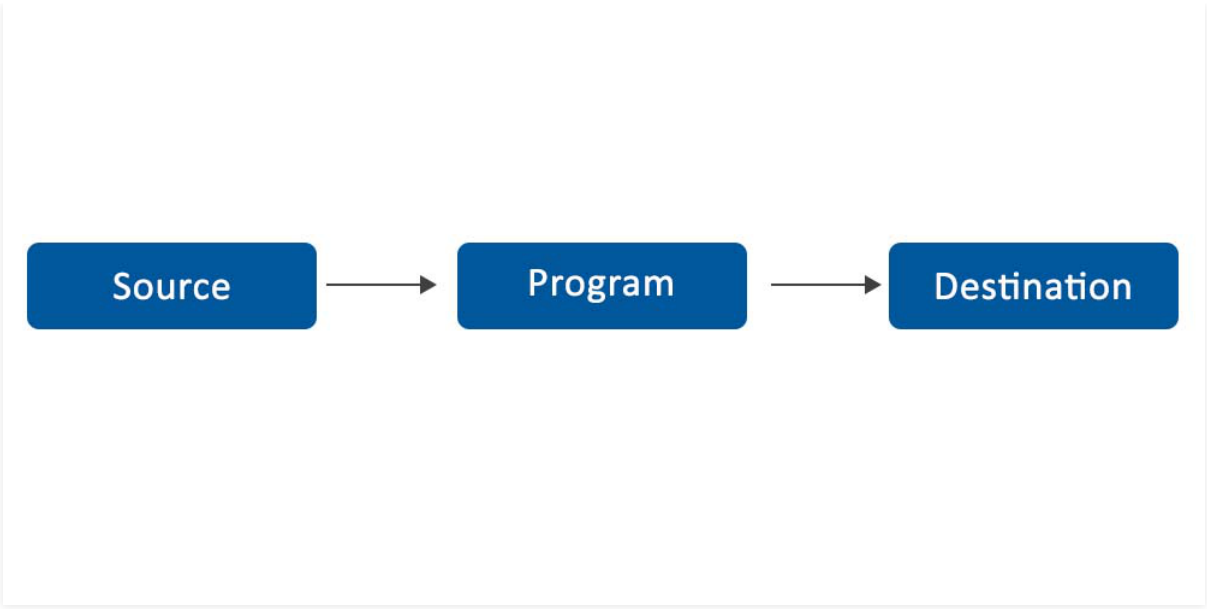
Java IO Tutorial

Last Updated : 17 Nov, 2021

Java programming language comes with a variety of APIs that helps the developers to code more efficiently. One of those APIs is **Java IO** API. Java IO API helps the users to read and write data. In simple words, we can say that the Java IO helps the users to take the input and produce output based on that input. Almost every application or code requires some input and output produced based on that input. So let's start with ***Java IO Tutorial***.

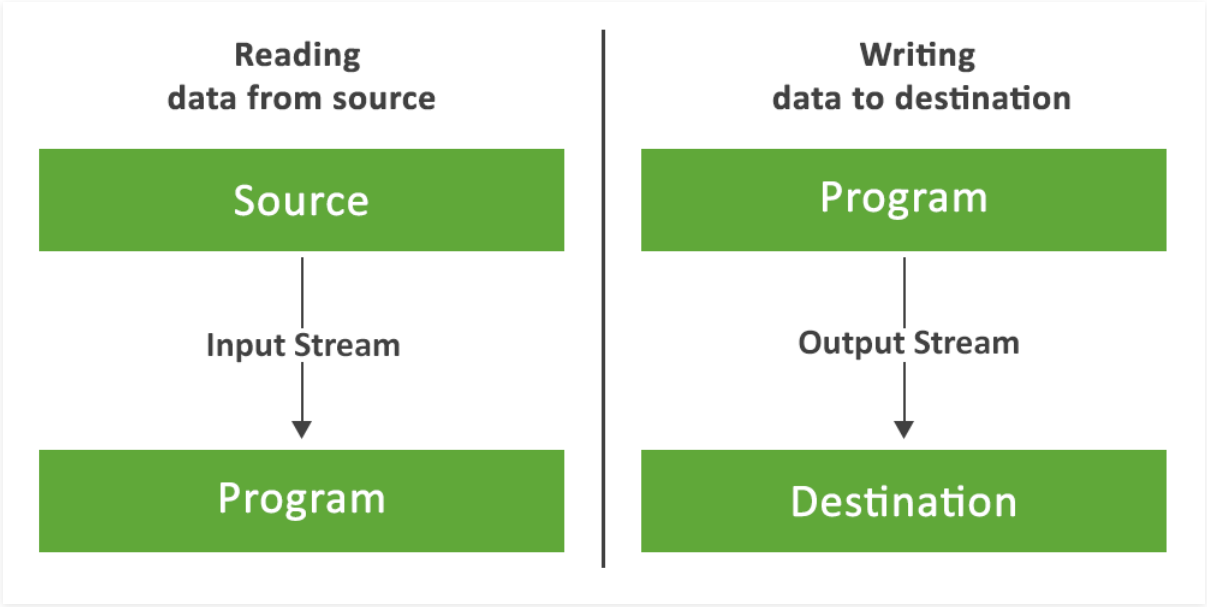
The Java IO API is placed in the **java.io package**. This package comprises almost all classes that a user requires to perform input and output (I/O) in Java. The Java IO package focuses on input and output to files, network streams, etc. However, the Java IO package does not include classes to open network sockets which are essential for network communication.

The **java.io** package generally involves reading basic information from a source and writing it to a destination. The below figure perfectly demonstrates the principle of a program taking input data from a source and producing output based on it to some destination.



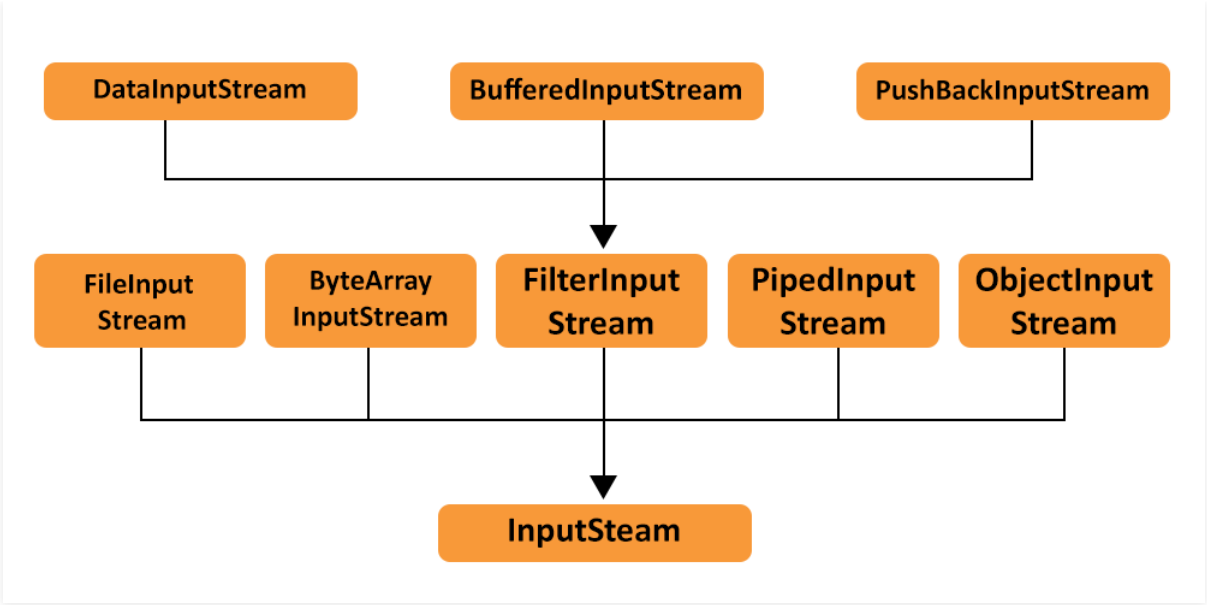
Java IO Streams

In the programming world, a stream can be described as a series of data. It is known as a stream because it is similar to a stream of water that continues to flow. Java IO streams are flows of data that a user can either read from or write to. Like an array, a stream has no concept of indexing the read or write data. A stream is attached to a data origin or a data target.



Java Streams uses two primary streams to read and write the data. These are:

1. InputStream – An input stream is used to read the data from a source in a Java application. Data can be anything, a file, an array, a peripheral device, or a socket. In Java, the class **java.io.InputStream** is the base class for all Java IO input streams.



Methods of Java IO InputStreams

1. **read() –** The **read()** method is used to read the next byte of data from the Input Stream. The value byte is passed on a scale of 0 to 255. If no byte is free because the end of the stream has arrived, the value -1 is passed.

2. **mark(int arg) –** The **mark(int arg)** method is used to mark the current position of the input stream. It sets read to limit, i.e., the maximum number of bytes that can be read before the mark position becomes invalid.

3. **reset() –** The **reset()** method is invoked by mark() method. It changes the position of the input stream back to the marked position.

4. **close() –** The **close()** method is used to close the input stream and releases system resources associated with this stream to Garbage Collector.

5. **read(byte [] arg) –** The **read(byte [] arg)** method is used to read the number of bytes of *arg.length* from the input stream to the buffer array arg. The bytes read by *read() method* are returned as an int. If the length is zero, then no bytes are read, and 0 is returned unless there is an effort to read at least one byte.

6. **skip(long arg) –** The **skip(long arg)** method is used to skip and discard arg bytes in the input stream.

7. **markSupported() –** The **markSupported()** method tests if the inputStream supports the mark and reset methods. The markSupported method of Java IO InputStream yields false by default.

Example of Java IO InputStream:

```
// Java program illustrates the use of
// the Java IO InputStream methods

import java.io.*;

public class InputStreamExample
{
    public static void main(String[] args) throws Exception
    {
        InputStream input = null;
        try {

            input = new FileInputStream("Text.txt");

            // read() method - reading and printing Characters
            // one by one
            System.out.println("Char - "+(char)input.read());
            System.out.println("Char - "+(char)input.read());

            // mark() - read limiting the 'input' input stream
            input.mark(0);

            // skip() - it results in skipping of 'e' in Ge'e'ksforGeeks
            input.skip(1);
            System.out.println("skip() method comes to play");
            System.out.println("mark() method comes to play");
            System.out.println("Char - "+(char)input.read());
            System.out.println("Char - "+(char)input.read());

            boolean check = input.markSupported();
            if (input.markSupported())
            {
                // reset() method - repositioning the stream to
                // marked positions.
                input.reset();
                System.out.println("reset() invoked");
                System.out.println("Char - "+(char)input.read());
                System.out.println("Char - "+(char)input.read());
            }
            else
                System.out.println("reset() method not supported.");


            System.out.println("input.markSupported() supported"+
                               " reset() - "+check);

        }
        catch(Exception e)
```

```
            {
                // in case of I/O error
                e.printStackTrace();
            }
            finally
            {
                if (input!=null)
                {
                    // Use of close() - closing the file
                    // and releasing resources
                    input.close();
                }
            }
        }
    }
```

Text.txt file used in the above code has the following content in it:

```
GeeksforGeeks
```

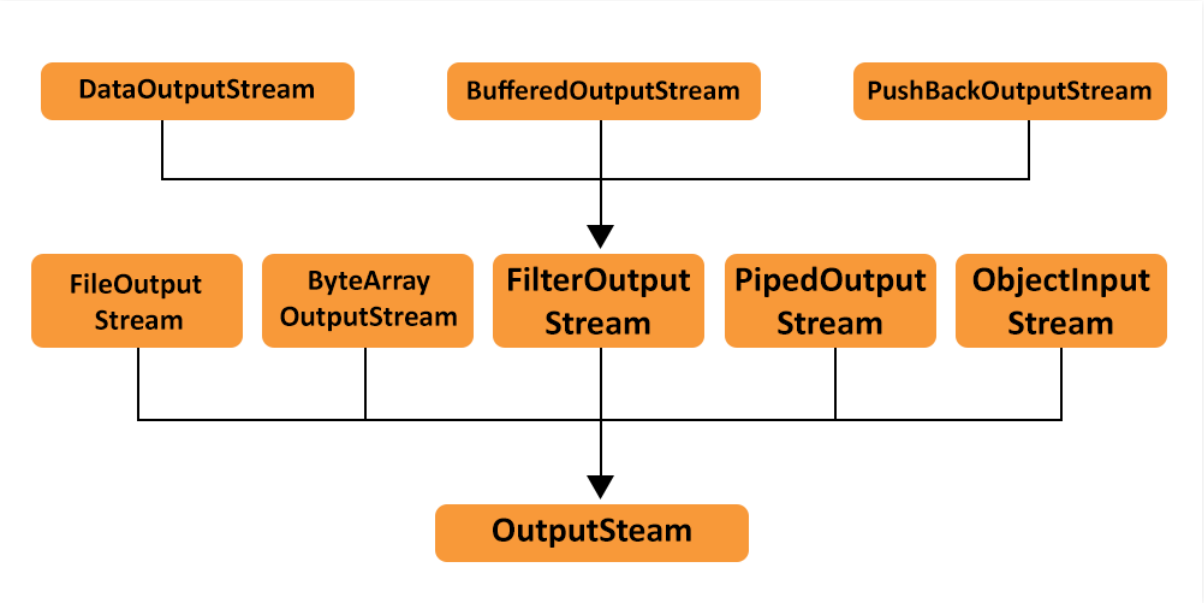
Output:

```
C:\Users\hp\Desktop>java InputStreamExample
Char - G
Char - e
skip() method comes to play
mark() method comes to play
Char - k
Char - s
reset() method not supported.
input.markSupported() supported reset() - false
```

> ***Note –*** *This code won't run on an online IDE as no such file is present there. You can run th is code on your System to check it's working.*

2. OutputStream – An output stream is used to write data (a file, an array, a peripheral device, or a socket) to a destination. In Java, the class **java.io.OutputStream** is the base class for all Java IO

output streams.



Methods of Java IO OutputStreams

1. **flush() –** The **flush()** method is used for flushing the outputStream This method forces the buffered output bytes to be written out.

2. **close() –** The **close()** method is used to close the outputStream and to release the system resources affiliated with the stream.

3. **write(int b) –** The **write(int b)** method is used to write the specified byte to the outputStream.

4. **write(byte [] b) –** The **write(byte [] b)** method is used to write bytes of length b.length from the specified byte array to the outputStream.

```
// Java program to demonstrate OutputStream
import java.io.*;

public class OutputStreamExample {
    public static void main(String args[]) throws Exception
    {
        OutputStream output
            = new FileOutputStream("file.txt");
        byte b[] = { 65, 66, 67, 68, 69, 70 };
```

```
            // illustrating write(byte[] b) method
            output.write(b);

            // illustrating flush() method
            output.flush();

            // illustrating write(int b) method
            for (int i = 71; i < 75; i++) {
                output.write(i);
            }

            output.flush();

            // close the stream
            output.close();
        }
    }
```

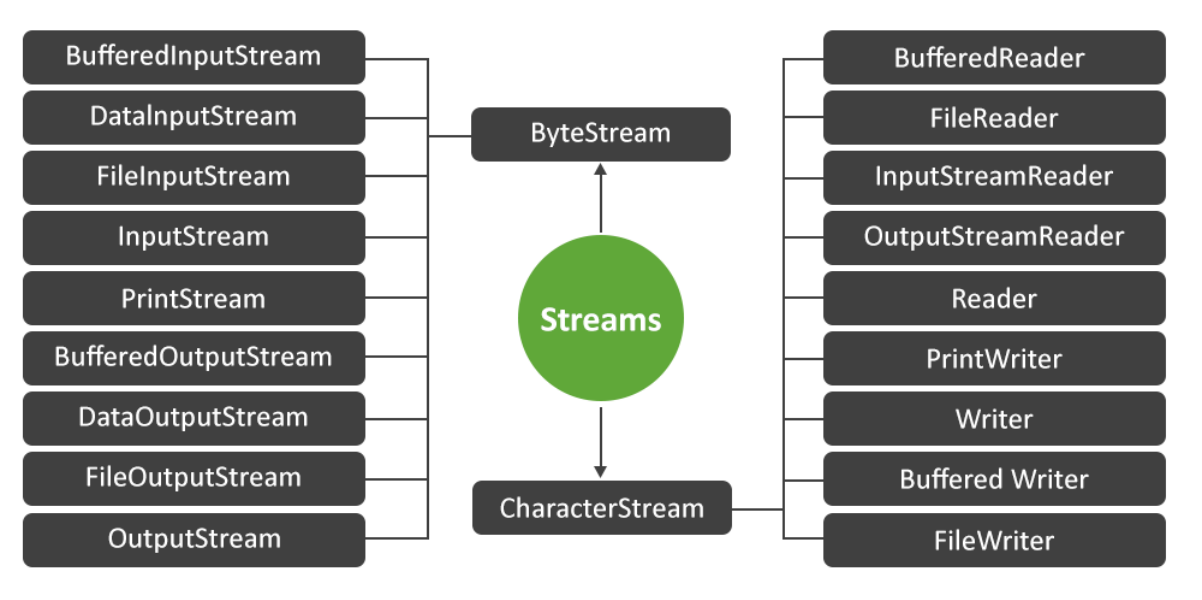
Output: When we run the program, the file.txt file is filled with the following content.

```
ABCDEFGHIJ
```

> ***Note –*** *This code won't run on an online IDE as no such file is present there. You can run th is code on your System to check it's working.*

Types of Streams in Java IO

The Streams in Java IO are of the following types:



1. Byte Streams – Java byte streams are the ones that are used to implement the input and output of 8-bit bytes. Several classes are affiliated with byte streams in Java. However, the most generally practiced classes are **FileInputStream** and **FileOutputStream**.

Different classes of Byte Streams

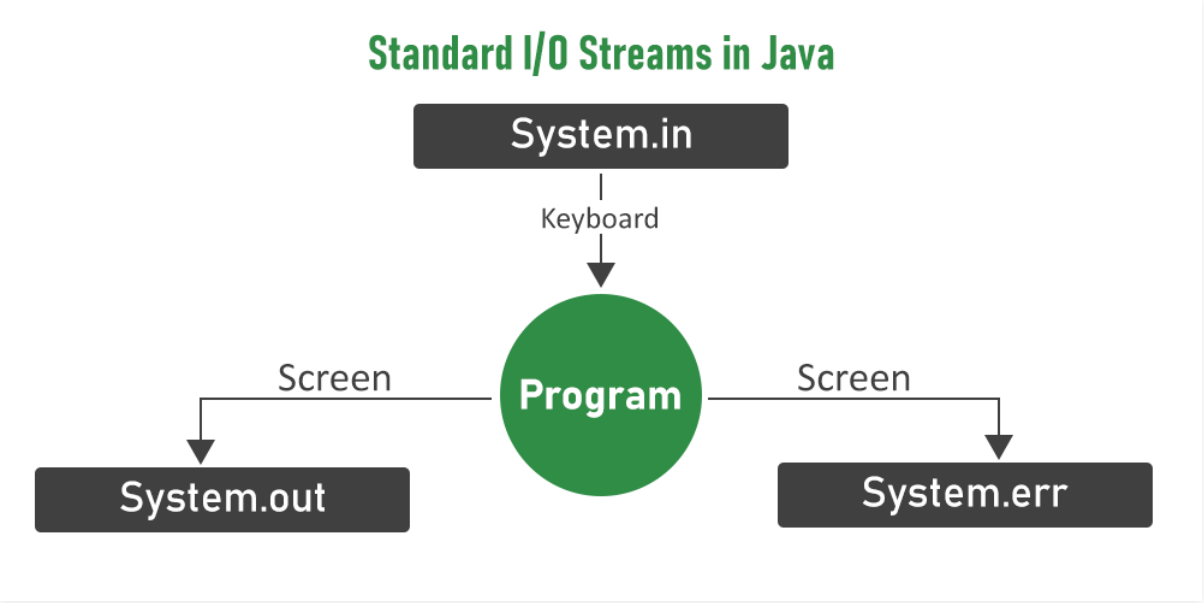
S. No.	Stream Class	Description
1	**InputStream**	This is an abstract class that defines stream input.
2	**FileInputStream**	This is used to reads from a file.
3	**DataInputStream**	It contains a method for reading java standard datatypes.
4	**BufferedInputStream**	It is used for Buffered Input Stream.
5	**PrintStream**	This class comprises the commonly used print() and println() methods.
6	**OutputStream**	This is an abstract class that describes stream output.
7	**FileOutputStream**	This class is used to write to a file.
8	**DataOutputStream**	This contains a method for writing java standard data types.
9	**BufferedOutputStream**	This is used for Buffered Output Stream.

2. Character Streams – Java Character streams are the ones that are used to implement the input and output for 16-bit Unicode. Several classes are associated with character streams in Java, but the most commonly used classes are **FileReader** and **FileWriter**. Internally, the FileReader class uses FileInputStream and the FileWriter class uses FileOutputStream. Nevertheless, the significant contrast is that FileReader and FileWriter read and write two bytes, respectively.

Different classes of Character Streams:

S. No.	Stream Classes	Description
1	**Reader**	This is an abstract class that defines character stream input.
2	**Writer**	This is an abstract class that defines character stream output.
3	**FileReader**	This is an input stream that reads from the file.
4	**FileWriter**	This is used to the output stream that writes to the file.
5	**BufferedReader**	It is used to handle buffered input streams.
6	**BufferedWriter**	This is used to handle buffered output streams.
7	**InputStreamReader**	This input stream is used to translate the byte to the character.
8	**OutputStreamReader**	This output stream is used to translate characters to bytes.
9	**PrintWriter**	This contains the most used print() and println() methods.

3. Standard Streams – All the programming languages assist standard I/O, where the user's program can take input from a keyboard and then produce an output on the computer screen. Just like C and C++ have three standard streams, STDIN, STDOUT, and STDERR, Java also provides the following three standard streams:



- **Standard Input –** The Standard Input class is used to accept input data to the user's program. Usually, a keyboard is utilized as a standard input stream and described as **System.in**.

- **Standard Output –** This class is used to output the data generated by the user's program, and usually, a computer screen is used for standard output stream and described as **System.out**.

- **Standard Error –** The Standard error class is used to output the data having an error that is generated by the user's program. Normally, a computer screen is utilized for standard error stream and described as **System.err**.

Example of Standard Streams in Java IO

```
// Java program to illustrate
// the Standard Java IO Streams

import java.io.*;
import java.util.*;

public class StandardStreamsExample {
    public static void main(String[] args)
    {
        try {

            // using System.in class to import the Scanner
            // that helps in taking input from the user
            Scanner s = new Scanner(System.in);
```

```
            // using System.out class print the output on
            // the screen using .print() ans .println()
            // methods
            System.out.print("Enter the value of A : ");
            int a = s.nextInt();
            System.out.println("Value of A is : " + a);

            System.out.print("Enter the value of B : ");
            int b = s.nextInt();
            System.out.println("Value of B is : " + b);

            System.out.println(
                "Result of division A/B is : " + (a / b));
        }
        catch (Exception ex) {

            // using System.err class to print the error
            // message
            System.err.print("Exception message : "
                             + ex.getMessage());
        }
    }
}
```

Output

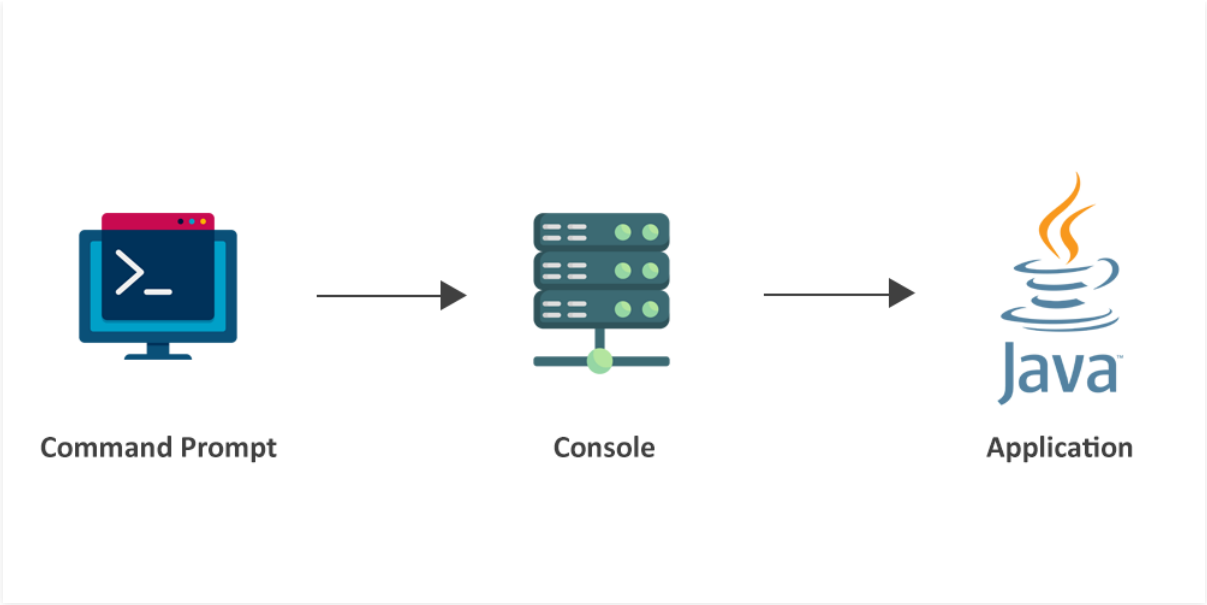
```
Enter the value of A : 8
Value of A is : 8
Enter the value of B : 4
Value of B is : 4
Result of division A/B is : 2
```

```
Enter the value of A : 4
Value of A is : 4
Enter the value of B : 0
Value of B is : 0
Exception message : / by zero
```

Java IO Console

The **Java.io.Console** class implements programs to receive the character-based console device, if any, affiliated with the prevailing Java virtual machine. The Console class was added to the package **java.io** in Java Development Kit 6 (JDK 6).

The Java Console class is be used to take input from the console. It presents methods to read texts and passwords. If a user reads the password using the Console class, it will not be disclosed to the user. The **java.io.Console** class is connected to the system console internally.



Characteristics of Java IO Console

- Java IO Console is used to read from and write to the console if one exists.
- The console is fundamentally a support class because most of its functionality is achievable through **System.in** and **System.out**. However, its effectiveness can clarify some console interactions, particularly when reading strings from the console.
- The console has no constructors. Instead, if a user wants to obtain a Console object, then it can be done by calling **System.console( ).**
- If a console is available, then a reference of that console is returned. Otherwise, null is returned. A console class is not feasible in all cases. Thus, in case of no console, null is returned.

Methods of Java IO Console

S.No.	Method's Name	Description
1.	writer()	This method is used to recover the individual PrintWriter object affiliated with the console.
2.	reader()	This method is used to retrieve the unique Reader object associated with the console.
3.	format(String format, Object… args)	This method expresses a formatted string to the output stream of the console utilizing the designated format string and arguments.
4.	printf(String format, Object… args)	This method is used to publish a formatted string to the console's output stream utilizing the stipulated format string and arguments.

S.No.	Method's Name	Description
5.	readLine(String format, Object… args)	This method is used to produce a formatted prompt. After that, this method reads a single line of text from the console.
6.	readLine()	This method is used to recite an individual line of data from the console.
7.	readPassword(String format, Object… args)	This method is used to provide a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
8.	readPassword()	This method is used to render a password or passphrase from the console.
9.	flush()	This method is used for flushing the console. This method limits any buffered output to be written rapidly.

Java IO Reader and Writer

Input and output streams are fundamentally byte-based. Java IO Reader and writer depends on the characters, which can have varying amplitudes depending on the character set. For instance, ASCII and ISO Latin-1 handle one-byte characters. Unicode uses two-byte characters. UTF-8 handles characters of different amplitudes (among one and three bytes). As characters are eventually comprised of bytes, readers practice their input from streams. However, they transform those bytes into characters according to a particularized encoding setup before catching them along. Furthermore, writers transform characters to bytes according to a particularized encoding before writing them onto any underlying stream.

The **java.io.Reader** and **java.io.Writer** classes are abstract superclasses. These superclasses help in reading and writing character-based data. These subclasses are important for managing the transformation among various character sets. In Java IO API, there are **nine reader** and **eight writer** classes, all in the **java.io** package.

Java IO Reader

A reader is used when a user wants to read character-based data from a source of the data. The Java IO Reader class is an abstract class that is used for expressing character streams. The only programs that a subclass needs to implement are read(char[], int, int) and close() methods. Many

subclasses of the Reader class will override some of the methods shown below. However, most subclasses produce tremendous efficiency, additional functionality, or both.

Methods of Java IO Reader

1. **close() –** This method is used to close the stream and release any associated system resources. Once the stream has been closed, further methods invocations like read(), mark(), or skip() will throw an IOException. In Java IO, closing an earlier-ended stream has no effect.

2. **mark(int readAheadLimit) –** This method is utilized to check the ongoing place in the stream. The following calls to reset() method will try to change the position of the stream to this point. In Java IO Reader, all the character-input streams do not support the mark() operation.

3. **markSupported() –** This process determines whether this stream sustains the mark() operation or not. The default implementation always returns false. All the subclasses of the Java IO Reader should call this method.

4. **read() –** This method is used to read a single character. The read() method is obstructed till a character is feasible, an I/O error happens, or the stream is ended. All the subclasses of the Java IO Reader class that intend to sustain adequate single-character input should override this method.

5. **read(char[] cbuf) –** This method is used to read characters into an array. This method is obstructed till some information is possible, an I/O error occurs, or the end of the stream is reached.

6. **read(char[] cbuf, int off, int len) –** This method is used to read the characters in a part of an array. This method is obstructed till a character is feasible, an I/O error happens, or the stream is ended.

7. **read(CharBuffer target) –** This method strives to read characters within the particularized character buffer. The buffer target is utilized like a container of characters.

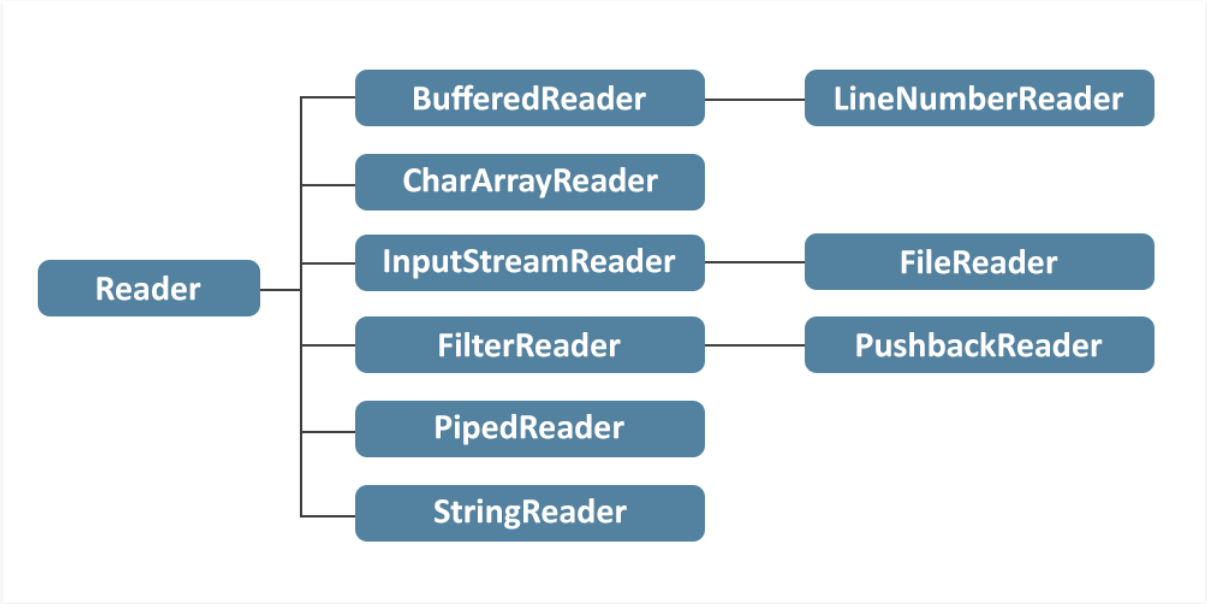
8. **ready() –** This method is used to tell whether this stream is ready to be read.

9. **reset() –** This method is used to reset the stream. If the stream has been marked, then this method tries to reposition the stream at the mark. If the stream has not been checked, then strive to reset it in some way relevant to the appropriate stream, for example, generally used by

repositioning it to its starting point. Not all character-input streams sustain the reset() process, and some help the reset() without sustaining mark().

10. **skip(long n) –** This method is used to skip characters. This method is blocked till some information is available, an I/O error occurs, or the end of the stream is reached.

Different classes of Java IO Reader



S. No.	Reader Classes	Description
1	**BufferedReader**	Java BufferedReader class is used to record the information from a character-based input stream. This method can be applied to read the data line by line with the help of the readLine() method.
2	**CharArrayReader**	The CharArrayReader class is used to read the character array as a reader (stream). It inherits Reader class.
3	**FileReader**	Java FileReader class is used to recite the information from the file. This method yields the data in a byte format, just like FileInputStream class.
4	**FilterReader**	Java FilterReader is used to perform filtering operations on the reader stream. FilterReader class is an abstract class for rendering filtered character streams.
5	**InputStreamReader**	An InputStreamReader class is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

S. No.	Reader Classes	Description
6	**LineNumberReader**	The LineNumberReaser class keeps track of line numbers of the read characters. Line numbering begins at 0.
7	**PushbackReader**	Java PushbackReader class is a character stream reader. It is used to pushes back a character into the stream and overrides the FilterReader class.
8	**PipedReader**	The PipedReader class is utilized to read the data of a pipe in the form of a stream of characters. The PipedReader class is commonly managed to read text.
9	**StringReader**	Java StringReader class is a stream of characters with string as an origin. It takes an input string and changes it into a character stream.

Example of Java IO Reader

```
// Java Program to illustrate the
// use of Java IO Reader methods
import java.io.*;

public class ReaderExample {
    public static void main(String[] args) {

        // Creates a character array of size 100.
        char[] array = new char[100];

        try {
            // Creating a reader object using the class FileReader
            Reader input = new FileReader("file.txt");

            // Checks if reader is ready
            System.out.println("Is there data in the stream?  " + input.ready(

            // Reads characters
            input.read(array);
            System.out.println("Data in the stream -");
            System.out.println(array);

            // Closes the reader
            input.close();
```

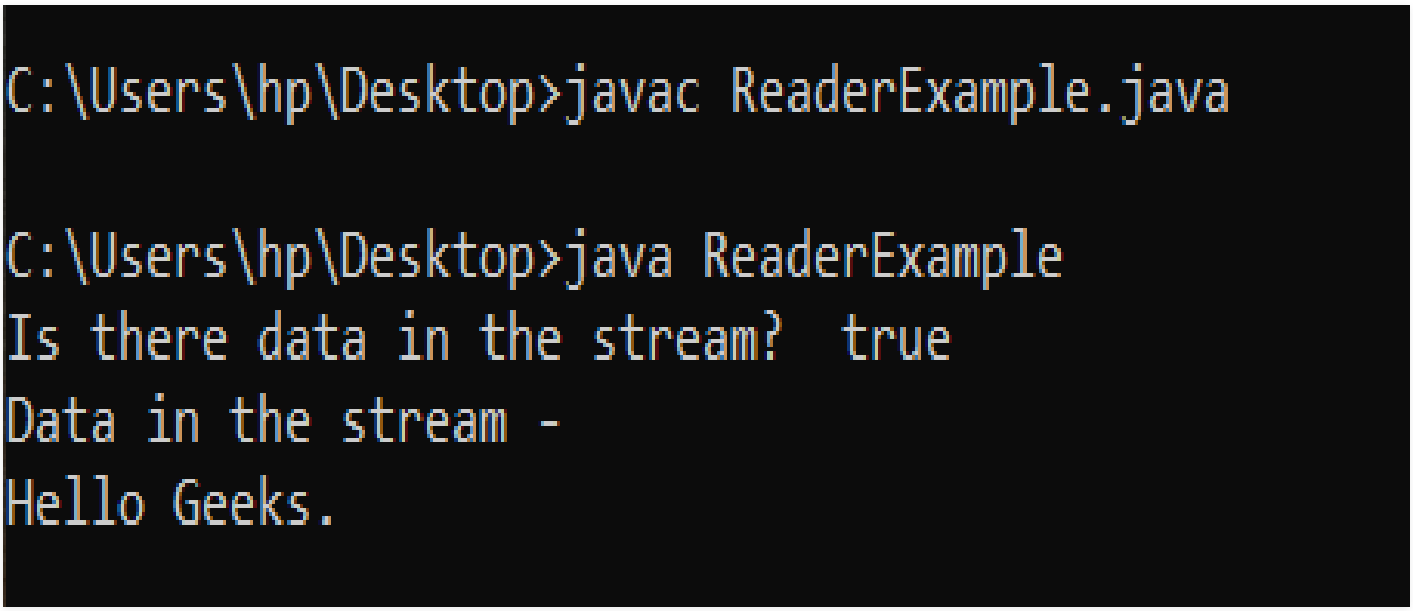
```
            }

            catch(Exception e) {
                e.getStackTrace();
            }
        }
    }
```

file.txt file used in the above code has the following content in it –

```
Hello Geeks.
```

Output



> ***Note –*** *This code won't run on an online IDE as no such file is present there. You can run th is code on your System to check it's working.*

Java IO Writer

Java IO Writer is an abstract class utilized for writing to the character streams. The only methods in Java IO Writer that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses, however, will override several methods explained here to implement higher efficiency, additional functionality, or both.

Methods of Java IO Writer:

1. **append(char c) –** This method is used to add the particularized character to the writer. An invocation of this process of the form out.append(c) performs the same way as the invocation – out.write(c).

2. **append(CharSequence csq) –** This method is used to add the designated character sequence to the writer. An invocation of this process of the form out.append(csq) functions in precisely the identical style as the invocation – out.write(csq.toString()).

3. **append(CharSequence csq, int start, int end) –** This method is used to add a subsequence of the particularized character sequence to the writer. It adds a subsequence of the detailed character sequence to the writer.

4. **close() –** This method is used to close the stream, flushing it first. Once the stream has been stopped, further write() or flush() methods will provoke an IOException to be thrown. In Java IO Writer, closing an earlier closed stream has no effect.

5. **flush() –** This method is used for flushing the stream. If the stream has preserved any characters from the multiple write() methods in a buffer, they should be written immediately to their proposed destination. After that, if that destination is a different character or byte stream, it should be flushed. Thus one invocation method flush() will flush all the barriers or buffers in a series of Writers and OutputStreams.

6. **write(char[] cbuf) –** This method is used to write an array of characters.

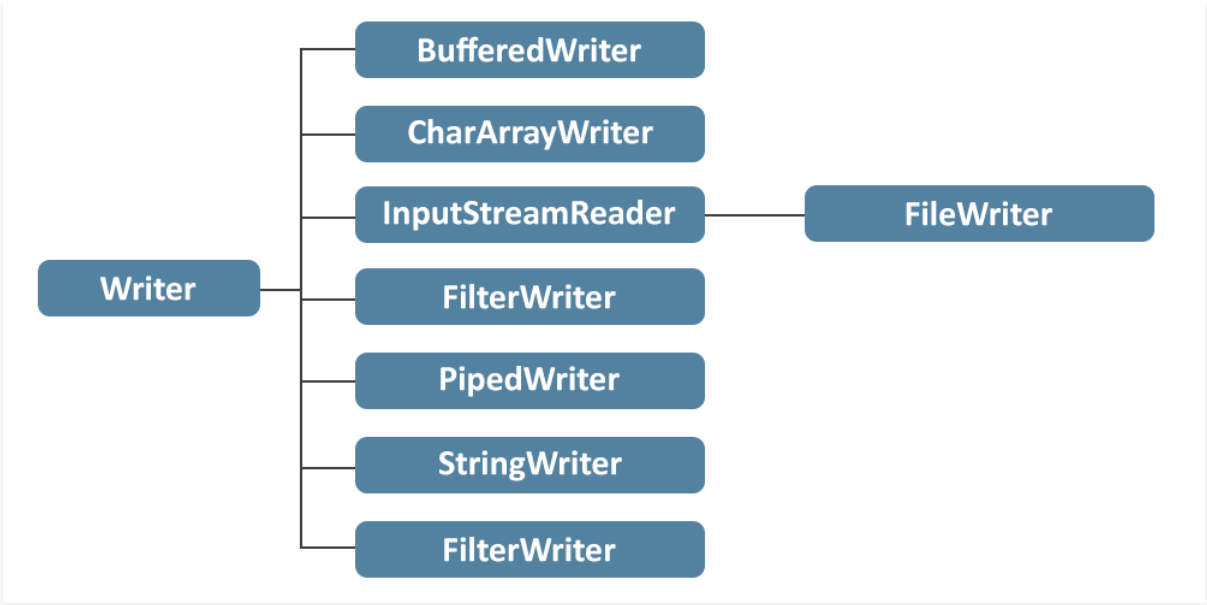
7. **write(char[] cbuf, int off, int len) –** This method is used to write a portion of an array of characters.

8. **write(int c) –** This method is used to print an individual character. The character to be rewritten is displayed in the 16 low-order bits of the provided integer value. It means that the 16 high-order bits are ignored. Subclasses that intend to support efficient single-character output should override this method.

9. **write(String str, int off, int len) –** This method is used to write a portion of a string.

Different classes of Java IO Writer

The list of different classes of Java IO Writer is given below –



S. No.	Writer Classes	Description
1	**BufferedWriter**	Java BufferedWriter class is used to implement buffering for the instances of the Writer class. It makes the performance fast. It inherits the Writer class.
2	**CharArrayWriter**	The CharArrayWriter class can be used to write common data to multiple files. This class inherits the Writer class.
3	**FileWriter**	Java FileWriter class is applied to write data to a file that is character-oriented. It is a character-oriented class that is utilized for the process of file handling in java.
4	**FilterWriter**	Java FilterWriter class is an abstract class. It is practiced to write only the filtered character streams. The subclass of the class FilterWriter should override a few of its processes, and it should contribute additional methods and fields
5	**OutputStreamWriter**	OutputStreamWriter is a class used to transform the character streams to the byte stream. In this class, the characters are encoded into bytes using a designated charset.
6	**PipedWriter**	The PipedWriter class is used to write a stream of characters in the form of a java pipe. This class is generally used for writing text.
7	**PrintWriter**	Java PrintWriter class can be described as an implementation of the Writer class. This class is used to imprint the formatted description of objects to the text-output stream.

S. No.	Writer Classes	Description
8	**StringWriter**	Java StringWriter class is used to collect the output from a string buffer, which can be utilized to build a string.

Example of Java IO Writer

```
// Java Program to illustrate the
// use of Java IO Writer methods
import java.io.*;

public class WriterExample {

    public static void main(String args[]) {

        String data = "Hello Geeks.";

        try {
            // Creates a Writer using FileWriter
            Writer output = new FileWriter("file.txt");


            // Writes string to the file
            output.write(data);

            // Closes the writer
            output.close();
        }

        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output

When we run the program, the **file.txt** file is filled with the following content.

```
Hello Geeks.
```

Note – *This code won't run on an online IDE as no such file is present there. You can run th is code on your System to check it's working.*

Java IO File

The **Java IO File** class is Java's description of a file or pathname of a directory. Since file and directory names have distinctive arrangements on various platforms, a naive string is not sufficient to describe them. The File class comprises various approaches for operating with the pathname, removing and renaming files, generating new directories, arranging the contents of a list, and managing numerous general properties of files and directories.

Characteristics of Java IO File

- Java IO File is an ideal illustration of files and pathnames of directories.
- A pathname, whether general or in string pattern, can be either fixed or contingent. The origin of an ideal pathname may be accomplished by invoking the getParent() process of the Java IO File class.
- First of all, a user should build an object of the File class by giving it the filename or directory name. A file method may execute limitations to specific actions on the real file-system object, such as reading and writing. These constraints are collectively identified as access permissions.
- Instances of the File class are immutable. That is, once created, the abstract pathname denoted by a File object will nevermore change.

Methods of Java IO File

1. **canExecute() –** This method is used to examine whether the application can accomplish the file expressed by the abstract pathname.

2. **canWrite() –** This method is used to test whether the application can transform the file expressed by the abstract pathname or not.

3. **canRead() –** This method is used to examine whether the application can read the given file signified by the abstract pathname.

4. **getName() –** This method is used to return the file or directory name denoted by the abstract pathname.

5. **getPath() –** This method is used to convert the abstract pathname into a pathname string.

6. **getAbsolutePath() –** This method is used to return the absolute pathname string of the abstract pathname.

7. **getParent() –** This method is used to return the pathname string of the abstract pathname's parent.

8. **exists() –** This method is used to test whether the file or directory expressed by the abstract pathname exists or not.

9. **length() –** This method is used to return the length of the file denoted by the abstract pathname.

10. **isDirectory() –** This method is used to test whether the file denoted by the pathname is a directory.

Example of Java IO File

```
// Java program to illustrate the use
// of Java IO File methods

import java.io.File;

public class FileExample
{
    public static void main(String[] args) {

        // taking the file or directory as a input
        // from the command line argument (args)
        String fname =args[0];

        // passing the file or directory name to File object
        File f = new File(fname);

        // applying the methods of the File
        // class on the File object
        System.out.println("File name - "+f.getName());
        System.out.println("Path - "+f.getPath());
        System.out.println("Absolute path - " +f.getAbsolutePath());
```

```
            System.out.println("Parent - "+f.getParent());
            System.out.println("Exists - "+f.exists());

        if(f.exists())
            {
                System.out.println("Is writable - "+f.canWrite());
                System.out.println("Is readable - "+f.canRead());
                System.out.println("Is a directory - "+f.isDirectory());
                System.out.println("File Size in bytes "+f.length());
            }
        }
    }
```

Output

```
C:\Users\hp>cd Desktop

C:\Users\hp\Desktop>javac FileExample.java

C:\Users\hp\Desktop>java FileExample Geeks.txt
File name - Geeks.txt
Path - Geeks.txt
Absolute path - C:\Users\hp\Desktop\Geeks.txt
Parent - null
Exists - true
Is writeable - true
Is readable - true
Is a directory - false
File Size in bytes 11
```

Geeks.txt file used in the above code has the following content in it –

```
Hello Geeks
```

> ***Note –*** *This code won't run on an online IDE as no such file is present there. You can run th is code on your System to check it's working.*