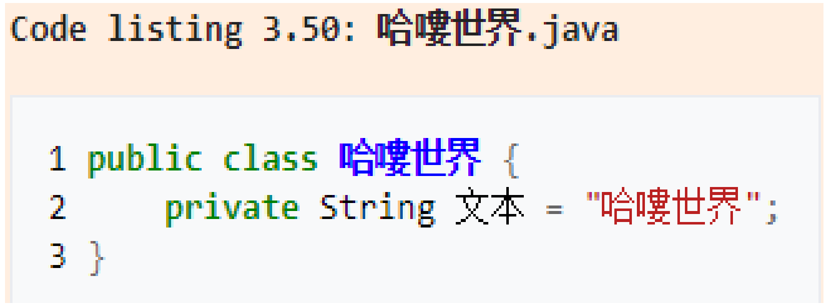
Files

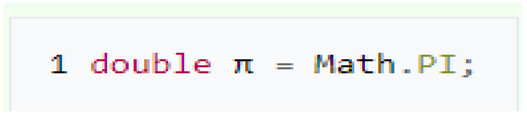
V P Jayachitra

Unicode

- Unicode is a character encoding
  - assigns a number to every character (and symbols) and hence every computer prints the same character (and symbols).
- Java uses a 16 bit Unicode UTF-16 to unify the world language characters. i.e 2^{16} - 65,536 characters
- UTF-Unicode Transformation unit
  - Hence java code can contain Chinese character(or symbol) as class name(or variable name) and string literal.


Code listing 3.50: 哈嘍世界.java

```
public class 哈嘍世界 {
    private String 文本 = "哈嘍世界";
}
```



```
double π = Math.PI;
```

I/O Streams

- Byte Streams handle I/O of raw binary data.

- Character Streams handle I/O of character data, automatic translation to and from the local character set.

- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.

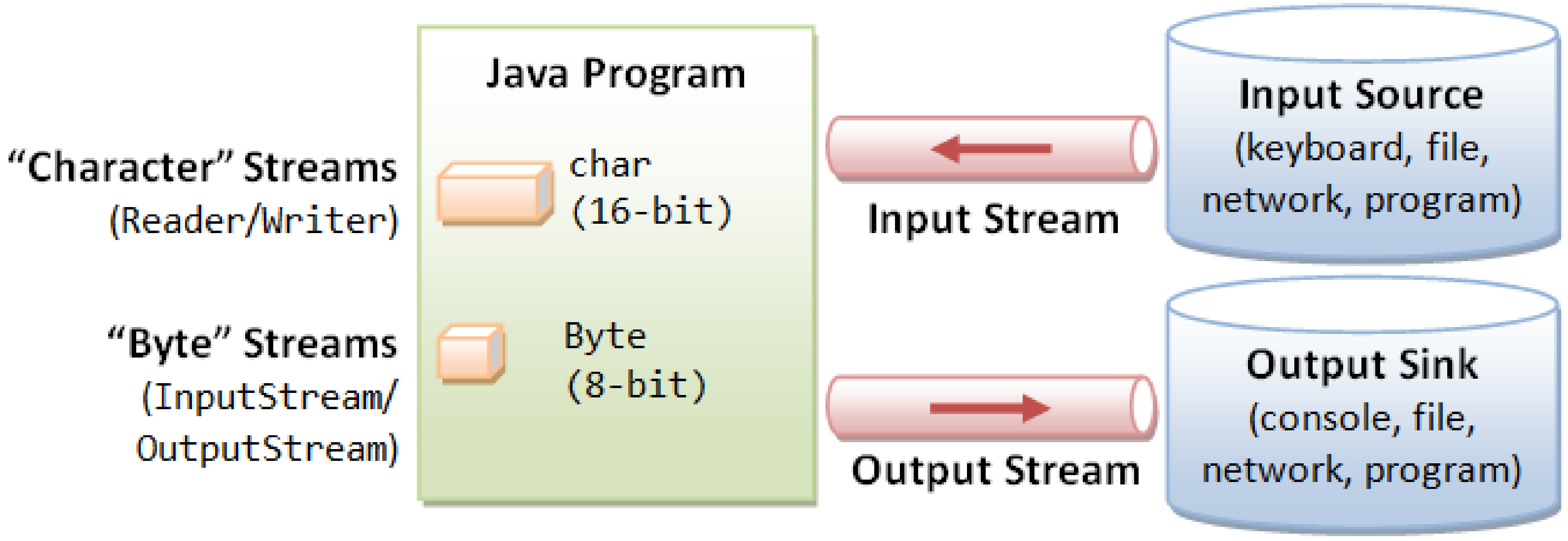
- I/O from the Command Line describes the Standard Streams and the Console object.

- Data Streams handle binary I/O of primitive data type and String values.

- Object Streams handle binary I/O of objects.

Contd…

- Java Programs read inputs from source(eg., File) and write outputs to destination(eg. File)
- Source or destination can also be a console, file, network, memory buffer, or another program.
- A ***stream*** is a sequence of data. In Java standard I/O, inputs and outputs are handled by *streams.*
- **Input Stream :** to read data from a source, one item at a time
- ***output stream:*** to write data to a destination, one item at time
- Stream I/O operations involve three steps:
  - *Open* an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
  - *Read* from the opened input stream until "end-of-stream" encountered, or *write* to the opened output stream (and optionally flush the buffered output).
  - *Close* the input/output stream.



Internal Data Formats:

- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Byte streams

- Byte streams are used to read/write *raw bytes* serially from/to an external device.
- Byte streams perform input and output on 8 bits.
- Byte streams should only be used for the most primitive I/O.
- Byte streams do not use **encoding scheme**
  - **(**Character streams use encoding scheme(UNICODE)).
  - Example: the text file uses **unicode encoding** to represent character in two bytes, the byte stream will read one byte at a time.
- Java.io.InputStream and java.io.OutputStream classes are used to read/write byte stream.

Why use character streams?

- easy to write programs
  - easy to internationalize i,.e that are not dependent upon a specific character encoding
- Efficient than byte streams.
  - To read a single character require two byte read operations.(char-2 byte)
- In java Char Data type is of two-byte,the only unsigned type in Java.
- Java.io.Reader and java.io.Writer classes are used to read/write char data.

Input Stream

- Input Stream
  - **To Read from an InputStream require read() method**

```
public abstract int read() throws IOException
```

- The read() method:
  - returns the int in the range of 0 to 255 –>byte that read
  - returns -1 -> end of stream
  - throws an IOException if it encounters an I/O error.

```
public int read(byte[] bytes, int offset, int length) throws IOException
             write(bytes, 0, bytes.length)
public int read(byte[] bytes) throws IOException
```

OutputStream

```
public void abstract void write(int unsignedByte) throws IOException
```

- to write a block of bytes ,byte-array :

```
public void write(byte[] bytes, int offset, int length) throws IOException

write(bytes, 0, bytes.length)
public void write(byte[] bytes) throws IOException
```

Reader

- superclass `Reader` operates on `char.`
  - It declares an `abstract` method `read()` to read one character from the input source.
  - `read()` returns the character as an `int` between 0 to 65535
    - (a `char` in Java can be treated as an unsigned 16-bit integer);
  - or **-1** if end-of-stream is detected;
  - or throws an `IOException` if I/O error occurs.

```
public abstract int read() throws IOException
public int read(char[] chars, int offset, int length) throws IOException
public int read(char[] chars) throws IOException
```

Writer

- superclass `Writer` operates on `char`.
  - It declares an `abstract` method `write()` to `write` one character into destination.
  - or throws an `IOException` if I/O error occurs.

```
public void abstract void write(int aChar) throws IOException

public void write(char[] chars, int offset, int length) throws IOException


public void write(char[] chars) throws IOException
```

Character-stream class	Description	Byte-stream class
Reader	Abstract class for character-input streams	InputStream
BufferedReader	Buffers input, parses lines	BufferedInputStream
CharArrayReader	Reads from a character array	ByteArrayInputStream
InputStreamReader	Translates byte stream into character stream (UTF-8/UTF-16 …)	-
FileReader	Translates bytes from a File into character stream	FileInputStream
Writer	Abstract class for character-output streams	OutputStream
BufferedWriter	Buffers Output, uses platform’s line separator	BufferedOutputStream
CharArrayWriter	Writes to a character array	ByteArrayOutputStream
OutputStreamWriter	Translates a character stream(UTF-8/UTF-16 …) into a byte stream	-
FileWriter	Translates character stream into a byte File	FileOutputStream

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBytes {
        public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
```

```
try {
   in = new FileInputStream(“SOURCE.txt");
  out =new FileOutputStream(“DESTINATION.txt");
  int c;
  while ((c = in.read()) != -1)
  {  out.write(c); }
  }
finally {
  if (in != null) {
      in.close(); }
  if (out != null)
   out.close(); }
}

}
}
```

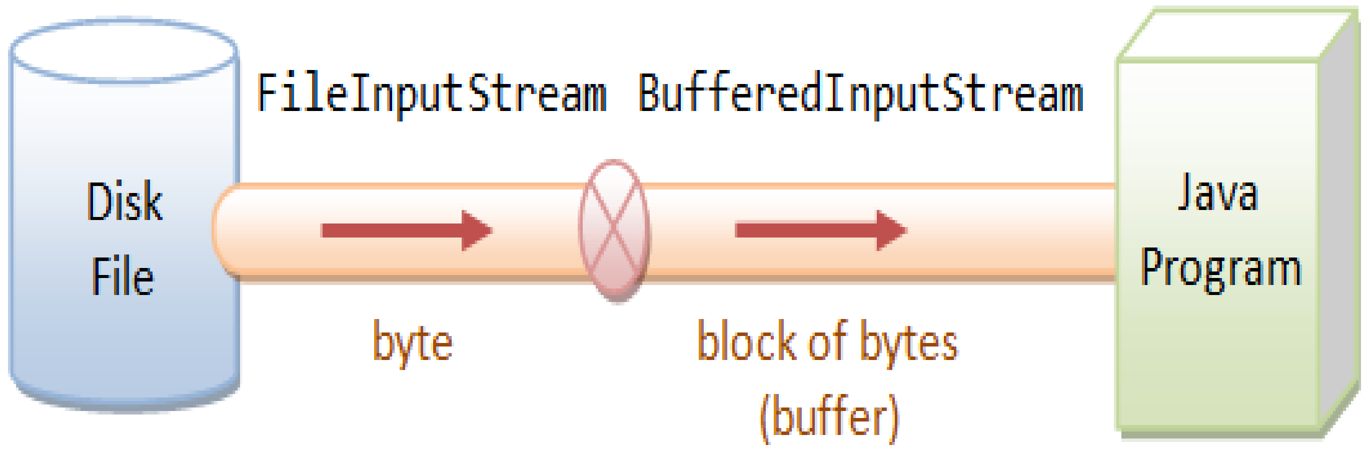
OR

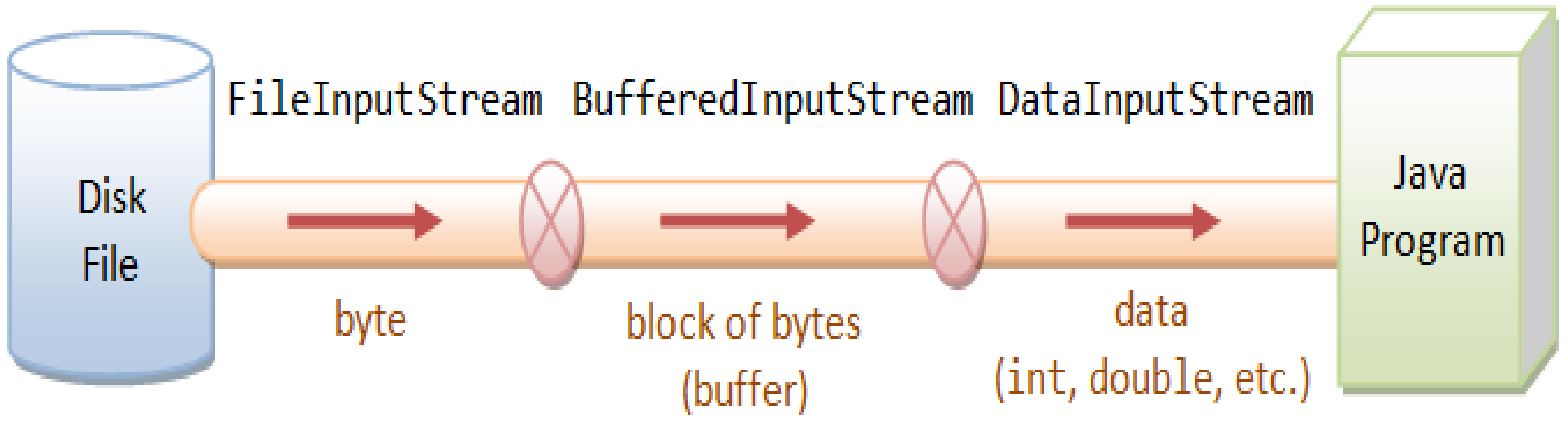
```
try (FileInputStream in = new
FileInputStream(“SOURCE.TXT”);
FileOutputStream out = new
FileOutputStream(“DESTINATION.TXT”))

{
int c;
while ((c = in.read()) != -1)
{ out.write(c); }
     }
}
}
```

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CopyCharacters {
        public static void main(String[] args) throws IOException {
                FileReader inputStream = null;
                FileWriter outputStream = null;
                try {
                        inputStream = new FileReader(“SOURCE.txt");
;                       outputStream = new FileWriter(“DESTINATION.txt”);
                        int c;
                        while ((c = inputStream.read()) != -1) {
                                outputStream.write(c); }
                        }
                finally {
                        if (inputStream != null) {
                                inputStream.close(); }
                        if (outputStream != null) {
                                outputStream.close(); }
                         }

 } }
```


Disk
File
FileInputStream
BufferedInputStream
byte
block of bytes
(buffer)
Java
Program


Disk
File
FileInputStream
BufferedInputStream
DataInputStream
byte
block of bytes
(buffer)
data
(int, double, etc.)
Java
Program

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBuffer{
    public static void main(String[] args) throws IOException {
  try (BufferedInputStream in = new BufferedInputStream(new FileInputStream("Source.txt"));
   BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("Dest.txt")))

   {
    int c;
   while ((c = in.read()) != -1)
    { out.write(c); }
      }
  }
  }
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBuffer{
    public static void main(String[] args) throws IOException {
try (BufferedReader in = new BufferedReader(new FileReader("Source.txt"));
 BufferedWriter out = new BufferedWriter(new FileWriter("Dest.txt")))

  {
   String s;
   while ((s = in.readLine()) != null) {

   out.write(l);
     }
}
}
```

BufferedReader and BufferedWriter perform buffered I/O, instead of character-by-character.
BufferedReader provides a new method readLine(), which reads a line and returns
a String (without the line delimiter).

Lines could be delimited by "\n"

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyBuffer{
    public static void main(String[] args) throws IOException {
      try (BufferedReader in = new BufferedReader(new FileReader("Source.txt"));
       PrintWriter out = new PrintWriter(new FileWriter("Dest.txt")))

       {
        String s;
        while ((s = in.readLine()) != null) {

        out.println(l);
          }
      }
      }
```

autoflush : PrintWriter object flushes the buffer on every invocation of println or format.
To flush a stream manually, invoke its flush method. The flush method is valid on any output stream.

```
import java.io.*;

class read{
 public static void main(String[] args) throws Exception{
// Read text file with specified unicode.
   FileInputStream fr=new FileInputStream("F:/java/eee.txt");
   InputStreamReader isr=new InputStreamReader(fr,"UTF-8");
   BufferedReader br=new BufferedReader(isr);
    String s;
   while ((s = in.readLine()) != null) {
          System.out.println(s);
}
}
}
```

```
public InputStreamReader(InputStream in) // Use default Unicode/charset
public InputStreamReader(InputStream in, String charsetName)throws UnsupportedEncodingException

public InputStreamReader(InputStream in, Charset cs)
```

`InputStreamReader/OutputStreamWriter` wraps in `BufferedReader/BufferedWriter` to read/write in multiple bytes.

Format Specifiers

- A format specifier begins with `'%‘.`
- A format specifier ends with a conversion-type character.

e.g. `"%d"` for integer, `"%f"` for `float` and `double`
optional parameters in between, as follows:

`%[argument_position$][flag(s)][width][.precision]conversion-type-character`

optional

•*argument_position* specifies the position of the argument in the argument list.
  •The first argument is `"1$"`, second argument is `"2$"`, and so on.

- *width* indicates the minimum number of characters to be output.
- *precision* restricts the number of decimal places for float-point numbers.

Mandatory

- *conversion-type-character* indicates how the argument should be formatted.

examples:
`'`, `'B'` (boolean), `'h'`, `'H'` (hex),
`'s'`, `'S'` (string), `'c'`, `'C'` (character),
`'d'`(decimal integer), `'o'` (octal integer), `'x'`, `'X'` (hexadecimal integer),
`'e'`, `'E'` (float-point number in scientific notation), `'f'` (floating-point number),

- `'%'` (percent sign).
- The uppercase conversion code (e.g., `'S'`) formats the texts in uppercase.

Contd…

```
System.out.printf("%2$2d %3$2d %1$2d%n", 1, 12, 123, 1234);
```

- 12 123 1

```
System.out.printf("Hello %4d %6.2f %s, and%n Hello again%n",
123, 5.5, "Hello");
```

- Hello 123 5.50 Hello , and

 Hello again