Unit 8: I/O and Streams in Java [LH - 2Hrs]

1. java.io Package

The `java.io` package provides classes for system input and output through data streams, serialization, and the file system. It is one of the core packages in Java for handling I/O operations.

2. Files and Directories

Java provides the `File` class in the `java.io` package to work with files and directories. This class can be used to create, delete, rename, and check the existence of files and directories.

Lab 1: Files and Directories

```
import java.io.File;

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

        try {
            if (file.createNewFile()) {//check for existence and the creation of the file happen as a single operation->returns true if the file was successfully created because it did not exist before and returns false if the file already exists, so no new file is created.
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Sample Output:

```
File does not exist.
File created: example.txt
```

Explanation:

- The `File` class is used to represent the file `example.txt`.
- The `createNewFile()` method checks for existence and creates a new file if it does not already exist.

3. Streams: Byte Streams and Character Streams

Streams in Java are used to perform input and output operations. There are two types of streams:

- **Byte Streams**: Handle I/O of raw binary data. Classes include `InputStream` and `OutputStream`.

- **Character Streams**: Handle I/O of character data. Classes include `Reader` and `Writer`.

Lab 2: Byte Stream:

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

public class ByteStreamExample {
    public static void main(String[] args) {
        // Use try-with-resources to automatically close the streams after use
        try (FileInputStream in = new FileInputStream("input.txt");    // Open a
stream to read raw bytes from "input.txt"
             FileOutputStream out = new FileOutputStream("output.txt")) {    // Open a
stream to write raw bytes to "output.txt"
            int c;    // Variable to hold each byte read from the input file
            while ((c = in.read()) != -1) {    //The read() method reads a byte of
data and returns -1 when there are no more bytes to read
                out.write(c);    //Write the byte to the output file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Sample Output:

- This program reads the contents of `input.txt` and writes them to `output.txt`.

Explanation:

- `FileInputStream` reads bytes from `input.txt`.
- `FileOutputStream` writes bytes to `output.txt`.
- The `read()` method reads a byte of data, and `write()` writes a byte of data.

Lab 3: Character Stream:

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

public class CharacterStreamExample {
    public static void main(String[] args) {
        try (FileReader in = new FileReader("input.txt");    // Open a stream to read
characters from "input.txt"
             FileWriter out = new FileWriter("output.txt")) {    // Open a stream to
write characters to "output.txt"
            int c;    // Variable to hold each character read from the input file
            while ((c = in.read()) != -1) {    //The read() method reads a character
and returns -1 when there are no more characters to read
                out.write(c);    // Write the character to the output file
            }
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Sample Output:

- This program reads the contents of `input.txt` and writes them to `output.txt`.

Explanation:

- `FileReader` reads characters from `input.txt`.
- `FileWriter` writes characters to `output.txt`.
- The `read()` method reads a character, and `write()` writes a character.

Difference Between Byte Streams and Character Streams:

- Byte streams handle binary data, while character streams handle text data.
- Byte streams use `InputStream` and `OutputStream`, while character streams use `Reader` and `Writer`.
- Use Byte Streams when working with binary data (e.g., images, audio, serialized objects).
- Use Character Streams when working with text data (e.g., .txt, .csv, or any file containing human-readable text).

Note

```
If we use a byte stream (e.g., InputStream, FileInputStream), the read() method reads bytes .
If we use a character stream (e.g., Reader, FileReader, InputStreamReader), the read() method reads characters .
```

4. Reading/Writing Console Input/Output

Java provides `System.in`, `System.out`, and `System.err` for console I/O. The `Scanner` class is commonly used for reading input from the console.

Lab 4: Reading/Writing Console Input/Output

```
import java.util.Scanner;

public class ConsoleIOExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name);
    }
}
```

Sample Output:

```
Enter your name: Sharat
Hello, Sharat
```

Explanation:

- `Scanner` reads input from the console.
- `nextLine()` reads a line of text input.

5. Reading and Writing Files

Java provides various classes for reading and writing files, such as `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.

Lab 5: Reading and Writing Files

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

public class FileReadWriteExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
// Opens a buffered stream to read lines from "input.txt"
             BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt")))
{    // Opens a buffered stream to write lines to "output.txt"
            String line;    // Variable to hold each line read from the input file
            while ((line = reader.readLine()) != null) {    //Read one line at a time
from the input file until the end of the file is reached and returns null when there
are no more lines to read
                writer.write(line);    // Write the current line to the output file
                writer.newLine();    // Add a newline character for formatting multi-
line text(\n for linux and \r\n for windows)
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Sample Output:

- This program reads lines from `input.txt` and writes them to `output.txt`.

Explanation:

- `BufferedReader` reads text from `input.txt`.
- `BufferedWriter` writes text to `output.txt`.
- `readLine()` reads a line of text, and `write()` writes a line of text.

6. The Serialization Interface

Serialization is the process of converting an object into a byte stream, and deserialization is the process of converting a byte stream back into an object. The `Serializable` interface is used to mark classes that can be serialized.

Lab 6: The Serialization Interface

```
import java.io.*;
```

```java
class Student implements Serializable {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{name='" + name + "', age=" + age + "}";
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        Student student = new Student("Sharat", 20);

        // Serialization
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("student.ser"))) {
            out.writeObject(student);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialization
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("student.ser"))) {
            Student deserializedStudent = (Student) in.readObject();
            System.out.println(deserializedStudent);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Sample Output:

```
Student{name='Sharat', age=20}
```

Explanation:

- The `Student` class implements the `Serializable` interface.
- `ObjectOutputStream` serializes the `Student` object and writes it to `student.ser`.
- `ObjectInputStream` deserializes the `Student` object from `student.ser`.

7. Serialization & Deserialization

- **Serialization**: Converts an object into a byte stream.
- **Deserialization**: Converts a byte stream back into an object.

a. Difference Between Serialization and Deserialization:

- Serialization is the process of converting an object to a byte stream, while deserialization is the process of converting a byte stream back to an object.

b. Purpose of Serialization

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

1. **Persisted**: Saved to a file or database for future use.
2. **Transmitted**: Sent over a network to another system.
3. **Cloned**: Used to create deep copies of objects.
4. **Cached**: Stored temporarily to improve performance.

c. Advantages of Serialization

1. **Simplicity**: Easy to implement with minimal code (just implement `Serializable`).
2. **Persistence**: Allows objects to be saved and restored later.
3. **Network Communication**: Enables objects to be sent across systems via sockets or APIs.
4. **Deep Copying**: Facilitates cloning of complex objects with nested references.
5. **Portability**: Serialized data can be transferred between different platforms (Java-to-Java).

d. Limitations of Serialization

1. **Performance**: Slower and more memory-intensive compared to custom binary formats.
2. **Versioning Issues**: Changes to the class structure (e.g., adding/removing fields) can break deserialization unless managed with `serialVersionUID`.
3. **Security Risks**:
    - Serialized data is not encrypted by default, making it vulnerable to tampering or exposure.
    - Deserialization of untrusted data can lead to serious vulnerabilities like remote code execution.
4. **Non-Interoperable**: Java's serialization format is proprietary and not compatible with other programming languages.
5. **Large File Size**: Serialized files can be larger than equivalent data in other formats like JSON or Protobuf.

e. Alternatives to Java Serialization

1. **JSON (JavaScript Object Notation)**:
2. **XML (eXtensible Markup Language)**:

f. How Serialization Enables Object Transfer

Serialization converts an object into a **byte stream**, making it possible to transfer or store the object in various ways:

1. **Sent Over a Network**:
    - The byte stream is transmitted using protocols like **TCP/IP** or **HTTP**.
    - On the receiving end, the byte stream is deserialized to reconstruct the original object.
2. **Saved to a File or Database**:

    - The byte stream is written to a file or database for long-term storage.
    - Later, the stored byte stream can be read and deserialized to recreate the object.

3. **Cloned or Cached**:
    - The byte stream can be used to create a **deep copy** of the object or temporarily store it in memory for caching purposes.

Summary

- The `java.io` package provides classes for I/O operations.
- Files and directories can be managed using the `File` class.
- Byte streams handle binary data, while character streams handle text data.
- Console I/O can be performed using `System.in`, `System.out`, and `Scanner`.
- Files can be read and written using classes like `FileReader`, `FileWriter`, `BufferedReader`, and `BufferedWriter`.
- Serialization and deserialization allow objects to be converted to and from byte streams using the `Serializable` interface.