Chaptter 12. Input Output

1. Types of Input Streams Classes :

Input stream's job is to represent classes that produce input from different sources. These sources could be

1) An array of bytes.
2) A **String** object.
3) A file.
4) A "pipe," which works like a physical pipe: You put things in at one end and they come out the other.
5) A sequence of other streams, so you can collect them together into a single stream.
6) Other sources, such as an Internet connection.

Hence we can see because of different sources of input we have corresponding classes.

Class	Function	Constructor arguments How to use it
ByteArray-InputStream	Allows a buffer in memory to be used as an **InputStream.**	The buffer from which to extract the bytes. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
StringBuffer-InputStream	Converts a **String** into an **InputStream.**	A **String**. The underlying implementation actually uses a **StringBuffer**. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
File-InputStream	For reading information from a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
Piped-InputStream	Produces the data that's being written to the associated **PipedOutput-Stream**. Implements the "piping" concept.	**PipedOutputStream** As a source of data in multithreading: Connect it to a **FilterInputStream** object to provide a useful interface.
Sequence-InputStream	Converts two or more **InputStream** objects into a single **InputStream**.	Two **InputStream** objects or an **Enumeration** for a container of **InputStream** objects. As a source of data: Connect it to a **FilterInputStream** object to provide a useful interface.
Filter-InputStream	Abstract class that is an interface for decorators that provide useful functionality to the other **InputStream** classes. See Table I/O-3.	See Table I/O-3. See Table I/O-3.

1.1 Above Class **StringBufferInputStream** has been deprecated.

1.2 ByteArrayInputStream and ByteArrayOutputStream classes :

➔ This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using toByteArray()and toString().

Constructor and Description
ByteArrayInputStream(byte[] buf) Creates a ByteArrayInputStream so that it uses buf as its buffer array.
ByteArrayInputStream(byte[] buf, int offset, int length) Creates ByteArrayInputStream that uses buf as its buffer array.

Example 1 : Reading from console.

```
package inputOutputStream;

import java.io.*;

public class ByteArrayStreamCls {

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

      ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);

      while( bOutput.size()!= 10 ) {
         // Gets the inputs from the user
         bOutput.write(System.in.read());
      }

      byte b [] = bOutput.toByteArray();
      System.out.println("Print the content");
      for(int x= 0 ; x < b.length; x++) {
         // printing the characters
         System.out.print((char)b[x]  + "   ");
      }
      System.out.println("   ");

      int c;

      ByteArrayInputStream bInput = new ByteArrayInputStream(b);

      System.out.println("Converting characters to Upper case " );
      for(int y = 0 ; y < 1; y++ ) {
         while(( c= bInput.read())!= -1) {
            System.out.println(Character.toUpperCase((char)c));
         }
         bInput.reset();
      }
   }
}
Output:
Hello Welcome
Print the content
```

```
H   e   l   l   o       W   e   l   c
Converting characters to Upper case
H
E
L
L
O

W
E
L
C
```

➔ ByteArrayInputStream only takes an 'Byte Array' as argument. Hence every input needs to be converted to a byte array before passing to this stream.

➔ In above example bOutput.write() method will writes the input data into byte array which could be retrived using toByteArray() and toString() methods. In above example how much bytes of data should go into buffer will be defined by bOutput.size().

➔ There is a write method in class ByteArrayOutputStream with arguments **write(byte[] b, int off, int len)**. Here we can directly write the byte array at offset **off** and length **len**. This method needs ready byte[] as argument and is useful when one wants to 'write' to a file or append a string.

➔ System.in.read() where 'in' is a InputStream which has read() method to read from console. This input will be stored in byte array by ByteArrayOutputStream.

➔ When we send argument as 'System.in' it means stream which has such argument will read from keyboard(console) directly and if argument is 'System.in.read()' it means InputStream defined in System class will read the input data.

➔ Read() method has <u>return type</u> as 'int' and write method has <u>argument type</u> as 'int'. Hence to read from console we have to use ByteArrayOutputStream class.

➔ While reading from String, one can convert it to a byte array and send it to ByteArrayInputStream(or CharArrayWriter) without sending it to ByteArrayOutputStream(or CharWriter). But while reading from console(or keyboard) one dont have any other option of writing first into the byte(or char) array and then read it.

Example 2 : Reading from a String

```
package inputOutputStream;

import java.io.*;

public class ByteArrayStringCls {

	public static void main(String[] args) throws IOException
	{
		String str = "Hello World";
		int c1;

		byte [] b = str.getBytes();

		ByteArrayInputStream bi = new ByteArrayInputStream(b);

			while ((c1 = bi.read()) != -1)
			{
				System.out.println((char)c1);
			}
		bi.close();
	}
}
Output:
H
e
l
l
o

W
o
r
l
d
```

1.3 FileInputStream and FileOutputStream classes :

➔ A FileInputStream obtains input bytes from a file in a file system. What files are available depends on the host environment. FileInputStream is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

```
package inputOutputStream;

import java.io.*;

public class FileIOStreamClass
{
	public static void main(String[] args) throws IOException
	{
		FileInputStream fin = null;
		FileOutputStream fout = null;

		try
		{

		fin = new FileInputStream("F:/Java material/java pgm
		files/javapgmfile1.txt");
```

```
            fout = new FileOutputStream("F:/Java material/java pgm
            files/javapgmfile2.txt");

            int c;

            while((c = fin.read())!= -1)
            {
                  fout.write(c);
            }
            }finally{
                  if (fin != null) {
                fin.close();
            }
            if (fout != null) {
                fout.close();
            }
      }
   }
}
```

1.4 SequenceInputStream Class

➔ There is only SequenceInputStream is available with read() methods but no SequenceOutputStream present. Hence no write() method either.

➔ Constructors :

Constructor and Description
SequenceInputStream(Enumeration<? extends InputStream> e) Initializes a newly created SequenceInputStream by remembering the argument, which must be an Enumeration that produces objects whose run-time type is InputStream.
SequenceInputStream(InputStream s1, InputStream s2) Initializes a newly created SequenceInputStream by remembering the two arguments, which will be read in order, first s1 and then s2, to provide the bytes to be read from this SequenceInputStream.

➔ A SequenceInputStream represents the logical concatenation of other input streams. It starts out with an ordered collection of input streams and reads from the first one until end of file is reached.

Example :

```
package testProg;

import java.io.*;

public class SequenceStreamCls
{

      public static void main(String[] args) throws IOException
      {
            int c;
            FileInputStream fin1 = new FileInputStream("F:/Java material/java pgm
            files/SequenceStream1.txt");
            FileInputStream fin2 = new FileInputStream("F:/Java material/java pgm
            files/SequenceStream2.txt");

            FileOutputStream fout = new FileOutputStream("F:/Java material/java pgm
```

```
                files/SequenceStream3.txt");

                SequenceInputStream sqi = new SequenceInputStream(fin1,fin2);

                while((c = sqi.read()) != -1)
                {
                      fout.write(c);
                }

                fout.close();
                sqi.close();
                fin1.close();
                fin2.close();
        }
}

Output:
SequenceStream1.txt = hi there!!
SequenceStream2.txt = how r u?
SequenceStream3.txt = hi there!!how r u?
```

2. Types of Output Stream Classes :

This category includes the classes that decide where your output will go: an array of bytes (but not a String—presumably, you can create one using the array of bytes), a file, or a "pipe."

Class	Function	Constructor arguments How to use it
ByteArray-OutputStream	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer.	Optional initial size of the buffer.
		To designate the destination of your data: Connect it to a **FilterOutputStream** object to provide a useful interface.
File-OutputStream	For sending information to a file.	A **String** representing the file name, or a **File** or **FileDescriptor** object.

		To designate the destination of your data: Connect it to a **FilterOutputStream** object to provide a useful interface.
Piped-OutputStream	Any information you write to this automatically ends up as input for the associated **PipedInput-Stream.** Implements the "piping" concept.	**PipedInputStream**
		To designate the destination of your data for multithreading: Connect it to a **FilterOutputStream** object to provide a useful interface.
Filter-OutputStream	Abstract class that is an interface for decorators that provide useful functionality to the other **OutputStream** classes. See Table 1/O-4-	See Table I/O-4.
		See Table I/O-4.

3. 3. Types of FilterInputStream Classes :

Data-InputStream	Used in concert with **DataOutputStream,** so you can read primitives **(int, char, long,** etc.) from a stream in a portable fashion.	**InputStream**
		Contains a full interface to allow you to read primitive types.
Buffered-InputStream	Use this to prevent a physical read every time you want more data. You're saying, "Use a buffer."	**InputStream,** with optional buffer size.
		This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.
LineNumber-InputStream	Keeps track of line numbers in the input stream; you can call **getLineNumber( )** and **setLineNumber (int)**.	**InputStream**
		This just adds line numbering, so you'll probably attach an interface object.
Pushback-InputStream	Has a one-byte pushback buffer so that you can push back the last character read.	**InputStream**
		Generally used in the

4. <u>Type of FilterOutputStream Classes</u> :

Data-OutputStream	Used in concert with **DataInputStream** so you can write primitives (**int**, **char**, **long**, etc.) to a stream in a portable fashion.	**OutputStream** Contains a full interface to allow you to write primitive types.
PrintStream	For producing formatted output. While **DataOutputStream** handles the *storage* of data, **PrintStream** handles *display*.	**OutputStream,** with optional **boolean** indicating that the buffer is flushed with every newline. Should be the "final" wrapping for your **OutputStream** object. You'll probably use this a lot.
Buffered-OutputStream	Use this to prevent a physical write every time you send a piece of data. You're saying, "Use a buffer." You can call **flush( )** to flush the buffer.	**OutputStream,** with optional buffer size. This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.

➔ The PrintStream Class prints all primitive data types and String objects in view-able format. The two important methods in **PrintStream** are **print( )** and **println( )** which are overloaded to print all the various types. The difference between **print( )** and **println( )** is that the latter adds a newline when it's done.

➔ You'll need to buffer your input almost every time, regardless of the I/O device you're connecting to, so it would have made more sense for the I/O library to have a special case (or simply a method call) for unbuffered input rather than buffered input. (Same goes for Buffered Output).

1.5 <u>FilterInputStream and FilterOutputStream</u>

1.5.1 <u>DataInputStream and DataOutputStream classes</u>

➔ A data input stream lets an application read primitive Java data types(`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as String values, from an underlying input stream in a machine-independent way. <u>An application uses a data output stream to write data that could later be read by a data input stream.</u>

➔ Syntax : public class DataInputStream extends FilterInputStream implements DataInput

➔ Example 1

```
package inputOutputStream;

import java.io.*;

public class DataStreamString {

	public static void main(String[] args) throws IOException
	{
		String dataFile = "F:/Java material/java pgm
files/DataOutputStream.txt";
		DataOutputStream dout = new DataOutputStream(new
FileOutputStream(dataFile));
		dout.writeUTF("Java");

		DataInputStream din = new DataInputStream(new
FileInputStream(dataFile));
		String k = din.readUTF();
		System.out.println(k);

		dout.close();
		din.close();
	}
}
Output:
Java
```

➔ Example 2:

```
package inputOutputStream;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.EOFException;

public class DataStreams {
    static final String dataFile = "F:/Java material/java pgm
files/invoicedata.txt";

    static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
            "Java Mug",
            "Duke Juggling Dolls",
            "Java Pin",
            "Java Key Chain" };

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

        DataOutputStream out = null;

        try {
            out = new DataOutputStream(new
                    BufferedOutputStream(new FileOutputStream(dataFile)));
```

```
            for (int i = 0; i < prices.length; i ++) {
                out.writeDouble(prices[i]);
                out.writeInt(units[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
        }

        DataInputStream in = null;
        double total = 0.0;
        try {
            in = new DataInputStream(new
                    BufferedInputStream(new FileInputStream(dataFile)));

            double price;
            int unit;
            String desc;

            try {
                while (true) {
                    price = in.readDouble();
                    unit = in.readInt();
                    desc = in.readUTF();
                    System.out.format("You ordered %d units of %s at $%.2f%n",
                            unit, desc, price);
                    total += unit * price;
                }
            } catch (EOFException e) { }
            System.out.format("For a TOTAL of: $%.2f%n", total);
        }
        finally {
            in.close();
        }
    }
}
Output:
You ordered 12 units of Java T-shirt at $19.99
You ordered 8 units of Java Mug at $9.99
You ordered 13 units of Duke Juggling Dolls at $15.99
You ordered 29 units of Java Pin at $3.99
You ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.88
```

1.5.2 BufferedInputStream and BufferedOutputStream classes

➔ During unbuffered stream call read or write request will be handled directly by underlying OS. This can make program less efficient since each such request triggers disk access, network activity or some other operation that is relatively expensive. To reduce overhead, we have buffered I/O streams. Buffered input streams read data from a memory area known as buffer, the native API will be called only when the buffer is empty. Similarly, buffered output streams write data to a buffer and native output API is called only when buffer is full. There are four buffered stream classes used to wrap unbuffered streams: **BufferedInputStream** and **BufferedOutputStream** create buffered byte streams, while **BufferedReader** and **BufferedWriter** create buffered character streams. Above class DataStream.java shows how one should use BufferedInputStream and BufferedOutputStream.

1.5.3. LineNumberInputStream

This class has been deprecated. LineNumberReader Class is present. Please check the notes and examples.

The Predefined Streams :

- As we know, all java programs automatically import java.lang package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. It contains three predefined stream variables **in**, **out** and **err**. These fields are declared as public, static and final.

- **System.out** refers to the standard output stream. By default, this is the console. **System.in** refers to the standard input stream, which is the keyboard by default. **System.err** refers to the System error stream, which is also console by default.

- System.in is of object type **InputStream**; System.out and System.err are of object type **PrintStream**. These are byte streams, even though typically they are used to read and write characters from and to the console. Also they can be wrapped in character based stream, if desired. Three variables in, out, err defined in class 'System' as below

```
public final static InputStream in = null

public final static PrintStream out = null;

public final static PrintStream err = null;
```

- How System.out.println() works?

    - As we can see above that all stream variables are set to null inside System.java class. Then System.out.println() should throw NullPointerException. But in reality null value will be set to a new value. But how come a 'final' value can be changed ? It can be done through 'native' methods i.e. `setIn0(InputStream in), setOut0(PrintStream out), setErr0(PrintStream err).`

    - The above native methods are defined as

```
private static native void setIn0(InputStream in);
private static native void setOut0(PrintStream out);
private static native void setErr0(PrintStream err);
```

      Note that native methods are defined same as methods of interfaces but with 'native' keyword.

    - These methods are called from another method called initializeSystemClass() which is called by JVM after 'static' and 'thread' initialization. It is defined in System.java class as

```
private static void initializeSystemClass()

{

//

}
```

1.5.5 PrintStream

➔ A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created so as to flush automatically; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte ('\n') is written.

➔ All characters printed by a PrintStream are converted into bytes using the platform's default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes.

➔ Example

```
package inputOutputStream;

import java.io.*;

public class PrintStreamCls
{
	public static void main(String[] args) throws IOException
	{
		String str = "New String";
		String fileStream = "F:/Java material/java pgm files/PrintStream.txt";
		String fileWriter = "F:/Java material/java pgm files/PrintWriter.txt";

		//Printing Without flush method
		System.out.println("Prints nothing without flush method");
		PrintStream ps1 = new PrintStream(System.out);
		ps1.println(str);
		System.out.println("------");

		//Printing With flush method
		System.out.println("Printing with flush method");
		PrintStream ps2 = new PrintStream(System.out);
		ps2.println(str);
		ps2.flush();
		System.out.println("------");

		//Auto flush
		System.out.println("Auto flush");
		PrintStream ps3 = new PrintStream(System.out, true);
		ps3.println(str);
		System.out.println("------");
```

```
            //Append
            System.out.println("Appending string");
            PrintStream ps4 = new PrintStream(System.out);
            ps4.println(str);
            ps4.append('A');
            ps4.append("Jayant");
            ps4.flush();
            System.out.println();
            System.out.println("------");

            //Writing to a file using PrintStream
            //Writes bytes not characters
            PrintStream ps5 = new PrintStream(fileStream);
            ps5.write(12);
            ps5.flush();
            ps5.close();

            //Writing to a file using PrintWriter
            //Writers characters
            PrintWriter pw = new PrintWriter(fileWriter);
            pw.write("Hello World..");
            pw.flush();
            pw.close();
      }
}
Output:
Prints nothing without flush method
New String
------
Printing with flush method
New String
------
Auto flush
New String
------
Appending string
New String
AJayant
------
```

InputStreamReader and OutputStreamWriter :

➔ An **InputStreamReader** is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

➔ An **OutputStreamWriter** is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.

➔ Each invocation of one of an InputStreamReader's read() methods may cause one or more bytes to be read from the underlying byte-input stream. To enable the efficient conversion of bytes to characters, more bytes may be read ahead from the underlying stream than are necessary to satisfy the current read operation. For top efficiency, consider wrapping an InputStreamReader within a BufferedReader. For example:

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

➔ Each invocation of a write() method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. The size of this buffer may be specified, but by default it is large enough for most purposes. Note that the characters passed to the write() methods are not buffered. For top efficiency, consider wrapping an OutputStreamWriter within a BufferedWriter so as to avoid frequent converter invocations. For example:

```
Writer out = new BufferedWriter(new OutputStreamWriter(System.out));
```

➔ Example

```
package testProg;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;


public class InputStreamReaderExample
{

    private static final String OUTPUT_FILE = "F:/Java material/java pgm
files/InputStreamReaderEx.txt";
    public static void main(String[] args)
    {
        char[] chars = new char[100];
        try
        {
            InputStreamReader inputStreamReader = new InputStreamReader(new
FileInputStream(OUTPUT_FILE),"UTF-8");
            // read 100 characters from the file
            while (inputStreamReader.read(chars) != -1)
                System.out.println(new String(chars));
        } catch (IOException e)
        {
          e.printStackTrace();
        }
     }
}
Output:
InputStreamReaderEx.txt: Hello world !! Today is a great day.
Console: Hello world !! Today is a great day.
```

5. Readers and Writers Classes :

➔ These classes has been added during Java 1.1 for Internationalization and not to replace InputStream and OutputStream Classes. Because the old I/O stream hierarchy supports only 8-bit byte streams and doesn't handle the 16-bit unicode characters well. Both **Reader** and **Writer** hierarchies supports unicodes in all I/O operations hence they are also called as Character Streams.

➔ There are times when you must use classes from the "byte" hierarchy *in combination* with classes in the "character" hierarchy. To accomplish this, there are "adapter" classes: **InputStreamReader** converts an **InputStream** to a **Reader**, and **OutputStreamWriter** converts an **OutputStream** to a **Writer**.

➔ Reader and Writer classes have all the corresponding I/O stream classes with the support for unicode.

➔ The package java.util.zip are byte oriented rather than char-oriented. So its better to use char-oriented I/O classes but there could be cases where one needs to use byte oriented classes(mostly during compiler gives error with char-oriented classes).

Sources & sinks: Java 1.0 class	Corresponding Java 1.1 class
InputStream	Reader adapter: InputStreamReader
OutputStream	Writer adapter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream (deprecated)	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

➔ Reader and Writer Classes for Filter Stream Classes

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract class with no subclasses)
BufferedInputStream	BufferedReader (also has readLine( ))
BufferedOutputStream	BufferedWriter
DataInputStream	Use DataInputStream (except when you need to use readLine( ), when you should use a
	BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream (deprecated)	LineNumberReader
StreamTokenizer	StreamTokenizer (Use the constructor that takes a Reader instead)
PushbackInputStream	PushbackReader

➔ Some Classes kept unchanged between java 1.0 and java 1.1 they are

Java 1.0 classes without corresponding Java 1.1 classes
DataOutputStream
File
RandomAccessFile
SequenceInputStream

5.1 CharArrayReader and CharArrayWriter Class :

➔ Example 1 : Reading from console

```
package inputOutputStream;
import java.io.*;

public class CharArrayClass {

	public static void main(String[] args)throws IOException
	{
		int i;
		CharArrayWriter cw = new CharArrayWriter();

		while( cw.size()!= 10 )
		{
			cw.write(System.in.read());
		}

		char [] c = cw.toCharArray();

		CharArrayReader cr = new CharArrayReader(c);

		while((i = cr.read())!= -1)
		{
			System.out.println((char)i);
		}
		cr.close();
	}
}
Output:
Hi how are you ?(input)
H
i

h
o
w

a
r
e
```

➔ Example 2 : Reading from a String

```
package inputOutputStream;
import java.io.*;

public class CharArrayString
{
	public static void main(String[] args) throws IOException
	{
		int i;
		String str = "Hello World";

		char [] c = str.toCharArray();

		CharArrayReader cr = new CharArrayReader(c);

		while((i = cr.read()) != -1)
		{
			System.out.println((char)i);
		}
	}
}
Output:
H
e
l
l
o

W
o
r
l
d
```

➔ Example 3 : Other methods of class CharArrayWriter like append() and writeto() has been used in the below example.

```
package inputOutputStream;

import java.io.*;

public class CharArrayOtherMeth {

	public static void main(String[] args) throws IOException
	{
		int i;
		char [] c1, c2;
		String str1 = "Hello how r u ?";
		String str2 = "I am fine";
		CharArrayOtherMeth cam = new CharArrayOtherMeth();
		CharArrayReader cr1, cr2;

		CharArrayWriter cw = new CharArrayWriter();

		//Reading from the writer
		cw.write(str1, 0, str1.length());
		c1 = cw.toCharArray();
		cr1 = new CharArrayReader(c1);

		System.out.println("Before append");
		cam.readMethod(cr1);
```

```
            //Appending the Writer
            cw.append(str2);
            c2 = cw.toCharArray();
            cr2 = new CharArrayReader(c2);

            System.out.println("After append");
            cam.readMethod(cr2);

            //Writing content of buffer to a file
            FileWriter fw = new FileWriter("F:/Java material/java pgm
            files/javapgmfile1.txt");
            cw.writeTo(fw);

            fw.close();
            cw.close();
      }

      void readMethod(CharArrayReader car) throws IOException
      {
            int c=0;
            while((c = car.read())!= -1)
            {
                  System.out.println((char)c);
            }
      }
}
Output:
Before append
H
e
l
l
o

h
o
w

r

u

?
After append
H
e
l
l
o

h
o
w

r

u

?
I

a
```

m

f
i
n
e

➔ Example 4

```
package inputOutputStream;

import java.io.*;

public class CharArrayStringPrint {

	public static void main(String[] args) throws IOException
	{
		String str;
		int c1, c2;
		char [] ch1;
		char [] ch2 = new char[1024];

		CharArrayWriter cw = new CharArrayWriter();

		while(cw.size()!= 10)
		{
			cw.write(System.in.read());
		}

		ch1 = cw.toCharArray();

		CharArrayReader cr1 = new CharArrayReader(ch1);
		CharArrayReader cr2 = new CharArrayReader(ch1);

		System.out.println("without string");
		while((c1 = cr1.read()) != -1)
		{
			System.out.println((char)c1);
		}

		System.out.println("with string");
		while((c2 = cr2.read(ch2)) != -1)
		{
			str = new String(ch2, 0, c2);
			System.out.println(str);
		}
	}
}
```

Output:

```
Hello World(console)
without string
H
e
l
l
o

W
o
r
```

```
l
with string
Hello Worl
```

How it works ?
We have the code

```
while((c2 = cr2.read(ch2)) != -1)
            {
                  str = new String(ch2, 0, c2);
                  System.out.println(str);
            }
```

Whats happening here is that cr2.read(ch2) will store input stream data into a char array 'ch2' and read(ch2) will return the number of characters read i.e. c2. Next a String 'str' will be constructed by using using ch2 and gets printed.

5.2 FileReader and FileWriter Class

```
package inputOutputStream;

import java.io.*;

public class FileReaderWriterClass {

      public static void main(String[] args) throws IOException
      {
            FileReader fr = null;
            FileWriter fw = null;

            try
            {
            fr = new FileReader("F:/Java material/java pgm files/CharReader1.txt");
            fw = new FileWriter("F:/Java material/java pgm files/CharReader2.txt");
                  int c;
            while((c = fr.read())!= -1)
            {
                  fw.write(c);
            }
            }
            finally
            {
                  if(fr != null)
                  {
                        fr.close();
                  }
                  if(fw != null)
                  {
                        fw.close();
                  }
            }
      }
}
```

6. FilterReader and FilterWriter Classes

6.1 BufferedReader and BufferedWriter Classes

Both works similar to BufferedInputStream and BufferedOutputStream classes. Please refer the notes and

check for examples. Only this classes extends FilterReader and FilterWriter classes respectively.

6.2 LineNumberReader Class

➔ A buffered Character input stream that keeps track of line numbers. This class defines methods setLineNumber(int) and getLineNumber() for setting and getting the current line number respectively.

➔ By default, line numbering begins at 0. This number increments at every line terminator as the data is read. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

➔ `public class LineNumberReader extends BufferedReader`

➔ Example :

```
package inputOutputStream;

import java.io.*;

public class LineNumberReaderCls
{
	public static void main(String [] args) throws IOException
	{
		int i;
		String str;
		String dataList = "F:/Java material/java pgm
		files/LineNumberReader.txt";

		try {
			FileReader fr = new FileReader(dataList);
			LineNumberReader lnr = new LineNumberReader(fr);

			//Read line till the end of the stream

			while((str=lnr.readLine()) != null)
			{
				i = lnr.getLineNumber();
				System.out.print("("+i+")");
				System.out.println(str);
			}
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
file:LineNumberReader.txt(contains)
apple
samsung
lg
nokia
htc

Output:
(1)apple
(2)samsung
(3)lg
(4)nokia
(5)htc
```

➔ setLineNumber() method

If we have one line

```
lnr.setLineNumber(20);
```

before while then output will be

```
Output:
(21)apple
(22)samsung
(23)lg
(24)nokia
(25)htc
```

As we can see, it will set line number to 20 and line reading will start from next line number i.e. 21. Hence we get the above output.

6.3 PushbackReader Class

➔ A character-stream reader that allows characters to be pushed back into the stream. When we read through a stream the control always goes to next byte/char. But with push back its possible to bring control back to the previous byte/char.

Constructor and Description
PushbackReader(Reader in) Creates a new pushback reader with a one-character pushback buffer.
PushbackReader(Reader in, int size) Creates a new pushback reader with a pushback buffer of the given size.

➔ Above we can see in the second constructor we can even define size for a push back buffer i.e. how many byte/char could be unread.

➔ Example

```
package inputOutputStream;

import java.io.*;

public class PushBackReaderCls {

	public static void main(String[] args) throws IOException
		{
		  String str = "a--b-cb---ab";
		  char charArray[] = str.toCharArray();
		  CharArrayReader chARed = new CharArrayReader(charArray);

		  PushbackReader pbr = new PushbackReader(chARed);
		  int c;
		  try {
		      while( (c = pbr.read())!= -1) {
		          if(c == '-') {
```

```
                        int nextC;
                       if( (nextC = pbr.read()) == '-'){
                            System.out.print("**");
                       }else {
                             //push backs a single character
                           pbr.unread(nextC);
                            System.out.print((char)c);
                       }
                     }else {
                             System.out.print((char)c);
                     }
                  }
            } catch(IOException ioe) {
                    System.out.println("Exception while reading" + ioe);
            }
        }
    }
    Output:
    a**b-cb**-ab
```

Above we can see that when two continuous hyphen(-) have not found then previous character will be pushed back and re-read.

➔ PushbackReader class have more unread methods as shown below

unread(char[] cbuf)

unread(char[] cbuf, int off, int len)

Through these methods one can unread array of characters or the portion of it(second method above). But if constructor has set the buffer limit than only that much will be allowed.

6.4 PrintWriter Class is same as PrintStream class. Please refer notes and examples.