**Assignment IO Operations**

Q1. What is Input and Output Stream in Java

In Java, **Input Stream** and **Output Stream** are abstract classes that represent the system of reading from and writing to data sources like files, network connections, or memory buffers. They are part of the java.io package and are primarily used to handle byte streams. Java also provides specialized classes for handling character streams (Reader and Writer), which are used for reading and writing textual data.

**1. Input Stream (java.io.InputStream):**

* Represents an input source from which data is read (e.g., file, network, etc.).
* It is an abstract class, meaning that concrete implementations (like FileInputStream, BufferedInputStream, etc.) are used to read data.
* The basic operations involve reading bytes, arrays of bytes, or streams of bytes from the input source.

**Common Methods**:

* int read(): Reads the next byte of data and returns it. If the end of the stream is reached, it returns -1.
* int read(byte[] b): Reads up to b.length bytes of data from the input stream into an array of bytes.
* void close(): Closes the stream and releases any resources associated with it.

**2. Output Stream (java.io.OutputStream):**

* Represents an output destination where data is written (e.g., file, network, etc.).
* Like InputStream, this is also an abstract class. Concrete implementations (like FileOutputStream, BufferedOutputStream, etc.) are used for writing data.
* The basic operations involve writing bytes, arrays of bytes, or streams of bytes to the output destination.

**Common Methods**:

* void write(int b): Writes the specified byte to the output stream.
* void write(byte[] b): Writes b.length bytes from the specified byte array to the output stream.
* void flush(): Flushes the output stream and forces any buffered output bytes to be written out.
* void close(): Closes the stream and releases any resources associated with it.

IMPLEMENTATION:

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class FileCopyExample {

public static void main(String[] args) {

FileInputStream inStream = null;

FileOutputStream outStream = null;

try {

// Open the input stream to read from the source file

inStream = new FileInputStream("source.txt");

// Open the output stream to write to the destination file

outStream = new FileOutputStream("destination.txt")

int byteData

// Read one byte at a time from the source file and write it to the destination file

while ((byteData = inStream.read()) != -1) {

outStream.write(byteData);

}

System.out.println("File copied successfully!");

} catch (IOException e) {

e.printStackTrace();

} finally {

// Close the streams in the finally block to ensure resources are released

try {

if (inStream != null) inStream.close();

if (outStream != null) outStream.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

Q2.What are the methods of OutputStream

The OutputStream class in Java is an abstract class that provides methods for writing bytes to a destination (such as a file, memory, or network socket). Concrete subclasses like FileOutputStream, BufferedOutputStream, and others implement these methods to provide actual functionality. Since it's a byte stream, it deals with binary data.

**Common Methods of OutputStream:**

1. **void write(int b)**:
   * **Description**: Writes the specified byte (the low eight bits of the int) to the output stream.
   * **Parameters**:
     + b: An integer representing a single byte. Only the lowest 8 bits of this integer are written; the higher 24 bits are ignored.
   * **Throws**: IOException if an I/O error occurs.
   * outputStream.write(65); // Writes the byte for character 'A' (ASCII 65)
2. **void write(byte[] b)**:

**Description**: Writes an array of bytes to the output stream.

**Parameters**:

b: The byte array to write.

**Throws**: IOException if an I/O error occurs.

**Example**:

byte[] byteArray = {65, 66, 67}; // A, B, C in ASCII

outputStream.write(byteArray);

**3.void write(byte[] b, int off, int len)**:

* **Description**: Writes a subarray of bytes from the byte array b, starting at the specified offset off and writing len bytes.
* **Parameters**:
  + b: The byte array to write from.
  + off: The start offset in the data.
  + len: The number of bytes to write.
* **Throws**: IOException if an I/O error occurs.

Ex:

byte[] byteArray = {65, 66, 67, 68}; // A, B, C, D in ASCII

* outputStream.write(byteArray, 1, 2); // Writes B, C

**void flush()**:

* **Description**: Flushes the output stream, forcing any buffered bytes to be written out. This method ensures that any data that may have been buffered is written immediately.
* **Throws**: IOException if an I/O error occurs.
* **Use Case**: Use flush() when you want to ensure that all data is written to the destination before proceeding.
* **Example**

outputStream.flush(); // Ensures all data is written

**void close()**:

* **Description**: Closes the output stream and releases any system resources associated with it. Once the stream is closed, further write operations will throw an IOException.
* **Throws**: IOException if an I/O error occurs.
* **Use Case**: Always close streams to free up resources, typically in a finally block or using a try-with-resources statement.
* **Example**:

outputStream.close(); // Closes the stream and releases resources

**Example Using OutputStream:**

Here’s a simple example demonstrating the use of OutputStream to write to a file:

import java.io.FileOutputStream;

import java.io.IOException;

public class OutputStreamExample {

public static void main(String[] args) {

String data = "Hello, OutputStream!";

byte[] byteArray = data.getBytes();

// Try-with-resources to ensure the stream is closed automatically

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

// Write the entire byte array to the output stream

outputStream.write(byteArray);

// Flush the stream to ensure all data is written out

outputStream.flush();

System.out.println("Data written to file.");

} catch (IOException e) {

e.printStackTrace();

}

}

}

Q3.What is serialization in Java

**Serialization in Java** is the process of converting an object into a byte stream, so it can be saved to a file, transmitted over a network, or persisted in memory. This byte stream can later be deserialized, which means converting it back into a copy of the original object.

**Key Points:**

1. **Serialization**: Converts an object into a byte stream.
2. **Deserialization**: Converts the byte stream back into the object.
3. **Usage**: Often used for saving objects to files, sending objects over the network, or storing them in memory.
4. **Implements Serializable Interface**: A class must implement the Serializable interface to be serializable.

**Example:**

import java.io.\*;

class Employee implements Serializable {

private static final long serialVersionUID = 1L;

String name;

int id;

public Employee(String name, int id) {

this.name = name;

this.id = id;

}

}

public class SerializationExample {

public static void main(String[] args) {

Employee emp = new Employee("John", 101);

// Serialization

try (FileOutputStream fileOut = new FileOutputStream("employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(emp);

} catch (IOException e) {

e.printStackTrace();

}

// Deserialization

try (FileInputStream fileIn = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

Employee deserializedEmp = (Employee) in.readObject();

System.out.println("Deserialized Employee: " + deserializedEmp.name + ", " + deserializedEmp.id);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

Q4. What is the Serializable interface in Java

The **Serializable** interface in Java is a **marker interface** (an interface with no methods) that enables an object to be serialized. Serialization allows the object to be converted into a byte stream, which can then be stored to a file, transmitted over a network, or saved in memory, and later reconstructed through deserialization.

**Key Points:**

* **Marker Interface**: Serializable has no methods, it only marks a class as serializable.
* **Required for Serialization**: A class must implement Serializable for its objects to be serialized.
* **Transient Keyword**: Fields marked as transient are not serialized.

**Example:**

import java.io.Serializable;

class Employee implements Serializable {

private static final long serialVersionUID = 1L;

String name;

int id;

public Employee(String name, int id) {

this.name = name;

this.id = id;

}

}

Q5. What is deserialization in Java

**Deserialization** in Java is the process of converting a byte stream back into a copy of the original object. This byte stream is typically obtained through serialization, which converts an object into a byte stream for storage or transmission.

**Key Points:**

Purpose: Reconstruct an object from a byte stream.

Requires Serializable: The class of the object must implement the Serializable interface.

Restores Object State: The object's state is restored from the byte stream.

**Example:**

If you have serialized an object to a file, you can deserialize it to read it back:

import java.io.FileInputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

public class DeserializationExample {

public static void main(String[] args) {

try (FileInputStream fileIn = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

Employee emp = (Employee) in.readObject();

System.out.println("Deserialized Employee: " + emp.name + ", " + emp.id);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

Q6. How is serialization achieved in Java

Serialization in Java is achieved by converting an object into a byte stream, which can be saved to a file, transmitted over a network, or stored in memory. The steps are as follows:

1. **Implement Serializable Interface**:
   * The class of the object must implement the Serializable interface, which is a marker interface with no methods.
2. **Use ObjectOutputStream for Serialization**:
   * Create an ObjectOutputStream to write the object's byte stream to a destination (e.g., a file).
3. **Use ObjectInputStream for Deserialization**:
   * Create an ObjectInputStream to read the byte stream and reconstruct the object.

**Example:**

**Serialization:**

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectOutputStream;

class Employee implements Serializable {

private static final long serialVersionUID = 1L;

String name;

int id;

public Employee(String name, int id) {

this.name = name;

this.id = id;

}

}

public class SerializationExample {

public static void main(String[] args) {

Employee emp = new Employee("John", 101);

try (FileOutputStream fileOut = new FileOutputStream("employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(emp);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Q7. How is deserialization achieved in Java

Deserialization in Java is achieved by converting a byte stream back into an object. The steps involved are:

1. **Ensure Serializable Interface**: The class of the object being deserialized must implement the Serializable interface.
2. **Use ObjectInputStream**:
   * Create an ObjectInputStream to read the byte stream that represents the serialized object.
3. **Read the Object**:
   * Use the readObject() method of ObjectInputStream to reconstruct the object from the byte stream.

**Deserialization:**

import java.io.FileInputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

public class DeserializationExample {

public static void main(String[] args) {

try (FileInputStream fileIn = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

Employee emp = (Employee) in.readObject();

System.out.println("Deserialized Employee: " + emp.name + ", " + emp.id);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

Q8.How can you avoid certain member variables of class from getting Serialized

To avoid certain member variables of a class from being serialized in Java, you can use the transient keyword. This keyword tells the Java serialization mechanism to skip the serialization of the marked field.

**How to Use transient:**

1. **Declare Fields as transient**:
   * Mark the fields you want to exclude from serialization with the transient keyword.
2. **Effect on Serialization**:
   * The transient fields will not be saved during serialization and will be initialized to their default values during deserialization.

**Example:**

import java.io.FileOutputStream;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.ObjectOutputStream;

import java.io.ObjectInputStream;

import java.io.Serializable;

class Employee implements Serializable {

private static final long serialVersionUID = 1L;

String name;

transient int salary; // This field will not be serialized

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

}

public class TransientExample {

public static void main(String[] args) {

Employee emp = new Employee("John", 50000);

// Serialize the Employee object

try (FileOutputStream fileOut = new FileOutputStream("employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(emp);

} catch (IOException e) {

e.printStackTrace();

}

// Deserialize the Employee object

try (FileInputStream fileIn = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

Employee deserializedEmp = (Employee) in.readObject();

System.out.println("Deserialized Employee Name: " + deserializedEmp.name);

System.out.println("Deserialized Employee Salary: " + deserializedEmp.salary); // Will be 0 because salary is transient

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

Q9.What classes are available in the Java IO File Classes API

In the Java I/O File Classes API (found in java.io and java.nio.file packages), several key classes are used for file handling. Here’s a brief overview:

**java.io Package:**

1. **File**:
   * Represents a file or directory path.
   * Provides methods for file and directory operations (e.g., createNewFile(), delete(), exists()).
2. **FileInputStream**:
   * Used for reading bytes from a file.
   * Extends InputStream.
3. **FileOutputStream**:
   * Used for writing bytes to a file.
   * Extends OutputStream.
4. **FileReader**:
   * Used for reading characters from a file.
   * Extends Reader.
5. **FileWriter**:
   * Used for writing characters to a file.
   * Extends Writer.
6. **BufferedReader**:
   * Buffers characters for efficient reading.
   * Wraps around FileReader.
7. **BufferedWriter**:
   * Buffers characters for efficient writing.
   * Wraps around FileWriter.
8. **PrintWriter**:
   * Provides methods for writing formatted text to a file.
   * Can be used with FileWriter or BufferedWriter.

**java.nio.file Package (introduced in Java 7):**

1. **Paths**:
   * Provides static methods to get Path objects.
   * Example: Paths.get("file.txt").
2. **Path**:
   * Represents a path in the file system.
   * Provides methods for file operations (e.g., toFile(), getFileName()).
3. **Files**:
   * Provides static methods for file operations (e.g., readAllBytes(), write(), copy(), delete()).
4. **FileSystems**:
   * Provides access to the FileSystem object.
5. **FileSystem**:
   * Represents a file system and provides methods to work with paths and files.

Q10. What is Difference between Externalizable and Serialization interface.

| **Serializable** | **Externalizable** |
| --- | --- |
| A serializable interface is used to implement serialization. | An externalizable interface used to implement Externalization |
| Serializable is a marker interface i.e. it does not contain any method. | The externalizable interface is not a marker interface and thus it defines two methods *writeExternal()* and *readExternal()*. |
| Serializable interface passes the responsibility of serialization to JVM and the programmer has no control over serialization, and it is a default algorithm. | The externalizable interface provides all serialization responsibilities to a programmer and hence JVM has no control over serialization. |
| Serialization using a serializable interface has bad performance. | Serialization using an externalizable interface has better performance. |
| Default serialization does not require any no-arg constructor. | A public no-arg constructor is required while using an Externalizable interface. |
| It is hard to analyze and modify class structure because any change in structure may break serialization. | It is relatively easy to analyze and modify class structure because of complete control over serialization logic. |
| Using a serializable interface we save the total object to a file, and it is not possible to save part of the object. | Base on our requirements we can save either the total object or part of the object. |
| Transient keywords play an important role here. | Transient keywords won’t play any role. |