Java I/O Package:

- Java performs input output operations through streams stream is an abstraction that either produces or consumes information.
- Input stream: a stream can extract information from any different form of sources keyboard, disk file or a network socket.
- Output stream: this refers to a disk file console or a network connection.

Streams are categorized into two types

- Byte streams (I/O of bytes, reading and writing binary data)
  Character streams(input and output of characters)

In cases, if we want to work on files that holds characters we use character stream, where we do not what kind of data that file contains (data from database, image, audio and serialized objects (process of converting an object from its memory representation into a format that can be easily stored transmitted or reconstructed later -> saving and restoring the state of objects in a persistent storage like files and databases)) then we use Byte Streams.

Internally character streams also use byte streams, it is much simpler and easier to use character streams if we know the file has only characters. Also the character streams read char by character.

1. **Byte Streams**:
   - **Input Streams**:
     - `InputStream`: Abstract superclass for all byte input streams.
       - `FileInputStream`: Reads bytes from a file.
       - `ByteArrayInputStream`: Reads bytes from a byte array.
       - `ObjectInputStream`: Deserializes objects from an input stream.
       - `PipedInputStream`: Reads bytes from a piped output stream.
       - `FilterInputStream`: Abstract subclass for filtered byte input streams.
         - `BufferedInputStream`: Adds buffering to an input stream for improved performance.
         - `DataInputStream`: Reads primitive data types from an input stream.
         - `PushbackInputStream`: Allows bytes to be unread from the stream.
       - `SequenceInputStream`: Concatenates multiple input streams into one.
   - **Output Streams**:
     - `OutputStream`: Abstract superclass for all byte output streams.
       - `FileOutputStream`: Writes bytes to a file.
       - `ByteArrayOutputStream`: Writes bytes to a byte array.
       - `ObjectOutputStream`: Serializes objects to an output stream.
       - `PipedOutputStream`: Writes bytes to a piped input stream.
       - `FilterOutputStream`: Abstract subclass for filtered byte output streams.
         - `BufferedOutputStream`: Adds buffering to an output stream for improved performance.
         - `DataOutputStream`: Writes primitive data types to an output stream.
       - `PrintStream`: Prints formatted representations of various data types to an output stream.
       - `SequenceOutputStream`: Concatenates multiple output streams into one.

2. **Character Streams**:
   - **Readers**:
     - `Reader`: Abstract superclass for all character input streams.
       - `FileReader`: Reads characters from a file.
       - `CharArrayReader`: Reads characters from a character array.
       - `StringReader`: Reads characters from a string.
       - `FilterReader`: Abstract subclass for filtered character input streams.
         - `BufferedReader`: Adds buffering to a reader for improved performance.
         - `LineNumberReader`: Keeps track of line numbers while reading.
       - `InputStreamReader`: Adapts an input stream into a character stream.
         - `FileReader`: Reads characters from a file.
   - **Writers**:
     - `Writer`: Abstract superclass for all character output streams.
       - `FileWriter`: Writes characters to a file.
       - `CharArrayWriter`: Writes characters to a character array.
       - `StringWriter`: Writes characters to a string.
       - `FilterWriter`: Abstract subclass for filtered character output streams.
         - `BufferedWriter`: Adds buffering to a writer for improved performance.
       - `OutputStreamWriter`: Adapts an output stream into a character stream.
         - `FileWriter`: Writes characters to a file.

Byte Stream: the two important abstract classes are **InputStream** and **OutputStream** which implement **read( )** and **write( )** methods to read and write bytes of data. Each method has a form that is abstract and must be overridden by derived stream classes.

Byte streams are used to perform I/O of 8-bit bytes, which can represent binary data. The classes for byte streams are typically suffixed with InputStream or OutputStream. Examples include FileInputStream, FileOutputStream, ByteArrayInputStream, and ByteArrayOutputStream

Character stream: it has two abstract classes named **Reader** and **Writer**.

Character streams handle I/O of 16-bit Unicode characters, making them suitable for text-based data. They are typically suffixed with Reader or Writer. Examples include FileReader, FileWriter, BufferedReader, and BufferedWriter.

FileReader and FileWriter: They are straightforward to use when you need to read from or write to a file character by character or string by string. However, for improved performance, it's often recommended to wrap them with BufferedReader and BufferedWriter (They are typically used for reading and writing text files line by line or in chunks. They are especially useful when dealing with large files or when efficiency is important.).

Note: all Java programs automatically import java.lang. package. This package defines a class called system. System contains 3 predefined stream variables in, out, and err these are declared as public static and final within the system i.e., they can be used in any other part of Java program without reference to a specific system object.

System.out : this is a standard output stream

System.in: this is a standard input stream(keyboard by default)

System.err: it refers to the standard error stream maybe redirect it to any compatible IO device.

- **System.out and System.err** are objects of **PrintStream**
- **System.in** is an object of **InputStream**

Reading console input: we can read console input by reading from System.in

-One way to obtain a character-based stream that is attached to the console is to wrap **System.in** in a **BufferedReader**. The BufferedReader class supports a buffered input stream

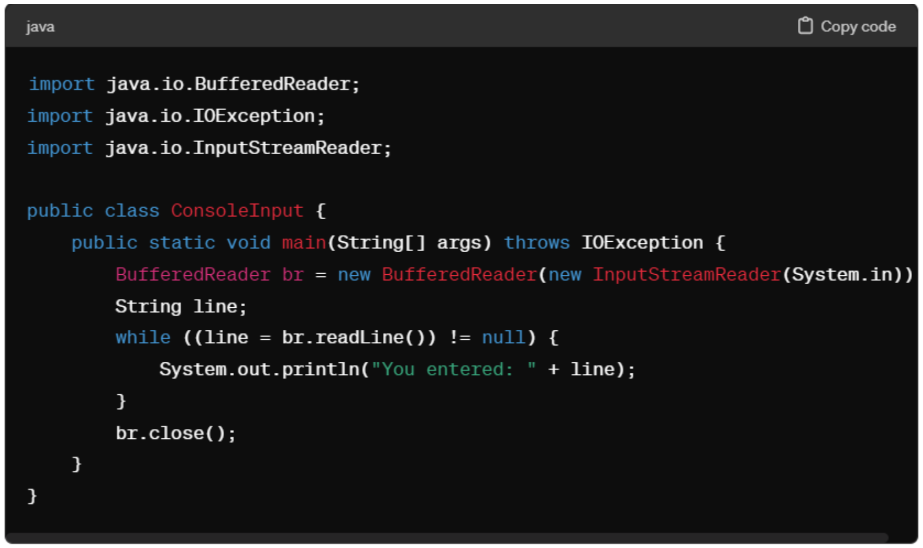
BufferedReader(Reader *inputReader*)

- Reading characters: To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is
- int read( ) throws IOException
- Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1, when an attempt is made to read at the end of the stream. As you can see, it can throw an IOException.

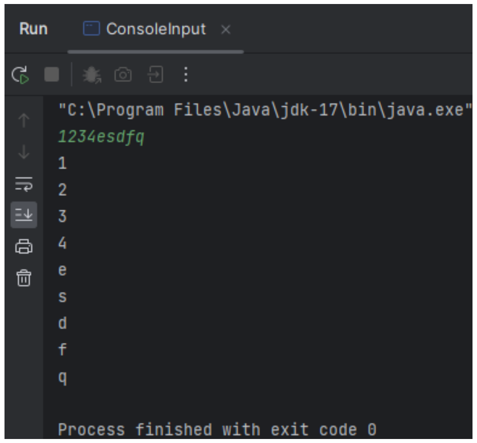
In the below code, InputStreamReader acts as a bridge between byte-oriented streams (such as InputStream) and character-oriented streams (such as Reader). It converts bytes read from the input stream into characters that can be processed by character-oriented classes like BufferedReader.

BufferedReader adds buffering functionality to the underlying reader. It reads data from the input stream in chunks and stores it in an internal buffer. This reduces the number of I/O operations and improves performance, especially when reading large amounts of data.

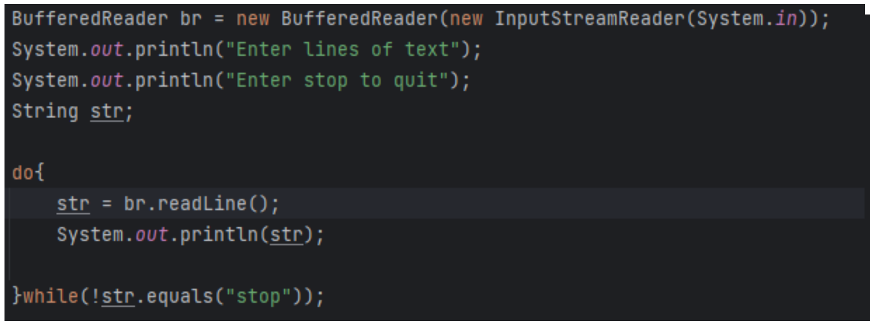
System.in is line buffered, by default. This means that no input is actually passed until you press enter, so here instead of read() we can use readLine() method, which is also a member of buffered reader class, which reads and displays lines of text until you specify certain conditions for the program to stop in the console



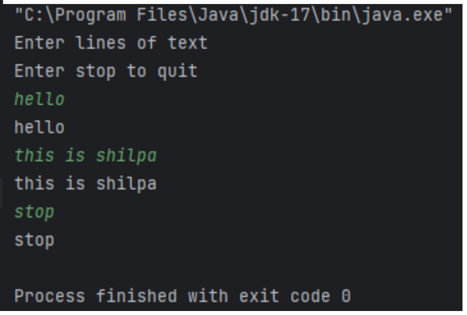
```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ConsoleInput {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in))
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println("You entered: " + line);
        }
        br.close();
    }
}
```



```
"C:\Program Files\Java\jdk-17\bin\java.exe"
1234esdfq
1
2
3
4
e
s
d
f
q

Process finished with exit code 0
```

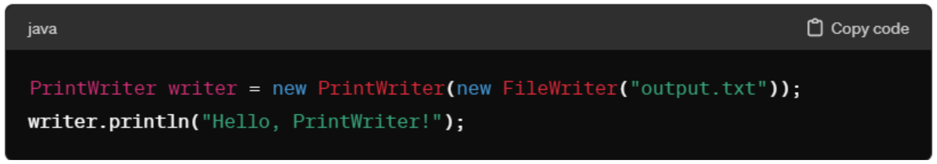


```java
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter lines of text");
System.out.println("Enter stop to quit");
String str;

do{
    str = br.readLine();
    System.out.println(str);

}while(!str.equals("stop"));
```

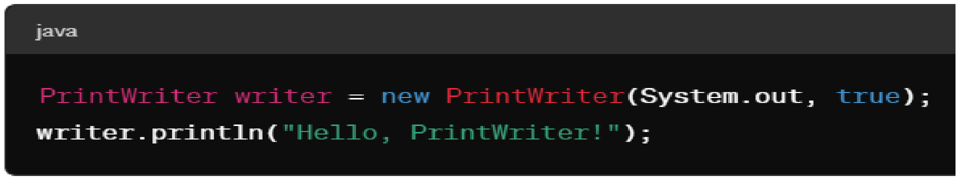


```
"C:\Program Files\Java\jdk-17\bin\java.exe"
Enter lines of text
Enter stop to quit
hello
hello
this is shilpa
this is shilpa
stop
stop

Process finished with exit code 0
```

PrintWriter: in cases where we wanted to write formatted text to a character based output streams we use print writer example for files sockets or any other character based streams. In the below example we are writing hello print writer to a new file called output.txt. In another example via printing the output to the console, the second parameter true enables auto flush. If flushingOn is true, flushing automatically takes place. If false, flushing is not automatic.



```java
PrintWriter writer = new PrintWriter(new FileWriter("output.txt"));
writer.println("Hello, PrintWriter!");
```



```java
PrintWriter writer = new PrintWriter(System.out, true);
writer.println("Hello, PrintWriter!");
```