



Java™
C-DAC PATNA

सीडैक
CDAC
पटना | PATNA

File Handling and I/O

Introduction: File

- File is a collection of bytes stored in secondary storage device i.e. disk. Thus, File handling is used to read, write, append or update a file without directly opening it.
- Types of File:
 1. Text File
 2. Binary File

C-DAC PATNA

Why we use File Handling?

- After the termination of program all the entered data is lost because primary memory is volatile. If the data has to be used later, then it becomes necessary to keep it in permanent storage device.
- File handling in Java refers to the process of managing data storage and retrieval operations, including reading from and writing data to a file. This involves creating, opening, reading, writing, and closing files using various classes.

File class

1. The File class in Java is part of the java.io package and is used to represent file and directory pathnames in an abstract manner. It provides several methods to perform file and directory operations.
2. The File class is used to represent the file or directory but does not contain methods to read/write the content of the file.
3. It does not represent the actual file content, but rather the file or directory pathname.
4. Always handle exceptions such as IOException when working with files.
5. The File class provides methods to:
 - a. Create files or directories.
 - b. Check file properties (e.g., file existence, readability, writability, etc.).
 - c. Rename or delete files and directories.
 - d. Work with absolute or relative paths.
 - e. List contents of directories.

File class: Constructor

→ Here are some constructors provided by the File class:

- `File(String pathname)`

Creates a new File instance from the given pathname string.

Example:

```
File file = new File("example.txt");
```

- `File(String parent, String child)`

Creates a new File instance from a parent directory pathname and a child pathname string.

Example:

```
File file = new File("/home/user", "example.txt");
```

- `File(File parent, String child)`

Creates a new File instance from a parent File object and a child pathname string.

Example:

```
File parentDir = new File("/home/user");
```

```
File file = new File(parentDir, "example.txt");
```

File Creation/Deletion Methods

Method	Description	Example
<code>createNewFile()</code>	Creates a new file. Throws <code>IOException</code> if the file cannot be created.	<code>file.createNewFile();</code>
<code>mkdir()</code>	Creates a single directory.	<code>file.mkdir();</code>
<code>mkdirs()</code>	Creates a directory along with all non-existent parent directories.	<code>file.mkdirs();</code>
<code>delete()</code>	Deletes the file or directory.	<code>file.delete();</code>

File Information/Operation Methods

Method	Description	Example
<code>exists()</code>	Checks if the file or directory exists.	<code>file.exists();</code>
<code>isFile()</code>	Checks if it is a file (not a directory).	<code>file.isFile();</code>
<code>isDirectory()</code>	Checks if it is a directory.	<code>file.isDirectory();</code>
<code>length()</code>	Returns the file size in bytes.	<code>file.length();</code>
<code>getName()</code>	Returns the name of the file or directory.	<code>file.getName();</code>
<code>getPath()</code>	Returns the relative or absolute path of the file.	<code>file.getPath();</code>
<code>getAbsolutePath()</code>	Returns the absolute path of the file.	<code>file.getAbsolutePath();</code>

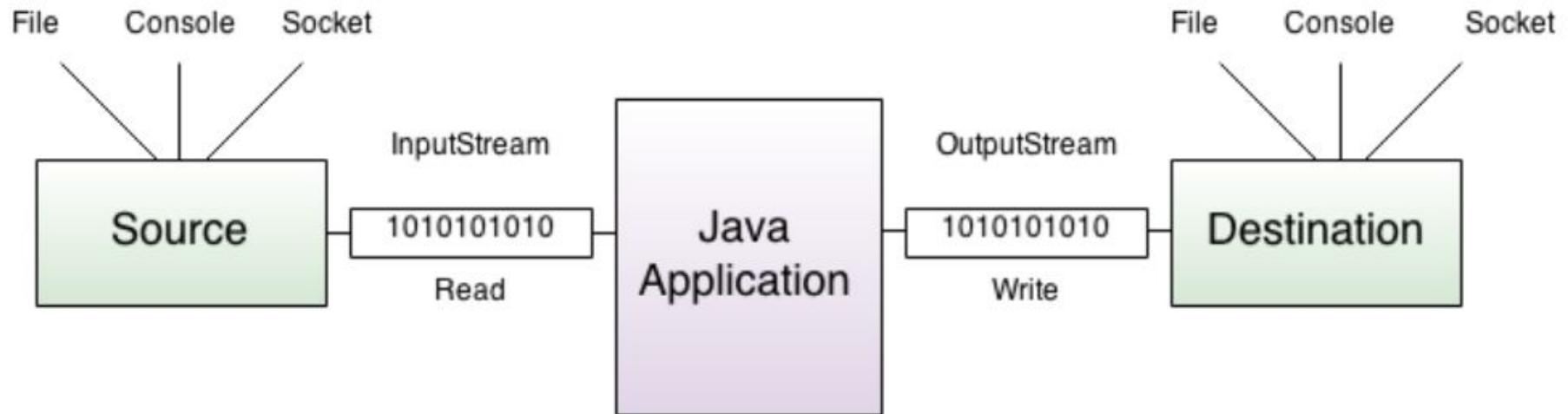
Method	Description	Example
<code>renameTo(File dest)</code>	Renames the file or directory to the given destination.	<code>file.renameTo(new File("newName.txt"));</code>
<code>list()</code>	Lists the names of files and directories in a directory.	<code>String[] files = directory.list();</code>
<code>listFiles()</code>	Returns an array of <code>File</code> objects for directory contents.	<code>File[] files = directory.listFiles();</code>

Stream

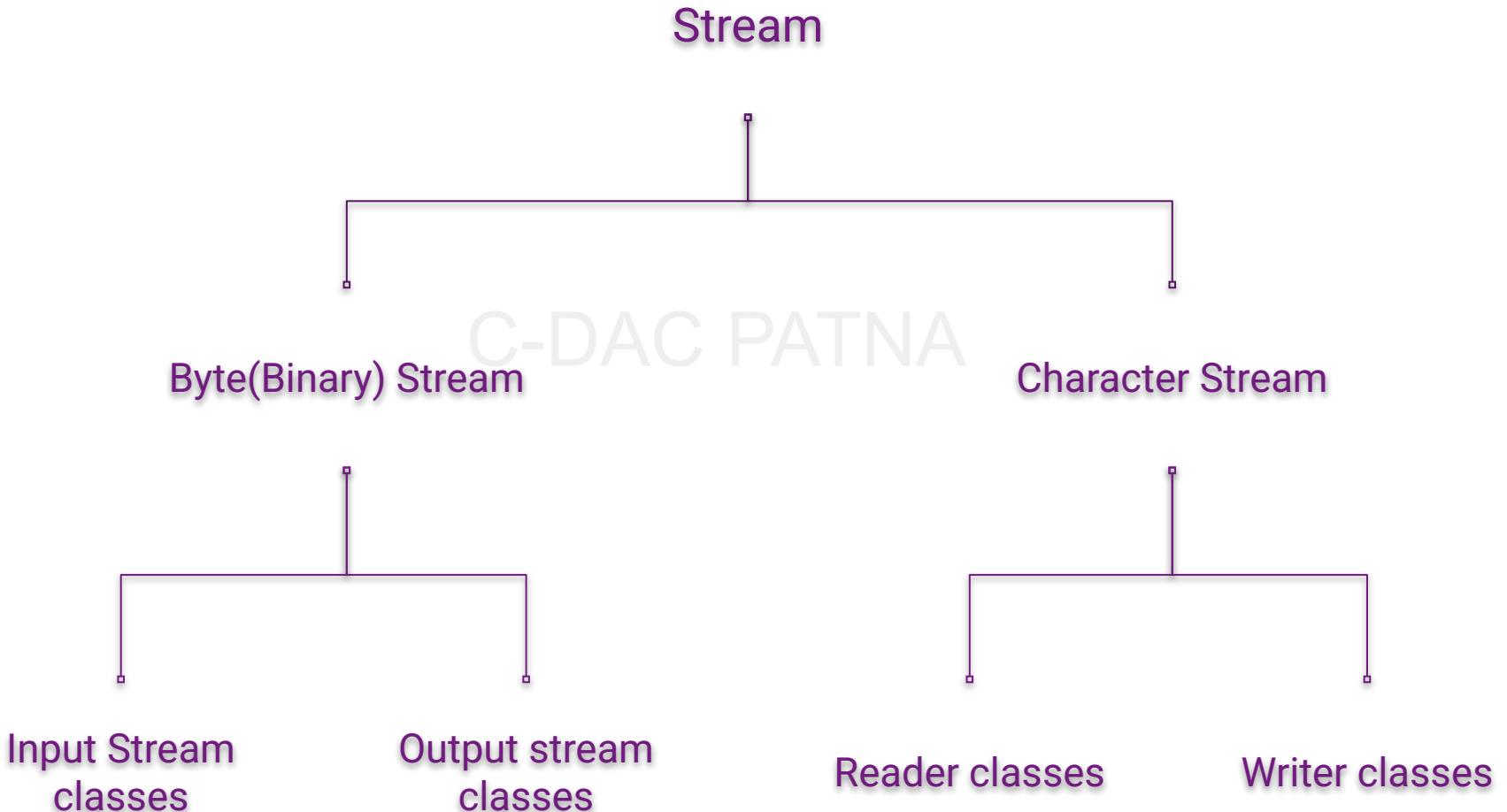
- A stream is simply a sequence of bytes that flows into or out of our program.
- It's an abstract representation of an input or output device.
- All these classes are present in `java.io` package.

Purpose : To achieve device independent I/O.

C-DAC PATNA



Stream Hierarchy

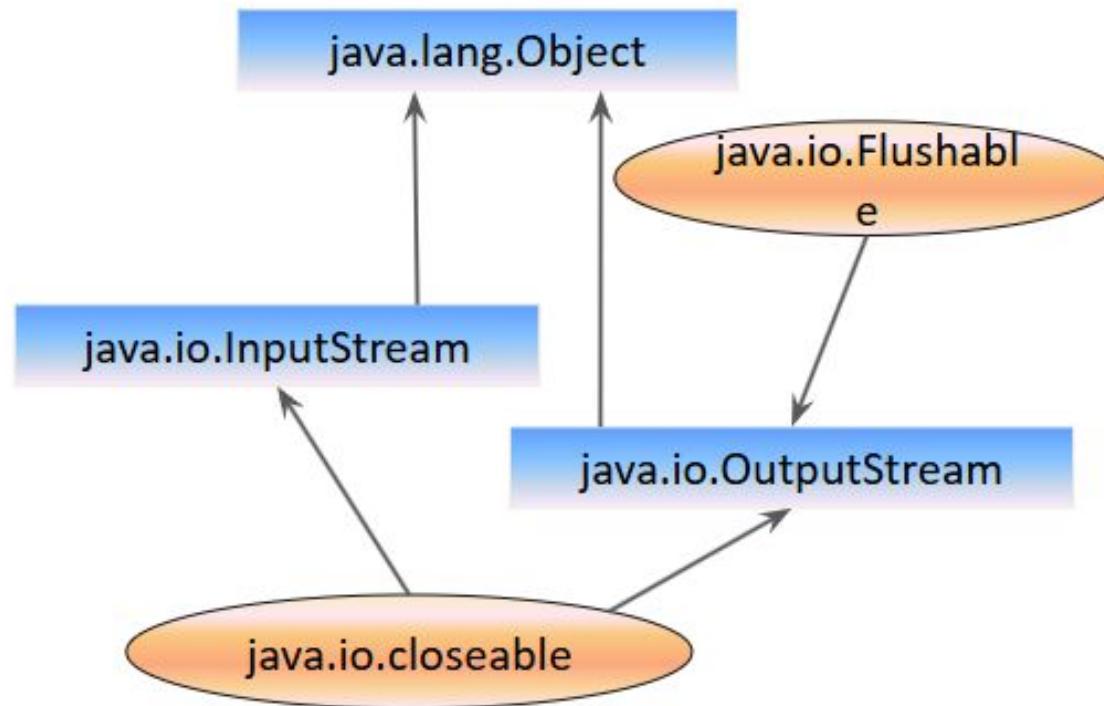


Byte(Binary) Stream

- While writing data to a Binary Stream, the data is written as a series of bytes, exactly as it appears in the memory. No data transformation takes place.
- It has 2 classes -
 - a. InputStream Classes
 - b. OutputStream Classes

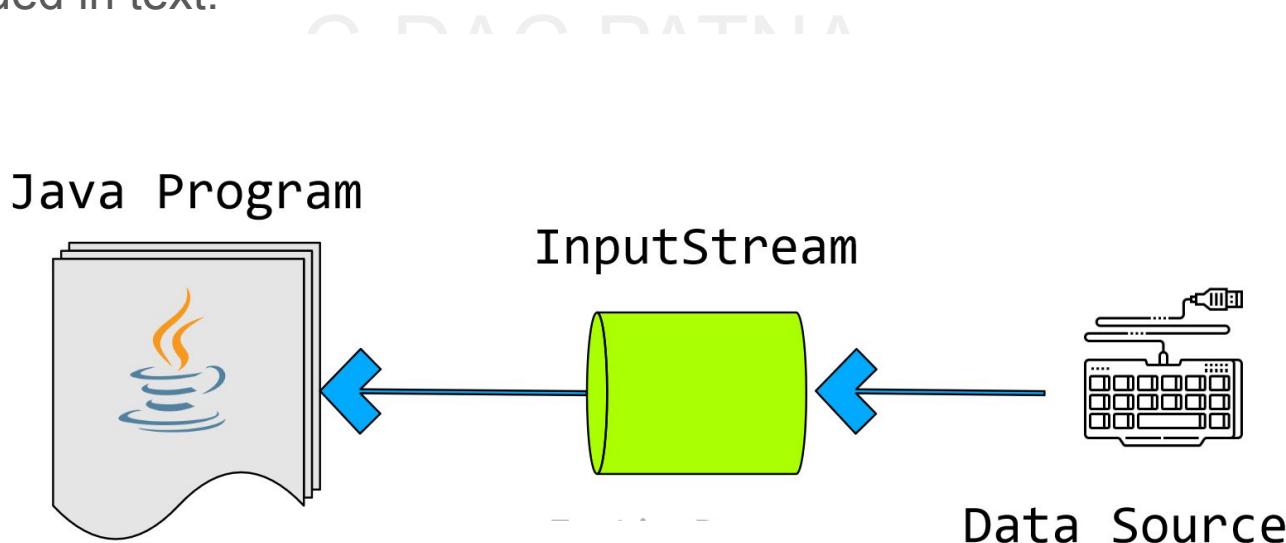
C-DAC PATNA

InputStream & OutputStream Class

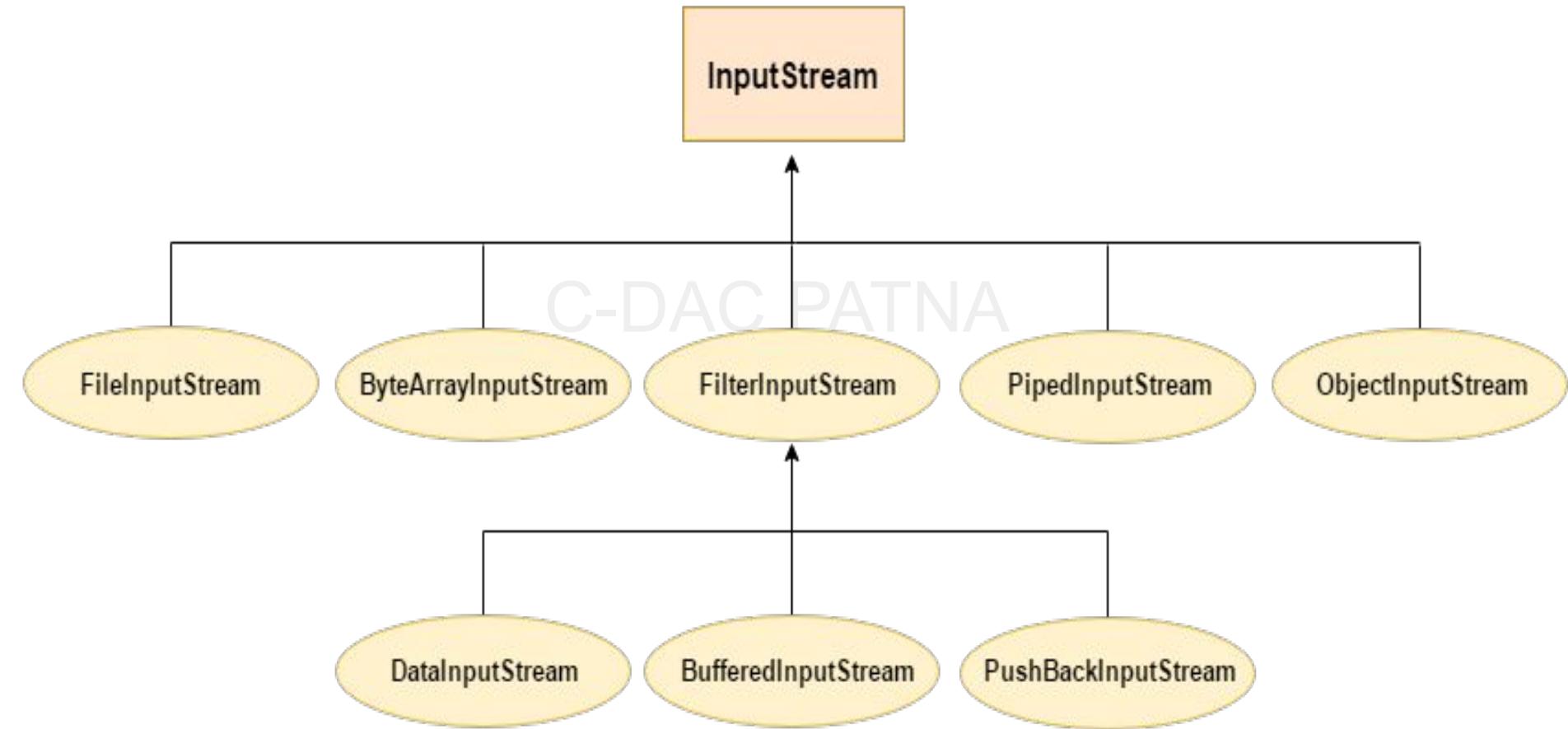


InputStream

- Represents a stream of data for reading raw byte data from a source (e.g., a file, network socket, or array). It serves as the superclass for all classes designed for byte input in Java. It is an **abstract class**.
- Reads data byte by byte (not character by character). This makes it suitable for reading binary data like images, audio files, and any data not encoded in text.



InputStream Classes



InputStream methods

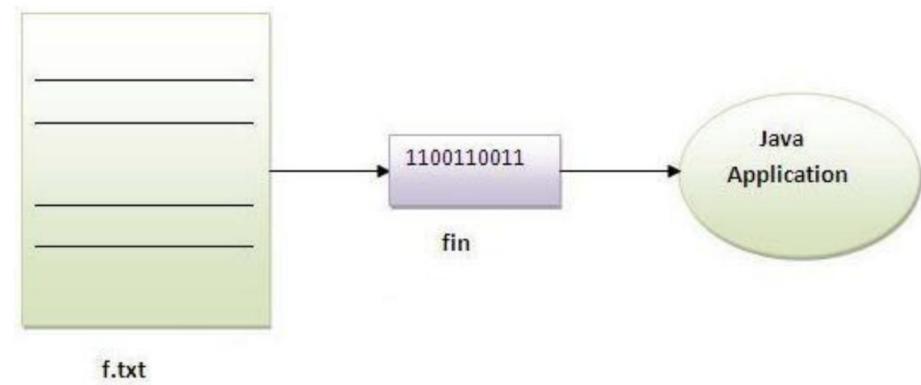
Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void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

FileInputStream

- FileInputStream class obtains input bytes from a file. It is used for reading streams of
- raw bytes such as image data.
- For reading streams of characters, consider using FileReader.
- It should be used to read byte-oriented data for example to read image, audio, video etc.

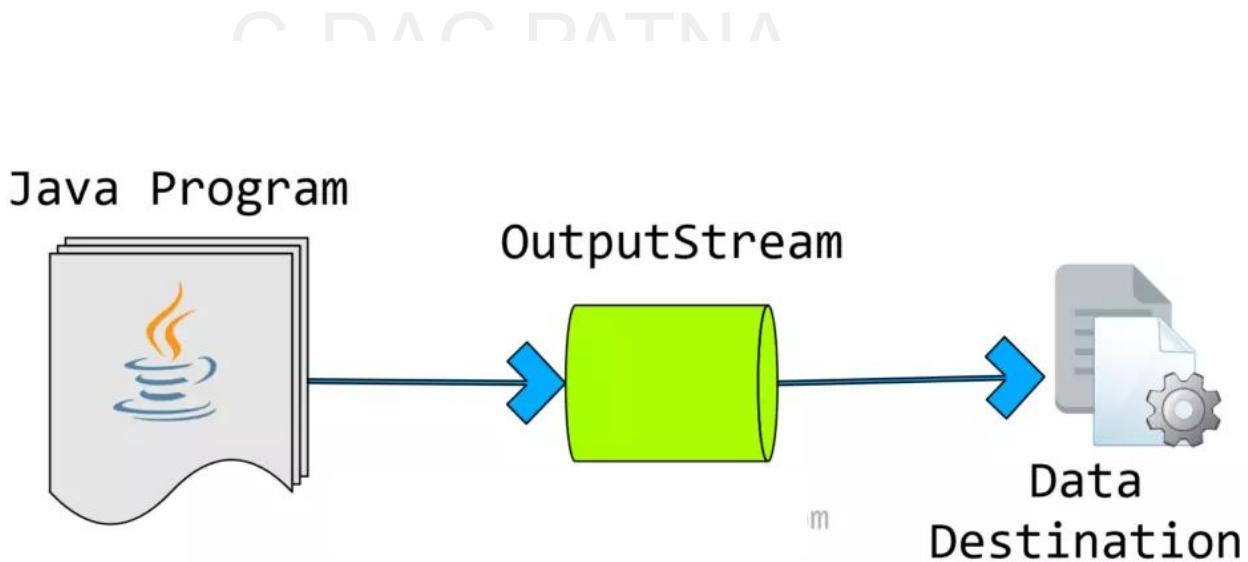
try{

```
FileInputStream fin = new FileInputStream("abc.txt");
int i=0;
while((i=fin.read())!=-1){
    System.out.print((char)i);
}
fin.close();
}catch(Exception e){System.out.println(e);}
}
```

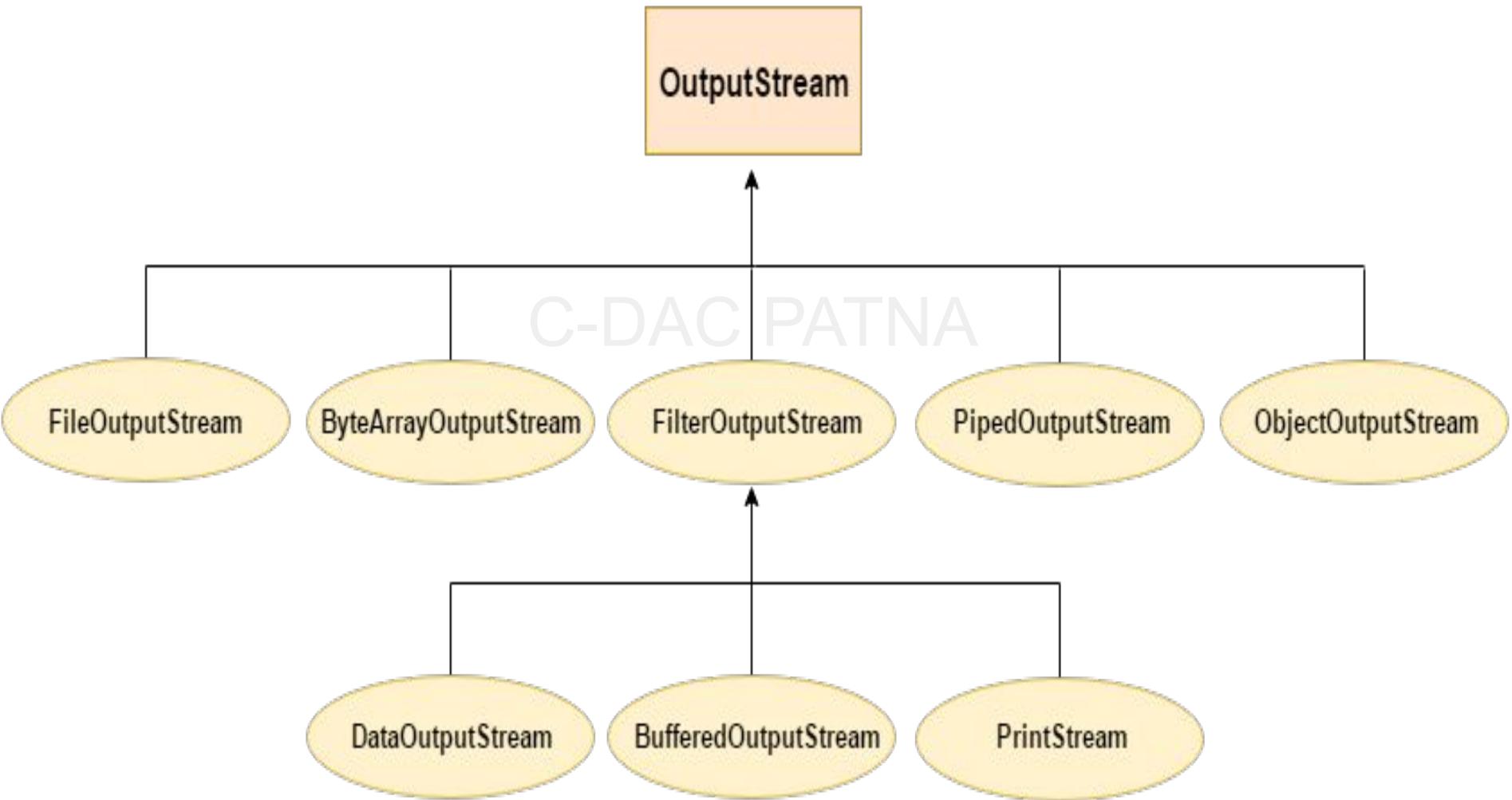


OutputStream

- Represents a stream of data for writing raw byte data to a destination (e.g., a file, network socket, or array). It is the superclass for all byte-oriented output classes in Java.
- It is an **abstract class**.
- OutputStream writes data byte by byte, which makes it suitable for binary data such as images, audio, and other non-textual content.



OutputStream Classes



OutputStream methods

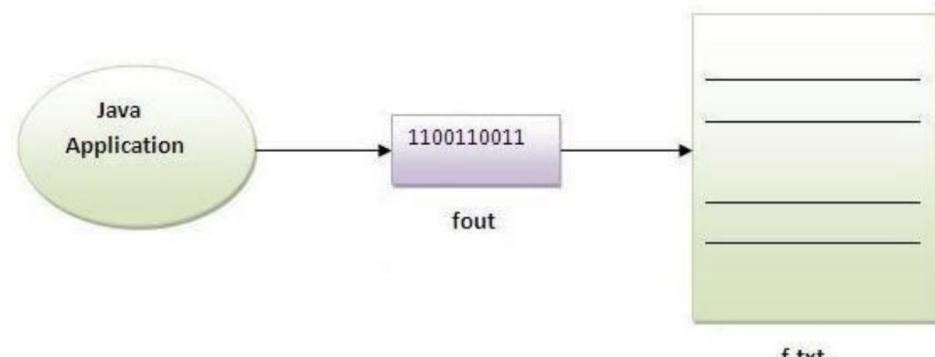
Method	Description
1) public void write(int) throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[]) throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush() throws IOException:	flushes the current output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

FileOutputStream

- Java FileOutputStream is an output stream for writing data to a file.

```
try{
```

```
    FileOutputStream fout=new FileOutputStream("abc.txt");
    String s="Sachin Tendulkar is my favourite player";
    byte b[]=s.getBytes();//converting string into byte array
    fout.write(b);
    fout.close();
    System.out.println(System.getProperty("user.dir"));
    System.out.println("success...");
}catch(Exception e)
{
    System.out.println(e);
}
```

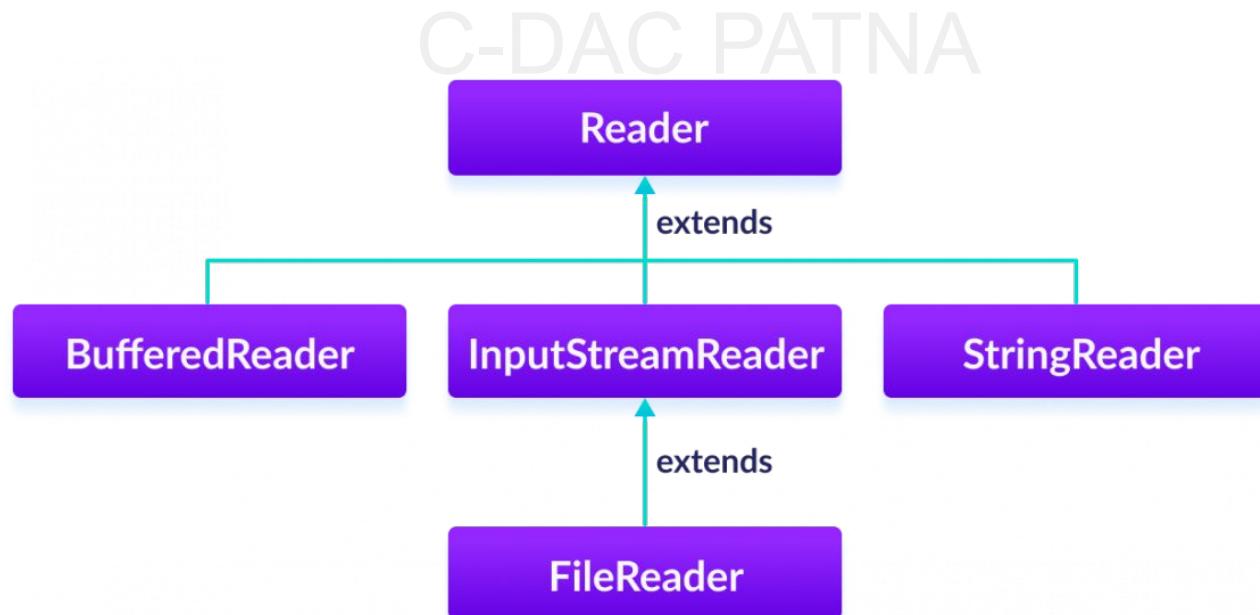


Character Stream

- It is designed to handle text data (characters). Unlike byte streams, which process raw bytes, character streams process data as characters, making them ideal for reading and writing text files.
- Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.
- It operates on 16-bit Unicode characters, ensuring support for international text.
- It has 2 classes -
 - a. Reader Classes
 - b. Writer Classes

Reader class

- Reader class is used to read the 16-bit characters from the input stream.
- It is an **abstract class** and can't be instantiated, but there are various subclasses that inherit the Reader class and override the methods of the Reader class.
- All methods of the Reader class throw an IOException.
- The subclasses of the Reader class are:



Reader class Methods

Method	Short Description	Example
<code>int read()</code>	Reads a single character from the stream.	<pre>int character = reader.read(); if (character != -1) { System.out.print((char) character); }</pre>
<code>int read(char[] cbuf)</code>	Reads characters into a specified character array.	<pre>char[] buffer = new char[1024]; int charsRead = reader.read(buffer); if (charsRead != -1) { System.out.print(new String(buffer, 0, charsRead)); }</pre>
<code>int read(char[] cbuf, int off, int len)</code>	Reads a portion of a character array with offset and length.	<pre>char[] buffer = new char[1024]; int charsRead = reader.read(buffer, 0, 1024); if (charsRead != -1) { System.out.print(new String(buffer, 0, charsRead)); }</pre>
<code>long skip(long n)</code>	Skips <code>n</code> characters in the stream.	<pre>long skipped = reader.skip(10);</pre>
<code>boolean ready()</code>	Checks if the stream is ready to be read.	<pre>if (reader.ready()) { System.out.println("Stream is ready to be read."); }</pre>
<code>void close()</code>	Closes the reader and releases resources.	<pre>reader.close();</pre>

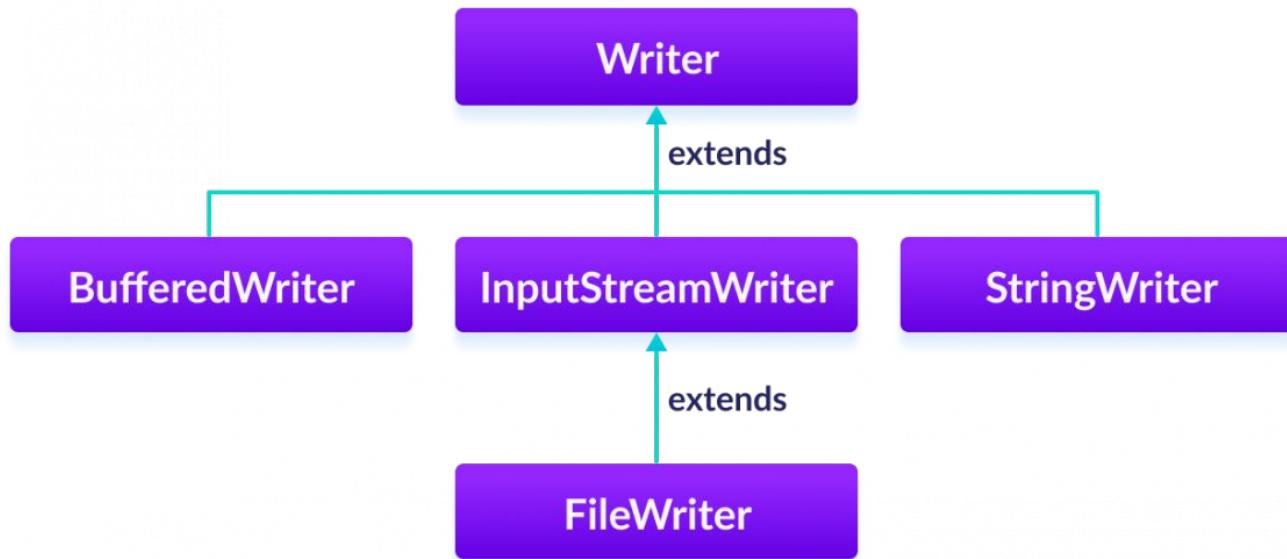
FileReader

```
try {  
  
    reader = new FileReader("sample.txt"); // Specify the path to the file  
  
    int character = reader.read();  
  
    if (character != -1) {  
  
        System.out.println("First character: " + (char) character);  
  
    }  
  
    // Read characters into a buffer  
  
    char[] buffer = new char[1024];  
  
    int charsRead = reader.read(buffer);  
  
    if (charsRead != -1) {  
  
        System.out.println("Characters read: " + new String(buffer, 0, charsRead));  
  
    }  
  
    // Skip a few characters  
  
    long skipped = reader.skip(5);  
  
    System.out.println("Skipped characters: " + skipped);  
  
  
    // Check if the stream is ready to be read  
  
    if (reader.ready()) {  
  
        System.out.println("Stream is ready to read more characters.");  
  
    }  
  
    reader.close();  
  
}  
} catch (IOException e) {  
  
    e.printStackTrace();  
  
}
```

C-DAC PATNA

Writer Classes

- Writer class is used to write 16-bit Unicode characters to the output stream.
- The methods of the Writer class generate IOException.
- Like Reader class, Writer class is also an **abstract class** that cannot be instantiated; therefore, the subclasses of the Writer class are used to write the characters onto the output stream.
- The subclasses of the Writer class are.



Writer class Methods

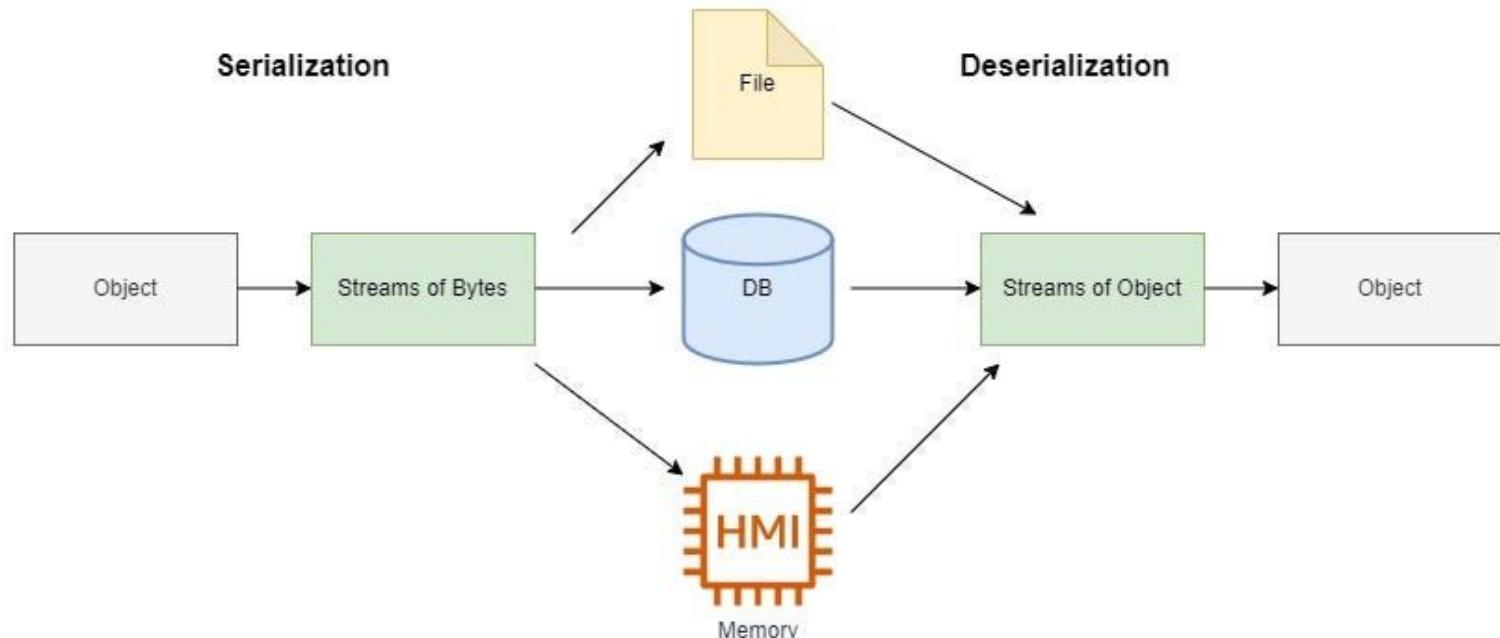
Method	Short Description	Example
<code>void write(int c)</code>	Writes a single character (given as an integer) to the stream.	<code>writer.write('A');</code>
<code>void write(char[] cbuf)</code>	Writes an array of characters to the stream.	<code>char[] buffer = {'H', 'e', 'l', 'l', 'o'}; writer.write(buffer);</code>
<code>void write(char[] cbuf, int off, int len)</code>	Writes a portion of an array of characters starting from an offset for a given length.	<code>char[] buffer = {'H', 'e', 'l', 'l', 'o'}; writer.write(buffer, 1, 3); // Writes "ell"</code>
<code>void write(String str)</code>	Writes a string to the stream.	<code>writer.write("Hello, world!");</code>
<code>void write(String str, int off, int len)</code>	Writes a portion of a string, starting from an offset for a given length.	<code>writer.write("Hello, world!", 7, 5); // Writes "world"</code>
<code>void flush()</code>	Flushes the stream, ensuring all data is written.	<code>writer.flush();</code>
<code>void close()</code>	Closes the stream, releasing any resources.	<code>writer.close();</code>

FileWriter

```
Writer writer = null;  
  
try {  
    writer = new FileWriter("output.txt");  
    // Write a single character  
    writer.write('H');  
    // Write a string  
    writer.write("ello, World!");  
    // Write a portion of a string  
    writer.write("Java Programming", 0, 4); // Writes "Java"  
    // Write an array of characters  
    char[] buffer = {'A', 'B', 'C'};  
    writer.write(buffer);  
    // Flush the writer to ensure all data is written  
    writer.flush();  
    writer.close();  
} catch (IOException e)  
{ e.printStackTrace(); }
```

Serialization and Deserialization

- If we want to store that object in permanent in memory then we cannot directly store object in memory we have to convert it into byte Stream and this is **called Serialization**.
- Serialization is the process of converting an object into a byte stream, which can then be saved to a file, sent over a network, or stored in a database.
- **Deserialization** is the reverse process of serialization, where a byte stream is converted back into a copy of the original object. This allows you to retrieve the object's state and use it in your application.
- The serialization and deserialization process is platform-independent.



Serializable interface

- A class must implement the **java.io.Serializable** interface. It is a marker interface (no methods to implement).
- Fields marked as **transient** are not serialized.
- If a class contains non-serializable fields, serialization will fail unless they are marked transient.
- we can serialize an object using the **ObjectOutputStream** class, which writes the object as a stream of bytes.

```
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;  
    }  
}
```

Hands-On: Serialization and Deserialization

ObjectOutputStream

- ObjectOutputStream is a class in Java used for serializing objects to an output stream.
- This class is part of the java.io package.
- Methods:
 - void writeObject(Object obj): Serializes an object and writes it to the stream.
 - Example: `objectOutputStream.writeObject(person);`
 - void write(byte[] b): Writes a byte array to the output stream.
 - void flush(): Ensures that any buffered data is written to the output stream.
 - void close(): Closes the output stream, freeing resources.

Serializable

```
public static void main(String[] args) {
    try {
        // Creating a Student object
        Student student = new Student("John Doe", 101);

        // Creating ObjectOutputStream to serialize the object
        FileOutputStream fileOut = new FileOutputStream("student.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);

        // Serializing the object
        out.writeObject(student);
        out.close();
        fileOut.close();

        System.out.println("Object serialized and saved to student.ser");
    } catch (IOException e) {
        System.out.println("Error during serialization: " + e.getMessage());
    }
}
```

C-DAC PATNA

ObjectInputStream

- ObjectInputStream is used for deserializing objects from an input stream. It reads the byte stream and reconstructs the original objects.
- This class is also part of the **java.io package**.
- Methods:
 - Object readObject(): Reads an object from the input stream and deserializes it.
 - Example: **Person person = (Person) objectInputStream.readObject();**
 - void close(): Closes the input stream, freeing resources.

Deserialization

```
public static void main(String[] args) {  
    try {  
        // Creating ObjectInputStream to read the serialized object  
        FileInputStream fileIn = new FileInputStream("student.ser");  
        ObjectInputStream in = new ObjectInputStream(fileIn);  
  
        // Deserializing the object  
        Student student = (Student) in.readObject();  
        in.close();  
        fileIn.close();  
  
        System.out.println("Deserialized Student:");  
        System.out.println("Name: " + student.getName());  
        System.out.println("Roll No: " + student.getRollNo());  
    } catch (IOException | ClassNotFoundException e) {  
        System.out.println("Error during deserialization: " + e.getMessage());  
    }  
}
```

C-DAC PATNA

Shallow Copy

- A shallow copy creates a new object but does not create copies of the objects referenced by the fields. Instead, it copies the references to those objects.
- Changes to nested objects affect both the original and the shallow copy.
- Use the `clone()` method of the `Object` class.

Hands-On: Shallow Copy

Deep Copy

C-DAC PATNA

- A deep copy creates a new object along with new instances of any objects referenced by its fields. It duplicates the entire object structure, including nested objects.
- Changes to nested objects in the copy do not affect the original object (and vice versa).
- Implement the `clone()` method and manually clone all referenced objects.
- Implement the `Cloneable` interface in the class to allow cloning.

Hands-On: Shallow Copy

Shallow Copy

```
class Person {  
    String name;  
    int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}  
  
class ShallowCopyExample {  
  
    public static void main(String[] args) throws CloneNotSupportedException {  
        Person person1 = new Person("John", 25);  
  
        // Create shallow copy  
  
        Person person2 = (Person) person1.clone(); // Using clone() for shallow copy  
  
        // Modify the original object  
  
        person1.name = "Jane";  
  
        System.out.println(person1.name); // Output: Jane  
        System.out.println(person2.name); // Output: Jane (same reference)  
    }  
}
```

C-DAC PATNA

Deep Copy

```

class Address implements Cloneable {
    String city;
    public Address(String city) {
        this.city = city;
    }
    @Override
    public Address clone() throws CloneNotSupportedException {
        return (Address) super.clone(); // Create a shallow copy of the Address
    }
}
class Person implements Cloneable {
    String name;
    Address address;
    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }
    @Override
    public Person clone() throws CloneNotSupportedException {
        Person cloned = (Person) super.clone(); // Shallow copy of the Person object
        cloned.address = this.address.clone(); // Create a deep copy of the Address object
        return cloned;
    }
}

```

```

public class DeepCopyWithClone {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Address address1 = new Address("New York");
        Person person1 = new Person("John", address1);
        // Create a deep copy of person1
        Person person2 = person1.clone();
        // Modify the original object's nested object
        person1.address.city = "Los Angeles"; // Affects only person1,
        person2 remains unchanged
        System.out.println(person1.address.city); // Output: Los Angeles
        System.out.println(person2.address.city); // Output: New York
        (independent copy)
    }
}

```

Assignments

1. Write a Java program to create a new file named "testfile.txt". Check if the file exists and print its absolute path, size, and readability status.
2. Write a Java program to write "Hello, Java!" into "output.txt" using FileOutputStream and read data from a file using FileInputStream.
3. Write a Java program to write "Welcome to File Handling in Java" into "notes.txt" using FileWriter.
4. Write a program that takes user input and appends it to "log.txt" without overwriting previous data.
5. Define a Student class with attributes: id, name, and grade. Implement serialization to save a student object to "student.ser". Implement deserialization to read the object back and display the student details.
6. Write a Java program that copies the content of "source.txt" into "destination.txt".
7. Create a Person class with an Address class as a field. Implement Shallow Copy using the clone() method. Modify the program to perform Deep Copy and explain the difference.

C-DAC PATNA
Thanks