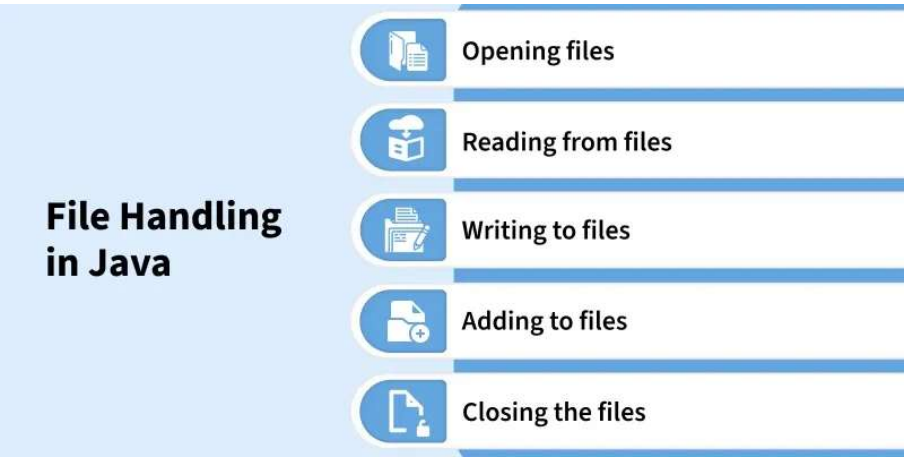
File Handling in Java

In Java, file handling means working with files like **creating** them**, reading data**, **writing data** or **deleting** them. It helps a program **save and use information permanently on the computer**.



Why is File Handling Required?

- To **store data permanently** instead of keeping it only in memory.
- To **read and write data** from/to files for later use.
- To **share data** between different programs or systems.
- To **organize and manage large data** efficiently.

To support file handling, Java provides the File class in the **java.io package**.

File Class

File class in Java (from the **java.io** package) is used to represent the name and path of a file or directory. It provides methods to create, delete, and get information about files and directories.

Example

```
// Importing File Class
import java.io.File;

class Files{
   public static void main(String[] args){
     // File name specified
       File obj = new File("myfile.txt");
       System.out.println("File Created!");  }}
```

Output:

File Created!

In Java, **I/O streams are used to perform input and output operations on files**. So, let's first understand what **streams** are.

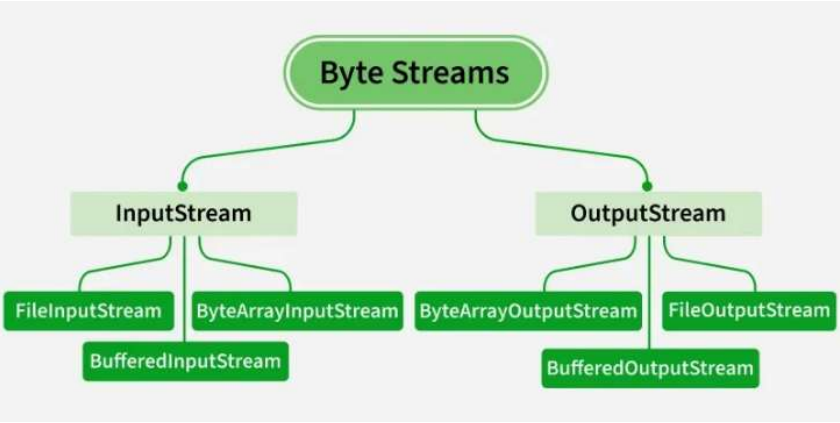
I/O Streams in Java

In Java, **I/O streams** are the fundamental mechanism for handling **input and output** operations. They provide a uniform way to read data from various sources (files, network, memory) and write data to different destinations.

Java I/O streams are categorized into **two main** types based on the type of data they handle:

1. Byte Streams

In Java, **Byte Streams are used to handle raw binary data** such as **images**, **audio files**, **videos** or **any non-text file**. They work with data in the form of **bytes**.



The **two main abstract classes** for **byte streams** are:

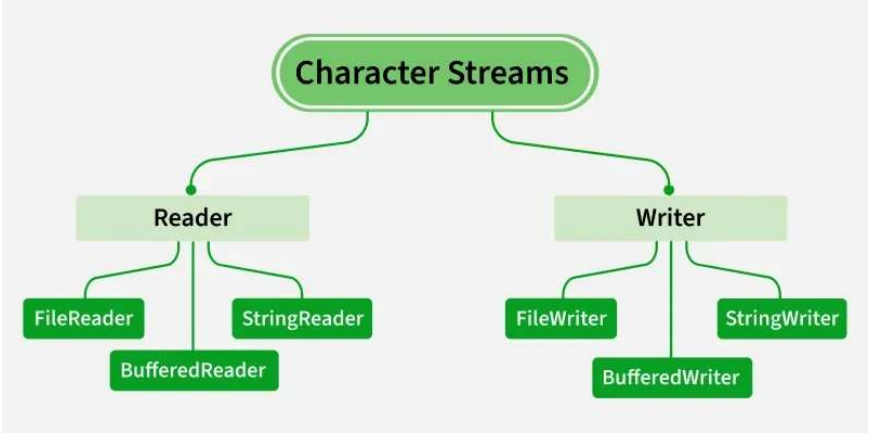
- **InputStream:** for reading data (input)
- **OutputStream:** for writing data (output)

Since abstract classes cannot be used directly, we use their implementation classes to perform actual **I/O operations**.

- **FileInputStream:** reads raw bytes from a file.
- **FileOutputStream:** writes raw bytes to a file.
- **BufferedInputStream / BufferedOutputStream:** use buffering for faster performance.
- **ByteArrayInputStream:** reads data from a byte array as if it were an input stream.
- **ByteArrayOutputStream:** writes data into a byte array, which grows automatically.

2. Character Streams

In Java, Character Streams are used to handle **text data**. They work with **16-bit Unicode characters**, making them suitable for international text and language support.



The **two main abstract classes** for character streams are:

- **Reader**: Base class for all character-based input streams (reading).
- **Writer**: Base class for all character-based output streams (writing).

Since abstract classes cannot be used directly, we use their implementation classes to perform actual **I/O operations**.

- **FileReader:** reads characters from a file.
- **FileWriter:** writes characters to a file.
- **BufferedReader:** reads text efficiently using buffering; also provides readLine() for reading lines.
- **BufferedWriter:** writes text efficiently using buffering.
- **StringReader:** reads characters from a string.
- **StringWriter:** writes characters into a string buffer.

File Operations

The following are the several operations that can be performed on a file in Java:

1. Create a File

- To create a file in Java, you can use the **createNewFile() method**.
- If the file is successfully created, it will **return a Boolean value true** and false if the **file already exists**.

Example

```
import java.io.File;

public class CreateFile{
   public static void main(String[] args)     {
        File Obj = new File("myfile.txt");
         // Creating File
        if (Obj.createNewFile()) {
            System.out.println("File created: " + Obj.getName());      }
         else {
            System.out.println("File already exists.");  }    }}
```

Output:

```
File created: myfile.txt
File already exists.
```

2. Write to a File

We use the FileWriter class along with its write() method to write some text to the file.

Example

```
import java.io.FileWriter;

public class WriteFile  {
   public static void main(String[] args)     {
      // Writing Text File
         FileWriter Writer = new FileWriter("myfile.txt");
         // Writing File
         Writer.write("Files in Java are seriously good!!");
         Writer.close();
         System.out.println("Successfully written."); }}}
```

Output:

Successfully written.

3. Read from a File

In Java, the **read() method** is used with **classes** like **FileReader** or **InputStream** to read data from a file one character or byte at a time.

- It **returns an integer value** representing the **character or byte read**.
- When the end of the file is reached, the method returns **-1** indicating no more data is available.

Example

```
import java.io.File;
import java.util.Scanner;

public class ReadFile  {
   public static void main(String[] args)     {
      // Reading File
         File Obj = new File("myfile.txt");
         Scanner Reader = new Scanner(Obj);
```

```
// Traversing File Data
 while (Reader.hasNextLine()) {
   String data = Reader.nextLine();
   System.out.println(data);       }
  Reader.close();    }}}
```

Output:

Files in Java are seriously good!!

4. canRead a File

canRead() check if the file is **readable**.

Example

```
import java.io.File;

public class CanReadExample {
   public static void main(String[] args) {
        // Create a new file
        File obj = new File("myfile.txt");
        // create a new, empty file
        if (obj.createNewFile()) {
           System.out.println("File created: " + obj.getName());}
        else {
           System.out.println("File already exists."); }
        // Check if the file is readable
        if (obj.canRead()) {
           System.out.println("The file is readable."); }
        else {
           System.out.println("The file is not readable.");   }}}}
```

Output:

```
File created: myfile.txt
The file is readable.
```

5. canWrite a File

canWrite() checks whether the file can be written to by the program.

Example

```
import java.io.File;

public class CanWriteExample {
   public static void main(String[] args) {
         // Create a new file
         File obj = new File("myfile.txt");
         if (obj.createNewFile()) {
            System.out.println("File created: " + obj.getName());}
         else {
            System.out.println("File already exists."); }
        // Check if the file is writable
         if (obj.canWrite()) {
            System.out.println("The file is writable."); }  // One-line explanation
         else {
            System.out.println("The file is not writable."); }}}
```

Output:

```
File created: myfile.txt
The file is writable.
```

6. Existance of a File

exists() checks whether the specified file or directory exists on the file system.

Example

```
import java.io.File;

public class ExistsExample {
   public static void main(String[] args) {
      // Create a File object
      File obj = new File("myfile.txt");
      // Check if the file exists
      if (obj.exists()) {
         System.out.println("The file exists.");  } // One-line explanation
      else {
         System.out.println("The file does not exist."); } }}
```

Output:

The file exists.

7. Getting a path

getAbsolutePath() returns the full path of the file or directory in the file system.

Example

```
import java.io.File;

public class AbsolutePathExample {
   public static void main(String[] args) {
      // Create a File object
      File obj = new File("myfile.txt");
      // Print the absolute path of the file
      System.out.println("Absolute Path: " + obj.getAbsolutePath());  }}
```

Output

Absolute Path: C:\Users\MSI-13\eclipse-workspace\gui1\myfile.txt

8. Delete a File

We use the **delete()** method to delete a file.

Example

```
import java.io.File;

public class DeleteFile  {
   public static void main(String[] args)     {
      File Obj = new File("myfile.txt");
      // Deleting File
      if (Obj.delete()) {
         System.out.println("The deleted file is : " + Obj.getName());     }
      else {
         System.out.println(
            "Failed in deleting the file."); }}}
```

Output:

The deleted file is : myfile.txt

Failed in deleting the file.

Writing data to a file

To write data to a file, you must create an instance of the **PrintWriter class**

To use the PrintWriter **import java.io.PrintWriter**.

PrintWriter writer = new **PrintWriter**("File.txt");

The PrintWriter class

The **PrintWriter class** allows you to write data to a file using the **print(str)** and **println(str)** methods that you have also been using with **System.out**

- **If close() is never called, the data will not be saved!**
- **The PrintWriter class always overwrites a file that already exists.**
- **Running the program multiple times will cause the data to get overwritten each time the program is run**
- **To avoid this, Java provides a FileWriter class, this is used alongside the PrintWriter class. It is not a direct replacement!**

Example

```
import java.io.PrintWriter;

public class Printfile {

    public static void main(String[] args) {

        PrintWriter outputFile = new PrintWriter("Names.txt");
        outputFile.println("Ann");
        outputFile.println("Bob");
        outputFile.println("Carol");
        outputFile.close(); }}
```

Appending text to a file

To append data to a file:

– Create an instance of **FileWriter**

– The first argument is the **filename**. The **second argument** is **true/false** for whether to append or overwrite:

– Pass the **FileWriter** instance into the constructor for the **PrintWriter**

```
FileWriter fw = new FileWriter("Names.txt", true);
PrintWriter outputFile = new PrintWriter(fw);
```

Example

```
import java.io.FileWriter;
import java.io.PrintWriter;

FileWriter fw = new FileWriter("Names.txt", true);
PrintWriter outputFile = new PrintWriter(fw);
outputFile.println("Ann");
outputFile.println("Bob");
outputFile.println("Carol");
outputFile.close();
```

Names.txt content after first run

```
Ann
Bob
Carol
```

Names.txt content after second run

```
Ann
Bob
Carol
Ann
Bob
Carol
```

As with any other file system operation you can write to a specific location by **specifying the full path for windows**. E.g.:

```
PrintWriter outFile = new PrintWriter("C:\\Users\\MSI-13\\eclipse-workspace\\gui1\\PriceList.txt")
```

Detecting the end of the file

The Scanner has a method called **hasNext()** which **returns true** if there is **more of the file which has yet to be read**.

```
File file = new File("Names.txt");
Scanner inputFile = new Scanner(file);
while (inputFile.hasNext()) {
   System.out.println(inputFile.nextLine());}
inputFile.close();
```