COURSE:- Java Programming
Sem:- I
Module: 4.1

By:
Dr. Ritu Jain,
Associate Professor,
DY Patil University,
Navi Mumbai

Java I/O

File Handling

- File Handling enables us to store the output of any particular program in a file and allows us to perform certain operations on it.
- File handling means reading and writing data to a file. We can:
  - access files of a file system with the help of **java.io.File** class
  - Read and write file data with the help of **I/O stream classes**
- File class and stream classes are in the **java.io** package.
- To interact with files in file system, we need to connect with them.
- Data is stored in file system / memory as 0 and 1 bytes.

Classes for file IO operations: File class

- The File class in Java is used to represent the path of a file or folder.
- It helps in creating, deleting, and checking details of files or directories, but not in reading or writing data.
- It acts as an abstract representation of file and directory names in the system.
- In Java, directories are often treated as files. Most of the times, we use same classes and methods to operate on files and directories.

File class

- File class is used to:
  - read information about existing files and directories
  - list the contents of a directory
  - creating new directories or files,
  - deleting and renaming directories or files, etc

File class

- Most of the classes defined by java.io operate on I/O streams for reading and writing data within file, however File class can't read or write data with in a file, although it can be passed as a reference to many stream classes to read or write data.
- **File class simply describes the properties of a file itself.**
- A file object often is initialized with a String containing either an absolute or relative path to the file or directory within the file system.
    - **Absolute path:** full path from root directory to file or directory,
        - **In windows:** An absolute path **starts with a drive letter** (like C: or D:)
            - C:\Users\Documents\data.txt

File class

- **Relative path**: path from cwd to file or directory.
  - File myFile = new File("data/Text1.txt");
  - In this example, data/Text1.txt is a relative path.
  - Java will look for a directory named ”data” within the current working directory, and then for a file named Text1.txt inside that data directory.

Constructors of Java File Class

1. **File(String pathname):** Creates a new File instance from a String path.

   Eg: File file1 = new File("c:/Java_Prg/Text1.txt");

- **File(String parent, String child):** Creates a new File instance from a parent and child String.
  - **Eg: File file1 = new File ("c:/Java_Prg", "Text1.txt");**

- **File(File parent, String child):** Creates a new File instance from a parent File instance and a child String.
  - Eg: **File par = new File("c:/Java_Prg");**

    File file2= new File(par, "MyFile1.txt");

- When working with an instance of File class, keep in mind that it only represent a path to a file.

Important Methods of File Class

- **String getName()**:retrieve the name of file or directory
- **String getParent():** returns the parent directory that the path is contained or null if there is none.
- **boolean exists():** Tests whether the file or directory denoted by this abstract pathname exists.
- **String getAbsolutePath():** Returns the absolute name of the file or directory.
- **boolean isDirectory():** It tests whether the file reference is a directory within the file system.
- **boolean isFile()**: It tests whether the file reference is a file within the file system
- **boolean isAbsolute():** It tests whether this abstract pathname is absolute.
- **boolean createNewFile() throws IOException:** It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
  - Returns `true` if the named file does not exist and was successfully created;
  - `false` if the named file already exists

Important Methods of File Class

- **long length()**: Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
- **boolean delete():** Delete the file or directory, return true only if successful. If the instance denotes a directory, then it must be empty in order to be deleted.
- **boolean mkdir():** It creates the directory named by this abstract pathname. Returns true if and only if the directory was created; false otherwise
- **boolean mkdirs():** It creates the directory named by this abstract pathname including any non existent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories. Returns `true` if and only if the directory was created, along with all necessary parent directories; `false` otherwise

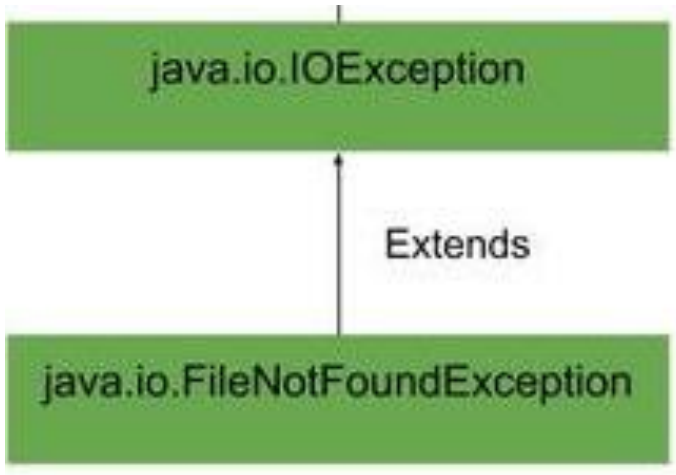
```java
import java.io.File;
class FileDemo {
  public static void main(String args[]) {
        File f1 = new File("C:/JavaPrg/JavaIO/output.txt");
        //File f1 = new File("output.txt"); //rel path
        //File f1 = new File("C:/JavaPrg/JavaIO", "output.txt");
        //File f1 = new File("C:/JavaPrg/JavaIO");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println(f1.canRead() ? "is readable" : "is not readable");
        System.out.println("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(f1.isFile() ? "is file" : "not a file");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File size: " + f1.length() + " Bytes");

        File f2=new File("newfile");
        try{
                if(f2.createNewFile())
                        System.out.println("Created new file");
                else
                        System.out.println("new file not created");
           }catch(Exception e){
                System.out.println("Exception occured");
        }

        File f3=new File("newdir1");
        if(f3.mkdir())
                System.out.println("Created new dir using mkdir");
        else
                System.out.println("new dir not created");

        File f4=new File("C:/JavaPrg/JavaIO/JavaIO_1/newdir2");

        if(f4.mkdirs())
                System.out.println("Created new dir using mkdirs");
        else
                System.out.println("new dir not created");

        File f5=new File("filedel.txt");
        if(f5.delete())
                System.out.println("file deleted");
        else
                System.out.println("file not deleted");

  }
}
```

```
File Name: output.txt
Path: C:\JavaPrg\JavaIO\output.txt
Abs Path: C:\JavaPrg\JavaIO\output.txt
Parent: C:\JavaPrg\JavaIO
exists
is writeable
is readable
is not a directory
is file
is absolute
File size: 74 Bytes
new file not created
new dir not created
new dir not created
file deleted
```

I/O Exceptions

- Two exceptions play an important role in I/O handling.
  - **IOException:** If an I/O error occurs, an **IOException** is thrown.
  - **FileNotFoundException:** In many cases, if a file cannot be opened, a FileNotFoundException is thrown.

- **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**.



I/O Stream

- Java programs perform I/O through streams.
- **Stream:** A stream is a sequence of data that flows from source to destination. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- Package **java.io** contains all the classes required for different types of input and output operations.

Input Stream and Output Stream

- There are two kinds of Streams in java:
  - **Input Stream** : Java application uses an input stream to read data from a source.
  - **Output Stream** : Java application uses an output stream to write data to a destination.

Byte Stream vs. Character Stream

- **Byte Stream:**
  - Java **byte streams** are used to perform reading and writing data byte by byte (8 bits).
  - Classes names end with InputStream, OutputStream
- **Character stream:**
  - are used to perform reading and writing data character by character.
  - Classes names end with Reader, Writer
- java.io has similar classes for both byte stream and character streams.

The Stream Classes

- Java's stream-based I/O is built upon four abstract classes:
- **InputStream**, **OutputStream**, **Reader**, and **Writer**.
- They are used to create several concrete stream subclasses.
- Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.
- **InputStream** and **OutputStream** are designed for byte streams.
- **Reader** and **Writer** are designed for character streams.
- The byte stream classes and the character stream classes form separate hierarchies.
- In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

IOStream classes

- InputStream, OutputStream, Reader, Writer are all abstract classes.

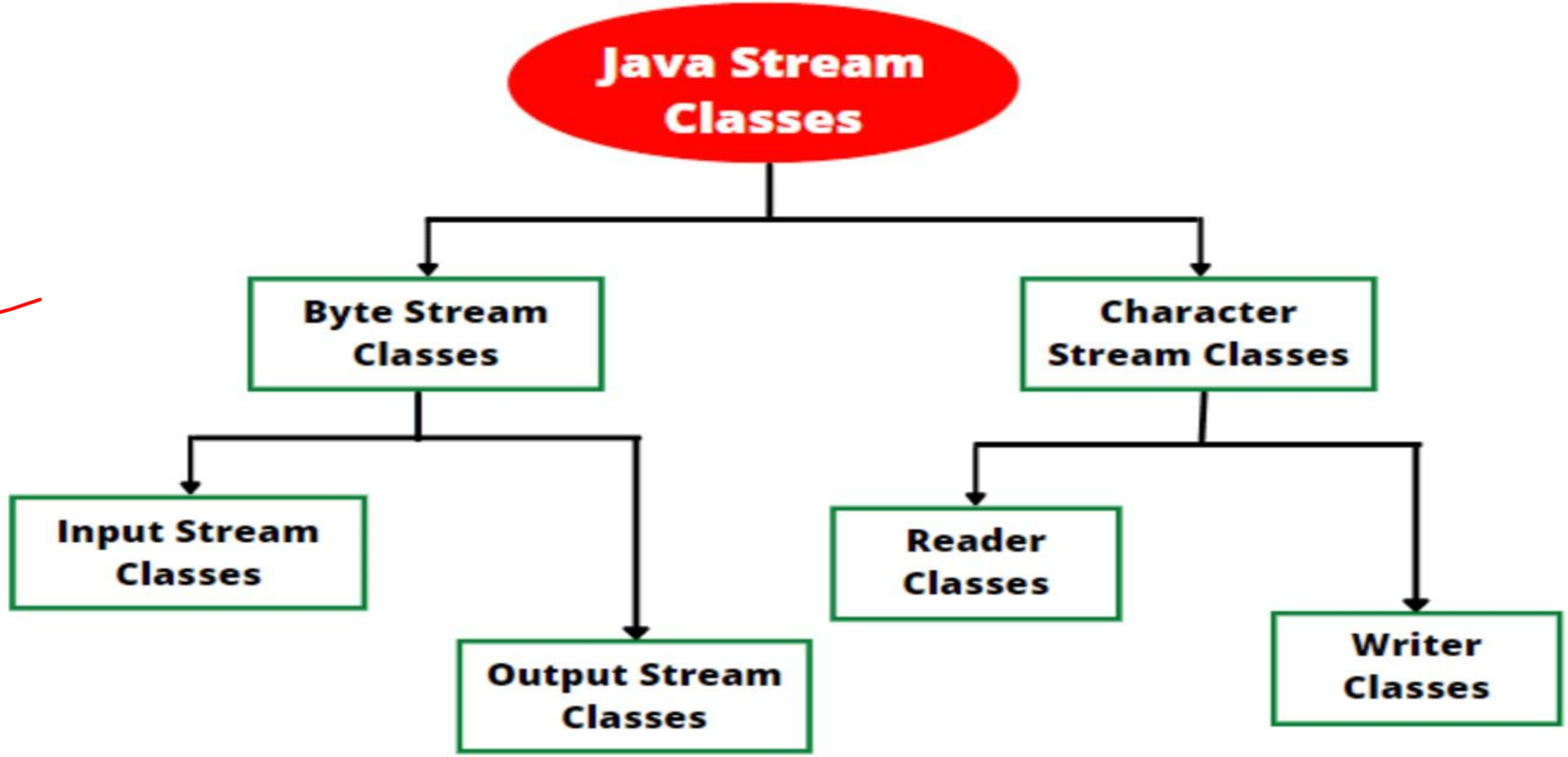


Fig: Classification of Java Stream Classes

Character Stream vs Byte Stream Differences

Aspect	Character Stream	Byte Stream
Data type handled	16-bit Unicode characters (text)	8-bit raw data (binary)
Classes end with	Reader / Writer	InputStream / OutputStream
Suitable for	Text files, Unicode data	Images, audio, video, binary files
Conversion	Converts bytes to characters automatically	No conversion, works with raw bytes
Examples	FileReader, FileWriter	FileInputStream, FileOutputStream

Important Byte stream classes

Stream class	Description
InputStream	It is an abstract class and a superclass of all classes representing an input stream of bytes.
OutputStream	This abstract class is the superclass of all classes representing an output stream of bytes.
FileInputStream	A `FileInputStream` obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing raw bytes to a `File`.
BufferedInputStream	Reads byte data from an existing InputStream in a buffered manner, which improves efficiency and performance
BufferedOutputStream	Write byte data to an existing OutputStream in a buffered manner, which improves efficiency and performance

Note: To release resources used by these streams, close()should be called directly or by try-with-resources.

Character Streams

- Java **Character streams** are used to perform input and output for 16-bit Unicode characters.
- There are two kinds of character stream classes:
  - **Reader stream classes**
    - Read 16 bit *characters* from files
    - very similar to input stream classes, except input streams use *bytes* as their fundamental unit of information, where reader streams use *characters*.
  - **Writer stream classes**
    - Perform output operations using characters
    - very similar to output stream classes, except output streams use *bytes* as their fundamental unit of information, where writer streams use *characters*.
- Though there are many classes related to character streams, the Stream classes used for file operations are **FileReader** and **FileWriter**.

Some important Character Stream classes.

Stream class	Description
Reader	Abstract class that define character stream input. Super class of all other Character oriented input stream classes.
Writer	Abstract class that define character stream output. Super class of all other Character oriented output stream classes.
FileReader	Input stream that reads from file character by character.
FileWriter	Output stream that writes to file character by character.
BufferedReader	Reads character data from an existing InputStream in a buffered manner, which improves efficiency and performance
BufferedWriter	Write character data to an existing OutputStream in a buffered manner, which improves efficiency and performance

Reader class

- **Reader** is an abstract class that defines Java's model of streaming character input.
- All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions.

Methods defined by Reader

Method	Description
void close( )	Closes the input source. Further read attempts will generate an IOException.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char buffer[ ])	Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
int read(char buffer[ ], int offset, int numChars)	Attempts to read up to numChars characters into buffer starting at buffer[offset], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
long skip(long numChars)	Skips over numChars characters of input, returning the number of characters actually skipped.

Methods defined by Reader

Method	Description
boolean markSupported()	Tests if this input stream supports the mark and reset methods.
public void mark(int readlimit)	• Marks the current position in this input stream. • A subsequent call to the reset method repositions this stream at the last marked position so that subsequent reads re-read the same bytes. • readlimit arguments tells this input stream to allow that many bytes to be read before the mark position gets invalidated.
public void reset()	Reset the pointer to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored. The `skip` method may end up skipping over some smaller number of bytes, if reaching end of file before `n` bytes have been skipped. The actual number of bytes skipped is returned.

Writer class

- Abstract class for writing to character streams.
- Most methods in the abstract Writer class and its standard subclasses are declared to throw IOException.
- This is because I/O operations inherently deal with external systems (files, network connections, etc.) where errors like disk full, permission denied, or network disconnection can occur, which are all represented by IOException

Methods of Writer class

Note: All the methods specified in the table throws IOException

Method	Description
void write(String text)	It is used to write the string into the stream.
void write(int c)	It is used to write the char into the stream
void write(char[] c)	It is used to write char array into the stream
void write (char[] cbuf, int off, int len)	Writes a portion of an array of characters: cbuf - Array of characters off - Offset from which to start writing characters len - Number of characters to write
void write(String str, int off, int len)	Writes a **substring** (part of the string). off - Offset from which to start writing characters len - Number of characters to write

Methods of Writer class

Note: All the methods specified in the table throws IOException

Writer append(char ch)	Appends ch to the end of the invoking output stream. Return a reference to the invoking stream
Writer append(CharSequence chars)	Appends chars to the end of the invoking output stream. Return a reference to the invoking stream
void flush()	It is used to flushes the buffered data through the stream
void close()	It is used to close the stream.

FileWriter Class: Constructors

Constructor	Description
FileWriter(String file)	Constructs a FileWriter object given a file name as String
FileWriter(File file)	Constructs a FileWriter object given a File object. It gets file name in File object.
FileWriter(String file, boolean append)	Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.
FileWriter(File file, boolean append)	Constructs a FileWriter object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning.

```
import java.io.FileWriter;
import java.io.IOException;
public class FileWriterMethodsExample {
    public static void main(String[] args) {
        try (FileWriter fw = new FileWriter("methodsExample.txt")) {
            fw.write(65);  // Writes ‘A’                          // 1. write(int c)

            char[] letters = {'J', 'a', 'v', 'a’};
            fw.write(letters);  // Writes "Java“                  //  2. write(char[] cbuf)

            fw.write(letters, 1, 3);  // Writes "ava“             //  3. write(char[] cbuf, int off, int len)

            fw.write("\nWelcome to FileWriter example!\n");          // 4. write(String str)

            String text = "Programming";                          //// 5. write(String str, int off, int len)
            fw.write(text, 0, 4);  // Writes "Prog"

            System.out.println("Data written successfully!");

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

Content of methodsExample.txt:
AJavaava
Welcome to FileWriter example!
Prog

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

public class FileWriterAppendExample {
    public static void main(String[] args) {
        try {
            // true → enables append mode
            FileWriter writer = new FileWriter("C:/JavaPrg/JavaIO/Sample1.txt", true);

            // Write new content at the end of the existing file
            writer.write("\nAppending new content using FileWriter.");
            writer.write("\nThis line will be added without deleting old data.");

            System.out.println("Data successfully appended to file.");

            // Always close the writer
            writer.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```
C:\JavaPrg\JavaIO>java FileWriterAppendExample
Data successfully appended to file.
```

FileReader

- The **FileReader** class creates a **Reader** that you can use to read the contents of a file.
- Its two most commonly used constructors are shown here:
  - FileReader(String *filePath*)
  - FileReader(File *fileObj*)
- Either can throw a **FileNotFoundException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

```
import java.io.FileReader;
public class FileReaderEx1 {
    public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("C:/JavaPrg/JavaIO/Sample.txt");
        int i;
        while((i=fr.read())!=-1)
        System.out.print((char)i);
        fr.close();
    }
}
```

```
C:\JavaPrg\JavaIO>java FileReaderEx1
Welcome to the
World of Java Programming
```

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

public class CopyFile {
    public static void main(String[] args) {
        // Source and destination file names
        String sourceFile = "input.txt";
        String destFile = "output.txt";

        // Try-with-resources (automatically closes streams)
        try (FileReader fr = new FileReader(sourceFile);
             FileWriter fw = new FileWriter(destFile)) {

            char[] buffer = new char[1024];   // read in chunks
            int charsRead;

            // Read characters into buffer and write to output
            while ((charsRead = fr.read(buffer)) != -1) {
                fw.write(buffer, 0, charsRead);
            }

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

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

```
C:\JavaPrg\JavaIO>java CopyFile
File copied successfully!
```

What is a Buffer?

- A buffer is a small portion of the device memory storage used to temporarily store some amount of data.
- Usually, Buffers use the RAM of the device to store the temporary data, and hence, accessing data from the buffer is much faster than accessing the same amount of data from the hard drive.

High level stream and low level stream

- In Java's I/O system, **low-level streams** connect directly to a data source (like a file or network socket), while **high-level streams** wrap around other streams to provide additional functionality and abstraction, such as buffering or reading/writing primitive data types.
- **Examples of low level stream:**
  - FileInputStream and FileOutputStream: For reading and writing raw bytes to a file.
  - FileReader and FileWriter: For reading and writing raw character data to a file.

- **Examples of High level Stream:**
  - BufferedInputStream and BufferedOutputStream (or BufferedReader and BufferedWriter for characters): Improve I/O performance by using a buffer to minimize physical access to the I/O device.
  - InputStreamReader is a **high-level** (or processing/wrapping) stream. It does not connect directly to a data source but instead wraps an underlying low-level byte stream to provide additional functionality: bridging byte streams to character streams.
  - PrintWriter: Provides convenient methods for printing formatted text data.

BufferedReader

- **BufferedReader** improves performance by buffering input.
- **Constructors:**
  - **BufferedReader(Reader *inputStream*):** creates a buffered character stream using a default buffer size.
- Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed.
- **BufferedReader** implements the **mark( )** and **reset( )** methods, and **BufferedReader.markSupported( )** returns **true**.
- **Method of BufferedReader in addition to overridden methods of Readers class:**
  - **public String readLine():**
    - Reads a line of text.
    - It returns a String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached without reading any characters

Example of BufferedReader class

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("input.txt"))) {

            String line;
            while ((line = br.readLine()) != null) {    // reads one line at a time
                System.out.println(line);
            }

        } catch (IOException e) {
             System.out.println("An error occurred while reading the file”);
}
    }
}
```

BufferedWriter

- A **BufferedWriter** is a **Writer** that buffers output.
- Using a **BufferedWriter** can improve performance by reducing the number of times data is actually physically written to the output device.
- A **BufferedWriter** has a constructor:
  - **BufferedWriter(Writer *outputStream*):** creates a buffered stream using a buffer with a default size.
- **Methods of BufferedWriter in addition to overridden methods of Writer class:**
  - **public void newLine():** Writes a line separator. The line separator string is defined by the system property line.separator

BufferedWriter Example

```
import java.io. *;
public class BufferedWriterExample {
    public static void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("output.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            bw.write("Hello from BufferedWriter!");
            bw.newLine(); // writes a platform-specific newline
            bw.write("This writes efficiently using a buffer.");
             bw.newLine();
            bw.write("BufferedWriter reduces disk I/O.");
              System.out.println("Data written successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
       finally{
            bw.close();
       }
    }
}
```

Example: Using BufferedReader and BufferedWriter to copy the content of one file to another file

```
import java.io.IOException;
public class FileCopyBuffered {

    public static void main(String[] args) {
        String sourceFilePath = "source.txt";
        String destinationFilePath = "destination.txt";

        try (BufferedReader reader = new BufferedReader(new FileReader(sourceFilePath));
             BufferedWriter writer = new BufferedWriter(new FileWriter(destinationFilePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                writer.write(line);
                writer.newLine(); // Add the new line character back
            }
            System.out.println("File copied successfully!");

        } catch (IOException e) {
            System.err.println("Error copying file: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

The Predefined Streams: Standard I/O Streams in Java

- All Java programs automatically import the **java.lang** package. This package defines a class called **System.**
- **System** also contains three predefined stream variables: **in**, **out**, and **err**.
- These fields are declared as **public**, **static**, and **final** within **System**.
- These streams are created automatically when your program starts.
- **System.out** refers to the **standard output stream**.
  - By default, it is connected to the console, allowing programs to display output to the user.
  - It's commonly used for printing messages, results, and other informational output.
  - System.out holds a **PrintStream** object.

The Predefined Streams: Standard I/O Streams in Java

- **System.in** refers to standard input, which is the keyboard by default.
- **System.err** refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.
- **System.in** is an object of type **InputStream**;
- **System.out** and **System.err** are objects of type **PrintStream**.
- These are byte streams, even though they are typically used to read and write characters from and to the console. But, they can be wrapped these within character-based streams, if desired.

System.out

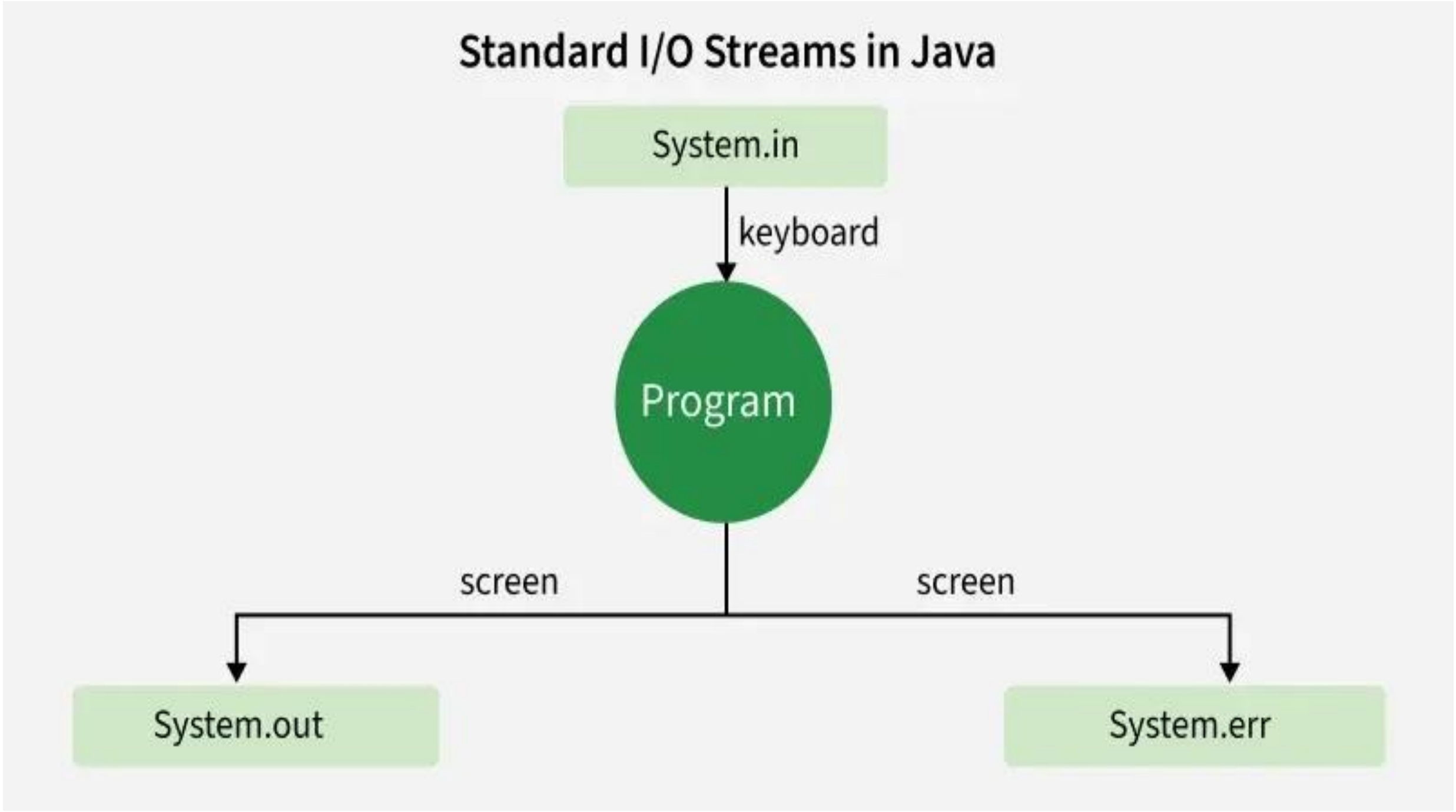
- **Type:** PrintStream
- **Purpose:** Standard output (normal program output)
- **Writes data to:** Console
- **Example how to use it:** System.out.println("Hello");
- **Advantages:**
  - PrintStream allows easy printing of String, numbers, characters, formatted output
  - It also **does NOT throw IOException**, making printing simpler.
- **Summary:** Used for **normal messages** to the console.

System.in

- **Type:** InputStream
- **Purpose:** Standard input (keyboard input)
- **Reads data from:** Keyboard (in console-based programs)
- **Usage:** int ch = System.in.read();
- Since System.in is a byte-based stream, we often wrap it:
  - Scanner sc = new Scanner(System.in);
  - BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
- **Summary:** Used to read user input.

System.err

- **Type:** PrintStream
- **Purpose:** Standard error output
- **Writes data to:** Console (often in red, depending on IDE/terminal)
- **Usage:**
  - System.err.println("Error occurred!");
- System.err is used to print:
  - error messages
  - warnings
  - diagnostic information


Standard I/O Streams in Java
System.in
keyboard
Program
screen
screen
System.out
System.err

Reading Console Input

- In Java, console input is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. **BufferedReader** supports a buffered input stream.
- BufferedReader(Reader *inputReader*)
- **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to
- **System.in**, use the following constructor:
- InputStreamReader(InputStream *inputStream*)
- Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*. Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
```

- After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

Code Example Using All Three Streams

```
import java.io.*;
public class PredefinedStreamsDemo {
    public static void main(String[] args) {
        // Wrap System.in into higher-level readers
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter your age: ");  // goes to System.out
        try {
            String input = br.readLine();      // reads from System.in
            int age = Integer.parseInt(input);
            if (age < 0) {
                System.err.println("Error: Age cannot be negative!"); // goes to System.err
            } else {
                System.out.println("Your age is: " + age); // goes to System.out
            }

        } catch (IOException e) {
            System.err.println("IO Error: " + e.getMessage()); // System.err
        } catch (NumberFormatException e) {
            System.err.println("Invalid number! Please enter digits only."); // System.err
        }
    }
```

PrintWriter Class

- PrintWriter is a **character stream** class used to **write** formatted **text data** to to various output destinations, such as files, console, or any character-output stream.
- It extends the Writer class and offers methods for printing different data types in a human-readable, formatted manner.
- It is part of **java.io** package.
- **Key Features of PrintWriter**
  - **Writes text data (characters), not bytes:** PrintWriter writes **Unicode text**.
  - **Supports print() and println():** Just like System.out (PrintStream):
    - pw.print("Hello");
    - pw.println("World");
  - **No need to handle IOException always:** Unlike FileWriter or BufferedWriter, many PrintWriter methods **do NOT throw IOException**.
- **Can be wrapped around other writers:** Example:
  - FileWriter
  - BufferedWriter
  - OutputStreamWriter
  - FileOutputStream

PrintWriter Class

- Unlike other writers, PrintWriter converts the primitive data(int, float, char, etc.) into the text format. It then writes that formatted data to the writer.
- Also, the PrintWriter class does not throw any input/output exception. Instead, we need to use the checkError() method to find any error in it.
- The PrintWriter class also has a feature of auto flushing. This means it forces the writer to write all data to the destination if one of the println() or printf() methods is called.

Constructors of PrintWriter

- PrintWriter pw = new PrintWriter(String fileName);
- PrintWriter pw = new PrintWriter(File file);
- PrintWriter pw = new PrintWriter(Writer out);
- PrintWriter pw = new PrintWriter(OutputStream out);
- PrintWriter pw = new PrintWriter(Writer out, boolean autoFlush);
- PrintWriter pw = new PrintWriter(OutputStream out, boolean autoFlush);

- autoFlush controls whether Java flushes the output stream every time a **println( )** method (among others) is called.
- If autoFlush is **true**, flushing automatically takes place. If **false**, flushing is not automatic.

```
import java.io.*;

public class PrintWriterExample {
    public static void main(String[] args) {
        try {
            PrintWriter pw = new PrintWriter("output.txt");

            pw.println("Hello World!");
            pw.println(100);
            pw.println(45.67);
            pw.println(true);
            pw.printf("Formatted Number: %.2f", 45.678);

            pw.close();
            System.out.println("Data written successfully.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

PrintWriter

Purpose:
Offers more advanced text formatting and printing capabilities, making it more convenient for writing human-readable text. It can write to various output streams, not just files.

Methods:
Provides a rich set of print(), println(), printf(), and format() methods for writing formatted representations of various data types (primitives, objects) and automatically handling line separators.

Buffering:
Often internally uses a BufferedWriter for efficiency, especially when constructed with a File or String filename.

PrintWriter

Encoding:
Allows specifying the character encoding in its constructors, providing better control over how characters are written to the file, ensuring cross-platform compatibility.

Automatic Flushing:
Can be configured for automatic flushing, where the buffer is written to the underlying stream after certain operations (like println(), printf(), format()).

Error Handling:
Does not throw IOException directly in its print() methods; instead, it sets an internal error flag that can be checked using checkError().

- **PrintWriter with FileWriter + Buffer**
  - PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("data.txt")));
- This gives:
  - Character writing
  - Buffering
  - Advanced print/println methods

- **PrintWriter with autoFlush**
  - PrintWriter pw = new PrintWriter(System.out, true);
  - pw.println("Hello");
  - true = auto flush on newline (println) or printf.

- **Writing to Console using PrintWriter**
  - PrintWriter pw = new PrintWriter(System.out, true);
  - pw.println("Printed using PrintWriter!");

When to Use PrintWriter?

Use PrintWriter when you need:

✓ Easy text printing
✓ printf(), print(), println()
✓ Unicode output
✓ Formatted text
✓ Auto-flush
✓ Writing to file or console

```java
import java.io.PrintWriter;

public class PrintWriterExample2 {
    public static void main(String[] args) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("Hello console using PrintWriter!");
        pw.println("Auto flush happens on println()");

        pw.printf("Value: %d\n", 10);
    }
}
```

```
C:\JavaPrg\JavaIO>java PrintWriterExample2
Hello console using PrintWriter!
Auto flush happens on println()
Value: 10
```

Byte Stream

8 bits carrier

InputStream

- **BufferedInputStream**
  Used for Buffered Input Stream
- **ByteArrayInputStream**
  Used for reading from a byte array
- **DataInputStream**
  Used for reading java standard data type
- **ObjectInputStream** - Input stream for objects
- **FileInputStream** - Used for reading from a File
- **PipedInputStream** - Input pipe
- **InputStream** - Describe stream input
- **FilterInputStream** - Implements **InputStream**

OutputStream

- **BufferedOutputStream**
  Used for Buffered Output Stream
- **ByteArrayOutputStream**
  Used for writing into a byte array
- **DataOutputStream**
  Used for writing java standard data type
- **ObjectOutputStream**- Output stream for objects
- **FileOutputStream**- Used for writing into a File
- **PipedOutputStream** - Output pipe
- **OutputStream**- Describe stream output
- **FilterOutputStream**- Implements **OutputStream**
- **PrintStream** - Contains **print( )** and **println( )**

read( ) and **write( )** both are key methods of byte stream

Character Stream

16 bits carrier - Unicode

Reader

- **BufferedReader**
  Used for Buffered Input Stream
- **CharArrayReader**
  Used for reading from an array
- **StringReader**
  Used for read from a string
- **FileReader** - Used for reading from a File
- **PipedReader** - Input pipe
- **InputStreamReader** - translates bytes to chatacter
- **FilterReader** - filtered reader
- **LineNumberReader** - used to count lines

Writer

- **BufferedWriter**
  Used for Buffered Output Stream
- **CharArrayWriter**
  Used for writing into an array
- **StringWriter**
  Used for write into a string
- **FileWriter** - Used for writing into a File
- **PipedWriter** - Output pipe
- **OutputStreamWriter** - characters to bytes
- **FilterWriter** - filtered writer
- **PrintWriter** - Contains **print( )** and **println( )**

Two Ways to Close a Stream

- In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation.
- **There are two basic ways in which you can close a stream.**
  - The first is to explicitly call close( ) on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, close( ) is typically called within a finally block. Thus, a simplified skeleton for the traditional approach is shown here:

```
try {
        // open and access file
} catch( I/O-exception) {
        // ...
} finally {
        // close the file
}
```

Two Ways to Close a Stream

- The second approach to closing a stream is to automate the process by using the **try**-with-resources statement The **try**-with- resources statement is an enhanced form of **try** that has the following form:

  - try (*resource-specification*) {

    // use the resource

    }

- Typically, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. Only objects of classes that implement **AutoCloseable interface** can be managed by **try**-with-resources. **Because AutoCloseable interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a try-with-resources statement.**

- When the **try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close( )** explicitly.

Advantage of try with resources

- The principal advantage of **try**-with-resources is that the resource (in this case, a stream) is closed automatically when the **try** block ends. Thus, it is not possible to forget to close the stream, for example.
- The **try**-with-resources approach also typically results in shorter, clearer, easier-to-maintain source code

Thanks!!!