File Handling in Java

Ming Jing

Software College

Spring 2019

Outline

Introduction

Java being one of the most popular programming languages provides extensive support to various functionalities like **database**, **sockets**, etc. One such functionality is File Handling in Java.

File Handling is necessary to perform various tasks on a file, such as read, write, etc.

I/O Overview

- I/O = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

What is File Handling in Java?

File handling in Java implies reading from and writing data to a file. The File class from the **java.io package**, allows us to work with different formats of files. In order to use the File class, you need to create an object of the class and specify the filename or directory name.

```
// Import the File class
import java.io.File

// Specify the filename
File file = new File("filename.txt");
```

Java uses the concept of a **stream** to make I/O operations on a file.

What is a Stream?

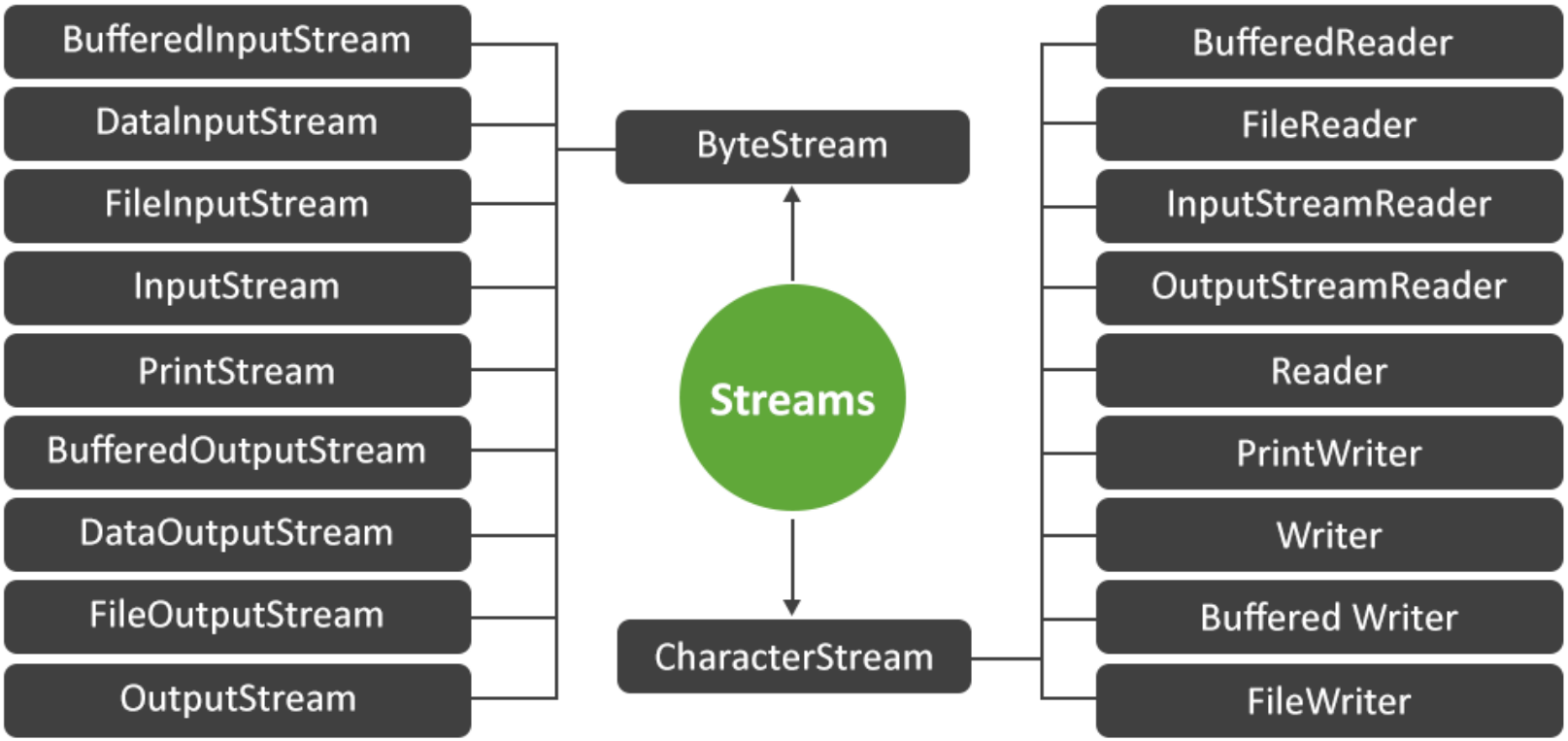
In Java, Stream is a sequence of data which can be of two types.

1. **Byte Stream**. When an input is provided and executed with byte data, then it is called the file handling process with a byte stream.
2. **Character Stream**.

Stream Overview

- Stream: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
- Input stream: a stream that provides input to a program
  - System.in is an input stream
- Output stream: a stream that accepts output from a program
  - System.out is an output stream
- A stream connects a program to an I/O object
  - System.out connects a program to the screen
  - System.in connects a program to the keyboard

Types of Streams in Java IO



Java File Methods

canRead()	Boolean	It tests whether the file is readable or not
canWrite()	Boolean	It tests whether the file is writable or not
createNewFile()	Boolean	This method creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	It tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

File Operations in Java

Create a File

```java
// Import the File class
import java.io.File;

// Import the IOException class to handle errors
import java.io.IOException;

public class CreateFile {
    public static void main(String[] args) {
        try {
            // Creating an object of a file
            File myObj = new File("filename.txt");
            if (myObj.createNewFile()) {
                System.out.println("File created: " + myObj.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

Get File information

```java
import java.io.File; // Import the File class

public class FileInformation {
    public static void main(String[] args) {
        // Creating an object of a file
        File myObj = new File("NewFileName.txt");
        if (myObj.exists()) {
            // Returning the file name
            System.out.println("File name: " + myObj.getName());
            // Returning the path of the file
            System.out.println("Absolute path: " + myObj.getAbsolutePath());
            // Displaying whether the file is writable
            System.out.println("Writeable: " + myObj.canWrite());
            // Displaying whether the file is readable or not
            System.out.println("Readable " + myObj.canRead());
            // Returning the length of the file in bytes
            System.out.println("File size in bytes " + myObj.length());
        } else {
            System.out.println("The file does not exist.");
        }
    }
}
```

```
File name: NewFileName.txt
Absolute path: D:/NewFileName.txt
Writable: true
Readable true
File size in bytes 52
```

Write to a File

```java
// Import the FileWriter class
import java.io.FileWriter;
// Import the IOException class to handle errors
import java.io.IOException;

public class WriteToFile {

  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("D:/FileHandlingNewFilef1.txt");
      // Writes this content into the specified file
      myWriter.write(
        "Java is the prominent programming language of the millenium!"
      );

      // Closing is necessary to retrieve the resources allocated
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

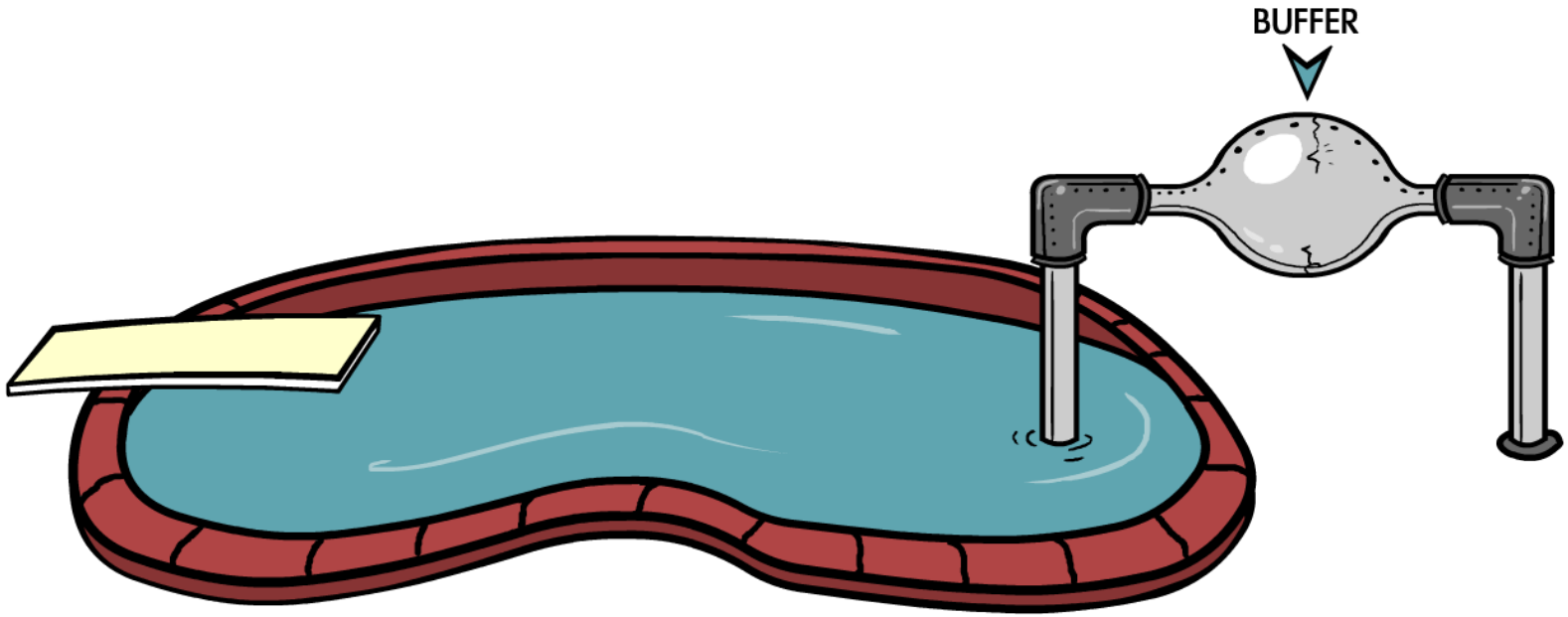
Read from a File

```java
// Import the File class
import java.io.File;
// Import this class to handle errors
import java.io.FileNotFoundException;
// Import the Scanner class to read text files
import java.util.Scanner;

public class ReadFromFile {

  public static void main(String[] args) {
    try {
      // Creating an object of the file for reading the data
      File myObj = new File("D:/FileHandlingNewFilef1.txt");
      Scanner myReader = new Scanner(myObj);
      while (myReader.hasNextLine()) {
        String data = myReader.nextLine();
        System.out.println(data);
      }
      myReader.close();
    } catch (FileNotFoundException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Delete a File

```java
// Import the File class
import java.io.File;

public class DeleteFile {

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

Buffer



In general, disk access is much slower than the processing performed in memory; that's why it's not a good idea to access the disk a thousand times to read a file of 1,000 bytes. To minimize the number of times the disk is accessed, Java provides buffers, which serve as reservoirs of data.

Unbuffered I/O

Each read or write request is handled directly by the underlying OS.

This can make a program much less efficient, since each such request often triggers **disk access**, **network activity**, or some other operation that is relatively expensive.

Buffered I/O

To reduce this kind of overhead, the Java platform implements buffered I/O streams.

Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty.

Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

Line-Oriented I/O

line terminator

- carriage-return/line-feed sequence ("\\r\\n")
- a single carriage-return ("\\r")
- a single line-feed ("\\n")

BufferedReader

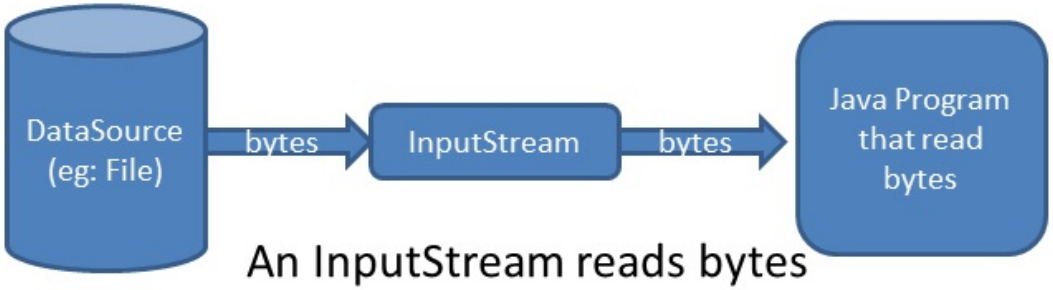
Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of **characters**, **arrays**, and **lines**.

```
BufferedReader br
    = new BufferedReader(new FileReader("foo.in"));
```

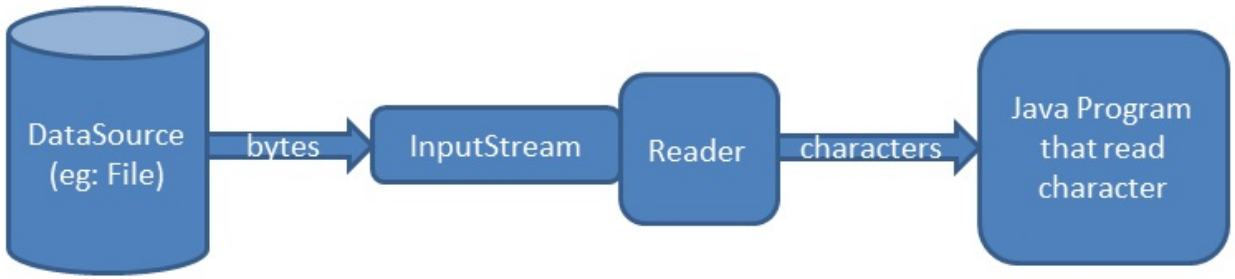
CopyLines

```
public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("input.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

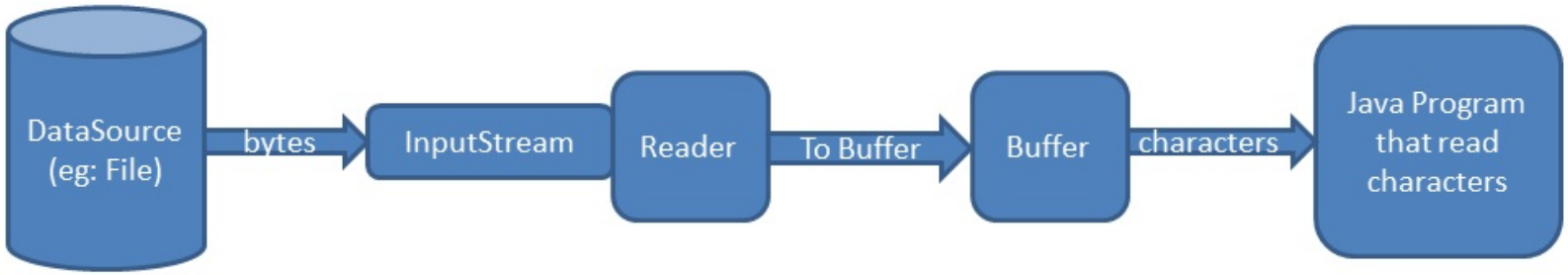
Conclusion



An InputStream reads bytes



A Reader works with an InputStream and reads characters



A BufferedReader pre-fetches characters in to a buffer for performance