Stream I/O & Random Access Files

Streams

- The Java I/O system is built upon *streams* .
- A *stream* is an abstraction of the flow of data. An *input stream* constitutes the flow of data *to* an application, and an *output stream* represents the flow of data *from* an application.

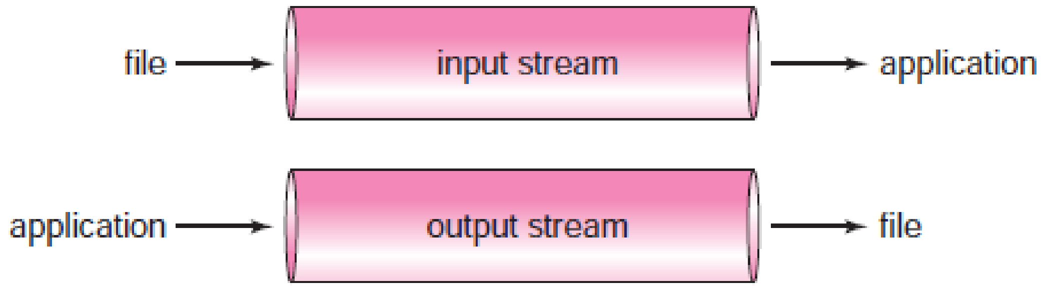


FIGURE 15.1 A *stream* is a flow of data.

Byte Stream & Character Stream Classes

- Java’s stream classes encapsulate all input and output. Java stores all data, even the most complex object, as a sequence of bytes. All objects are built from bytes. Bytes flow to and from an application via streams. Accordingly, Java provides the *Byte Stream* classes for byte I/O. The Byte Stream classes are the foundation of all Java I/O.
- Indeed, the Byte Stream classes can be used independently or as helpers for another hierarchy of I/O classes called the *Character Stream* classes. Character I/O is usually accomplished with the Character Stream classes.

Why does Java provide two separate hierarchies of stream classes?

- If all character data are composed of bytes, and I/O can be accomplished using the Byte Stream classes, why complicate matters with the Character Stream classes?
- Recall that Java stores character data using the Unicode encoding scheme, which requires *two* bytes for each character, rather than the one byte used by the ASCII code. Unicode allows Java to handle normal ASCII characters (1 byte each) as well as international character sets such as Chinese, Hebrew, or Arabic.
- Using the Character Stream classes you can process data independent of a particular character code. These classes are smart enough to automatically and invisibly handle ASCII, Unicode, or any other character code.

ASCII control characters		
00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters					
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters							
128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Continue..

- Each collection of stream classes is split into a pair of hierarchies, one for input and one for output. For the Byte Stream collection, the root classes of these two hierarchies are InputStream and OutputStream, respectively. The Reader and Writer classes fill this role for the Character Stream classes.

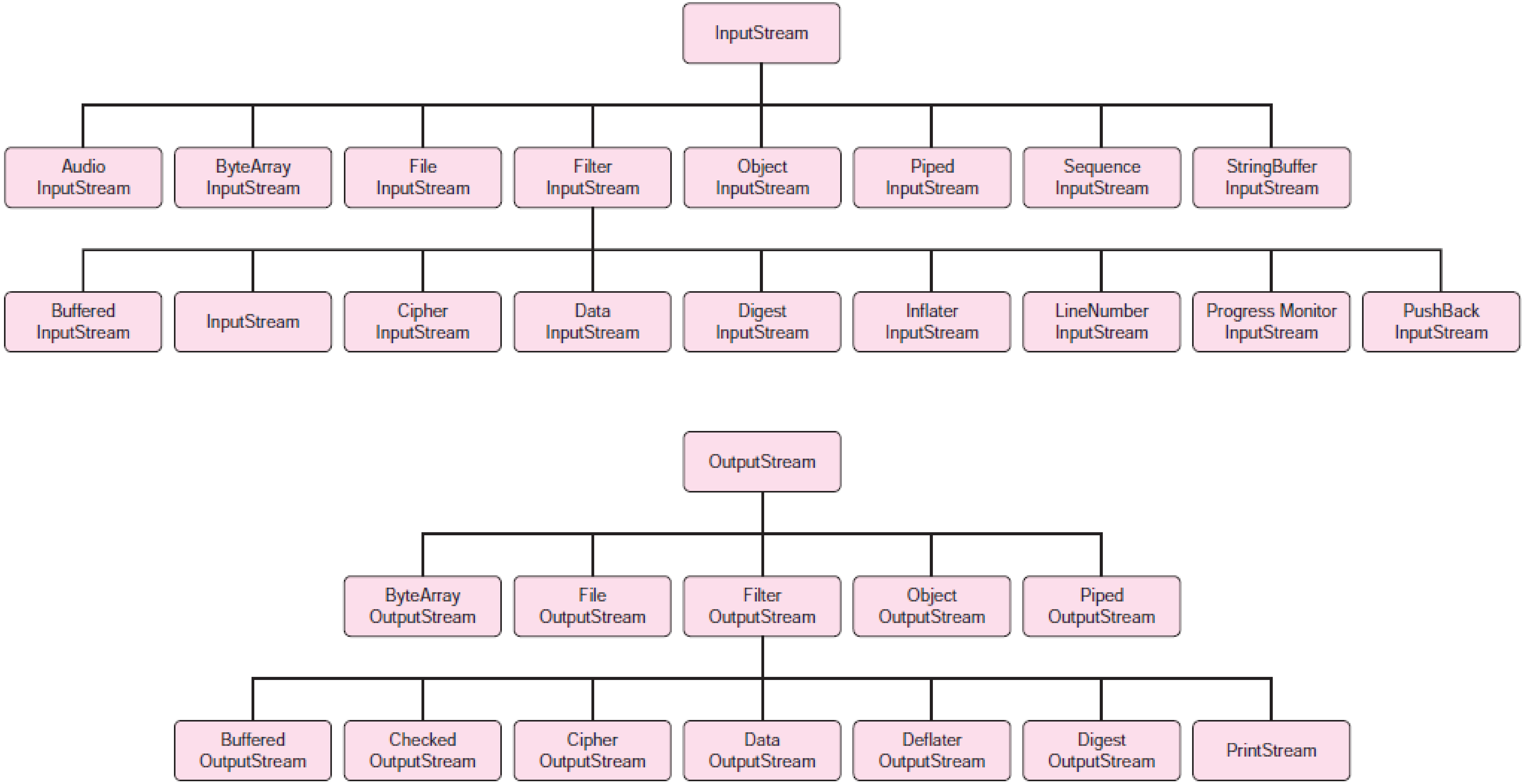


FIGURE 15.2 The *InputStream* and *OutputStream* hierarchies for byte I/O

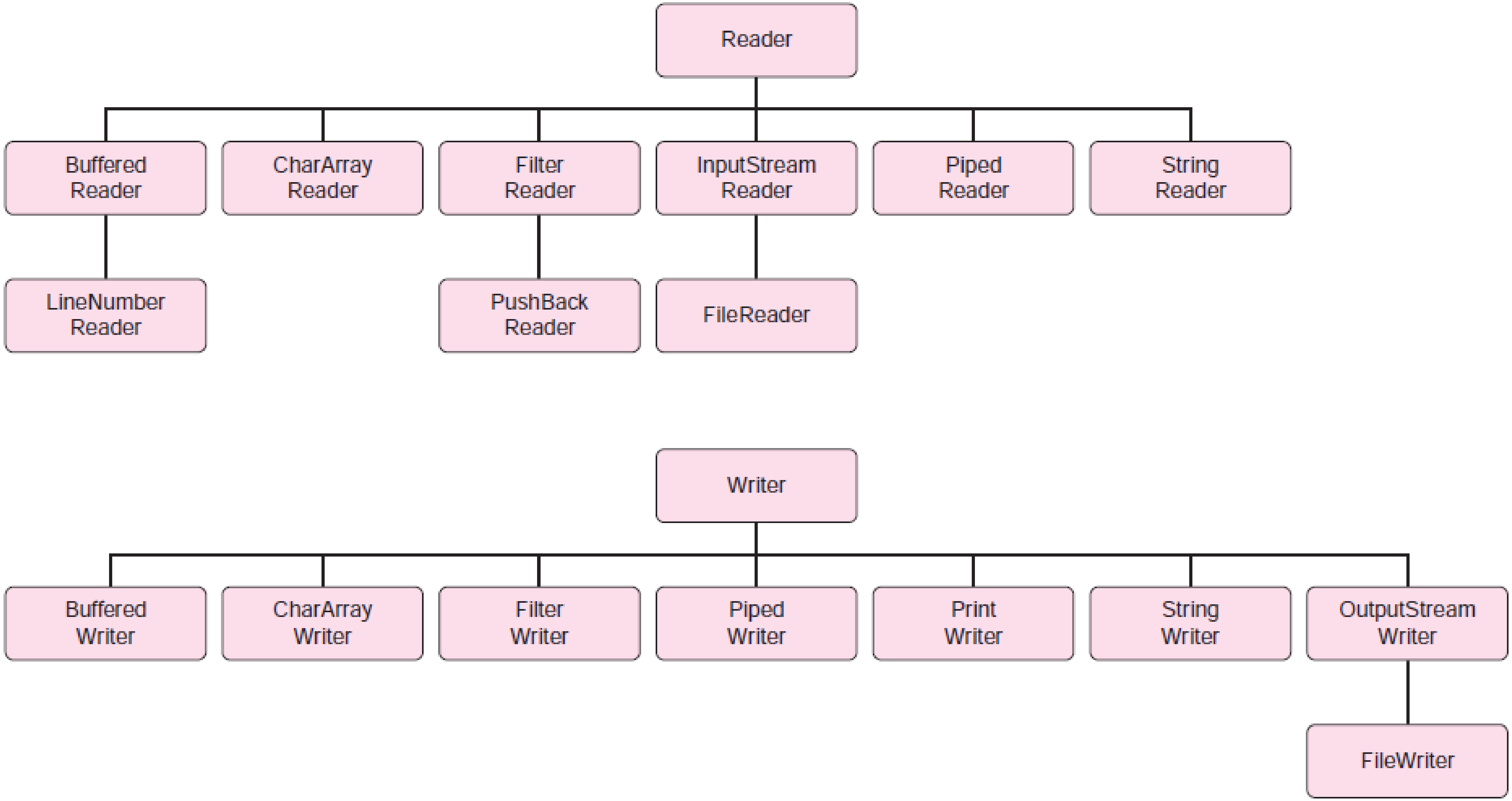


FIGURE 15.3 The *Reader* and *Writer* classes for character I/O

Byte Stream & Character Stream Classes For

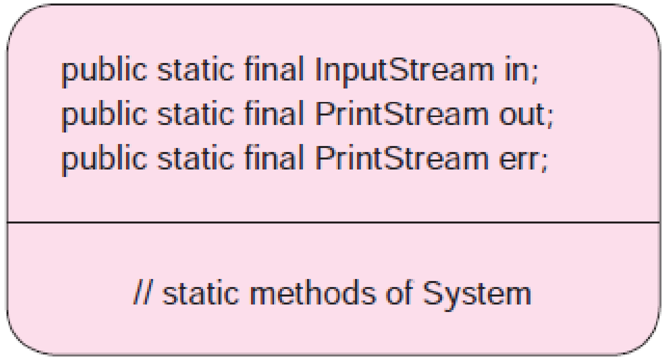
- console I/O,
- text files,
- binary files, and
- random access files

Console I/O -> Console Input

- Console Input Via ByteStream Classes
  - All console input is accomplished using System.in.
  - For example, a Scanner object that effects console I/O is connected to System.in via the constructor:
    - Scanner input = new Scanner(**System.in**);
    - public static final InputStream in;
  - This declaration states that the reference, in , refers to an InputStream object. But InputStream , a member of the Byte Stream classes, is abstract and cannot be instantiated. In fact, in is an instance of the concrete class BufferedInputStream , which extends InputStream. The declaration
    - public static final InputStream in;
  - is one more example of upcasting.

Continue..

- And what is BufferedInputStream? The BufferedInputStream class offers the capability to handle I/O efficiently.
- A *buffer* is primary memory used to temporarily store data. Using a buffer increases the efficiency and speed of I/O.
- With a buffer, data is moved in large blocks (many bytes in each block) between slower devices (like disks) and the faster buffer. A program can retrieve individual bytes more quickly from a buffer. Both the Byte Stream and Character Stream classes provide subclasses with the capability for buffered I/O.
- The relevant classes are BufferedInputStream and BufferedOutputStream for the Byte Stream classes, and BufferedReader and BufferedWriter for the Character Stream classes.



```
public static final InputStream in;
public static final PrintStream out;
public static final PrintStream err;

// static methods of System
```

FIGURE 15.4 The fields of *System*. All are *static*.

Some of the methods declared in InputStream , inherited by BufferedInputStream , and thus available to the object System.in , include:

- int read() throws IOException
  - returns the next byte in the stream (an int in the range 0..127)
  - returns –1 at the end of the stream
- int read(byte[ ] b) throws IOException
  - reads up to b.length bytes
  - returns the number of bytes read, or –1 at the end of the stream
- void close() throws IOException
  - closes the stream
- int available() throws IOException
  - returns the number of bytes that can be read (or skipped over) from this input stream without waiting. If another method tries to read from the input stream, then other methods are *blocked* temporarily and must wait.
- long skip(long n) throws IOException
  - skips n bytes in the stream before the next read

Console I/O -> Console Input

- Console Input via Character Stream Classes
  - In contrast to the Byte Stream classes, the Character Stream methods are *character* oriented.
  - A call to read() via a Character Stream object returns a Unicode character code (two bytes). Consequently, the Character Stream classes can read and write many international character sets such as Chinese, Arabic, or Hebrew.
  - Not all programs, however, process characters using two bytes. When you type characters at the terminal, your operating system encodes the characters using just eight bits, a 0 followed by a 7-bit ASCII code, giving 128 possibilities. Moreover, a simple text editor such as Notepad stores and interprets each character using just eight bits.
  - Fortunately and conveniently, the Character Stream classes are smart enough to invisibly adapt to an 8-bit scheme.

Continue..

- The BufferedReader class (in the Character Stream classes), similar in purpose to the BufferedInputStream class (in the Byte Stream classes), is used to accomplish efficient character input via the methods:
  - int read() throws IOException
    - reads a single character and returns its code number, and
  - String readLine() throws IOException
    - reads a line of text and returns the line as a String.

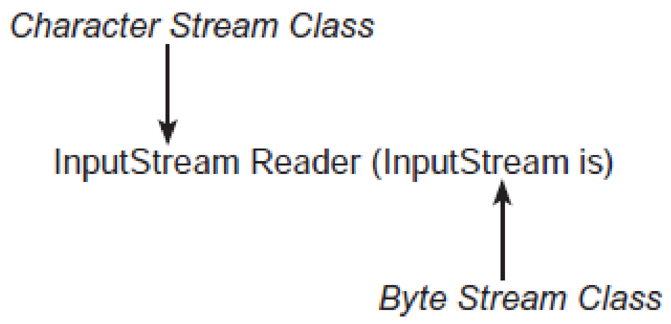
- Character Stream classes do not work independently of the Byte Stream classes.

Continue..

- The class constructor is
  - BufferedReader(Reader in)
- A BufferedReader object uses System.in, an object from the Byte Stream hierarchy, to accomplish console input. In fact, System.in is the workhorse of *all* console input. Being an InputStream object, System.in is *byte* —not character—oriented.
  - new Scanner (System.in) **//** No problem here
  - new BufferedReader(System.in) **//** BUT THIS DOES NOT WORK
- Unfortunately, this does not work. A problem occurs because the BufferedReader constructor is of the form
  - BufferedReader (Reader in)

Continue..

- To overcome this little difficulty, Java provides a link or bridge between the Character Stream classes and the Byte Stream classes. This bridge is InputStreamReader. As the name suggests, an object belonging to InputStreamReader (a Character Stream class) reads bytes and converts those bytes to characters.
- One of the constructors for an InputStreamReader has the form:

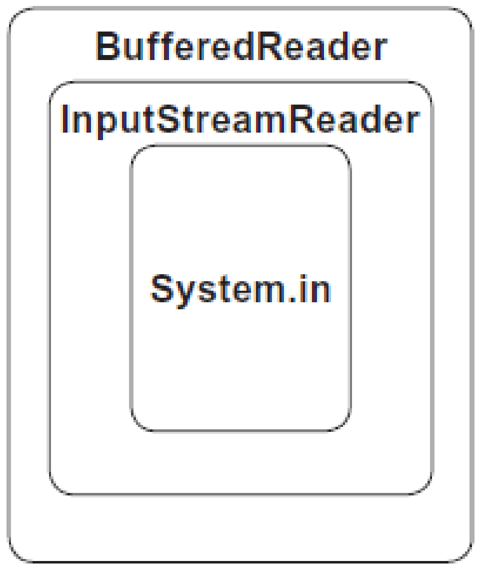


Continue..

- The Character Stream and Byte Stream classes are linked via InputStreamReader.
  - InputStreamReader link = new InputStreamReader(System.in); // link is a Reader object
  - BufferedReader br = new BufferedReader(link); // wrap a Reader with BufferedReader
- We say that the InputStreamReader , link , *wraps* System.in , and the BufferedReader , br , *wraps* link . Wrappers are a common technique in stream I/O and in object-oriented programming in general.
- Wrapping an object means that the functionality of the wrapped object is accessed via the wrapper.

Continue..

Console input using the BufferedReader class



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

     // link connects the two Stream hierarchies

BufferedReader br = new BufferedReader (link);

int c = br.read();          // character is returned as an int
String str = br.readLine();
```

FIGURE 15.5 *System.in* is wrapped in an *InputStreamReader*, which is then wrapped with a *BufferedReader*

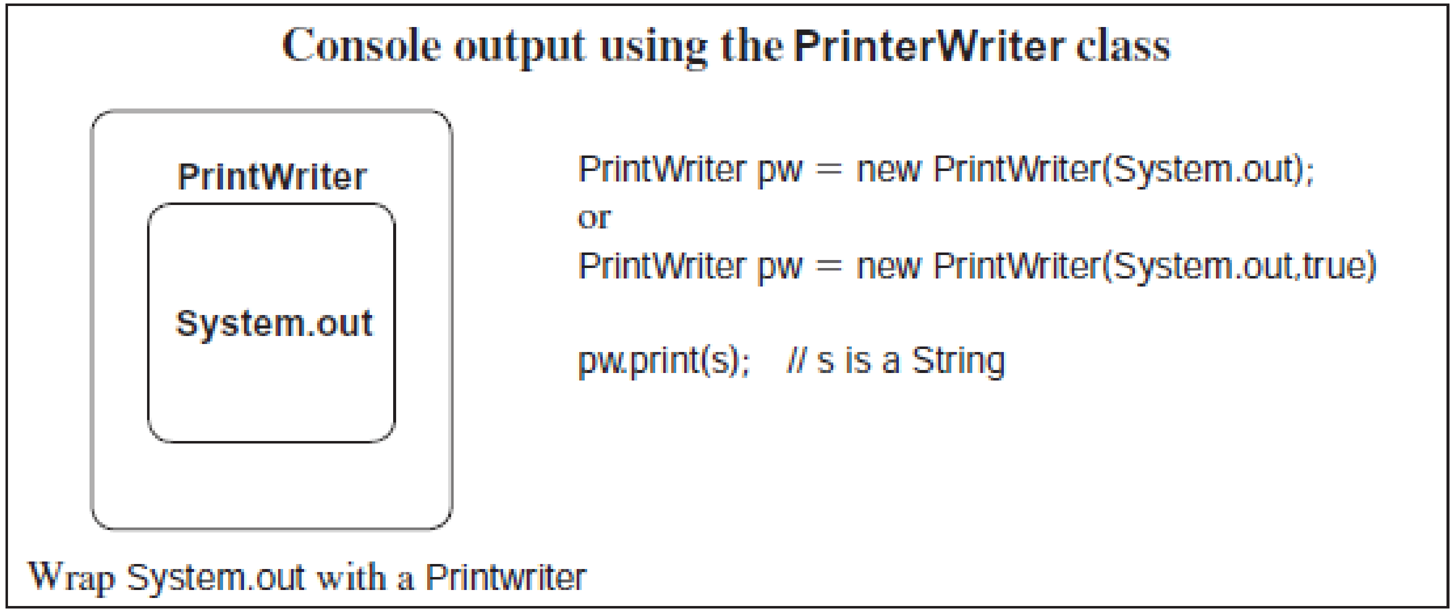
Console I/O -> Console Output

- Console Output Via ByteStream Classes
  - Console output is usually effected with the familiar System.out.print() and System.out.println() methods. In addition to in , the System class declares a field out :
    - public static final PrintStream out;
  - The methods print() and println() are defined in the PrintStream class. Since we have used these methods for all console output, no further discussion is necessary, but now you finally know what it all means.
  - void write(int b) throws IOException
    - writes a byte (an integer in the range 0..127) to the output stream
  - void write(byte[ ] b) throws IOException
    - write up to b.length bytes
  - void close() throws IOException
    - closes the stream
  - void flush() throws IOException
    - flushes the stream; write out any data remaining in a buffer

Console I/O -> Console Output

- Console Output Via Character Stream Classes
  - The PrintWriter class provides an easy mechanism for console output. Like the byte-oriented PrintStream class, PrintWriter methods include print() and println() methods. Two of the PrintWriter constructors have the following form:
    - PrintWriter(OutputStream os);
    - PrintWriter(OutputStream os, boolean flush);
  - Notice that these constructors accept a parameter belonging to OutputStream , a member of the Byte Stream hierarchy. This is in contrast to BufferedReader , which requires a parameter belonging to Reader , that is, a Character Stream reference. Consequently, PrintWriter *can* accept System.out as an argument. That's one less wrapper!
  - The second PrintWriter constructor accepts a boolean argument flush . When fl ush is set to true , automatic line flushing is enabled. This means that the stream is flushed, that is, all characters are sent to the corresponding output device whenever println() is invoked. By default, automatic line flushing is *not* enabled—a call to println(…) does not automatically print a line of text, but sends it to the stream. PrintWriter methods do not throw exceptions.

Continue..



Console output using the PrinterWriter class

```
PrintWriter pw = new PrintWriter(System.out);
or
PrintWriter pw = new PrintWriter(System.out,true)

pw.print(s);   // s is a String
```

Wrap System.out with a Printwriter

FIGURE 15.6 A *PrintWriter* object

FILES

- We define a file as a sequence of bytes and classify a file as either:
  - A text file or
  - A binary file
- A *text file* is a sequence of readable characters, that is, a file that you can create and read with a text editor.
- When you open an ASCII text file with a text editor such as Notepad, the program reads the numeric code for each character and displays the corresponding character on the screen.
- A Unicode text file encodes each character with two bytes, thus allowing many more possible character codes. In fact, the Unicode standard character set consists of more than 100,000 characters.

Continue..

- A *binary file* is *any* sequence of binary digits.
- In contrast to a text file, each byte in a binary file does not necessarily correspond to a character. An attempt to read a binary file using a text editor produces some very odd looking symbols.
- You might say that text files are readable by humans and binary files are not.
- Binary files can save space, and they facilitate specialized formatting specific to the needs of a program.
- Binary files are more efficient for both storing and manipulating *numeric* data.

Continue..

- In a text file, the symbols 123456 might be encoded as
  - 00110001 00110010 00110011 00110100
  - 00110001, the binary equivalent of 49, is the ASCII code for ‘1’;
  - 00110010, the binary equivalent of 50, is the ASCII code for ‘2’;
  - 00110011, the binary equivalent of 51, is the ASCII code for ‘3’; and
  - 00110100, the binary equivalent of 52, is the ASCII code for ‘2’;
- In a binary file, 1234 might be stored as an integer using its 32-bit binary representation of 1234:
  - 00000000 00000000 00000100 11010010

Continue..

- Both representations require four bytes of memory. However, a longer string of symbols such as “1234567890” requires 10 bytes of storage in an ASCII text file but still only four bytes as an integer in binary format. Storing large integers in a binary file rather than a text file saves space.
- Saving space is not the only advantage gained by storing numeric data in a binary file; processing time can be reduced. The CPU expects that a number has a 32-bit binary representation. If an integer such as 123456789 is stored as a sequence of nine characters (’1’,’2’,’3’,’4,’…,’9’), the character sequence must ultimately be converted to a “real” integer before any arithmetic operations can be performed, and that takes time. Furthermore, if the digits of a number are stored as characters, some type of separator, such as a space, between character sequences is required to distinguish one number from another, and these separators must also be processed.

Continue..

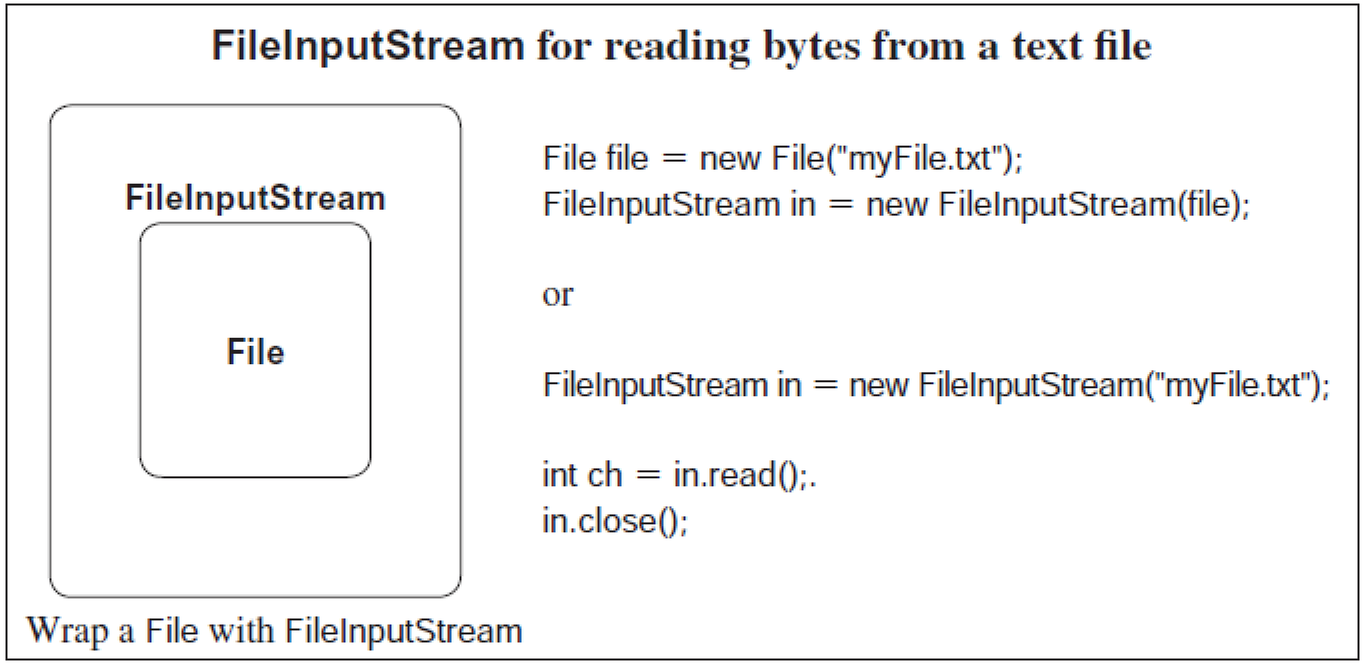
- File provides constructors and methods.
    - File name = new File(String filename);
    - The constructor throws a NullPointerException (unchecked) if file name is null.
- A few methods supplied by the File class are:
    - public boolean exists()
        - returns true if the physical file exists; otherwise returns false
    - public boolean canRead()
        - returns true if the application can, in fact, read from a file; otherwise returns false
    - public boolean canWrite()
        - returns true if an application has permission to write to a file, otherwise returns false
    - public boolean delete()
        - attempts to delete a file from the disk and returns true if operation was successful
    - public long length()
        - returns the size of the file in bytes

Continue..

- If file access is denied for any reason, each of these methods throws a SecurityException , which *is-a* RunTimeException and consequently unchecked.

Text File Input Via ByteStream Classes

- To read from a file, an application must connect a FileInputStream object to a File object, that is, wrap a File with a FileInputStream.
- Each constructor throws a FileNotFoundException if the file does not exist. If file access is denied, the constructor throws a SecurityException which *is-a* RuntimeException and hence unchecked.


FileInputStream for reading bytes from a text file
FileInputStream
File
Wrap a File with FileInputStream

```
File file = new File("myFile.txt");
FileInputStream in = new FileInputStream(file);

or

FileInputStream in = new FileInputStream("myFile.txt");

int ch = in.read();.
in.close();
```

FIGURE 15.7 A *FileInputStream* to read bytes

Continue..

- **Problem Statement:** Implement a class ShowFile with a single static utility method that reads characters from a text file, byte by byte, and displays the contents of the file on the screen. Construct a second class that demonstrates the capability of ShowFile .

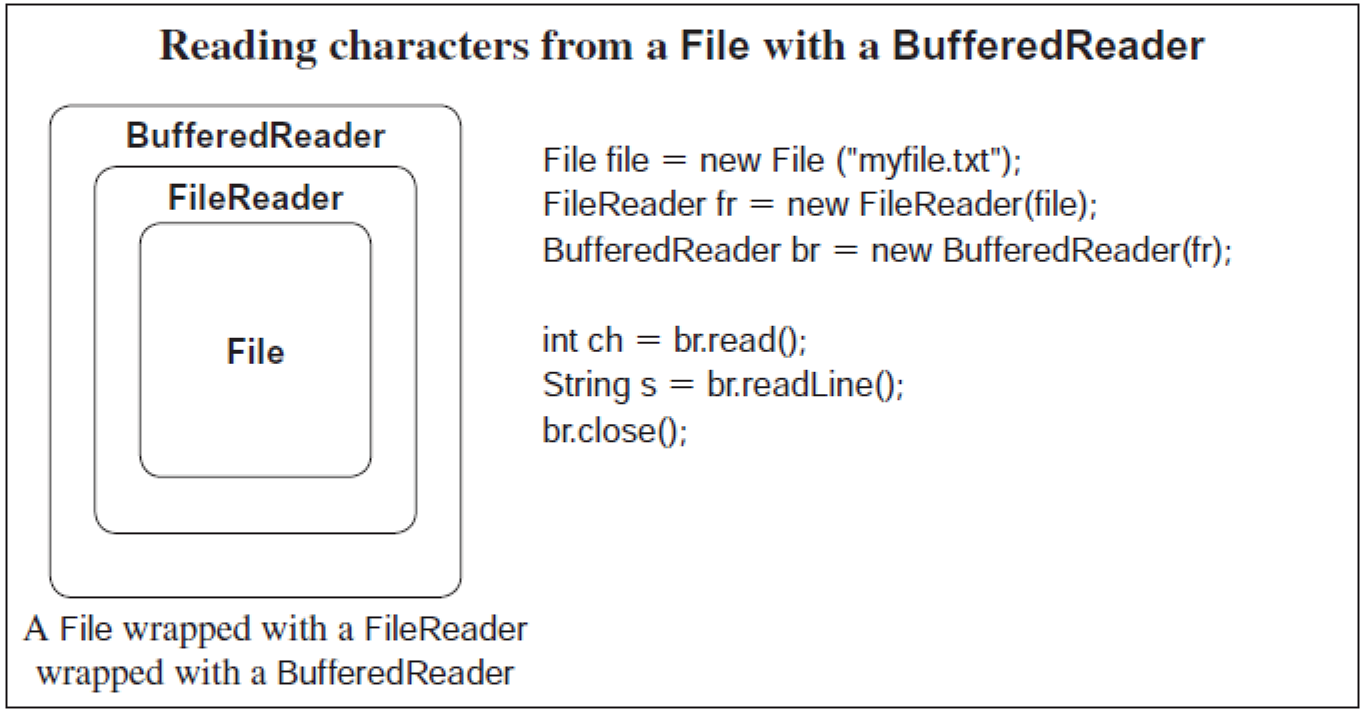
Text File Input Via CharacterStream Classes

- FileReader, a Character Stream class, includes methods that read characters from a file, one by one—in other words, very slowly. This class needs help. As you know, a *buffer* is an area of primary memory used to temporarily store data. Efficiency improves if an application reads characters from a buffer rather than directly from a disk file.
- BufferedReader provides methods that read and store a group or *block* of characters in a buffer. An application subsequently reads those characters from the buffer.

Continue..

- For example, the read() method of BufferedReader reads a single character from a buffer and not directly from a file. When read() is first invoked, a block of characters is copied from a file to a buffer. Subsequent calls to read() take characters from the buffer. When the characters stored in the buffer are consumed, the next call to read() brings another block of characters into the buffer. By reading a block of characters into a buffer, disk access is minimized and program efficiency improves. For example, using a block size of 100 bytes, an application can read 1000 bytes from a file with just 10 disk accesses. This is much faster than accessing the disk 1000 times and reading data one byte each time.

Continue..


Reading characters from a File with a BufferedReader
BufferedReader
FileReader
File
A File wrapped with a FileReader wrapped with a BufferedReader

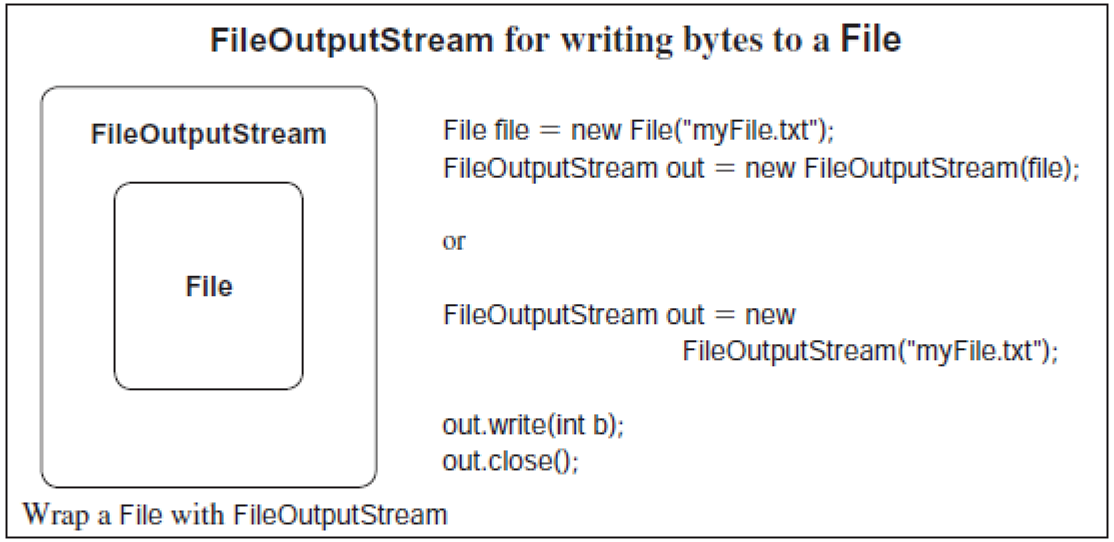
```
File file = new File ("myfile.txt");
FileReader fr = new FileReader(file);
BufferedReader br = new BufferedReader(fr);

int ch = br.read();
String s = br.readLine();
br.close();
```

FIGURE 15.8 A *BufferedReader* wrapped around a *FileReader*

Text File Output Via ByteStream Classes

- Writing to a text file using one of the Byte Stream classes is no more difficult than reading from a text file. To send output to a file:
    - wrap a File with a FileOutputStream, a Byte Stream class,
    - use the write() method of FileOutputStream , and
    - close the stream.
- The constructor FileOutputStream(File file) throws a FileNotFoundException.
- Some Useful Methods
    - void write(int b) throws IOException
        - writes a single byte
    - void close() throws IOException
        - flushes and closes the stream, and
    - void flush() throws IOException
        - Forces the data in the stream to be written to the appropriate file.

Continue..


FileOutputStream for writing bytes to a File
FileOutputStream
File
Wrap a File with FileOutputStream

```
File file = new File("myFile.txt");
FileOutputStream out = new FileOutputStream(file);

or

FileOutputStream out = new
                        FileOutputStream("myFile.txt");

out.write(int b);
out.close();
```

FIGURE 15.9 A *FileOutputStream* object

Text file output with the Byte Stream classes can be made more efficient by wrapping a BufferedOutputStream around a FileOutputStream:

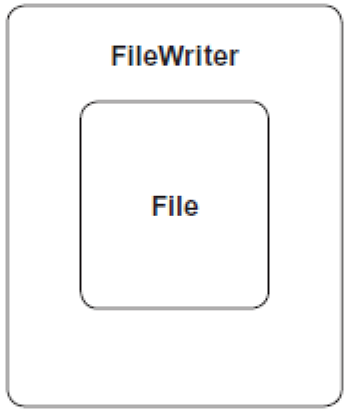
```
File output = new File ("myOutput.txt");
FileOutputStream out = new FileOutputStream(output);
BufferedOutputStream brOutput = new BufferedOutputStream(out);
```

Text File Output Via CharacterStream Classes

- The FileWriter class provides several low-level methods for writing character data to a file.
- However, because these methods do no buffering, they are rather inefficient. Indeed, these methods write just one character at a time. Therefore, FileWriter methods usually work in conjunction with other classes. So, we begin at the bottom of the food chain with the FileWriter class and work upward.
- FileWriter objects are instantiated using the constructors:
  - FileWriter(File file); throws IOException;
  - FileWriter(String filename); throws IOException;
- The FileWriter methods include:
  - void write(int ch) throws IOException;
  - void write(String s) throws IOException;
  - void close() throws IOException;
  - void flush() throws IOException;

Continue..

Filewriter does File output, but very slowly

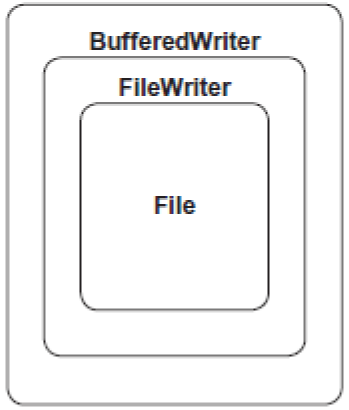


Wrap a File with a FileWriter

```
File file = new File("myfile.txt");
FileWriter fw = new FileWriter(file)

or

FileWriter fw = new FileWriter("myfile.txt");

char ch = 'a';
fw.write(ch):
fw.flush();
fw.close();
```

FIGURE 15.10 A *FileWriter* does low-level file output.

BufferedWriter wrapping a FileWriter



A File wrapped with a FileWriter wrapped with a Bufferedwriter

```
File file = new File("myfile.txt");
FileWriter = new FileWriter(file);
BufferedWriter bw = new BufferedWriter(fw);

bw.write(ch);
bw.flush();
bw.close()
```

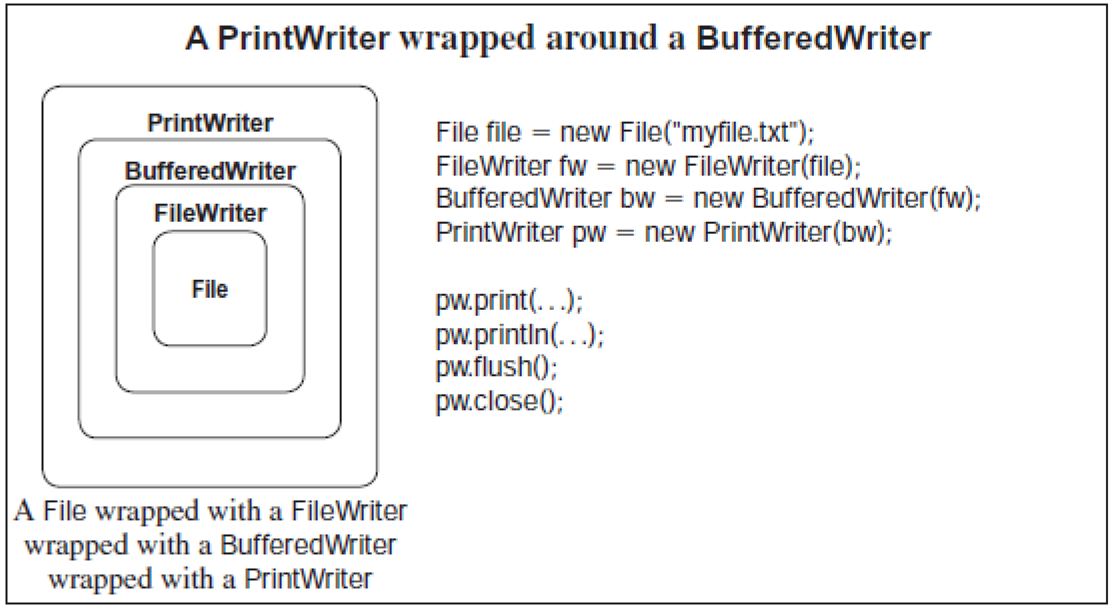
FIGURE 15.11 A *BufferedWriter* is more efficient than a *FileWriter.*

Continue..

- It is faster to write characters to a buffer and then write the contents of the buffer to a file, than it is to write each character one at a time to the file. Unlike a FileWriter, which writes characters one by one, a BufferedWriter saves characters in a buffer and writes them to a file when the buffer is full.
- A BufferedWriter can be instantiated as
  - BufferedWriter(Writer wr);
- The methods of the BufferedWriter class include:
  - void write (int ch) throws IOException;
  - void write (String s) throws IOException;
  - void close() throws IOException;
  - void flush() throws IOException;

Continue..

- The PrintWriter
  - The BufferedWriter provides efficiency and the PrintWriter class adds the familiar print() and println() methods, which facilitate *formatted* output. Two PrintWriter constructors are:
    - PrintWriter(Writer out);
    - PrintWriter(Writer out, boolean flush);


A PrintWriter wrapped around a BufferedWriter
PrintWriter
BufferedWriter
FileWriter
File
A File wrapped with a FileWriter wrapped with a BufferedWriter wrapped with a PrintWriter

```
File file = new File("myfile.txt");
FileWriter fw = new FileWriter(file);
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter pw = new PrintWriter(bw);

pw.print(...);
pw.println(...);
pw.flush();
pw.close();
```

FIGURE 15.12 The *PrintWriter* class has *print()* and *println()* methods.

Problem

- **Problem Statement:** Write a class, NumberLines , that reads lines from a text file, numbers the lines sequentially, and writes the numbered lines to a second file. The class should have two constructors:
- NumberLines() ,

  prompts for the names of the input and output files, and
- NumberLines(String inputFile, String outputFile)
  - accepts the names of the input and output files.
- Include a second class that demonstrates the NumberLines class.
- **Java Solution** The following class uses a BufferedReader object to effect reading and a PrintWriter object for output. FileNotFoundExceptions and IOExceptions are thrown to the caller. It is the caller’s responsibility to handle these exceptions with a catch block or a throws clause. The test program uses the try-catch construction.

1. Twinkle twinkle little stars
2. How I wonder what you are
3. Up above the world so high
4. Like a diamond in the sky