# File Handling in Java

---

## 1. Introduction to File Handling

File handling in Java allows programs to **read from** and **write to** files. Java provides multiple classes in the java.io and java.nio packages to perform file operations efficiently.

**Why File Handling?**

- Store data permanently.
- Read configuration files.
- Process large datasets.
- Log application activities.

---

## 2. Java File Class

The File class (java.io.File) represents file and directory paths.

### Example: Check if a File Exists

```
import java.io.File;
public class FileExample {
    public static void main(String[] args) {
        File file = new File("example.txt");

        if (file.exists()) {
            System.out.println("File exists!");
        } else {
            System.out.println("File does not exist.");
        }
    }
}
```

```
Output:

File does not exist.

(If example.txt does not exist.)
```

---

## 3. File Input and Output Streams

**Byte Streams (**FileInputStream **&** FileOutputStream**)**

- Used for reading/writing binary data (images, videos, etc.).

**Example: Writing to a File**

```
import java.io.FileOutputStream;

import java.io.IOException;


public class WriteFileExample {

    public static void main(String[] args) {

        try (FileOutputStream fos = new FileOutputStream("output.txt")) {

            String text = "Hello, Java File Handling!";

            byte[] bytes = text.getBytes();

            fos.write(bytes);

            System.out.println("Data written successfully.");

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

**Output (in** output.txt**):**

```
text
Hello, Java File Handling!
```

**Example: Reading from a File**

```
import java.io.FileInputStream;
import java.io.IOException;

public class ReadFileExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("output.txt")) {
            int content;
            while ((content = fis.read()) != -1) {
                System.out.print((char) content);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Output:**

```
text
Hello, Java File Handling!
```

---

## 4. Reading and Writing Text Files

**Using** FileReader **and** FileWriter

**Example: Writing to a File**

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

public class FileWriterExample {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("demo.txt")) {
            writer.write("This is a text file.");
            System.out.println("Successfully written to file.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Output (in** demo.txt**):**

**This is a text file.**

**Example: Reading from a File**

```
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("demo.txt")) {
            int character;
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Output:**

**This is a text file.**

## Using BufferedReader and BufferedWriter (Efficient for Large Files)

**Example: Writing with** BufferedWriter

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

public class BufferedWriterExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("buffered.txt"))) {
            bw.write("Using BufferedWriter for efficiency.");
            bw.newLine(); // Adds a new line
            bw.write("Second line.");
            System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Output (in** buffered.txt**):**

**Using BufferedWriter for efficiency.**

**Second line.**

**Example: Reading with** BufferedReader

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("buffered.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Output:**

```
Using BufferedWriter for efficiency.
Second line.
```

---

**5. Best Practices**

- Use try-with-resources for automatic resource management.
- Prefer BufferedReader/BufferedWriter for large files.
- Close streams properly to avoid memory leaks.
- Use NIO for better performance in modern Java.

---