Zoho Schools for Graduate Studies

Notes

File Handling in Java

What is File Handling?

File handling in Java is the process of creating, reading, writing, updating, and deleting files using Java I/O classes.

Why use File Handling?

- To store data permanently (even after program ends)
- To read or write data from files
- For logging, configuration, report generation, user data, etc.

When to use it?

- Save application state
- Load external data
- Generate logs or reports
- Read/write configuration files
- Work with external file systems

Volatile data

- Temporary: Lost when the program or system ends.
- Stored in: RAM or cache.
- Used for: Intermediate computations, session data.
- Examples:
    - Local variables
    - Objects in heap
    - Session memory

Persistent Data

- Permanent: Stored until explicitly deleted.
- Stored in: Disk, database, or file system.
- Used for: Saving user data, configurations, logs.
- Examples:
    - Text files
    - Serialized objects
    - Database records

File Class :

The File class in Java is part of the java.io package.

It represents the name and path of a file or directory on the disk.

It is used to manage file system operations like:

- Creating files or directories
- Deleting them
- Checking file properties (readable, writable, size, etc.)
- Listing directory contents

The File class plays a supporting role in file handling by managing the file structure before you use streams to read/write data.

Common Uses in File Handling:

Task	Usage Example
Check if file exists	- file.exists()
Create a new file	- file.createNewFile()
Delete a file	- file.delete()
Check if file or directory	- file.isFile(), file.isDirectory()
Get file metadata	- file.length(), file.getName(), file.getPath()
Create directory	- file.mkdir() or file.mkdirs()
List files in folder	- file.list() or file.listFiles()

Example : Using File in File Handling in java

```
import java.io.File;
import java.io.IOException;

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

        try {
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }

            if (file.exists()) {
                System.out.println("File name: " + file.getName());
                System.out.println("Path: " + file.getAbsolutePath());
                System.out.println("Writable: " + file.canWrite());
                System.out.println("Readable: " + file.canRead());
                System.out.println("File size: " + file.length() + "
bytes");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

Streams:

Streams in Java are part of the java.io package and are used for reading from and writing to files (or other input/output sources). They allow you to process data sequentially (one byte or one character at a time) or in chunks.

Types of Streams

Streams are divided into two main categories based on how the data is processed:

1. Byte Streams (for binary data)

- Used for: Reading and writing binary data like images, audio files, PDFs, etc.
- Classes: InputStream, OutputStream, FileInputStream, FileOutputStream, etc.
- Data Processing: Works at the byte level.

2. Character Streams (for text data)

- Used for: Reading and writing text data (characters).
- Classes: Reader, Writer, FileReader, FileWriter, BufferedReader, BufferedWriter, etc.
- Data Processing: Works at the character level (handles encoding/decoding automatically).

How Streams are Used in File Handling?

Streams enable you to read from and write to files in a more efficient and manageable way. Here's how they fit in:

1. File Creation and Metadata:

- File class is used to create or check for files, directories, and paths.
- Streams are used to read from or write to these files once they exist.

2. Reading and Writing Data:

- For text files, you would use character streams like FileReader and FileWriter.
- For binary files, you would use byte streams like FileInputStream and FileOutputStream.

Reading and Writing with Streams

Example : Writing to a File using FileOutputStream

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

public class FileWriteExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String data = "Hello, Java Streams!";
            fos.write(data.getBytes());  // Writes byte data to file
            System.out.println("Data written to file successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example : Reading from a File using FileInputStream

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

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

Key Points About Streams

- Sequential: Streams read and write data one piece at a time (byte or character).
- Buffered vs Unbuffered: Buffered streams (like BufferedReader) store data in a buffer for more efficient reading/writing.
- Automated Encoding: Character streams handle character encoding/decoding automatically, whereas byte streams work directly with raw bytes.

Most Frequently Used in Real Projects:

Scenario	Stream Classes Used
Copying a file (image/pdf)	- FileInputStream, FileOutputStream
Reading large text file line by line	- BufferedReader, FileReader
Writing logs or text output	- BufferedWriter, PrintWriter
Saving objects to disk	- ObjectOutputStream, FileOutputStream
Reading object from file	- ObjectInputStream, FileInputStream

Commonly Used Methods of FileReader

Method	Description
int read()	- Reads one character at a time and returns its Unicode int value; returns -1 if end of file
Int read(char[]cbuf)	- Reads characters into a char array
int read(char[] cbuf, int offset, int length)	- Reads up to length characters into cbuf starting at offset
void close()	- Closes the stream and releases system resources

Commonly Used Methods of FileWriter

Method	Description
void write(int c)	- Writes a single character
void write(char[] cbuf)	- Writes an array of characters
void write(char[] cbuf, int off, int len)	- Writes part of a char array
void write(String str)	- Writes a full string
void write(String str, int off, int len)	- Writes a part of a string
void flush()	- Forces any buffered output to be written
void close()	- Closes the stream and releases resources

Console Class in Java

What is the Console Class?

- The Console class is part of java.io package.
- It is used for reading user input from the console and writing output.
- Provides secure methods like readPassword() to read sensitive data without showing it on screen.

How Console supports File Handling?

We can:

- Take user input using Console.
- Write it to a file using FileWriter, BufferedWriter, etc.

Example: Take input from user and write to file

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

public class ConsoleToFile {
    public static void main(String[] args) {
        Console console = System.console();

        if (console != null) {
            String name = console.readLine("Enter your name: ");

            try (FileWriter writer = new FileWriter("userdata.txt", true)) {
                writer.write("Name: " + name + "\n");
                System.out.println("Data saved to file.");
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("Console is not available. Run from
terminal.");
        }
    }
}
```

Common Methods in Console Class

Method		Description
readLine()	-	Reads a line of text from console
readLine(String fmt, ...)	-	Reads formatted line
readPassword()	-	Reads password input (hidden)
printf(String fmt, ...)	-	Writes formatted output to console
format(String fmt, ...)	-	Alias of printf()
flush()	-	Flushes the output buffer
reader()	-	Returns a Reader for console input
writer()	-	Returns a PrintWriter for output

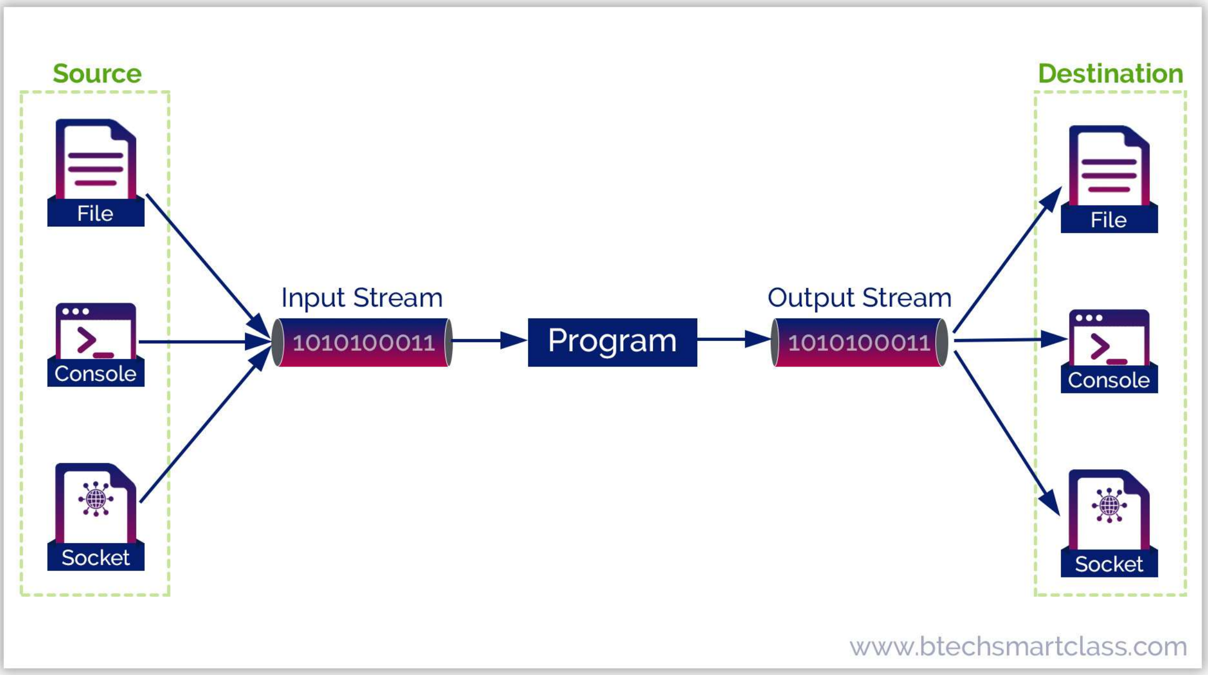
FileDescriptor Class in Java

What is FileDescriptor?

- FileDescriptor is a class in the java.io package.
- It is a wrapper for the OS-level file handle (descriptor).
- It represents an open file, socket, or another byte stream source/sink.
- Used internally by stream classes like FileInputStream, FileOutputStream.

Why Use FileDescriptor?

- To get low-level control of a file or stream.
- Useful when you want to reuse the same file handle across multiple streams.
- It allows access to standard input/output and other system resources.



Serialization and Deserialization

Serialization

Serialization is the process of converting a Java object into a byte stream. so it can be stored (in a file, database) or transmitted (over a network).

Why Serialization?

Serialization is used to:

- Persist object state (save it for later use)
- Send objects over network
- Store objects in files
- Cache session data
- Deep copy objects

How to Serialize an Object

1. Implement Serializable interface

```
import java.io.Serializable;

class Employee implements Serializable {
    int id;
    String name;

    Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Note : Serializable is a marker interface — it has no methods.

2. Use ObjectOutputStream to serialize

```
import java.io.*;

public class SerializeExample {
    public static void main(String[] args) throws Exception {

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

        FileOutputStream fos = new FileOutputStream("employee.ser");

        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(emp);  // Serializing object

        oos.close();

        System.out.println("Object serialized successfully.");
    }
}
```

Variables and Their Roles in Serialization

1. Normal Data Members

- Serialized by default
- Saved into the byte stream
- Restored during deserialization

2. Transient Variables

- NOT serialized
- Skipped while writing object to stream
- Becomes default value (0, null, false, etc.) when deserialized

3. Static Variables

- NOT serialized
- Belong to the class, not the object
- Not saved in the object's byte stream

Deserialization

Deserialization is the process of reconstructing a Java object from a byte stream. It's the reverse of serialization.

Why Deserialization?

- To retrieve saved objects (from files or databases)
- To receive objects over a network
- To restore application state

How to Perform Deserialization

You need:

- A .ser file (or byte stream) created by serialization
- The original class must still implement Serializable

Step 1: Class must be Serializable

```
import java.io.Serializable;

class Employee implements Serializable {
    int id;
    String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Step 2: Deserialize using ObjectInputStream

```
import java.io.*;

public class DeserializeExample {
    public static void main(String[] args) throws Exception {
        FileInputStream fis = new FileInputStream("employee.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Employee emp = (Employee) ois.readObject();  // Deserialization
        ois.close();

        System.out.println("ID: " + emp.id);
        System.out.println("Name: " + emp.name);
    }
}
```

Frequently Asked Interview Questions on File Handling and (De)Serialization

1. What is file handling in Java?
2. Which package in Java supports file handling?
3. What is the purpose of the File class?
4. How do you create a new file using Java?
5. How do you check if a file or directory exists?
6. How can you delete a file in Java?
7. What are byte streams and character streams?
8. What is the difference between FileInputStream and FileReader?
9. How do you read a text file line by line?
10. What is the role of BufferedReader and BufferedWriter?
11. What is the difference between PrintWriter and FileWriter?
12. How do you append data to an existing file?
13. How do you list all files in a directory?
14. What is the use of the FileDescriptor class?
15. What is the difference between absolute and relative file paths?
16. What is serialization in Java?
17. What is deserialization in Java?
18. Which interface must a class implement to be serializable?
19. What is serialVersionUID and why is it important?
20. What happens if the serialVersionUID doesn't match?

21. What is the use of the transient keyword in serialization?
22. Are static fields serialized? Why or why not?
23. How do you serialize and deserialize an object in Java?
24. Can a class be serialized if its superclass is not serializable?
25. What are some common exceptions during serialization and deserialization?

References:

https://www.geeksforgeeks.org/java/file-handling-in-java/

https://medium.com/@AlexanderObregon/navigating-javas-file-api-8505b8d395f4

https://w3schools.tech/tutorial/java/java_file_class

https://www.geeksforgeeks.org/java/file-class-in-java/

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

https://www.sanfoundry.com/file-handling-in-java/