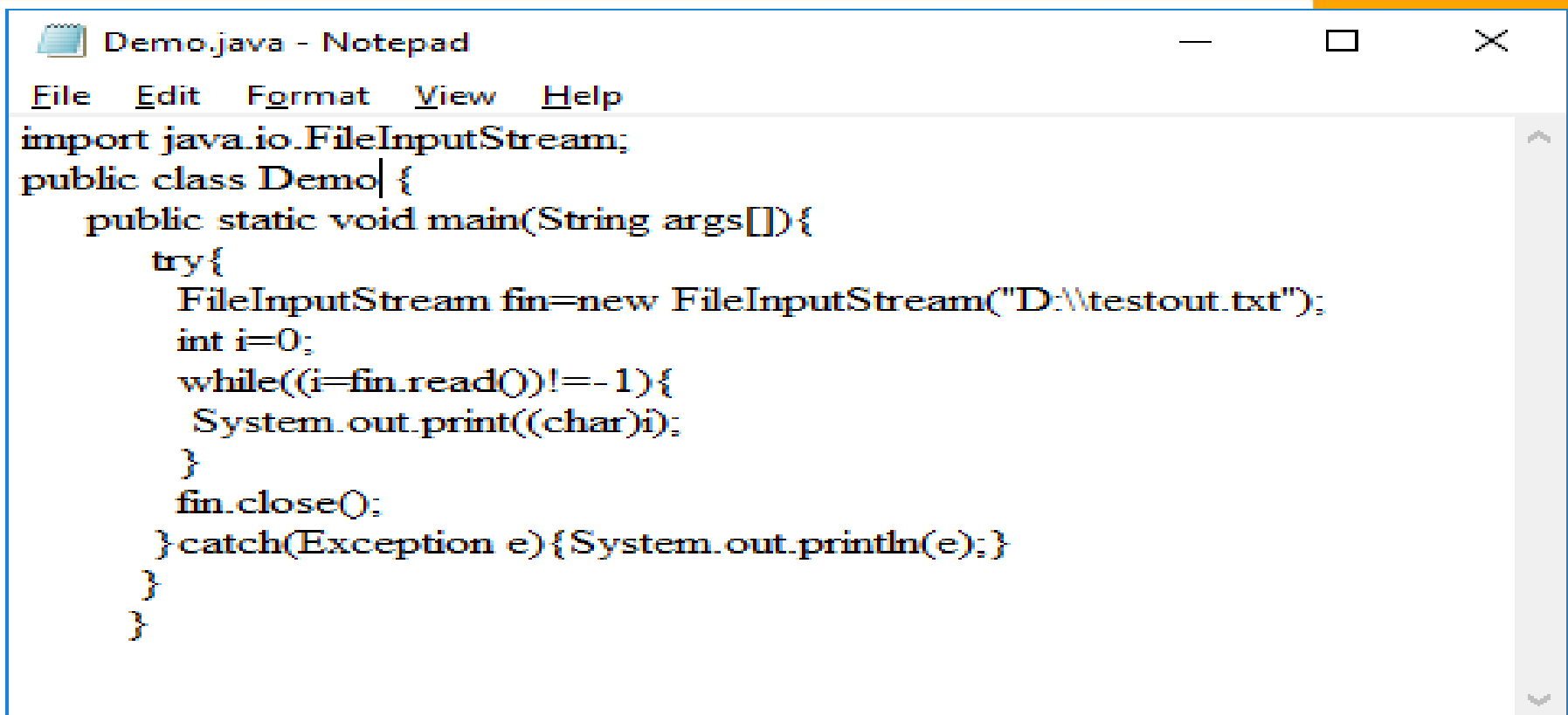
public class FileInputStream extends InputStream

- **Constructors**
- **FileInputStream(File f)**
- **FileInputStream(FileDescriptor obj)**
- **FileInputStream(String path)**

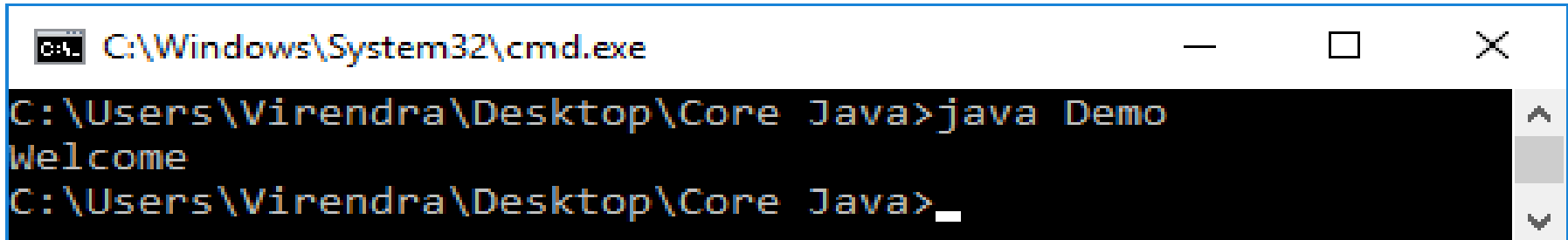
Reading File using Byte Stream Classes

- Java FileInputStream class methods
- Method Description
- **int available()** - It is used to return the estimated number of bytes that can be read from the input stream.
- **int read()** - It is used to read the byte of data from the input stream.
- **long skip(long x)** - It is used to skip over and discards x bytes of data from the input stream.
- **protected void finalize()** - It is used to ensure that the close method is called when there is no more reference to the file input stream.
- **void close()** - It is used to closes the stream.

Reading File using Byte Stream Classes



```
File Edit Format View Help
import java.io.FileInputStream;
public class Demo {
    public static void main(String args[]) {
        try {
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            int i=0;
            while((i=fin.read())!=-1){
                System.out.print((char)i);
            }
            fin.close();
        } catch (Exception e) { System.out.println(e); }
    }
}
```



```
C:\Windows\System32\cmd.exe
C:\Users\Virendra\Desktop\Core Java>java Demo
Welcome
C:\Users\Virendra\Desktop\Core Java>_
```

Writing Files using Character Stream Classes

- Java **FileWriter** class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.
- Unlike **FileOutputStream** class, you don't need to convert string into byte array because it provides method to write string directly.
- The declaration for **Java.io.FileWriter** class:

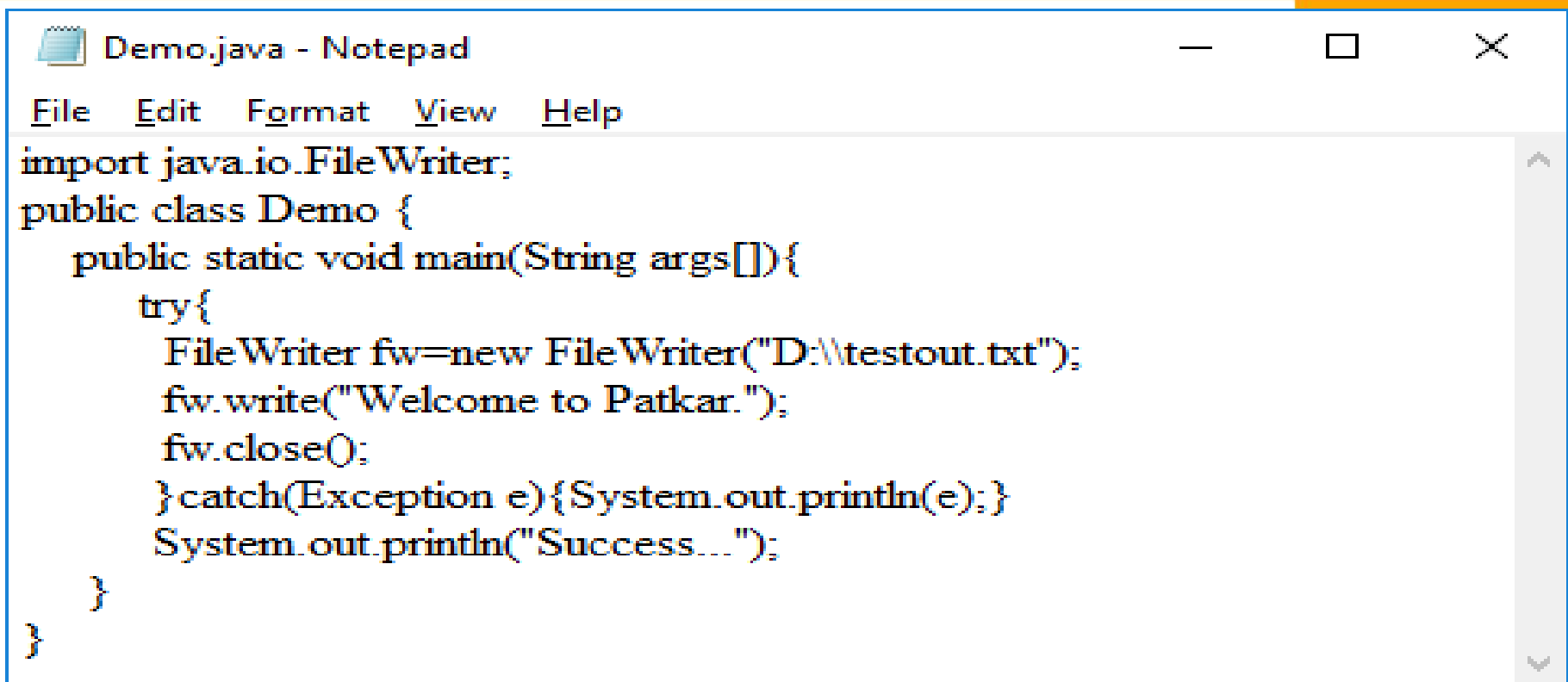
public class FileWriter extends OutputStreamWriter

- **Constructors of FileWriter class**
- **FileWriter(String f)** - Creates a new file. It gets file name in string.
- **FileWriter(File f)** - Creates a new file. It gets file name in File object.
- **FileWriter (File f, Boolean b)**
- **FileWriter(FileDescriptor obj)**
- **FileWriter(Boolean append)**

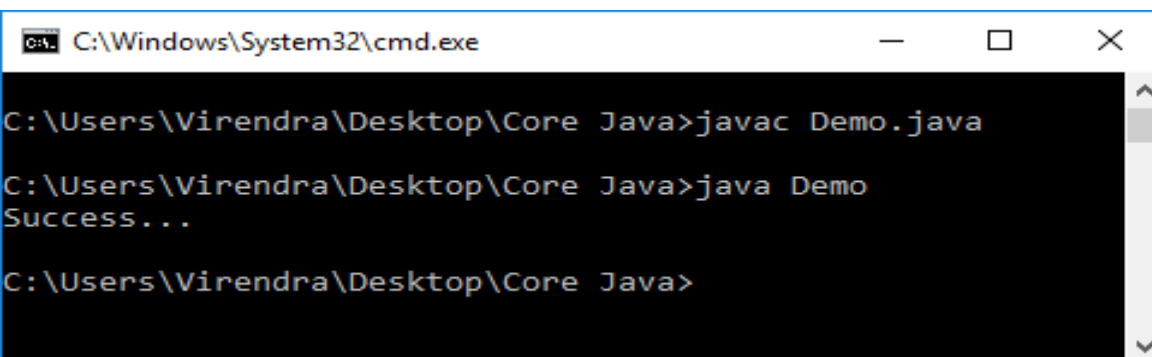
Writing Files using Character Stream Classes

- Methods of `FileWriter` class
- Method Description
- **`void write(String text)`** - It is used to write the string into `FileWriter`.
- **`void write(char c)`** - It is used to write the char into `FileWriter`.
- **`void write(char[] c)`** - It is used to write char array into `FileWriter`.
- **`void flush()`** - It is used to flushes the data of `FileWriter`.
- **`void close()`** - It is used to close the `FileWriter`.

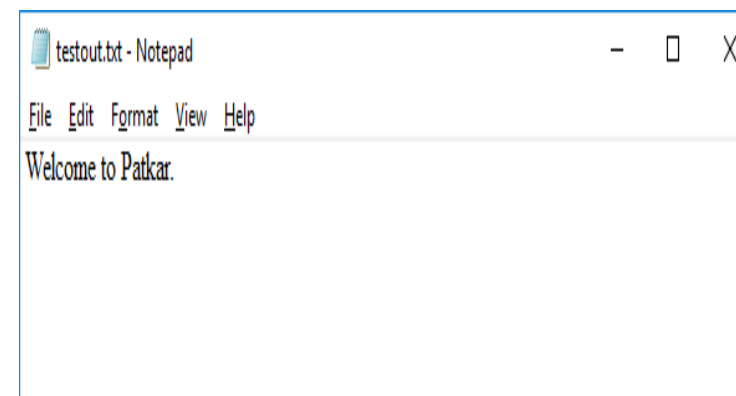
Writing Files using Character Stream Classes



```
import java.io.FileWriter;
public class Demo {
    public static void main(String args[]){
        try{
            FileWriter fw=new FileWriter("D:\\testout.txt");
            fw.write("Welcome to Patkar.");
            fw.close();
        }catch(Exception e){System.out.println(e);}
        System.out.println("Success...");
    }
}
```



```
C:\Users\Virendra\Desktop\Core Java>javac Demo.java
C:\Users\Virendra\Desktop\Core Java>java Demo
Success...
C:\Users\Virendra\Desktop\Core Java>
```



```
File Edit Format View Help
Welcome to Patkar.
```

Reading Files using Character Stream Classes

- Java **FileReader** class is used to **read data from the file**. It returns data in byte format like **FileInputStream** class.
- It is **character-oriented class** which is used for file handling in java.
- The declaration for **Java.io.FileReader** class:

public class FileReader extends InputStreamReader

- **Constructors of FileReader class**

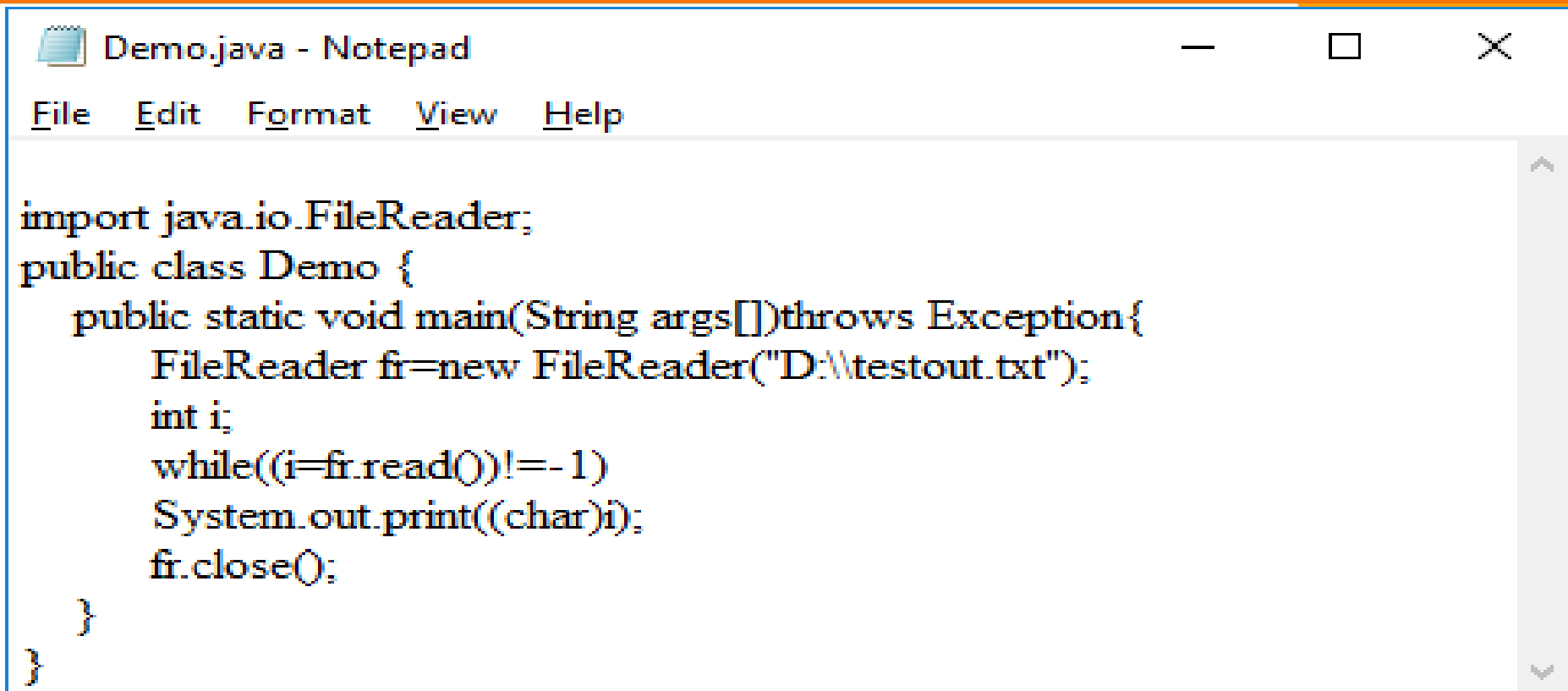
FileReader(String file) - It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws **FileNotFoundException**.

FileReader(File file) - It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws **FileNotFoundException**.

Reading Files using Character Stream Classes

- Java FileInputStream class methods
- Method Description
- **int available()** - It is used to return the estimated number of bytes that can be read from the input stream.
- **int read()** - It is used to read the byte of data from the input stream.
- **long skip(long x)** - It is used to skip over and discards x bytes of data from the input stream.
- **protected void finalize()** - It is used to ensure that the close method is called when there is no more reference to the file input stream.
- **void close()** - It is used to closes the stream.

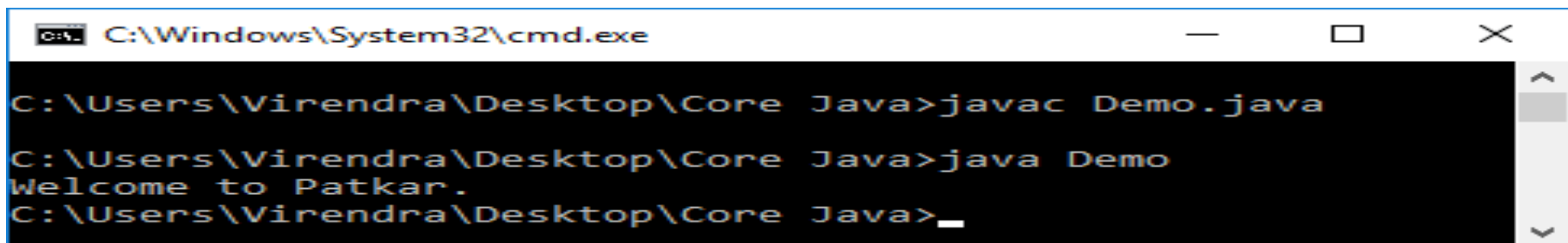
Reading Files using Character Stream Classes



A screenshot of a Notepad window titled "Demo.java - Notepad". The window has a menu bar with "File", "Edit", "Format", "View", and "Help". The text area contains the following Java code:

```
import java.io.FileReader;

public class Demo {
    public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        int i;
        while((i=fr.read())!=-1)
            System.out.print((char)i);
        fr.close();
    }
}
```



A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The prompt shows the compilation and execution of the Java program:

```
C:\Users\Virendra\Desktop\Core Java>javac Demo.java
C:\Users\Virendra\Desktop\Core Java>java Demo
Welcome to Patkar.
C:\Users\Virendra\Desktop\Core Java>_
```



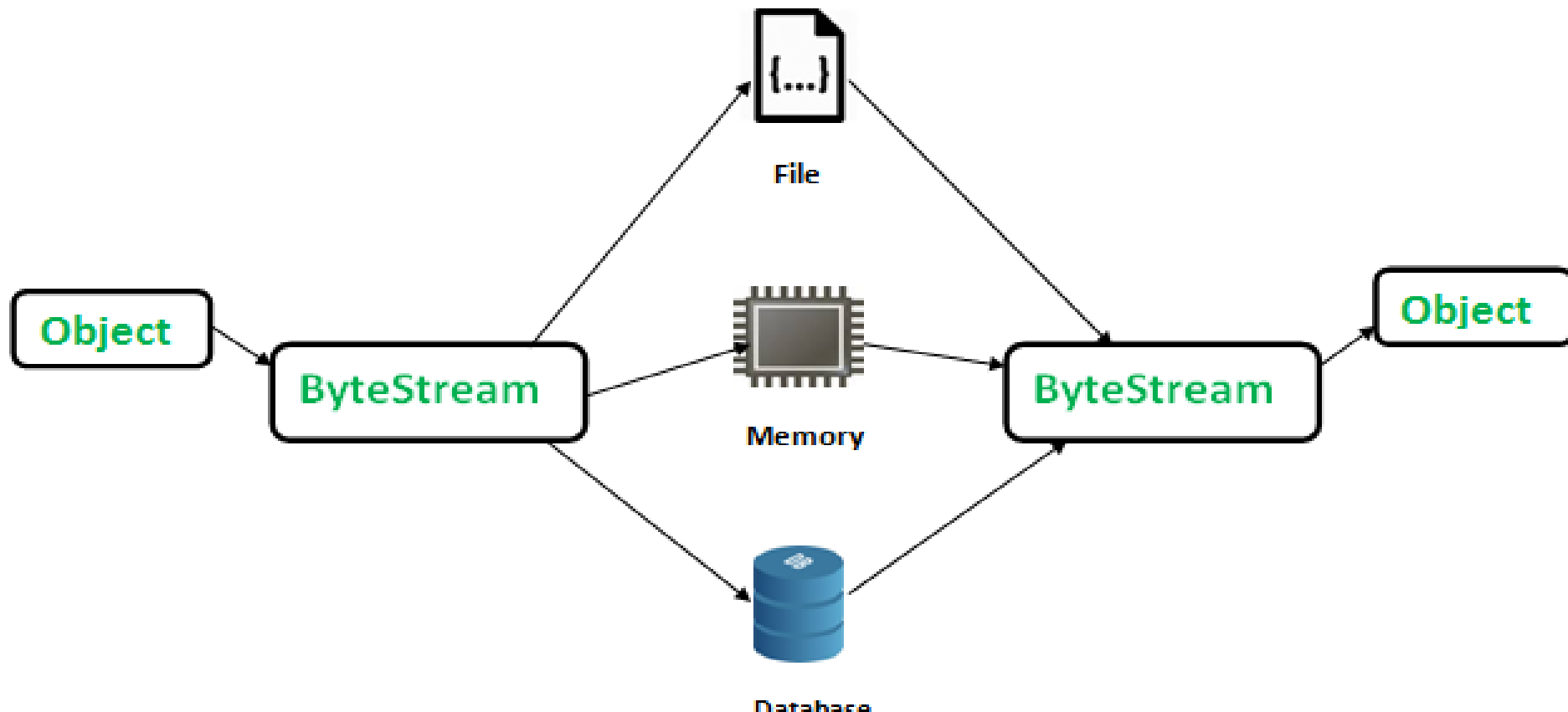
Serialization: Serialization, Storing objects via serialization Metadata and File attributes: Metadata and File attributes.

Serialization

java.io.Serializable, Class implements Serializable interface

Serialization

De-Serialization



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.**
- To **serialize an object** means to convert its **state to a byte stream** so that the byte stream can be reverted back into a copy of the object.
- A Java object is **serializable** if its class or any of its superclasses implements either the **java.io.Serializable** interface or its subinterface, **java.io.Externalizable**.
- **Deserialization is the process of converting the serialized form of an object back into a copy of the object.**
- Serialization is mainly used to travel object's state on the network (that is known as marshalling).
- The basic concept is that we will convert the object into the form of a byte array and we will send that byte array to the server that we will again convert this byte array into the object. We convert an object into a byte stream because the byte stream is platform-independent.

Serialization (Contd.)

ObjectOutputStreamClass

- The class `ObjectOutputStream` is a high-level stream that contains the methods for serializing an object into a byte stream. The **`ObjectOutputStream` class writes objects and primitive data types to an `OutputStream`.**
- The rule is that we can write only that object to the streams that support the `java.io.Serializable` interface.

S.N	Method	Description
1.	<code>public final void writeObject(Object obj) throws IOException {}</code>	This method writes the specified object to the <code>ObjectOutputStream</code> class.
2.	<code>public void flush() throws IOException {}</code>	This method flushes the current output stream object.
3.	<code>public void close() throws IOException {}</code>	This method closes the current output stream object.

Storing objects via Serialization Example

```
import java.io.Serializable;

public class Teacher implements
                               Serializable
{
    int id;
    String name;
    public Teacher(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
}
```

```
import java.io.*;

public class Serialization
{
    public static void main(String args[])
    {
        try {
            Teacher t1 = new Teacher(1011,"Yahvi");

            FileOutputStream file = new FileOutputStream("myF.txt");

            ObjectOutputStream output = new ObjectOutputStream(file);
            output.writeObject(t1);
            output.flush();
            output.close();
            System.out.println("Successful");
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

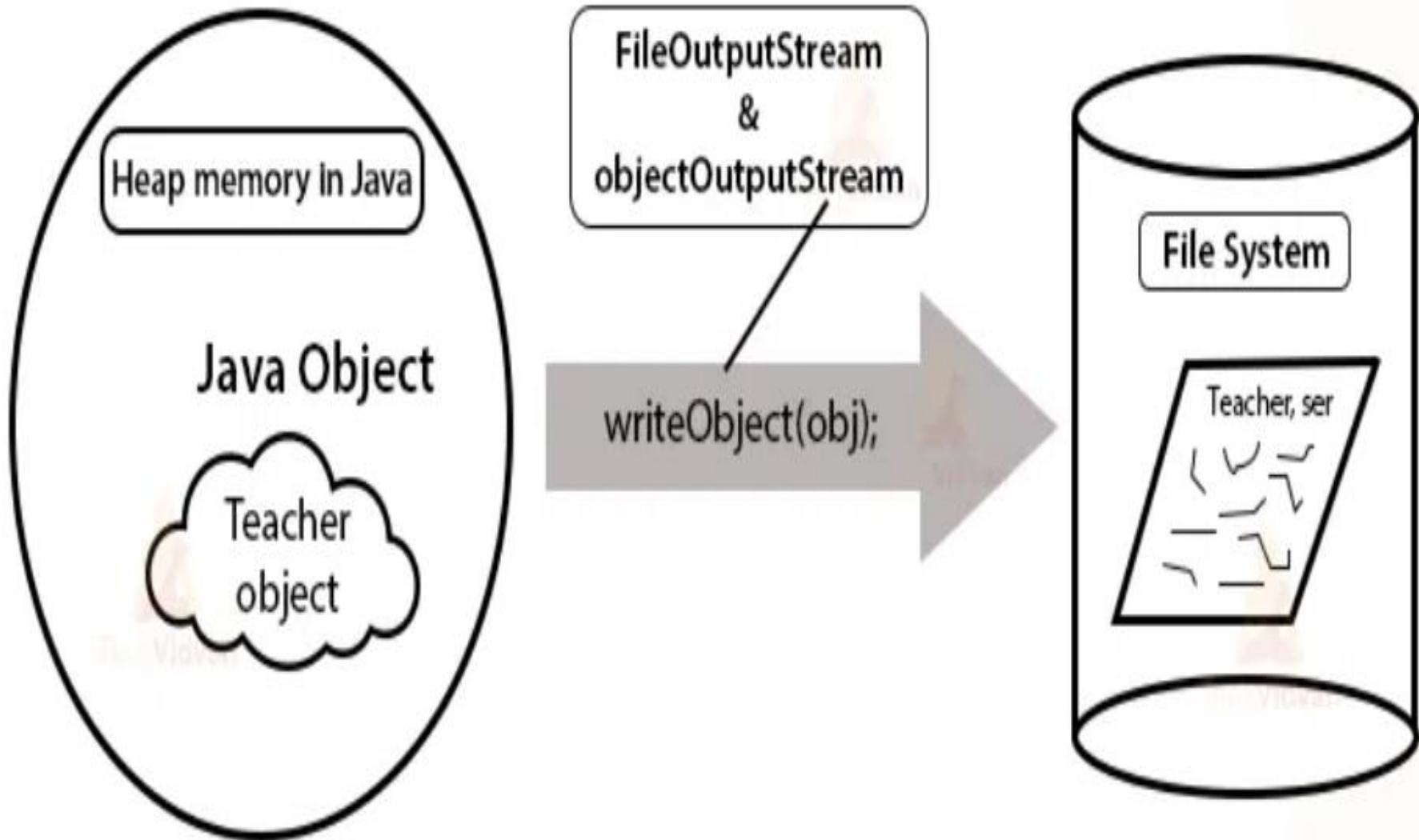
Deserialization Example

```
import java.io.*;
public class Deserialization
{
    public static void main(String args[])
    {
        try
        {
            //Creating stream to read the object
            ObjectInputStream input = new ObjectInputStream(new FileInputStream("myF.txt"));
            Teacher teacher = (Teacher)input.readObject();

            //printing the data of the serialized object
            System.out.println("The id of the teacher is:" +teacher.id);
            System.out.println("The name of the teacher is:" +teacher.name);

            //closing the stream
            input.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Serialization (Contd.)



Deserialization

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

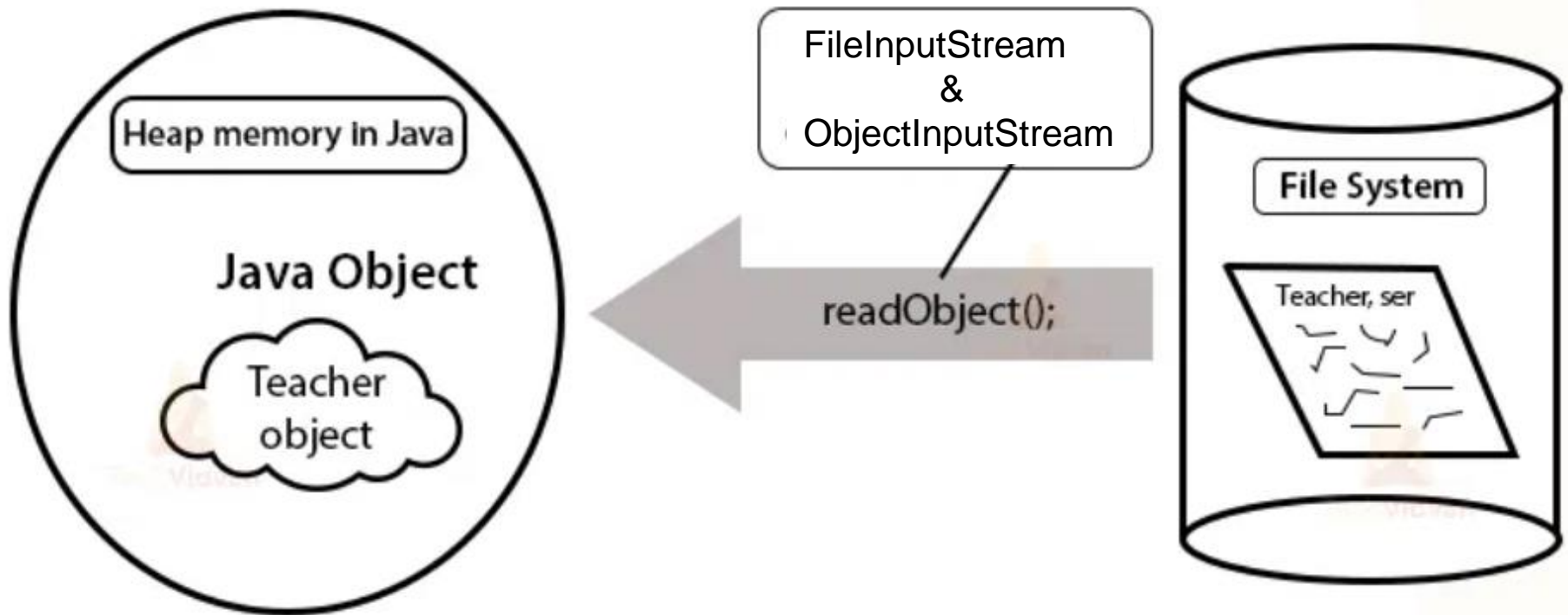
ObjectInputStreamClass

An ObjectInputStream is a high-level stream class that contains methods that help us in deserializing a byte stream back to the state of the object. It deserializes the objects and primitive data that are written using an ObjectOutputStream.

S.N	Method	Description
1.	public final Object readObject() throws IOException, ClassNotFoundException {}	This method reads an object from the input stream.
2.	public void close() throws IOException {}	This method is used to close the object of ObjectInputStream.

Deserialization (Contd.)

Deserialization in Java



Metadata

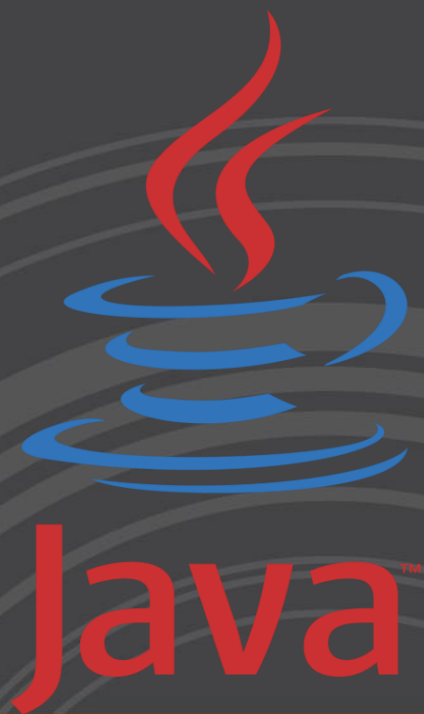
- The definition of **metadata** is "**data about other data.**" With a file system, the data is contained in its files and directories, and the metadata tracks information about each of these objects: Is it a regular file, a directory, or a link? What is its size, creation date, last modified date, file owner, group owner, and access permissions?
- A file system's metadata is typically referred to as its **file attributes**. The Files class includes methods that can be used to obtain a single attribute of a file, or to set an attribute.

Methods	Comment
size(Path)	Returns the size of the specified file in bytes.
isDirectory(Path, LinkOption)	Returns true if the specified Path locates a file that is a directory.
isRegularFile(Path, LinkOption...)	Returns true if the specified Path locates a file that is a regular file.
isSymbolicLink(Path)	Returns true if the specified Path locates a file that is a symbolic link.
isHidden(Path)	Returns true if the specified Path locates a file that is considered hidden by the file system.
getLastModifiedTime(Path, LinkOption...) setLastModifiedTime(Path, FileTime)	Returns or sets the specified file's last modified time.
getOwner(Path, LinkOption...) setOwner(Path, UserPrincipal)	Returns or sets the owner of the file.

File Attributes

- If a **program needs multiple file attributes around the same time**, it can be inefficient to use methods that retrieve a single attribute. Repeatedly accessing the file system to retrieve a single attribute can adversely affect performance.
- For this reason, the Files class provides two readAttributes methods to fetch a file's attributes in one bulk operation.

Methods	Comment
readAttributes(Path, String, LinkOption...)	Reads a file's attributes as a bulk operation. The String parameter identifies the attributes to be read.
readAttributes(Path, Class, LinkOption...)	Reads a file's attributes as a bulk operation. The Class parameter is the type of attributes requested and the method returns an object of that class.



Basic File Attribute: Basic File Attribute File Visitor Interface: File Visitor Interface, Random Access File.

Random Access File

- The `Java.io.RandomAccessFile` class file behaves like a **large array of bytes stored in the file system**. Instances of this class support both reading and writing to a random-access file.
- There is a cursor implied to the array called **file pointer**, by moving the cursor we do the read write operations.
- If end-of-file is reached before the desired number of bytes has been read than `EOFException` is thrown. It is a type of `IOException`.

Constructor	Description
<code>RandomAccessFile(File file, String mode)</code>	Creates a random-access file stream to read from, and optionally to write to, the file specified by the <code>File</code> argument.
<code>RandomAccessFile(String name, String mode)</code>	Creates a random-access file stream to read from, and optionally to write to, a file with the specified name.

Random Access File Example

```
public class RandomFA {  
    static final String FILEPATH = "D:/myFile.TXT"; //STRING  
    public static void main(String[] args) {  
        try {  
            System.out.println(new String(readFromFile(FILEPATH, 0, 18)));  
            writeToFile(FILEPATH, "I am Studying JAVA", 31);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
static byte[] readFromFile(String filePath, int position, int size)  
throws IOException {  
    RandomAccessFile file = new RandomAccessFile(filePath, "r");  
    file.seek(position);  
    byte[] bytes = new byte[size]; //BYTE ARRAY  
    file.read(bytes); //arjit  
    file.close();  
    return bytes;  
}
```

```
static void writeToFile(String filePath, String data, int position)  
throws IOException {  
    RandomAccessFile file = new RandomAccessFile(filePath, "rw");  
    file.seek(position);  
    file.write(data.getBytes());  
    file.close();  
}
```