

# Computer System Design & Application

## 计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn



# Lecture 5

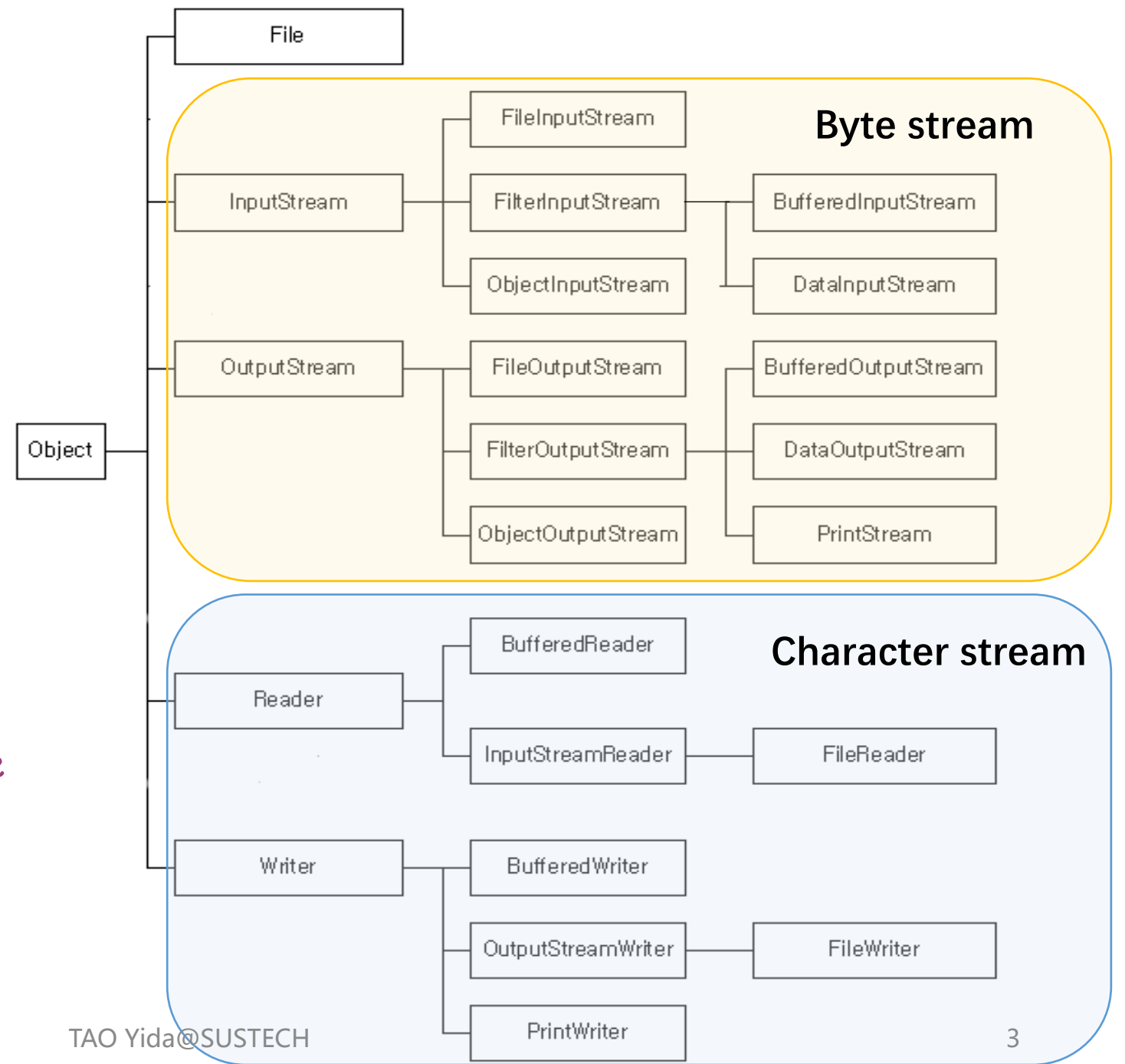
---

- I/O Overview
- i18n & Character Encoding
- Byte Streams & Character Streams
- Combining Stream Filters
- Reading/Writing Text Input/Output
- I/O from Command Line

# Overview

- Java I/O and File are in java.io package
- I/O classification
  - Input and output
  - Byte stream vs Character stream

Character Stream is used to handle  
Internationalization





## Internationalization (i18n)

- Internationalization refers to making programs that can take input and output that is tailored to different locations and languages.
- As different languages contain a wide variety of letters and characters, **character encoding** is an important element to make software systems international or language/location independent.



# Encoding

- Convert characters (字符) to other formats, often numbers, in order to store and transmit them more effectively

The International Morse Code (摩斯电码, 1837) encodes A-Z, numbers, and some other characters.

... — — — ...  
S O S

## International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.



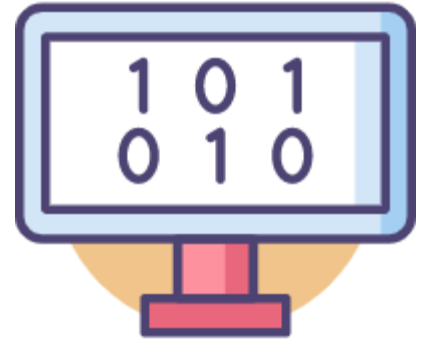
000	0181 俯	0161 仉	0141 伏	0121 佞	0101 仕	0081 上	0061 了	0041 乍	0021 丁	0001 一
1	0182 佳	0162 位	0142 伐	0122 伏	0102 仁	0082 亡	0062 屯	0042 乎	0022 丫	0002 丁
2	0183 併	0163 低	0143 休	0123 仰	0103 仃	0083 亢	0063 予	0043 弟	0023 中	0003 七
3	0184 倍	0164 住	0144 伙	0124 仔	0104 仆	0084 交	0064 事	0044 乏	0024 丰	0004 丈
4	0185 份	0165 佐	0145 伯	0125 伶	0105 仇	0085 亥	0065 乖	0045 串	0025 三	0005 三
5	0186 使	0166 佑	0146 估	0126 仲	0106 今	0086 亦	0066 乘	0046 上	0026 上	0006 上
6	0187 侃	0167 佔	0147 佚	0127 伙	0107 介	0087 享	0067 二	0047 下	0027 下	0007 下
7	0188 來	0168 何	0148 你	0128 低	0108 仍	0088 庇	0068 于	0048 丷	0028 丷	0008 丷
8	0189 侈	0169 伐	0149 僂	0129 仵	0109 仔	0089 亨	0069 云	0049 九	0029 九	0009 九
9	0190 例	0170 余	0150 伴	0130 伴	0110 仕	0090 京	0070 互	0050 乙	0030 凡	0010 丑
0	0191 侍	0171 余	0151 伶	0131 攸	0111 他	0091 亭	0071 五	0051 九	0031 丹	0011 且
1	0192 侏	0172 佛	0152 仲	0132 价	0112 仗	0092 亮	0072 井	0052 乞	0032 主	0012 丕
2	0193 值	0173 作	0153 伺	0133 任	0113 付	0093 毫	0073 亘	0053 也	0033 世	0013 世
3	0194 侑	0174 佞	0154 伴	0134 仿	0114 仙	0094 亘	0074 互	0054 乚	0034 丘	0014 丘
4	0195 侑	0175 佟	0155 似	0135 企	0115 全	0095 疊	0075 况	0055 乳	0035 丙	0015 丙
5	0196 侑	0176 佩	0156 伽	0136 伉	0116 仞	0096 些	0076 乾	0056 父	0036 丞	0016 丞
6	0197 侑	0177 侗	0157 佃	0137 伊	0117 仟	0097 亞	0077 亂	0057 乃	0037 丢	0017 丢
7	0198 供	0178 佺	0158 但	0138 伋	0118 代	0098 亟	0078 亟	0058 久	0038 並	0018 並
8	0199 依	0179 伴	0159 佇	0139 伍	0119 令	0099 令	0079 令	0059 么	0039 之	0019 之
9	0200 伽	0180 伯	0160 佈	0140 伎	0120 以	0100 人	0080 丁	0060 之	0040 之	0020 之

# Chinese Telegraph Code (中文电码, 1872)

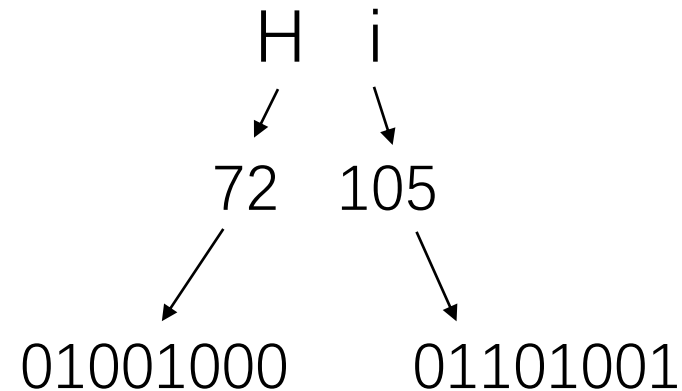
One-to-one mapping between Chinese characters and four-digit numbers from 0000 to 9999

# ASCII

- Represent text in computers
- Using 7 bits to represent 128 characters
- Extended ASCII uses 8 bits for 256 characters



Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	~	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL



# GB2312, GBK, GB18030

- GB stands for 国标
- GB2312 uses 2 bytes (cover 99% daily usages)
- GBK (国标扩展) extends GB2312 to encode more characters
- GB18030 extends GBK

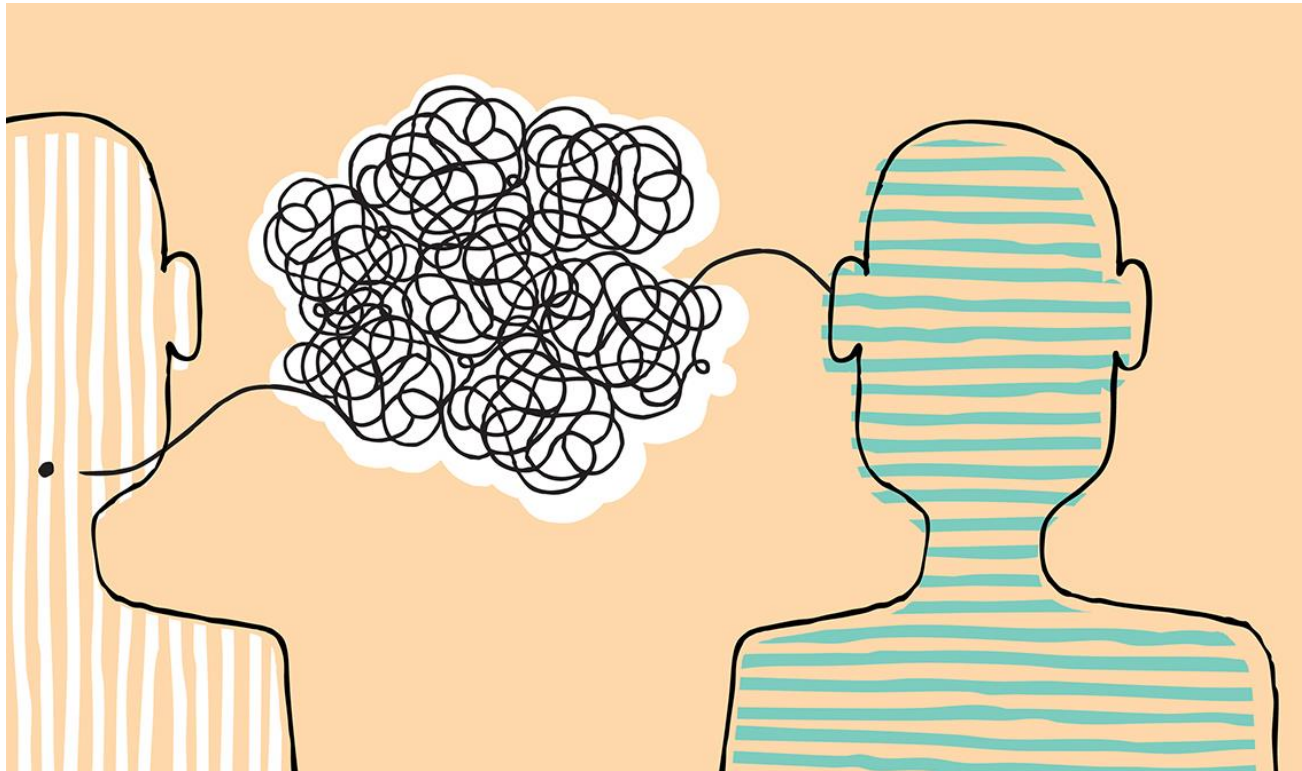
编 码

B1E0 C2EB

1011000111100000 1100001011101011



# Problems?



## Communication
















Different countries with different language systems implement their own character encoding (e.g., sending text message)

# Unicode (统一码、万国码、单一码)

<https://unicode-table.com/en/>

- Motivated by the need to encode characters in all languages consistently without conflicts
- A character maps to something called a code point (A: U+0041)
- Unicode comprises 1,114,112 code points in the range  $0_{\text{hex}}$  to  $10\text{FFFF}_{\text{hex}}$

A	a	B	b	C	c	D	d
U+0041	U+0061	U+0042	U+0062	U+0043	U+0063	U+0044	U+0064
E	e	F	f	G	g	H	h
U+0045	U+0065	U+0046	U+0066	U+0047	U+0067	U+0048	U+0068
I	i	J	j	K	k	L	l
U+0049	U+0069	U+004A	U+006A	U+004B	U+006B	U+004C	U+006C
M	m	N	n	O	o	P	p
U+004D	U+006D	U+004E	U+006E	U+004F	U+006F	U+0050	U+0070

				
U+1F602	U+2764	U+1F60D	U+1F923	U+1F60A
				
U+1F525	U+1F618	U+1F44D	U+1F970	U+1F60E
				
U+1F914	U+1F605	U+1F614	U+1F644	U+1F61C

# Encoding Scheme

- Unicode is a standard (defines the mapping to code point)
- An encoding scheme follows the Unicode standard and defines how code points are stored in memory
- There are different encoding schemes for packaging Unicode code points into bytes, e.g., UTF-8, UTF-16, UTF-32

## UTF-8

- Uses a minimum of 1 byte, but if the character is bigger, then it can use 2, 3 or 4 bytes.
- is compatible with the ASCII table

## UTF-16

- uses a minimum of 2 bytes. UTF-16 can not take 3 bytes, it can either take 2 or 4 bytes
- is not compatible with the ASCII table

## UTF-32

- always uses 4 bytes
- is not compatible with the ASCII table

character	encoding	bits
A	UTF-8	01000001
A	UTF-16	00000000 01000001
A	UTF-32	00000000 00000000 00000000 01000001
あ	UTF-8	11100011 10000001 10000010
あ	UTF-16	00110000 01000010
あ	UTF-32	00000000 00000000 00110000 01000010

# UTF-8

- UTF-8 stands for “Unicode Transformation Format – 8-bit”
- Characters are encoded with varied lengths (1~4 bytes)
  - For example: "T" in UTF-8 is "01010100"
  - "汉" in "UTF-8" is "11100110 10110001 10001001 "

Character Range	Encoding
0...7F	0a <sub>6</sub> a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
80...7FF	110a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> 10a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
800...FFFF	1110a <sub>15</sub> a <sub>14</sub> a <sub>13</sub> a <sub>12</sub> 10a <sub>11</sub> a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> 10a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>
10000...10FFFF	11110a <sub>20</sub> a <sub>19</sub> a <sub>18</sub> 10a <sub>17</sub> a <sub>16</sub> a <sub>15</sub> a <sub>14</sub> a <sub>13</sub> a <sub>12</sub> 10a <sub>11</sub> a <sub>10</sub> a <sub>9</sub> a <sub>8</sub> a <sub>7</sub> a <sub>6</sub> 10a <sub>5</sub> a <sub>4</sub> a <sub>3</sub> a <sub>2</sub> a <sub>1</sub> a <sub>0</sub>

Image: Core Java  
Volume II, 2.2.4

# UTF-8

<https://stackoverflow.com/a/27939161/636398>

A Chinese character: 汉  
its Unicode value: U+6C49  
convert 6C49 to binary: 01101100 01001001

To computer, its simply 0110110001001001  
(don't know whether its 1 or 2 character)

1st Byte	2nd Byte	3rd Byte	4th Byte	Number of Free Bits
0xxxxxxx				7
110xxxxx	10xxxxxx			(5+6)=11
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21

Add header to free bits

Header	Place holder	Fill in our Binary	Result
1110	xxxx	0110	11100110
10	xxxxxx	110001	10110001
10	xxxxxx	001001	10001001

11100110 10110001 10001001



# Java char

```
int v1 = 0x0454; // Hex
System.out.printf("%c\n", v1); //e
System.out.printf("%c\n", (char)v1); //e

int v2 = 1108; // Decimal
System.out.printf("%c\n", v2); //e
System.out.printf("%c\n", (char)v2); //e

int v3 = 0x10454; // Hex
System.out.printf("%c\n", v3); //d
System.out.printf("%c\n", (char)v3); //e
```

## Java char implementation

- 16-bit unsigned int (U+0000~U+FFFF), corresponding to Unicode code points
- Conversion between int and char refers to the Unicode mapping

<https://unicode-table.com/en>

# Java char

- Unicode legal range now: U+0000 to U+10FFFF
- Characters whose code points are greater than U+FFFF are called supplementary characters
- Supplementary characters are represented as a pair of char values

```
int v1 = 0x0454;  
int v3 = 0x10454;  
char[] c1 = Character.toChars(v1); // length 1  
char[] c2 = Character.toChars(v3); // length 2  
System.out.println(c1); //e  
System.out.println(c2); //ð
```



# Lecture 5

---

- I/O Overview
- i18n & Character Encoding
- Byte Streams & Character Streams
- Combining Stream Filters
- Reading/Writing Text Input/Output
- I/O from Command Line

# Java I/O Streams

- A Stream is a continuous flow of data that can be accessed sequentially (not like an array for which we could use index to move back and forth)
- A Stream, as a data container, is linked to a data source and a data destination

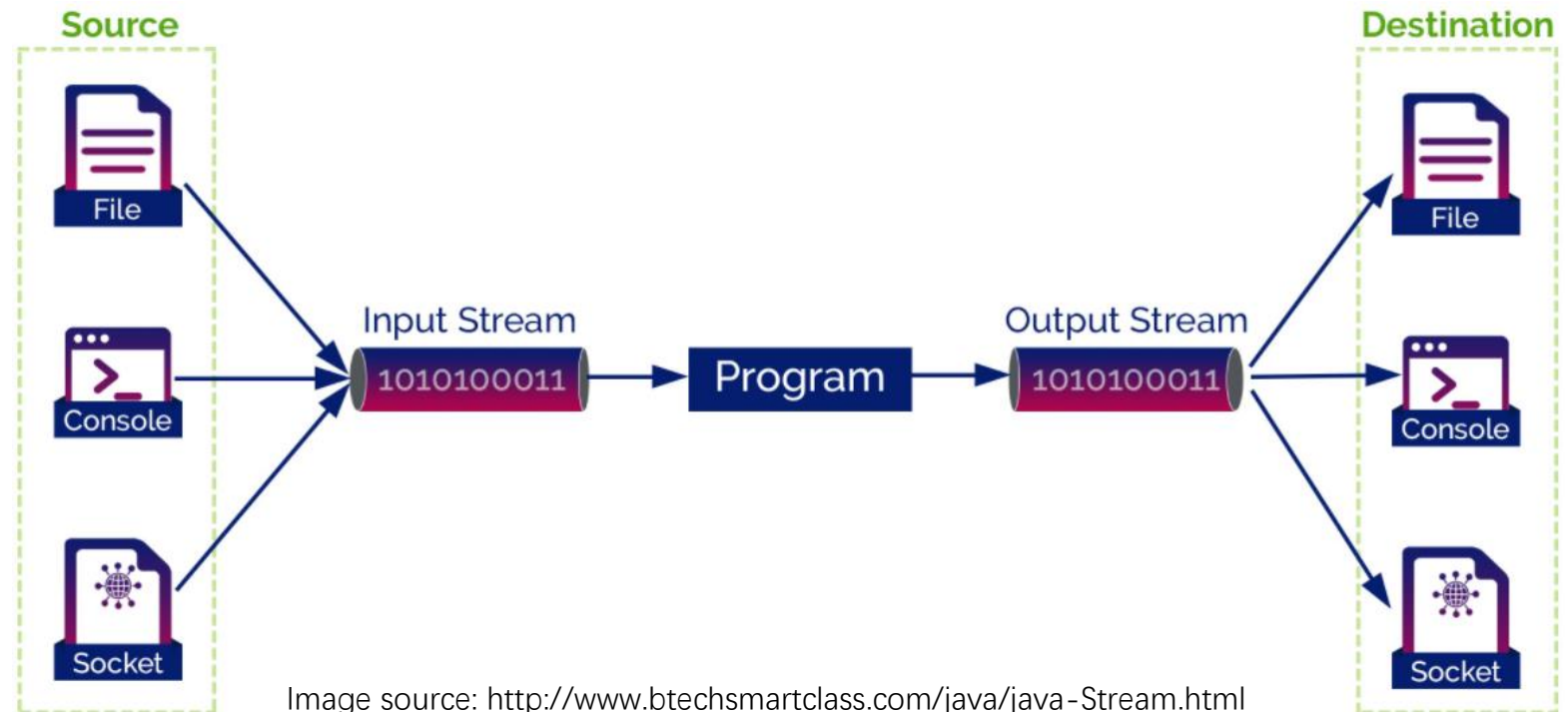


Image source: <http://www.btechsmartclass.com/java/java-Stream.html>

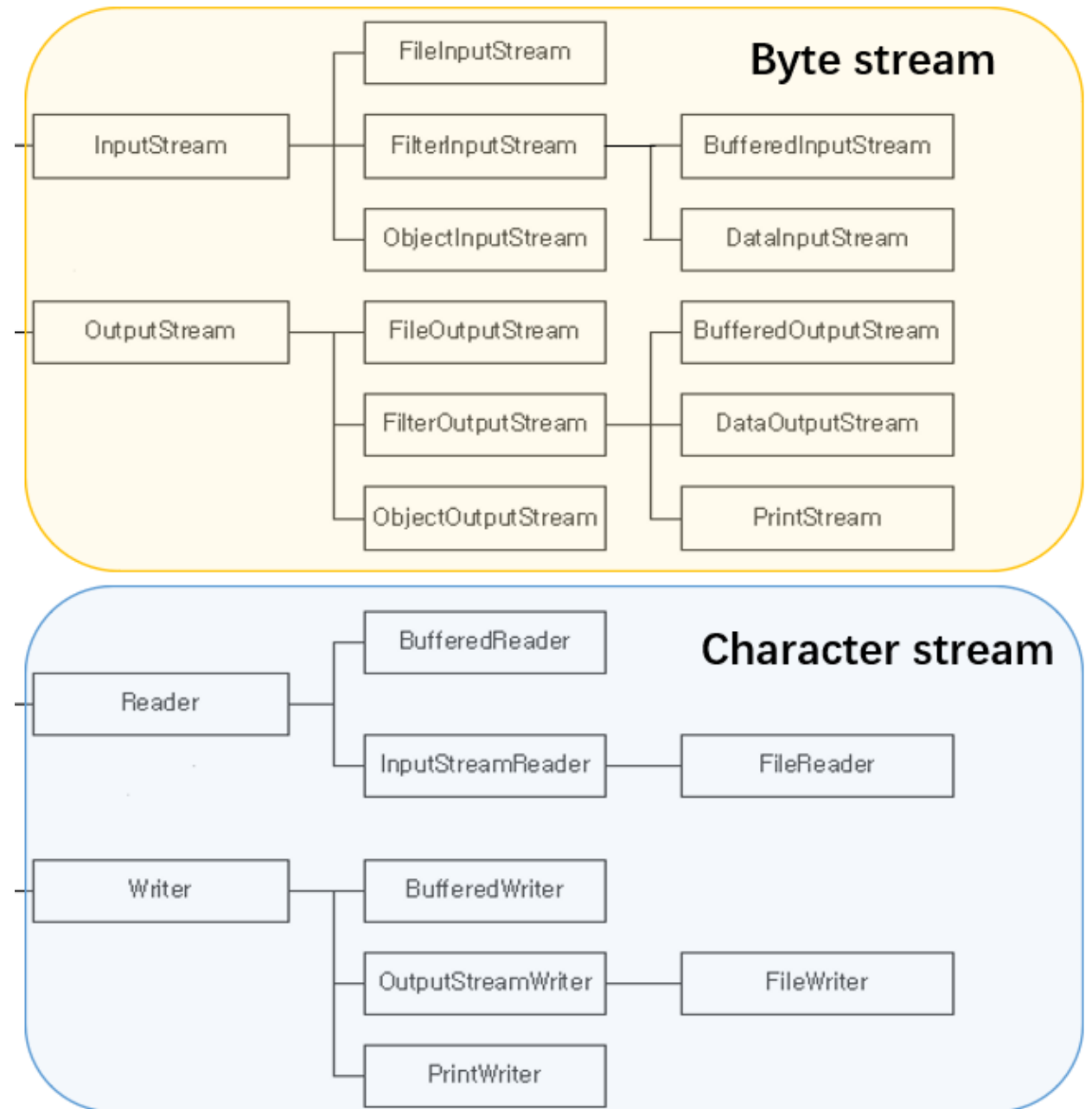
# Java I/O Streams

- Byte Stream

- *Input stream*: an object from which we can read a sequence of bytes
- *Output stream*: an object to which we can write a sequence of bytes
- Byte streams are inconvenient for processing info stored in Unicode

- Character Stream

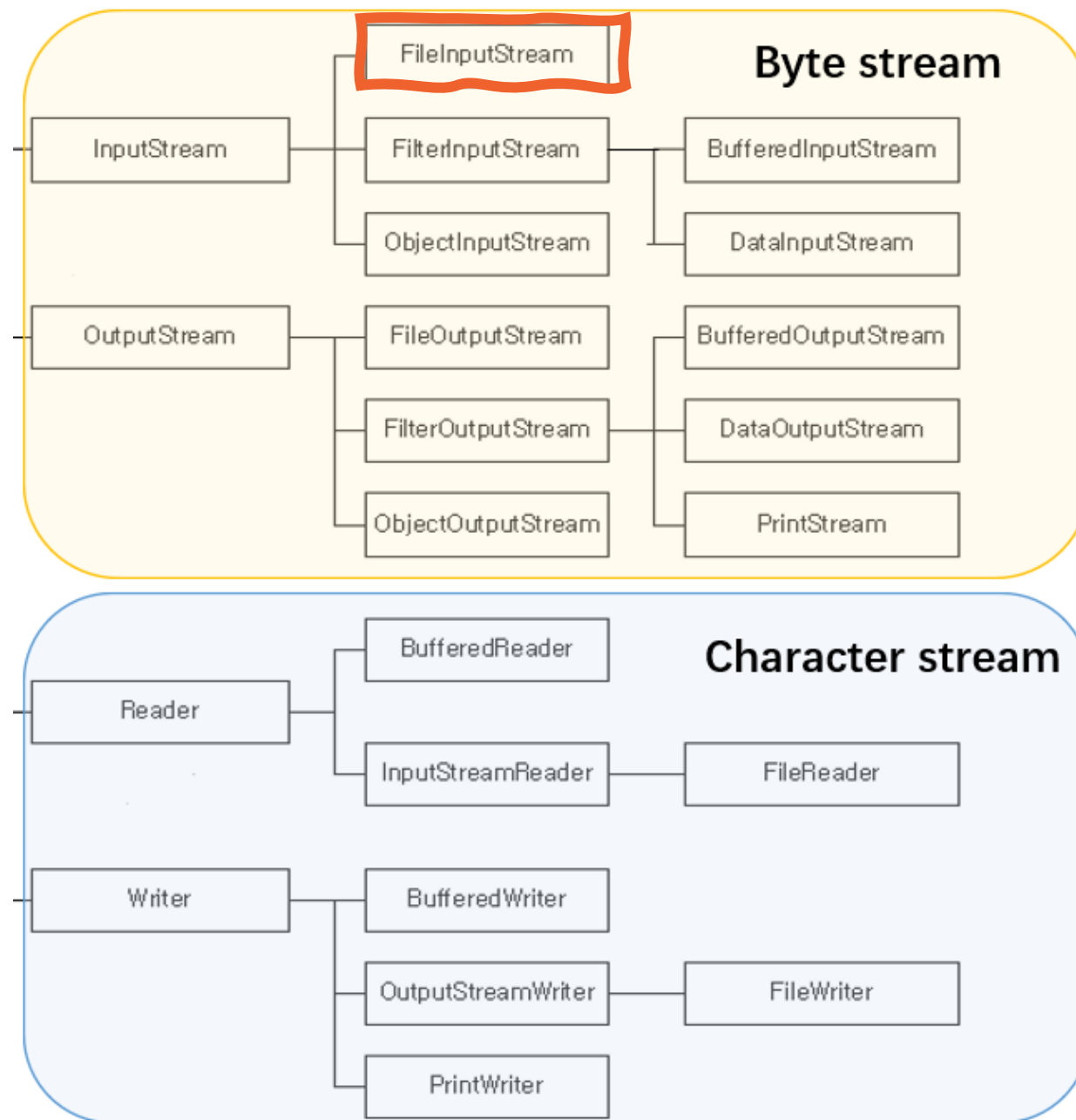
- A separate hierarchy provides classes, inheriting from Reader and Writer, for processing Unicode characters
- These classes have read and write operations that are based on char values rather than byte values





# Similarity

- InputStream & OutputStream, Reader & Writer are abstract classes
- Subclasses are all called “xxxStream” or “xxxReader” & “xxxWriter”
- Subclasses for InputStream or Reader must implement read()
- Subclasses for OutputStream or Writer must implement write()



# FileInputStream

Used for reading streams of raw bytes

```
public void readFile() throws IOException {  
    try (InputStream input = new FileInputStream("src/test.txt")) {  
        int n;  
        while ((n = input.read()) != -1) {  
            System.out.println(n);  
        }  
    }  
}
```

Reading 1 byte a time until  
there is no more data (-1)

What is the output when test.txt  
contains the text "Hello World"?  
(e.g., file encoding is UTF-8)

72 101 108 108 111 32 87 111 114 108 100



# FileInputStream

Used for reading streams of raw bytes

- What if test.txt contains “计算机系统” ?

```
try (InputStream input = new FileInputStream("src/test.txt")) {  
    int n;  
    while ((n = input.read()) != -1) {  
        System.out.print(" " + n);  
    }  
}
```

## If file encoding is UTF-8

232 174 161 231 174 151 230 156 186  
231 179 187 231 187 159

In UTF-8, normal Chinese characters often take  
3 bytes

## If file encoding is GBK

188 198 203 227 187 250 207 181 205 179

1 Chinese Character requires more than 1 byte  
to store (2 bytes for GBK encoding)

GBK encoding for 计: BCC6

# FileInputStream

Used for reading streams of raw bytes

- How to get meaningful characters? Can we directly cast bytes to char?

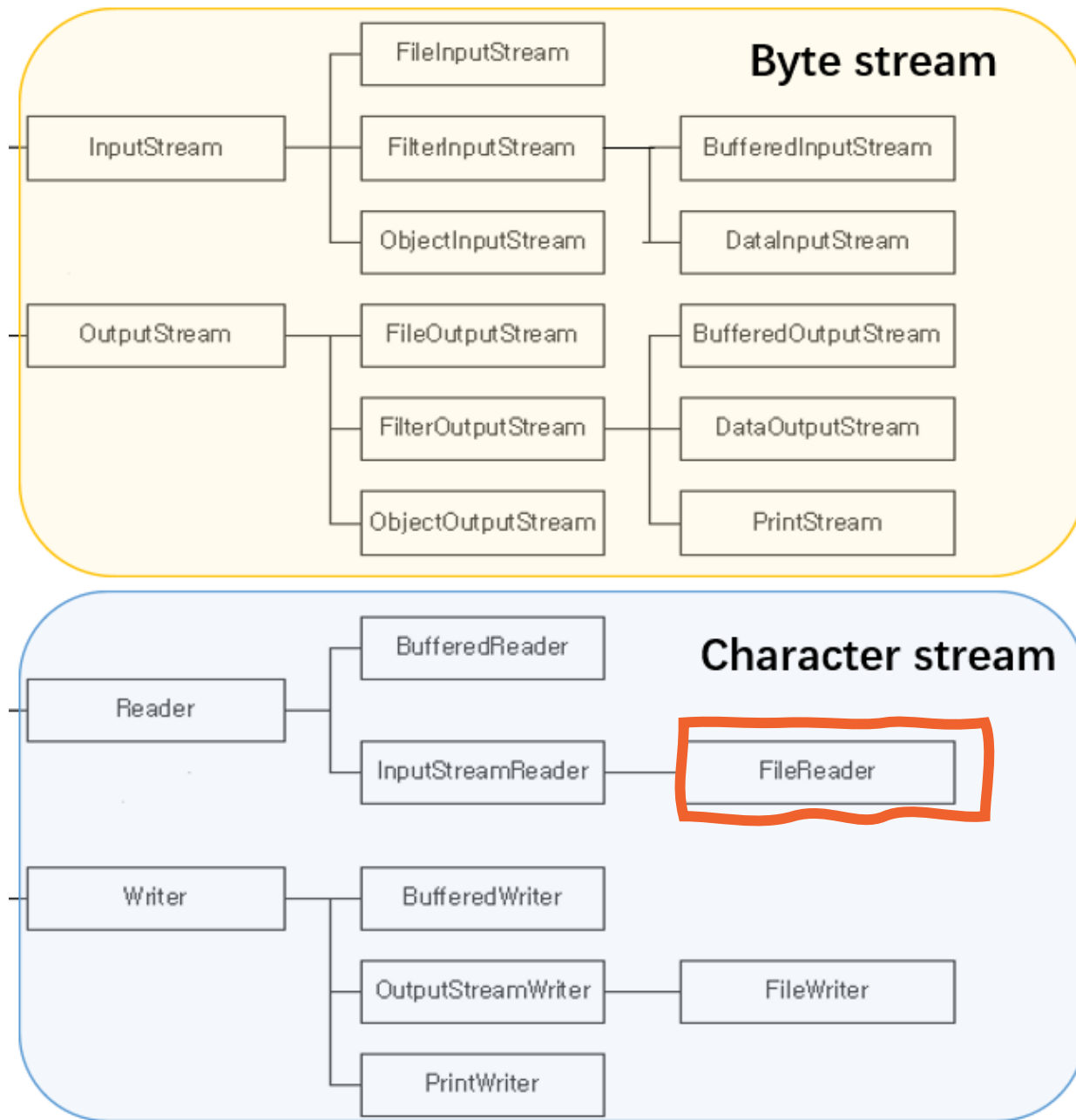
```
try (InputStream input = new FileInputStream("src/test.txt")) {  
    int n;  
    while ((n = input.read()) != -1) {  
        System.out.print((char)n);  
    }  
}
```

# FileReader

Used for reading streams of **characters** (instead of streams of bytes)

Q: Data is still read as streams of 0s and 1s. How to decide the corresponding character?

A: We need to specify an encoding scheme. If not specified, use the default encoding scheme.







# Java Default Encoding

---

- Java system/platform default encoding
  - The default encoding when JVM starts (i.e., used when deciding bytes for a character)
  - Differs from OS and language settings (e.g., GBK on 中文操作系统)
  - Could be changed (environment variable, IDE, code)
- File encoding
  - Independent from Java
  - Could be changed
  - When using Java to read a file, the Java system default encoding and the file encoding should be consistent

# Encoding Support for Java

- Every implementation of the Java platform must support the following standard charsets (see `StandardCharsets`)

Charset	Description
US-ASCII	Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
ISO-8859-1	ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8	Eight-bit UCS Transformation Format
UTF-16BE	Sixteen-bit UCS Transformation Format, big-endian byte order
UTF-16LE	Sixteen-bit UCS Transformation Format, little-endian byte order
UTF-16	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

# FileReader

- What if the input txt contains “计算机系统” and has UTF-8 file encoding? (consistent with my Java default encoding UTF-8)

```
try(Reader reader = new FileReader( fileName: "src/io/sample2.txt")){  
    int n;  
    while((n=reader.read())!=-1){  
        System.out.printf("%d %c\n", n, n);  
    }  
}
```

35745	计
31639	算
26426	机
31995	系
32479	统

Return the read char as an integer (range 0 to 65535 or  $2^{16}$ )      计: U+8BA1

# FileReader

- What if the input txt contains “计算机系统” and has GBK file encoding? (inconsistent with my Java default encoding UTF-8)

```
try(Reader reader = new FileReader( fileName: "src/io/sample2.txt")){  
    int n;  
    while((n=reader.read())!=-1){  
        System.out.printf("%d %c\n", n, n);  
    }  
}
```

```
65533 ?  
65533 ?  
65533 ?  
65533 ?  
65533 ?  
1013 €  
883 T
```

“计算机” GBK bytes are malformed UTF-8 bytes, which are replaced by default string; “系统” GBK bytes happen to be valid UTF-8 bytes, which map to different characters though.

# FileReader

- What if the input txt contains “计算机系统” and has **GBK** file encoding? (inconsistent with my Java default encoding UTF-8)

```
try(Reader reader = new FileReader( fileName: "src/io/sample3.txt" Charset.forName("gb2312"))){  
    int n;  
    while((n=reader.read())!=-1){  
        System.out.printf("%d %c\n", n, n);  
    }  
}
```

Set FileReader encoding to be consistent with the file encoding

35745	计
31639	算
26426	机
31995	系
32479	统



# InputStream to Reader

- FileReader under the hood: using FileInputStream for reading bytes, then convert them to characters based on the given encoding
- Use InputStreamReader to transform InputStream to Reader

```
// create FileInputStream
InputStream input = new FileInputStream("src/test.txt");
// convert to FileReader by specifying encoding
Reader reader = new InputStreamReader(input, "UTF-8");
```

OutputStream and Writer have the same pattern



# Lecture 5

---

- I/O Overview
- i18n & Character Encoding
- Byte Streams & Character Streams
- Combining Stream Filters
- Reading/Writing Text Input/Output
- I/O from Command Line

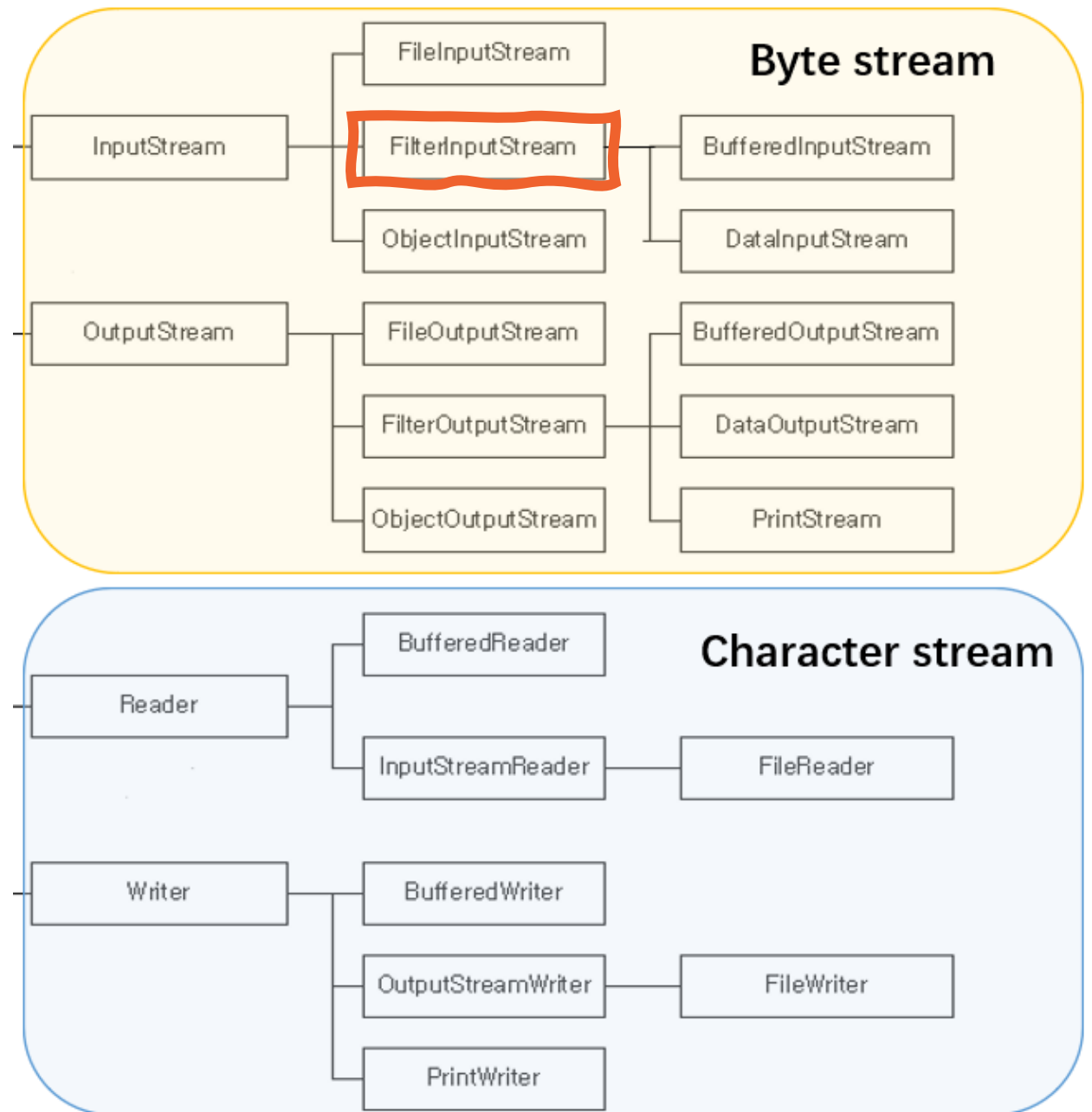
# Java I/O Streams

- Byte Stream

- *Input stream*: an object from which we can read a sequence of bytes
- *Output stream*: an object to which we can write a sequence of bytes
- Byte streams are inconvenient for processing info stored in Unicode

- Character Stream

- A separate hierarchy provides classes, inheriting from Reader and Writer, for processing Unicode characters
- These classes have read and write operations that are based on char values rather than byte values



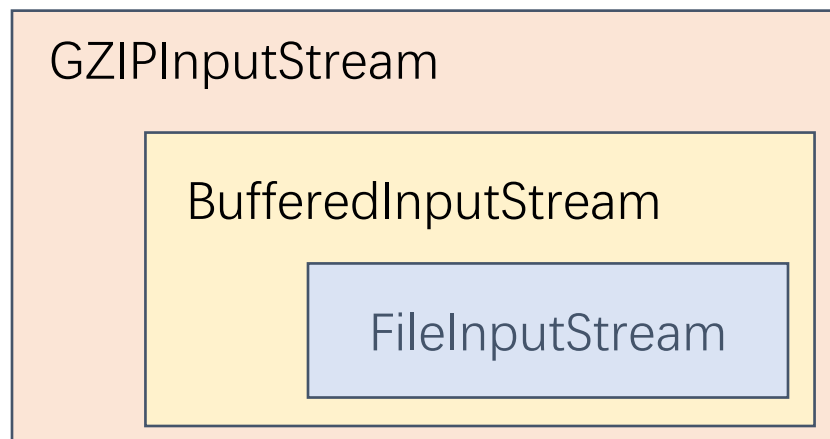
# FilterInputStream

an example of the Decorator design pattern

- Contains some other InputStream as its basic source of data
- Subclasses of FilterInputStream provide additional functionality on top of the original stream
- Direct known subclasses
  - BufferedInputStream
  - DataInputStream
  - DigestInputStream
  - InflaterInputStream
  - LineNumberInputStream
  - PushbackInputStream
  - etc.....

# FilterInputStream Example I

- + gzip functionality
- + buffered functionality
- original data



```
InputStream zfile = new  
GZIPInputStream(bfile);
```

```
InputStream bfile = new  
BufferedInputStream(file);
```

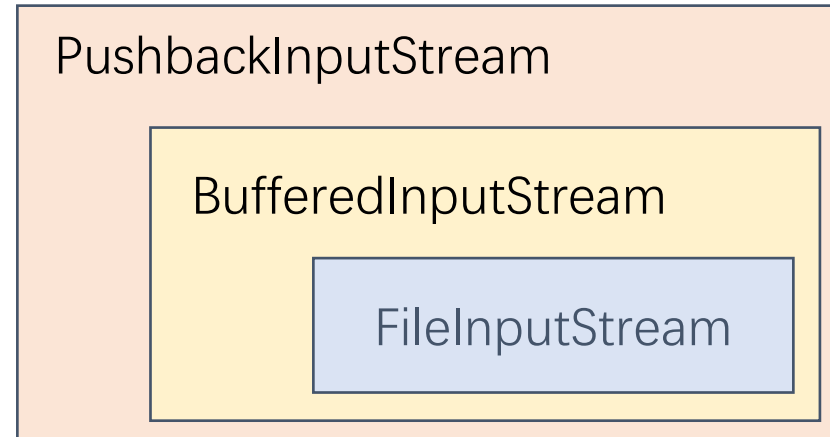
```
InputStream file = new  
FileInputStream("src/test.zip");
```

# FilterInputStream Example II

+ pushback functionality

+ buffered functionality

original data



```
PushbackInputStream pbin = new PushbackInputStream(  
    new BufferedInputStream(  
        new FileInputStream("test.txt"))));  
  
int b = pbin.read();  
pbin.unread(b); // added functionality to push back bytes
```

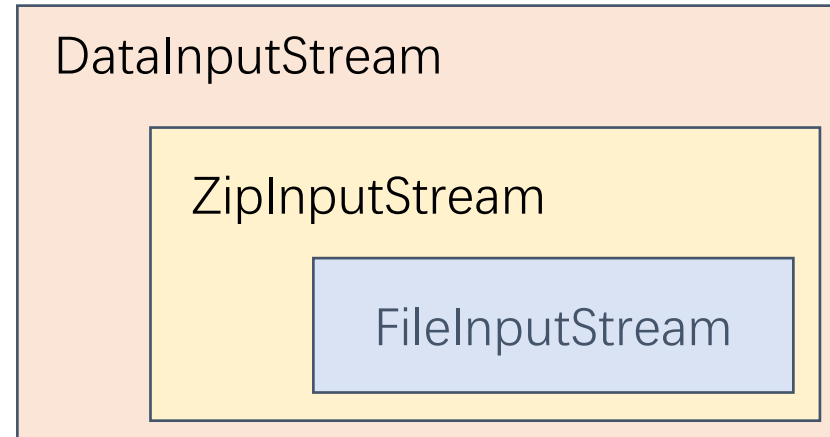


# FilterInputStream Example III

+ read-numbers functionality

+ zip functionality

original data



```
DataInputStream din=new DataInputStream(  
    new ZipInputStream(  
        new FileInputStream("test.zip")));
```

```
// added abilities to read numbers  
din.readDouble();  
din.readInt();
```

An abstract graphic on the left side of the slide, featuring concentric circles and various digital patterns like binary code and pixelated shapes in shades of blue, green, and white.

# Lecture 5

---

- I/O Overview
- Character Encoding
- Byte Streams & Character Streams
- Combining Stream Filters
- Reading/Writing Text Input/Output
- I/O from Command Line

# Reading/Writing Text Input/Output

- When working with I/O, we often work with human-readable text rather than binary data
- Java provide two APIs to assist working with text I/O
  - Scanning: useful for breaking down formatted input into tokens and translating individual tokens according to their data type (Scanner).
  - Formatting: assembles data into nicely formatted, human-readable form (PrintWriter)

# Using Scanner for reading text files

To begin, construct a `File` object with the name of the input file:

```
File inputFile = new File("input.txt");
```

Then use the `File` object to construct a `Scanner` object:

```
Scanner in = new Scanner(inputFile);
```

This `Scanner` object reads text from the file `input.txt`. You can use the `Scanner` methods (such as `nextInt`, `nextDouble`, and `next`) to read data from the input file.

For example, you can use the following loop to process numbers in the input file:

```
while (in.hasNextDouble()) {  
    double value = in.nextDouble();  
    Process value.  
}
```

# Using PrintWriter for writing text files

To write output to a file, you construct a `PrintWriter` object with the desired file name, for example

```
PrintWriter out = new PrintWriter("output.txt");
```

If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.

The `PrintWriter` class is an enhancement of the `PrintStream` class that you already know—`System.out` is a `PrintStream` object. You can use the familiar `print`, `println`, and `printf` methods with any `PrintWriter` object:

```
out.println("Hello, World!");  
out.printf("Total: %8.2f\n", total);
```



## Constructing a Scanner with a String

When you construct a `PrintWriter` with a string, it writes to a file:

```
PrintWriter out = new PrintWriter("output.txt");
```

However, this does *not* work for a `Scanner`. The statement

```
Scanner in = new Scanner("input.txt"); // Error?
```

does *not* open a file. Instead, it simply reads through the string: `in.next()` returns the string `"input.txt"`. (This is occasionally useful.)

You must simply remember to use `File` objects in the `Scanner` constructor:

```
Scanner in = new Scanner(new File("input.txt")); // OK
```



An abstract graphic on the left side of the slide, featuring concentric circles and various digital patterns like binary code and pixelated shapes in shades of blue, green, and white.

# Lecture 5

---

- I/O Overview
- Character Encoding
- Byte Streams & Character Streams
- Combining Stream Filters
- Reading/Writing Text Input/Output
- I/O from Command Line

# I/O from the Command Line

- A program is often run from the command line and interacts with the user in the command line environment
- Java supports this kind of interaction in two ways:
  - Standard Streams (often used, e.g., `System.out`):
  - Console (more advanced, e.g., `System.console()`)

# Standard Streams

- Standard streams read input from the keyboard and write output to the display.
- Java platform supports three Standard Streams

Fields		
Modifier and Type	Field and Description	
static <code>PrintStream</code>	<code>err</code> The "standard" error output stream.	<code>System.err</code>
static <code>InputStream</code>	<code>in</code> The "standard" input stream.	<code>System.in</code>
static <code>PrintStream</code>	<code>out</code> The "standard" output stream.	<code>System.out</code>

# System.in `public static final InputStream in`

- Standard input, often read keyboard input
- System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

Decorator

To Reader

InputStream

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
String str = "";  
while (!str.equals("quit")) {  
    str = br.readLine();  
    System.out.println(str);  
}
```

# System.in with Scanner

java.lang.Object  
java.util.Scanner

- Parse the input into different primitive types and strings

```
Scanner input = new Scanner(System.in);
```

```
System.out.println("Enter an int: ");  
int data = input.nextInt();  
System.out.println("Get int: " + data);
```

```
Enter an int:  
2|
```

```
System.out.println("Enter a float: ");  
float data2 = input.nextFloat();  
System.out.println("Get float: " + data2);
```

```
System.out.print("Enter a word: ");  
String value = input.next();  
System.out.println("Get word: " + value);
```

```
input.close();
```

# System.out

```
public static final PrintStream out
```

- `System.out` is defined as a `PrintStream` object.
- Although it is technically a byte stream, `PrintStream` utilizes an internal character stream object to emulate many of the features of character streams (same for `PrintWriter`)
  - `print` and `println` format individual values in a standard way.
  - `format` formats almost any number of values based on a format string, with many options for precise formatting.

<https://docs.oracle.com/javase/tutorial/essential/io/formatting.html>



# System.out

```
public static final PrintStream out
```

- Could use `setOut()` to redirect the output to other resources

```
// construct a new PrintStream with a specified file
PrintStream out = new PrintStream(new File("src/sysout.txt"));
// re-assign the standard output from console to file
System.setOut(out);
// this will be written to file
System.out.println("where am I?");
```

# System.out.println

- Performance could be affected for many println()
- All things will be printed with no filter (flooded console)
- Alternatives: logging
  - java.util.logging
  - Open-source logging framework: Log4J, SLF4J, etc.

## Log4j software bug is 'severe risk' to the entire internet

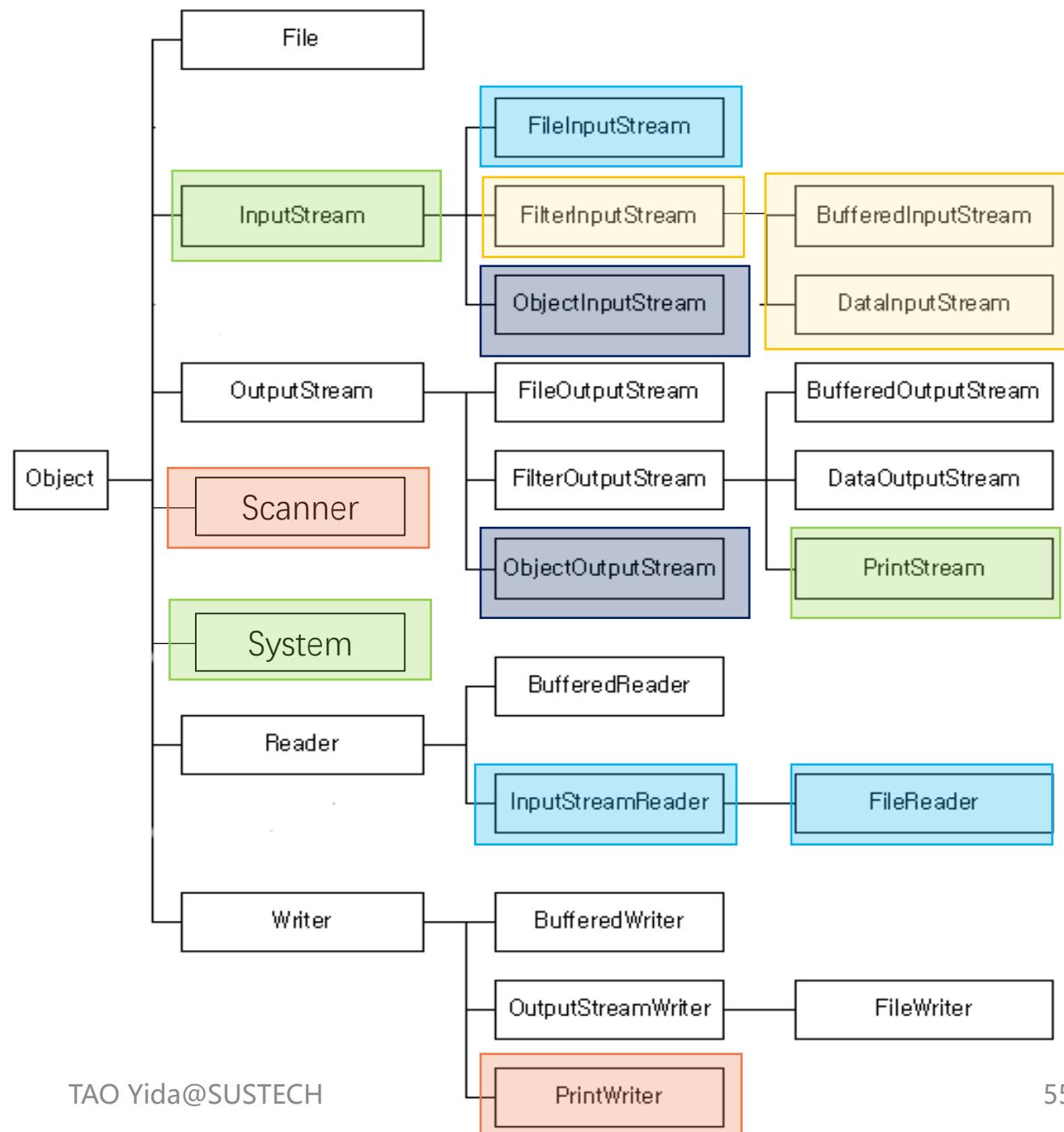
A flaw in a commonly used piece of software has left millions of web servers vulnerable to exploitation by hackers



TECHNOLOGY 13 December 2021

Level
FATAL
ERROR
WARNING
INFO
DEBUG
TRACE

# Let's Review



# Next Lecture

- Working with Files
- Serialization
- Exception Handlings