1. File Handling - Read and Write a Text File

📌 Problem Statement:

Write a Java program that reads the contents of a text file and writes it into a new file. If the source file does not exist, display an appropriate message.

Requirements:

- Use `FileInputStream` and `FileOutputStream`.
- Handle `IOException` properly.
- Ensure that the destination file is created if it does not exist.

---

2. Buffered Streams - Efficient File Copy

📌 Problem Statement:

Create a Java program that copies a large file (e.g., 100MB) from one location to another using **Buffered Streams** (`BufferedInputStream` and `BufferedOutputStream`). Compare the performance with normal file streams.

Requirements:

- Read and write in chunks of **4 KB (4096 bytes)**.
- Use `System.nanoTime()` to measure execution time.
- Compare execution time with **unbuffered streams**.

---

3. Read User Input from Console

📌 Problem Statement:

Write a program that asks the user for their **name, age, and favorite programming language**, then saves this information into a file.

Requirements:

- Use `BufferedReader` for console input.
- Use `FileWriter` to write the data into a file.
- Handle exceptions properly.

---

4. Serialization - Save and Retrieve an Object

📌 Problem Statement:

Design a Java program that allows a user to **store a list of employees in a file** using **Object Serialization** and later retrieve the data from the file.

Requirements:

- Create an `Employee` class with fields: `id, name, department, salary`.
- Serialize the list of employees into a file (`ObjectOutputStream`).
- Deserialize and display the employees from the file (`ObjectInputStream`).
- Handle `ClassNotFoundException` and `IOException`.

5. ByteArray Stream - Convert Image to ByteArray

📌 Problem Statement:

Write a Java program that **converts an image file into a byte array** and then writes it back to another image file.

Requirements:

- Use `ByteArrayInputStream` and `ByteArrayOutputStream`.
- Verify that the new file is identical to the original image.
- Handle `IOException`.

---

6. Filter Streams - Convert Uppercase to Lowercase

📌 Problem Statement:

Create a program that reads a text file and writes its contents into another file, converting all uppercase letters to lowercase.

Requirements:

- Use `FileReader` and `FileWriter`.
- Use `BufferedReader` and `BufferedWriter` for efficiency.
- Handle character encoding issues.

---

7. Data Streams - Store and Retrieve Primitive Data

📌 Problem Statement:

Write a Java program that stores **student details** (roll number, name, GPA) in a binary file and retrieves it later.

Requirements:

- Use `DataOutputStream` to write primitive data.
- Use `DataInputStream` to read data.
- Ensure proper closing of resources.

---

8. Piped Streams - Inter-Thread Communication

📌 Problem Statement:

Implement a Java program where one thread **writes data** into a `PipedOutputStream` and another thread **reads data** from a `PipedInputStream`.

Requirements:

- Use **two threads** for reading and writing.
- Synchronize properly to prevent data loss.
- Handle `IOException`.

---

9. Read a Large File Line by Line

📌 Problem Statement:

Develop a Java program that efficiently reads a **large text file** (500MB+) **line by line** and prints only lines containing the word **"error"**.

Requirements:

- Use `BufferedReader` for efficient reading.
- Read line-by-line instead of loading the entire file.
- Display only lines containing `"error"` (case insensitive).

---

10. Count Words in a File

📌 Problem Statement:

Write a Java program that **counts the number of words in a given text file** and displays the **top 5 most frequently occurring words**.

Requirements:

- Use `FileReader` and `BufferedReader` to read the file.
- Use a `HashMap<String, Integer>` to count word occurrences.
- Sort the words based on frequency and display the top 5.