Java I/O Streams

1️⃣ Introduction to Streams

Concept

- In Java, a **Stream** is a continuous flow of data between a **source** and a **destination**.
- Streams are used for **input (reading)** and **output (writing)** operations.

Types of Streams

1. **Byte Stream** – Handles **binary data** like images, audio, or video.
2. **Character Stream** – Handles **text data** (Unicode characters).

Diagram: Flow of Data

Input Device → Input Stream → Program → Output Stream → Output Device

---

2️⃣ Byte Streams

Definition

- Byte streams deal with **8-bit bytes**.
- They are mainly used for reading and writing **binary files**.

Important Classes

Operation	Class Name	Description
Input	FileInputStream	Reads data from a file (byte by byte)
Output	FileOutputStream	Writes data to a file (byte by byte)

Example 1: Copying a File Using Byte Stream

```
import java.io.*;


public class ByteStreamExample {
   public static void main(String[] args) {
      try {
         // Step 1: Open input and output streams for files
         FileInputStream fin = new FileInputStream("input.jpg");   // Source file
         FileOutputStream fout = new FileOutputStream("output.jpg"); // Destination file


         int i;
         // Step 2: Read data byte by byte until end of file (-1)
         while ((i = fin.read()) != -1) {
            fout.write(i); // Write each byte to the destination file
         }


         // Step 3: Close the streams
         fin.close();
         fout.close();


         System.out.println(" ✅ File copied successfully using Byte Stream!");
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

💻 **Output:**

✅ File copied successfully using Byte Stream!

3️⃣ Character Streams

Definition

- Character streams use **16-bit Unicode** and automatically handle **encoding**.
- Ideal for **text files** (e.g., .txt, .csv).

Important Classes

Operation	Class Name	Description
Input	FileReader	Reads text from a file (character by character)
Output	FileWriter	Writes text to a file (character by character)

Example 2: Reading and Writing Using Character Streams

```
import java.io.*;


public class CharacterStreamExample {
   public static void main(String[] args) {
      try {
         // Step 1: Create FileReader and FileWriter objects
         FileReader fr = new FileReader("input.txt");     // Reads text from source file
         FileWriter fw = new FileWriter("output.txt");    // Writes text to destination file


         int ch;
         // Step 2: Read each character and write it to output file
         while ((ch = fr.read()) != -1) {
            fw.write(ch);  // Writing character by character
         }
```

```
            // Step 3: Close both streams
            fr.close();
            fw.close();

            System.out.println("✅ File copied successfully using Character Stream!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

💻 **Output:**

✅ File copied successfully using Character Stream!

4️⃣ Readers and Writers

Overview

- **Reader** and **Writer** are **abstract classes** used for handling character-based data.
- They serve as **superclasses** for many file handling classes.

Common Subclasses

Reader Class	Writer Class	Purpose
FileReader	FileWriter	Basic file reading and writing
BufferedReader	BufferedWriter	Faster operations (buffered I/O)
InputStreamReader	OutputStreamWriter	Convert byte stream to character stream
PrintWriter	—	Used for formatted printing

Example 3: Using BufferedReader and BufferedWriter

```
import java.io.*;


public class BufferedReaderWriterExample {
   public static void main(String[] args) {
      try {
         // Step 1: Create buffered reader and writer for text files
         BufferedReader br = new BufferedReader(new FileReader("input.txt"));
         BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt"));


         String line;
         // Step 2: Read each line and write it to output file in uppercase
         while ((line = br.readLine()) != null) {
            bw.write(line.toUpperCase()); // Convert to uppercase
            bw.newLine(); // Move to next line
         }
```

```
            // Step 3: Close the streams
            br.close();
            bw.close();

            System.out.println("✅ Data written successfully using BufferedReader & BufferedWriter!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

💻 **Output:**

✅ Data written successfully using BufferedReader & BufferedWriter!

5️⃣ The File Class

Definition

- The File class represents a **file or directory** path, not the file content.
- It is used to **create, delete, check existence, and get information** about files.

Common Methods

Method	Description
exists()	Checks if file or directory exists
createNewFile()	Creates a new empty file
mkdir()	Creates a new directory
getName()	Returns the name of the file
length()	Returns file size in bytes
delete()	Deletes the file

Example 4: Using the File Class

```
import java.io.*;


public class FileClassExample {
    public static void main(String[] args) {
        // Step 1: Create File object representing a text file
        File file = new File("example.txt");


        try {
            // Step 2: Check if file exists, otherwise create it
            if (file.createNewFile()) {
                System.out.println("✅ File created: " + file.getName());
            } else {
                System.out.println("ℹ️ File already exists.");
```

```
        }

        // Step 3: Display file details
        System.out.println("File path: " + file.getAbsolutePath());
        System.out.println("Can read: " + file.canRead());
        System.out.println("Can write: " + file.canWrite());
        System.out.println("File size: " + file.length() + " bytes");

      } catch (IOException e) {
        e.printStackTrace();
      }
   }
}
```

💻 **Output (Example):**

✅ File created: example.txt

File path: C:\Users\Admin\example.txt

Can read: true

Can write: true

File size: 0 bytes

6️⃣ FileInputStream and FileOutputStream

Purpose

Used to handle **binary data** (like PDFs, images, or raw data files).

Class	Description
FileInputStream	Reads bytes from a file
FileOutputStream	Writes bytes to a file

Example 5: Copying Text File Using FileInputStream and FileOutputStream

```
import java.io.*;


public class FileStreamExample {
   public static void main(String[] args) {
      try {
         // Step 1: Open input and output streams
         FileInputStream fis = new FileInputStream("source.txt");
         FileOutputStream fos = new FileOutputStream("destination.txt");


         int data;
         // Step 2: Read and write byte by byte
         while ((data = fis.read()) != -1) {
            fos.write(data);
         }


         // Step 3: Close the streams
         fis.close();
         fos.close();
```

```
            System.out.println("✅ Data copied successfully using FileInputStream & FileOutputStream!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

💻 **Output:**

✅ Data copied successfully using FileInputStream & FileOutputStream!

🔍 **Summary Table**

Stream Type	Data Type	Main Classes	Used For
Byte Stream	Binary (8-bit)	FileInputStream, FileOutputStream	Images, audio, etc.
Character Stream	Text (16-bit)	FileReader, FileWriter	Text files
Buffered Stream	Buffered data	BufferedReader, BufferedWriter	Faster I/O
File Class	File metadata	File	Managing file properties

🧠 **Key Points to Remember**

- Always **close** streams after use with .close().
- Prefer **try-with-resources** for automatic closure (Java 7+).
- Use **Character Streams** for text files and **Byte Streams** for binary files.
- The File class deals with **file properties**, not contents.