1. Introduction to Java I/O

What is I/O?

Java I/O (Input–Output) refers to how a Java program **exchanges data with the outside world**.

A Java program does not work in isolation. It frequently needs to:

- Read input (keyboard, file, network, memory)
- Write output (monitor, file, network, memory)

This exchange of data is what we call **Input–Output (I/O)**.

- **Input**: Data flowing *into* the program Examples:
    - Typing on keyboard
    - Reading from a file
    - Receiving data from a network
- **Output**: Data flowing *out of* the program Examples:
    - Displaying text on monitor
    - Writing to a file
    - Sending data over a network

I/O is simply the transfer of data between a program and resources.

Java I/O Packages

Java provides two major packages:

- **java.io** → Traditional stream-based I/O
- **java.nio / java.nio.file** → New I/O (non-blocking, better file handling)

This syllabus mainly focuses on **java.io** with basic file handling.

---

2. The `java.io` Package

The `java.io` **package** provides classes for:

- File handling
- Reading and writing data
- Serialization

3. Files and Directories

The `File` Class

The `File` **class** represents:

- A file

- A directory path

> ⚠ It does **not** read or write data itself.

Creating a File Object

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

Common File Methods

Method	Description
`exists()`	Checks if file exists
`createNewFile()`	Creates a file
`delete()`	Deletes file
`isFile()`	Checks if it is a file
`isDirectory()`	Checks if directory
`listFiles()`	Lists directory contents

Directory Example

```
File dir = new File("documents");
dir.mkdir();
```

---

4. Streams in Java

A **stream** is a flow of data between a source and a destination.

Data can flow:

- From resource → program (Input Stream)
- From program → resource (Output Stream)

When you type:

```
A → S → D
```

Each character flows **one after another** as a stream of:

- bytes
- or characters

This continuous flow is what defines a stream.

1 Byte Streams

- Handle raw bytes
- Used for binary data (images, files, network)
- Base classes:
  - `InputStream`
  - `OutputStream`

---

2 Character Streams

- Handle characters (Unicode)
- Each character = **2 bytes** in Java
- Used for text data
- Base classes:
  - `Reader`
  - `Writer`

---

5. Byte Streams

Definition

Byte Streams handle **raw binary data (8-bit bytes)**.

Used for:

- Images
- Audio files
- Binary data

`InputStream` and `OutputStream` are the **base (parent) classes** for all **byte stream** classes in Java.

- `FileOutputStream` → Used to **write bytes** to a file.
- `FileInputStream` → Used to **read bytes** from a file.

Example: Writing Bytes

- Use **forward slashes** (`/`) in the path.
- If the file is in the same directory as your `.class` file, just use the filename.

[FileExample.java]

```
public class FileExample
{
  public static void main(String[] args) {
    try {
      FileOutputStream fos = new FileOutputStream("C:/MyJava/Test.txt");
      String str = "Learn Java Programming";

      byte[] b = str.getBytes(); // Convert string to bytes

      fos.write(b); // Write all bytes at once

      //Writing byte by byte
      // for (byte x : b) {
      //   fos.write(x);
      // }


      fos.close();
    } catch(FileNotFoundException e) {
      System.out.prinln(e);
    } catch(IOException e) {
      System.out.prinln(e);
    }
  }
}
```

Example: Reading Bytes

[FileExample.java]

```
public class FileExample
{
  public static void main(String[] args) {
    try {
      FileInputStream fis = new FileInputStream("C:/MyJava/Test.txt")

      int x;
      while ((x = fis.read()) != -1) { // read each byte until EOF (-1)
        System.out.print((char)x); // convert byte to character and print
      }

      fis.close();
    } catch(FileNotFoundException e) {
      System.out.prinln(e);
    } catch(IOException e) {
      System.out.prinln(e);
    }
  }
}
```

6. Character Streams

Definition

Character Streams handle **text data (16-bit Unicode)**.

Used for:

- Text files
- Console input/output

Main Classes

- `FileReader`
- `FileWriter`
- `BufferedReader`
- `BufferedWriter`

Example: Writing Text

[FileExample.java]

```
public class FileExample
{
  public static void main(String[] args) {
    try {
      FileWriter writer = new FileWriter("text.txt");
      writer.write("Hello Java");
      writer.close();

      fos.close();
    } catch(FileNotFoundException e) {
      System.out.prinln(e);
    } catch(IOException e) {
      System.out.prinln(e);
    }
  }
}
```

Example: Reading Text

[FileExample.java]

```
public class FileExample
{
  public static void main(String[] args) {
    try {
      FileReader fr = new FileReader("C:/MyJava/Test.txt")
```

```
      int x;
      while ((x = fr.read()) != -1) {
        System.out.print((char)x);
      }

      fr.close();
    } catch(FileNotFoundException e) {
      System.out.prinln(e);
    } catch(IOException e) {
      System.out.prinln(e);
    }
  }
}
```

---

7. Console Input and Output

Console Output

Using:

- `System.out.print()`
- `System.out.println()`

Example:

```
System.out.println("Welcome to Java I/O");
```

Console Input

Using `BufferedReader` or `Scanner`

Example using `BufferedReader`:

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in)
);
String name = br.readLine();
```

---

8. Reading and Writing Files

Reading Files

Approaches:

- Byte streams → `FileInputStream`

- Character streams → `FileReader`, `BufferedReader`

Buffered reading improves performance.

Example:

```
BufferedReader br = new BufferedReader(new FileReader("data.txt"));
String line = br.readLine();
```

Writing Files

Approaches:

- `FileWriter`
- `BufferedWriter`

Example:

```
BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt"));
bw.write("Java I/O Example");
bw.close();
```

---

9. Serialization Interface

What is Serialization?

Serialization is the process of converting an object into a **byte stream** so it can be:

- Stored in a file
- Sent over a network

Deserialization

Deserialization is the reverse process: converting byte stream back into an object.

`Serializable` Interface

To serialize an object, a class must implement:

```
implements Serializable
```

No methods to override — it is a **marker interface**.

---

10. Serialization and Deserialization

Serializable Class Example

```
class Student implements Serializable {
    int id;
    String name;
}
```

Serialization Example

```
ObjectOutputStream oos =
  new ObjectOutputStream(new FileOutputStream("student.ser"));
oos.writeObject(student);
oos.close();
```

Deserialization Example

```
ObjectInputStream ois =
  new ObjectInputStream(new FileInputStream("student.ser"));
Student s = (Student) ois.readObject();
ois.close();
```

---

11. Case Study: Saving User Data

Scenario

A university system saves student records to a file.

Steps

1. Create a `Student` class
2. Implement `Serializable`
3. Serialize object to file
4. Deserialize when needed

Benefits

- Data persistence
- Easy data transfer
- Object-based storage

---

12. Common Issues and Best Practices

Common Issues

- `FileNotFoundException`
- `IOException`
- Class version mismatch during deserialization

Best Practices

- Always close streams
- Use buffering
- Handle exceptions properly
- Use `transient` keyword for sensitive data
