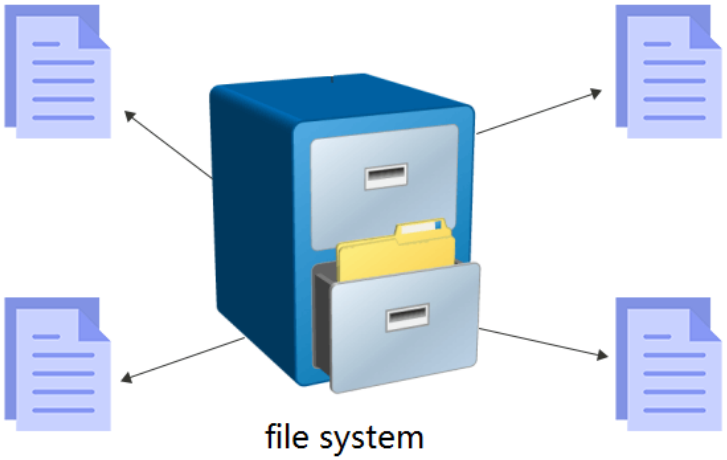
IO Operations In Java (File Handling)

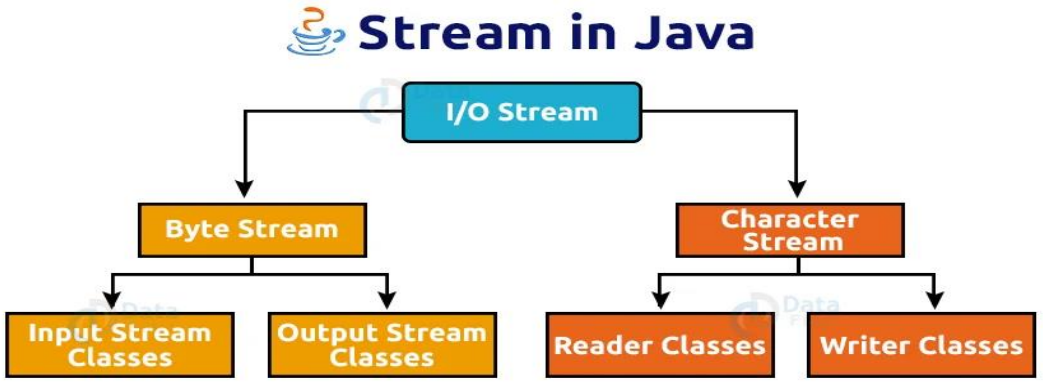


Created By : Gayathri ramadurgum

Date : 14 September, 2023

INTRODUCTION

- File handling is a crucial aspect of programming in Java, allowing you to read, write, create, delete, and manipulate files and directories on your computer's file system.
- Java provides a rich set of classes and methods in its standard library to perform file handling operations.
  - The **java.io package** contains classes for working with files and streams. However, starting from Java 7, the **java.nio package**, also known as the New I/O API, provides more flexible and efficient file handling capabilities.



FILE CLASS

- The File class in Java is used for working with file and directory paths, but it doesn't directly provide methods for reading from or writing to files like InputStream and OutputStream do. Instead, the File class offers functionality for creating, deleting, checking the existence of, and inspecting files and directories on the file system.

```
import java.io.*;

class FileDemo {
    public static void main(String[] args) {
        File f = new File("cricket.txt");
        System.out.println(f.exists()); // false

        try {
            f.createNewFile();
            System.out.println(f.exists()); // true
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The line : *File f=new File(“cricket.txt");* checks whether the sbc.txt file is already available or not

- If available, the reference of the object “f” will simply refer to it
- Else, it wont create any physical file but it will create a java object representing the name of the file.

IMPORTANT METHODS OF FILE CLASS

1. boolean exists(): Returns true if the physical file or directory is available.

2. boolean createNewFile() : This method 1st checks whether the physical file is already available or not if it is already available then this method simply returns false without creating any physical file. If this file is not already available then it will create a new file and returns true

3. boolean mkdir(): This method 1st checks whether the directory is already available or not. If it is already available then this method simply returns false without creating any directory. If this directory is not already available then it will create a new directory and returns true

4. boolean isFile(): Returns true if the File object represents a physical file.

5. boolean isDirectory(): Returns true if the File object represents a directory.

FILEWRITER CLASS

- In Java, FileWriter is a class in the java.io package that is used for writing character data to a file. It's specifically designed for writing text-based data to files. FileWriter is a subclass of the Writer class, an abstract class in the java.io package that provides an abstract representation of a character stream.

Ways to Declare :

❖The above 2 constructors are meant for overriding the data to the file.

```
FileWriter fw=new FileWriter(String name);

FileWriter fw=new FileWriter(File f);
```

❖Instead of overriding if we want append operation then we should go for the following 2 constructors.

```
FileWriter fw=new FileWriter(String name, true);

FileWriter fw=new FileWriter(File f, true);
```

Example :

```
import java.io.*;
public class FileWriterExample {
    public static void main(String[] args) {
        File fileName = new File("example.txt");
        try  {
            FileWriter obj = new FileWriter(fileName,true);
            // Write a string to the file
            String data = "Hello, FileWriter!";
            obj.write(data);

            // Append additional text to the file
            obj.write("\nThis line will get appended.");

            // Flush any buffered data to the file
            obj.flush();

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

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

BUFFERED STREAMS

Using BufferedReader and BufferedWriter can be more efficient and convenient when working with files compared to directly using FileReader and FileWriter because :

- Buffered streams use internal buffers to read and write data in larger chunks. This reduces the number of I/O operations, which can be slow, and improves overall performance.
- While reading data by FileReader we have to read character by character instead of line by line which is not convenient to the programmer.
- While writing data by FileWriter compulsory we should insert line separator(\n) manually, but BufferedWriter automatically uses the appropriate platform-specific newline separator when you call write(String str) method.

Example :

```java
import java.io.*;

public class BufferedWriter {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("example.txt");
            BufferedWriter buffer = new BufferedWriter(writer);
            buffer.write("Hello World.");
            buffer.close();
            System.out.println("Success");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Other methods of BufferedWriter class :

1. write(int ch);
2. write(char[] ch);
3. write(String s);
4. flush();
5. close();
6. newline(); ---> ensures that you insert the newline separator in a platform-independent manner. Different operating systems use different line separators (e.g., "\n" on Unix/Linux, "\r\n" on Windows, etc.). By using newLine(), your code remains portable across different platforms, and Java handles the translation of line separators.

Example :

```java
import java.io.*;

public class BufferedReaderExample {
    public static void main(String args[]) {
        try {
            FileReader fr = new FileReader("D:\\testout.txt");
            BufferedReader br = new BufferedReader(fr);
            int i;
            //Used to read a single character
            while ((i = br.read()) != -1) {
                System.out.print((char) i);
            }
            // Close the BufferedReader and FileReader in a finally block
            br.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Other methods of BufferedReader class :

1. int read();
2. int read(char[] ch);
3. void close();
4. String readLine(); ---> It attempts to read the next line and return it , from the File. if the next line is not available then this method returns null.

SERIALIZATION

- The process of saving (or) writing the state of an object to a file is called serialization
  - By using FileOutputStream and ObjectOutputStream classes we can achieve a serialization process.
- The process of reading the state of an object from a file is called Deserialization
  - By using FileInputStream and ObjectInputStream classes we can achieve DeSerialization.