Lecture 2:

STREAMS

elshaikh@unimap.edu.my

Objectives

- **DESCRIBE** and **IDENTIFY** basic concepts of (I/O) in Java:
  - Input streams read data
  - Output streams write data
  - Filter Streams
  - Readers and writers streams

Java I/O

- A large part of what network programs do is simple input and output:
  - moving bytes from one system to another.
- Reading data a server sends → reading a file
- Sending text to a client → writing a file

Java I/O

- Java I/O (Input and Output) is used to process the input and produce the output.
- Java uses the concept of stream to make I/O operation fast.
- The java.io package contains all the classes required for input and output operations.
- We can perform file handling in java by Java I/O API.

Stream

- A stream is a **sequence of data**.
- 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.
  - Input streams = read data
  - Output streams = write data
- In Java, 3 streams are created for us automatically. All these streams are attached with console.

1. **System.out**: standard output stream
2. **System.in**: standard input stream
3. **System.err**: standard error stream

System.out

- System.out is the first instance of the OutputStream class most programmers encounter.
- Specifically, System.out is the static out field of the java.lang.System class.
- It's an instance of java.io.PrintStream, a subclass of java.io.OutputStream.
- Normally, output sent to System.out appears on the console
  - console converts the numeric byte data System.out sends to it into ASCII or ISO Latin-1 text.

```
//For example, this will print Hello World to the console

byte[] hello = {72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33, 10};
System.out.write(hello);
```

System.in

- System.in is the input stream connected to the console.
- System.in is the static in field of the java.lang.System class. It is an instance of java.io.InputStream.
- When the user types into the console using the platform's default character set (typically ASCII), the data is converted into numeric bytes when read.

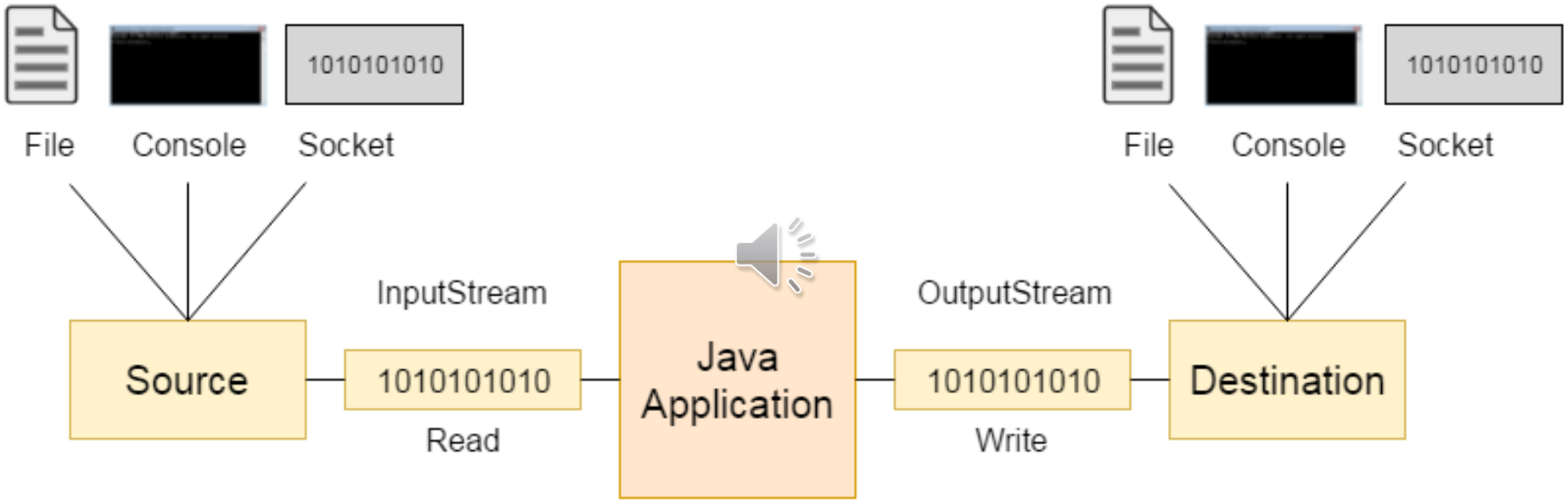
```
//For example, if the user types "Hello World!" and hits the return or enter
key, the following bytes will be read from System.in in this order:

72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33, 10, 13
```

System.err

- System.err is commonly used for error messages.
- System.err is an instance of java.io.PrintStream , a subclass of java.io.OutputStream .
- System.err is most commonly used **inside the catch clause of a try/catch block**, which is useful for debugging.

```
try {
        // Do something that may throw an exception.
}catch (Exception e) {
        System.err.println(e);
}
```

Java Input and Output Streams



- Java application uses an input stream to read data from a source, and an output stream to write data to a destination.
- It may be a file, an array, peripheral device or socket.

Output Streams

- Java's basic output class is **java.io.OutputStream**

```
public abstract class OutputStream
```

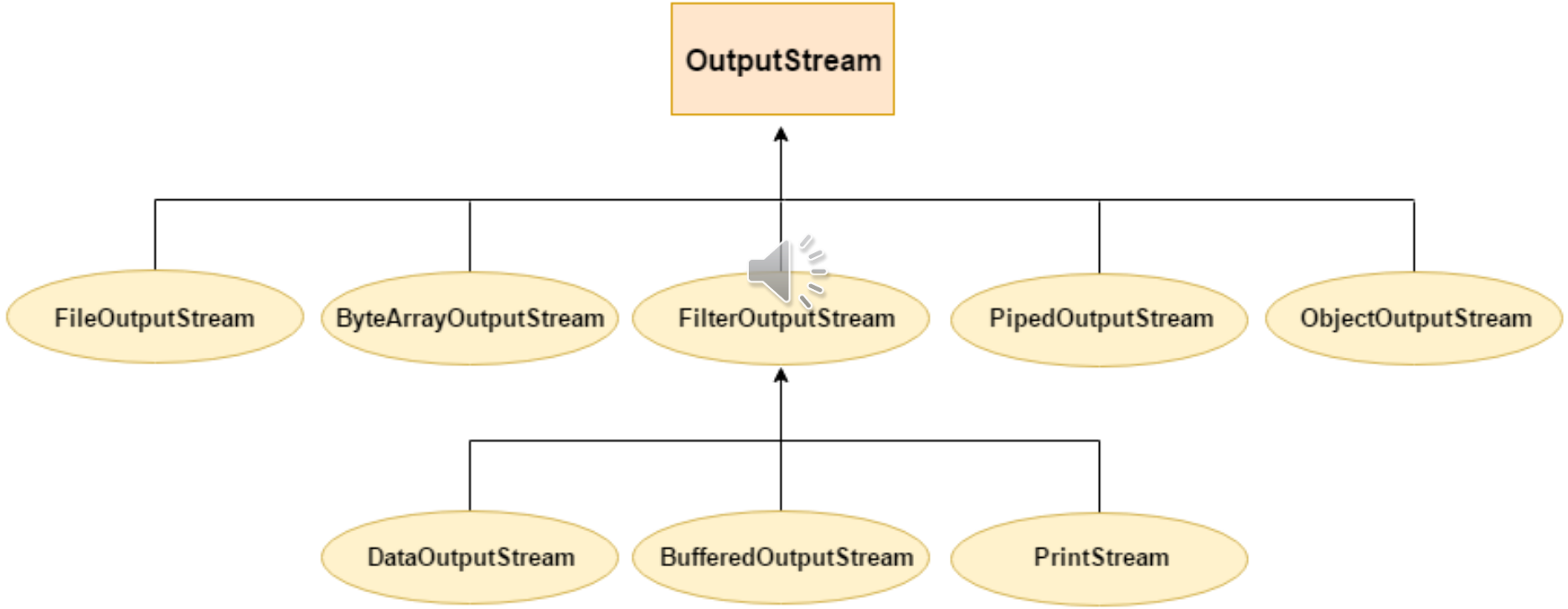
- This class provides the fundamental methods needed to write data:

```
public abstract void write(int b) throws IOException
public void write(byte[] data) throws IOException
public void write(byte[] data, int offset, int length)
                                        throws IOException
public void flush() throws IOException
public void close() throws IOException
```

Useful Methods of OutputStream class

Method	Description
`public void write(int data) throws IOException`	takes an integer from 0 to 255 as an argument and writes the corresponding byte to the output stream
`public void write(byte[] data) throws IOException`	used to write an array of byte to the current output stream.
`public void flush() throws IOException`	flushes the current output stream.
`public void close() throws IOException`	is used to close the current output stream.

OutputStream class Hierarchy

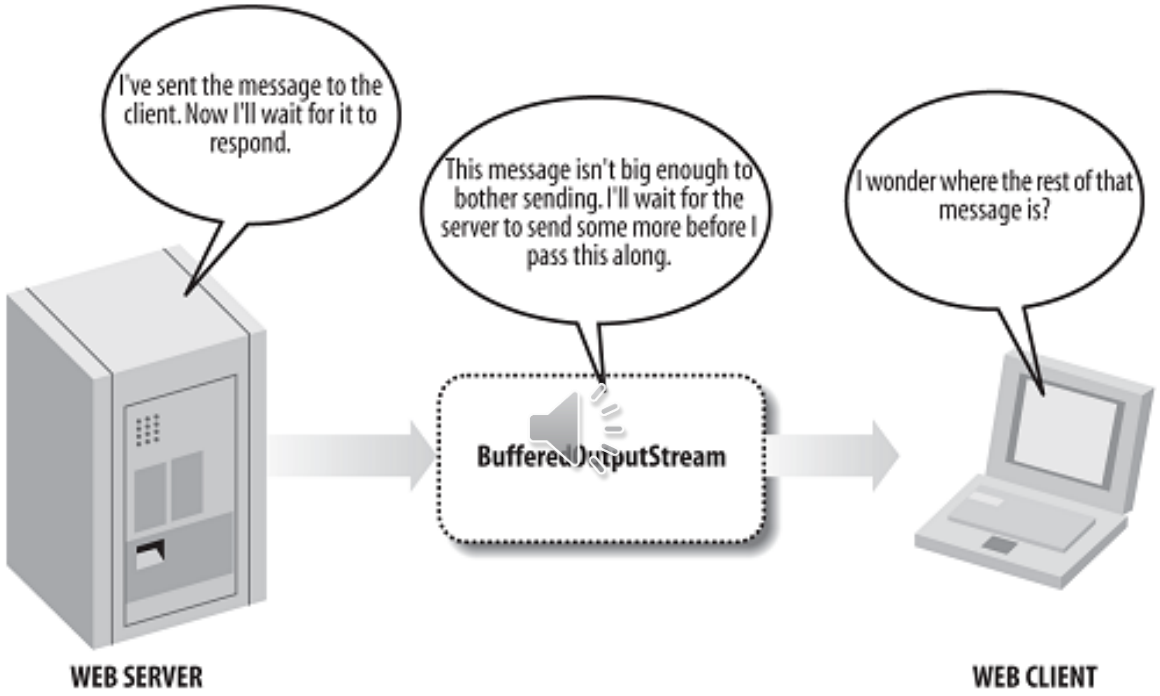


Buffer and Flush

- Streams can also be buffered in software, typically by chaining a BufferedOutput Stream or a BufferedWriter to the underlying stream.

- If you are done writing data, it's important to flush the output stream

Why Flush?



- No more data will be written onto the stream until the server response arrives, but the response is never going to arrive because the request has not been sent yet!
- The flush( ) method breaks the deadlock by forcing the buffered stream to send its data even if the buffer isn't yet full

FileOutputStream

- Java FileOutputStream is a subclass of OutputStream used for writing data to a file.
- If you have to write primitive values into a file, use FileOutputStream class.
- You can write **byte-oriented** as well as **character-oriented** data through FileOutputStream class.
- But, for character-oriented data, it is preferred to use FileWriter class instead.

FileOutputStream
Example 1: write byte

```
import java.io.FileOutputStream;

public class FileOutputStreamExample {

	public static void main(String args[]){

		try{

			FileOutputStream fout = new FileOutputStream("D:\\testout.txt");

			fout.write(65);

			fout.close();

			System.out.println("Success...");

		}catch(Exception e){

			System.out.println(e);

		}

	}

}
```

Output:

```
Success...
```

* The content of a text file testout.txt is set with the data A.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

FileOutputStream
Example 2: write String

```
import java.io.FileOutputStream;

public class FileOutputStreamExample {

        public static void main(String args[]){

                try{

                        FileOutputStream fout = new FileOutputStream("D:\\testout.txt");

                        String s = "Welcome to java.";

                        byte b[] = s.getBytes();//converting string into byte array

                        fout.write(b);

                        fout.close();

                        System.out.println("Success...");

                }catch(Exception e){System.out.println(e);}

        }

}
```

Output:

```
Success...
```

* The content of a text file testout.txt is set with the data Welcome to java.

Input Streams

- Java's basic input class is **java.io.InputStream**

```
public abstract class InputStream
```

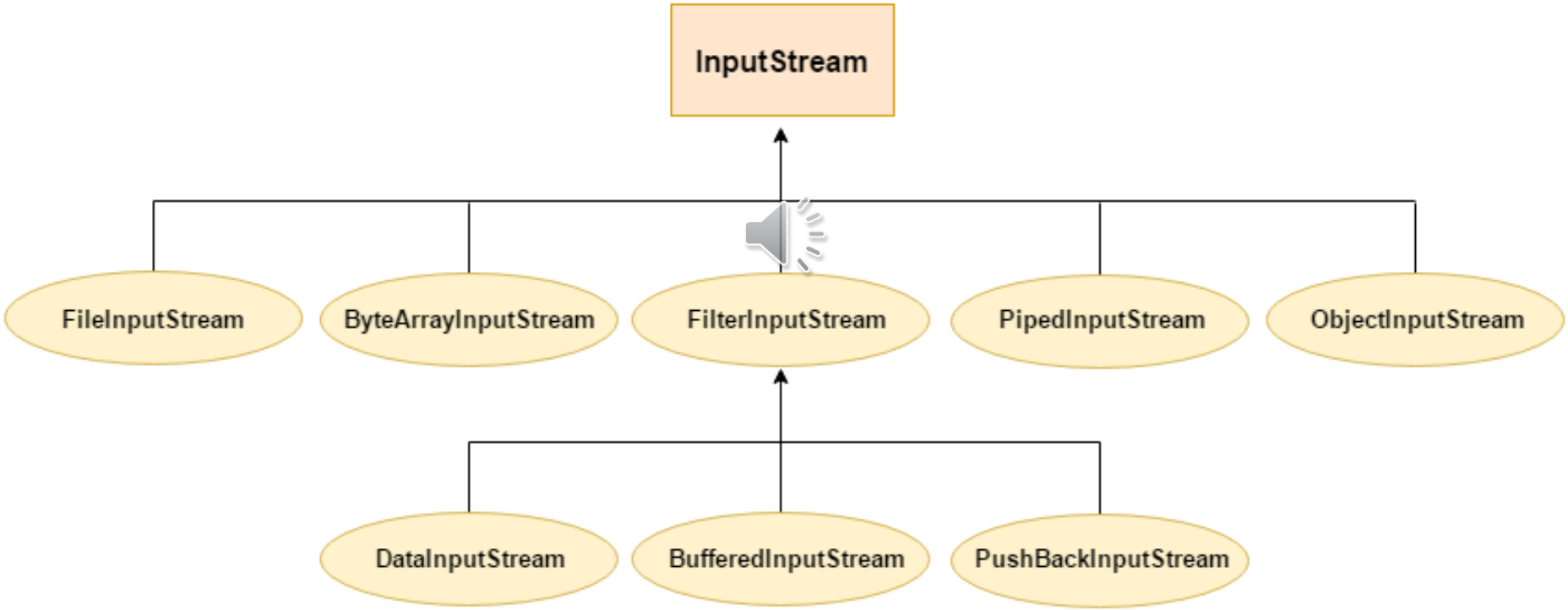
- This class provides the fundamental methods needed to read data:

```
public abstract int read() throws IOException
public int read(byte[] input) throws IOException
public int read(byte[] input, int offset, int length)
                                        throws IOException
public long skip(long n) throws IOException
public int available() throws IOException
public void close() throws IOException
```

Useful Methods of InputStream class

Method	Description
`public abstract int read() throws IOException`	reads the next byte of data from the input stream. It returns -1 at the end of file.
`public int available() throws IOException`	returns an estimate of the number of bytes that can be read from the current input stream.
`public void close() throws IOException`	used to close the current input stream.

InputStream class Hierarchy



FileInputStream

- Java FileInputStream class obtains input bytes from a file.
- It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.
- You can also read character-stream data.
- But, for reading streams of characters, it is recommended to use FileReader class.

FileInputStream
Example 1: read single character

```
import java.io.FileInputStream;

public class DataStreamExample {

        public static void main(String args[]){

                try{

                        FileInputStream fin = new FileInputStream("D:\\testout.txt");

                        int i = fin.read();

                        System.out.print((char)i);

                        fin.close();

                }catch(Exception e){System.out.println(e);}

        }

}
```

Note: Before running the code, a text file named as **"testout.txt"** is required to be created. In this file, we are having following content: **Welcome to java.**
After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.
Therefore, Output:

```
W
```

FileInputStream
Example 2: read all characters

```
import java.io.FileInputStream;

public class DataStreamExample {

    public static void main(String args[]){

        try{

            FileInputStream fin=new FileInputStream("D:\\testout.txt");

            int i=0;

            while((i=fin.read()) != -1){

                System.out.print((char)i);

            }

            fin.close();

        }catch(Exception e){System.out.println(e);}

    }

}
```

Note: Before running the code, a text file named as **"testout.txt"** is required to be created. In this file, we are having following content: **Welcome to java.**

Output:

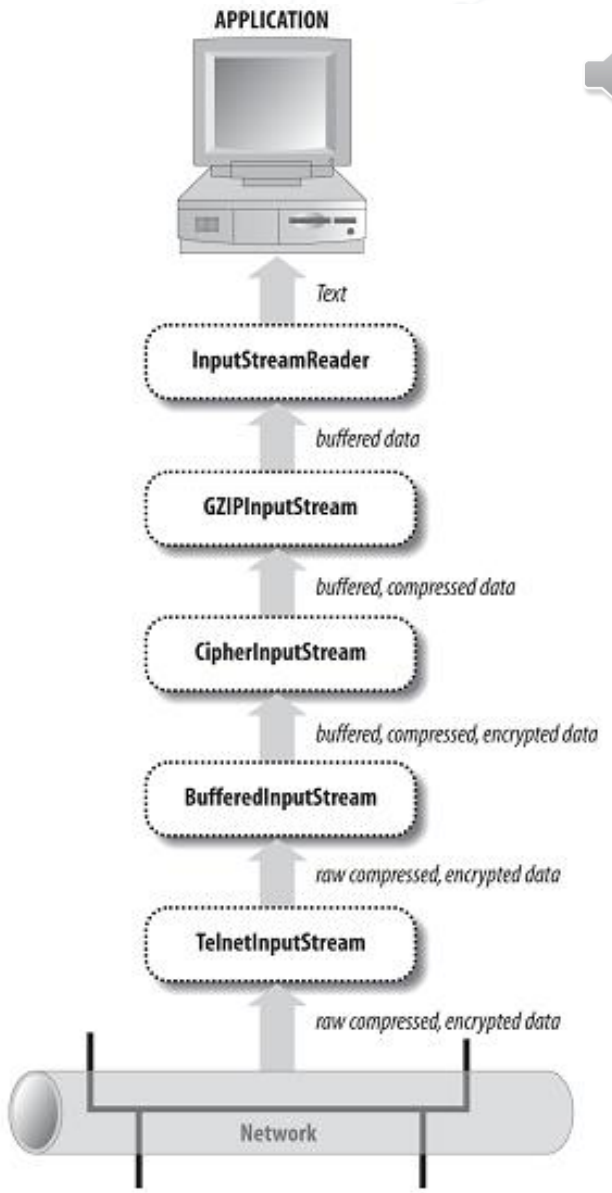
```
Welcome to java.
```

Filter Classes

- InputStream and OutputStream are fairly **raw** classes.
- They only read and write bytes singly or in groups, but does not recognize the data format.
- Java provides a number of **filter** classes you can attach to raw streams to translate the raw bytes to and from these and other formats.
- *2 versions of filters:*
  - Filter Streams
  - Readers and Writers.

Filter Streams

- Filters are organized in a chain.
- Each link in the chain receives data from the previous filter or stream and passes the data along to the next link in the chain.
- Every filter output stream has the same write(), close(), and flush() methods as java.io.OutputStream.
- Every filter input stream has the same read(), close(), and available() methods as java.io.InputStream.



In this example, a compressed, encrypted text file arrives from the local network interface

Chaining Filters Together

- Defines how bits and bytes of data are organized into the larger groups called packets, and the addressing scheme by which different machines find each other.

Chaining Filters Together

- Filters are connected to streams by their constructors. E.g:

```
FileInputStream     fin = new FileInputStream("data.txt");
BufferedInputStream bin = new BufferedInputStream(fin);
```

- Most of the time, you should only use the last filter in the chain to do the actual reading or writing
(by: overwrite the reference to the underlying input stream)

```
InputStream in = new FileInputStream("data.txt");
in = new BufferedInputStream(in);
```

*Connection is permanent. Filters cannot be disconnected from a stream.

Buffered Streams

- The BufferedOutputStream class stores written data in a buffer (`protected byte[] buf`) until the buffer is full or the stream is flushed.
- Then it writes the data onto the underlying output stream **all at once**.
- A single write of many bytes is almost always much faster than many small writes that add up to the same thing.
- BufferedInputStream has 2 constructors:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)
```

- BufferedOutputStream also has 2 constructors:

```
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)
```

Example

```
import java.io.*;

public class BufferedOutputStreamExample{

            public static void main(String args[])throws Exception{
                    FileOutputStream fout=new FileOutputStream("D:\\testout.txt");

                    BufferedOutputStream bout=new BufferedOutputStream(fout);

                    String s="Welcome to JAVA.";

                    byte b[]=s.getBytes();

                    bout.write(b);

                    bout.flush();

                    bout.close();

                    fout.close();

                    System.out.println("success");

             }

}
```

Readers and Writers

- APIs for reading and writing characters:
  - java.io.Reader API by which **characters** are read.
  - java.io.Writer API by which **characters** are written.
- Wherever input and output streams use bytes, readers and writers use **Unicode characters**.
- Concrete subclasses of Reader and Writer allow particular sources to be read and targets to be written.
- Filter readers and writers can be attached to other readers and writers to provide additional services or interfaces.
- 2 most important concrete R&W subclasses:
  - OutputStreamWriter class
  - InputStreamReader class

Writer

- The Writer class mirrors the java.io.OutputStream class. It's abstract and has two protected constructors.

```
protected Writer()
protected Writer(Object lock)
```

- Like OutputStream, the Writer class is never used directly; instead, it is used polymorphically, through one of its subclasses.

- It has five write() methods as well as a flush() and a close()

```
public abstract void write(char[] text, int offset, int length)
                                                    throws IOException
public void write(int c) throws IOException
public void write(char[] text) throws IOException
public void write(String s) throws IOException
public void write(String s, int offset, int length) throws IOException
public abstract void flush() throws IOException
public abstract void close() throws IOException
```

OutputStreamWriter

- The most important concrete subclass of Writer.
- Receives characters from a Java program, then
  - converts these into bytes according to a specified encoding and
  - writes them onto an underlying output stream.
- Its **constructor** has 2 parameters:

1. output stream to write to
2. the encoding to use (if no encoding is specified, the default encoding for the platform is used)

```
public OutputStreamWriter(OutputStream out, String encoding)
                                    throws UnsupportedEncodingException
```

OutputStreamWriter

Example:

```
OutputStreamWriter w;

w = new OutputStreamWriter(

        new FileOutputStream("OdysseyB.txt"), "Cp1253");

w.write("ἦμος δ᾽ ἠριγένεια φάνη ῥοδοδάκτυλος Ἠώς");
```

Reader

- Reader is the base class of all Reader's in the Java IO API. Subclasses include a BufferedReader, PushbackReader, InputStreamReader, StringReader and several others.

- It is abstract with two protected constructors. Like InputStream and Writer, the Reader class is never used directly, only through one of its subclasses.

```
Reader r = new FileReader("c:\\data\\myfile.txt");

int data = r.read();
while(data != -1) {
      char dataChar = (char)data;
      data = r.read();
}
```

InputStreamReader

- InputStreamReader is the most important concrete subclass of Reader.
- An InputStreamReader reads bytes from an underlying input stream such as a FileInputStream or TelnetInputStream. It converts these into characters according to a specified encoding and returns them.
- The **constructor** specifies the input stream to read from and the encoding to use:

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String encoding)
                                    throws UnsupportedEncodingException
```

InputStreamReader

Example:

```
InputStream is = new FileInputStream("c:\\data\\input.txt");
Reader isr = new InputStreamReader(is);

int data = isr.read();
while(data != -1){
    char theChar = (char)data;
    data = isr.read();
}

isr.close();
```

THANK YOU