

File

Motivations

- Data stored in a text file is represented in human-readable form.
- Data stored in a binary file is represented in binary form. You cannot read binary files. They are designed to be read by programs.
- For example, Java source programs are stored in text files and can be read by a text editor
- But Java classes are stored in binary files and are read by the JVM.
- The advantage of binary files is that they are more efficient to process than text files.

Text File vs. Binary File

- ❑ Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters
- ❑ A binary file consists of a sequence of bits.
- ❑ For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file and
- ❑ the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7.

File

- A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.
- `File a = new File("filename");`

Text I/O

- In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file.
- This section introduces how to read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

Writing Data Using PrintWriter

java.io.PrintWriter

+PrintWriter(filename: String)
+print(s: String): void
+print(c: char): void
+print(cArray: char[]): void
+print(i: int): void
+print(l: long): void
+print(f: float): void
+print(d: double): void
+print(b: boolean): void

Also contains the overloaded
println methods.

Also contains the overloaded
printf methods.

Creates a PrintWriter for the specified file.

Writes a string.

Writes a character.

Writes an array of character.

Writes an int value.

Writes a long value.

Writes a float value.

Writes a double value.

Writes a boolean value.

A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix. The printf method was introduced in §3.6, “Formatting Console Output and Strings.”



[WriteData](#)

Run

LISTING 12.13 WriteData.java

```
1 public class WriteData {
2     public static void main(String[] args) throws IOException {
3         java.io.File file = new java.io.File("scores.txt");
4         if (file.exists()) {
5             System.out.println("File already exists");
6             System.exit(1);
7         }
8
9         // Create a file
10        java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12        // Write formatted output to the file
13        output.print("John T Smith ");
14        output.println(90);
15        output.print("Eric K Jones ");
16        output.println(85);
17
18        // Close the file
19        output.close();
20    }
21 }
```

The diagram illustrates the output of the Java program being written to a file named `scores.txt`. The file contains the following text:

John T Smith 90
Eric K Jones 85

Arrows indicate the mapping from the code to the file content:

- Line 13: `output.print("John T Smith ");` points to the first line of the file.
- Line 14: `output.println(90);` points to the number 90 on the first line of the file.
- Line 15: `output.print("Eric K Jones ");` points to the second line of the file.
- Line 16: `output.println(85);` points to the number 85 on the second line of the file.

Try-with-resources

Programmers often forget to close the file. JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {  
    Use the resource to process the file;  
}
```



[WriteDataWithAutoClose](#)

Run

LISTING 12.14 WriteDataWithAutoClose.java

```
1 public class WriteDataWithAutoClose {
2     public static void main(String[] args) throws Exception {
3         java.io.File file = new java.io.File("scores.txt");
4         if (file.exists()) {
5             System.out.println("File already exists");
6             System.exit(0);
7         }
8
9         try (
10             // Create a file
11             java.io.PrintWriter output = new java.io.PrintWriter(file);
12         ) {
13             // Write formatted output to the file
14             output.print("John T Smith ");
15             output.println(90);
16             output.print("Eric K Jones ");
17             output.println(85);
18         }
19     }
20 }
```

Reading Data Using Scanner

java.util.Scanner

+Scanner(source: File)
+Scanner(source: String)
+close()
+hasNext(): boolean
+next(): String
+nextByte(): byte
+nextShort(): short
+nextInt(): int
+nextLong(): long
+nextFloat(): float
+nextDouble(): double
+useDelimiter(pattern: String):
Scanner

Creates a Scanner object to read data from the specified file.
Creates a Scanner object to read data from the specified string.
Closes this scanner.
Returns true if this scanner has another token in its input.
Returns next token as a string.
Returns next token as a byte.
Returns next token as a short.
Returns next token as an int.
Returns next token as a long.
Returns next token as a float.
Returns next token as a double.
Sets this scanner's delimiting pattern.



[ReadData](#)

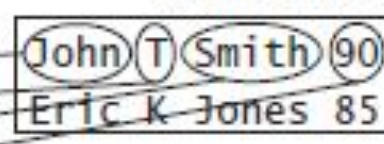
Run

LISTING 12.15 ReadData.java

```
1 import java.util.Scanner;
2
3 public class ReadData {
4     public static void main(String[] args) throws Exception {
5         // Create a File instance
6         java.io.File file = new java.io.File("scores.txt");
7
8         // Create a Scanner for the file
9         Scanner input = new Scanner(file);
10
11         // Read data from a file
12         while (input.hasNext()) {
13             String firstName = input.next();
14             String mi = input.next();
15             String lastName = input.next();
16             int score = input.nextInt();
17             System.out.println(
18                 firstName + " " + mi + " " + lastName + " " + score);
19         }
20
21         // Close the file
22         input.close();
23     }
24 }
```

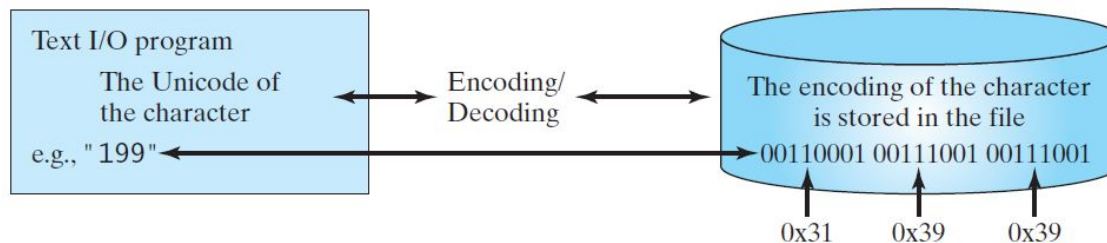
scores.txt

John	T	Smith	90
Eric	K	Jones	85

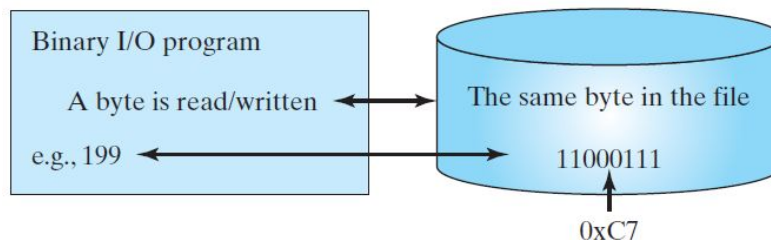


Binary I/O

Text I/O requires encoding and decoding. The JVM converts a Unicode to a file specific encoding when writing a character and converts a file specific encoding to a Unicode when reading a character. Binary I/O does not require conversions. When you write a byte to a file, the original byte is copied into the file. When you read a byte from a file, the exact byte in the file is returned.

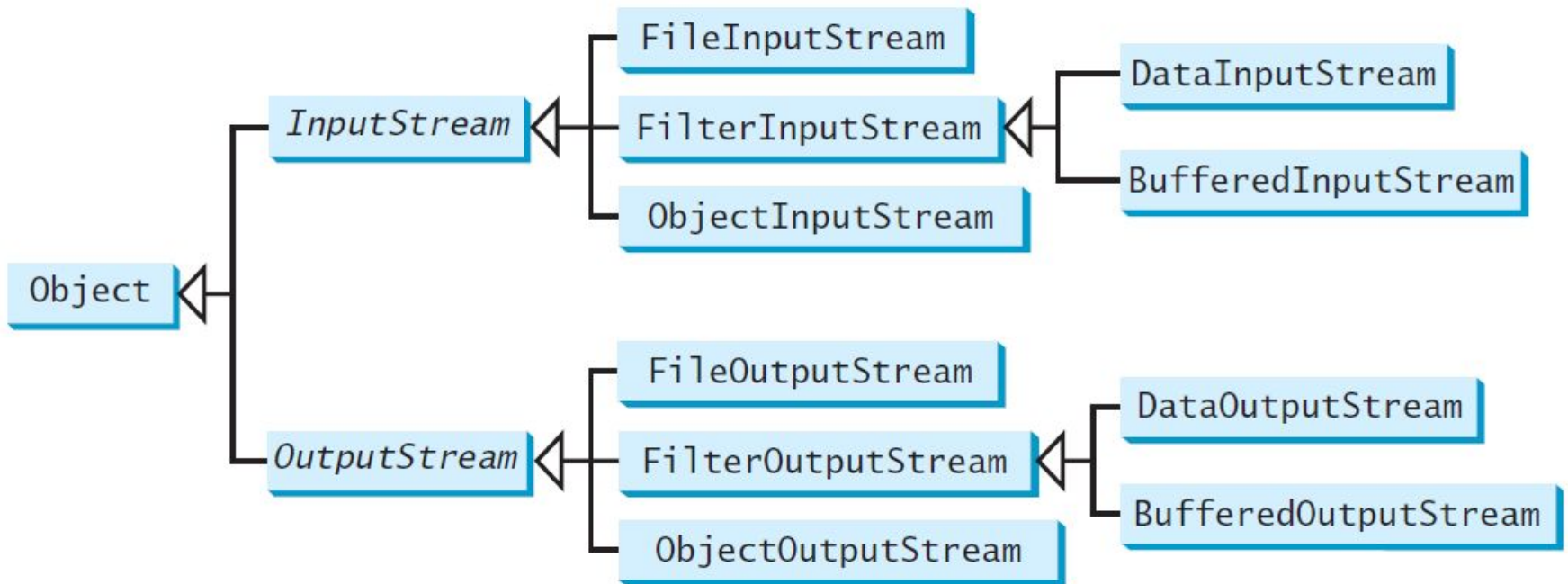


(a)



(b)

Binary I/O Classes



InputStream

The value returned is a byte as an int type.

java.io.InputStream

+*read(): int*

Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned.

+*read(b: byte[]): int*

Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.

+*read(b: byte[], off: int, len: int): int*

Reads bytes from the input stream and stores into b[off], b[off+1], ..., b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.

+*available(): int*

Returns the number of bytes that can be read from the input stream.

+*close(): void*

Closes this input stream and releases any system resources associated with the stream.

+*skip(n: long): long*

Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.

+*markSupported(): boolean*

Tests if this input stream supports the mark and reset methods.

+*mark(readlimit: int): void*

Marks the current position in this input stream.

+*reset(): void*

Repositions this stream to the position at the time the mark method was last called on this input stream.

OutputStream

The value is a byte as an int type.

java.io.OutputStream

+ *write(int b): void*

+ *write(b: byte[]): void*

+ *write(b: byte[], off: int, len: int): void*

+ *close(): void*

+ *flush(): void*

Writes the specified byte to this output stream. The parameter *b* is an int value. (byte)*b* is written to the output stream.

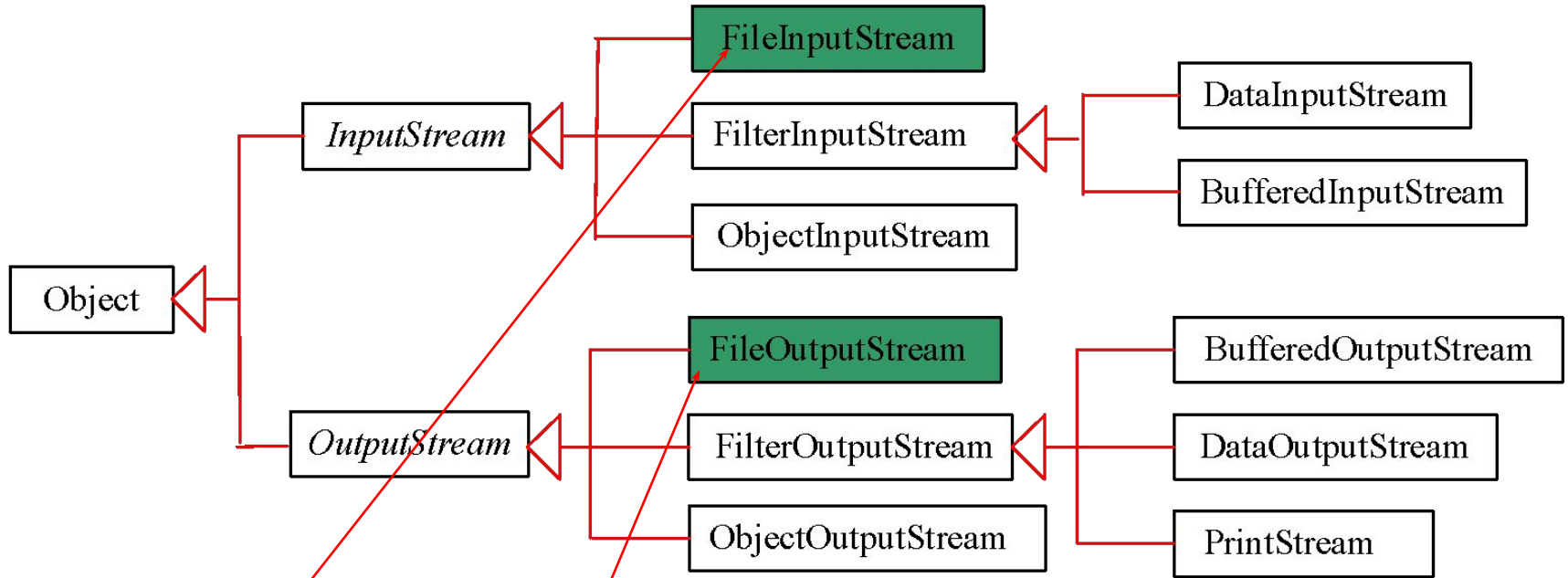
Writes all the bytes in array *b* to the output stream.

Writes *b[off]*, *b[off+1]*, ..., *b[off+len-1]* into the output stream.

Closes this output stream and releases any system resources associated with the stream.

Flushes this output stream and forces any buffered output bytes to be written out.

FileInputStream/FileOutputStream



`FileInputStream/FileOutputStream` associates a binary input/output stream with an external file. All the methods in `FileInputStream/FileOuptputStream` are inherited from its superclasses.

FileInputStream

To construct a `FileInputStream`, use the following constructors:

```
public FileInputStream(String filename)
```

```
public FileInputStream(File file)
```

A `java.io.FileNotFoundException` would occur if you attempt to create a `FileInputStream` with a nonexistent file.

FileOutputStream

To construct a FileOutputStream, use the following constructors:

```
public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename, boolean append)
public FileOutputStream(File file, boolean append)
```

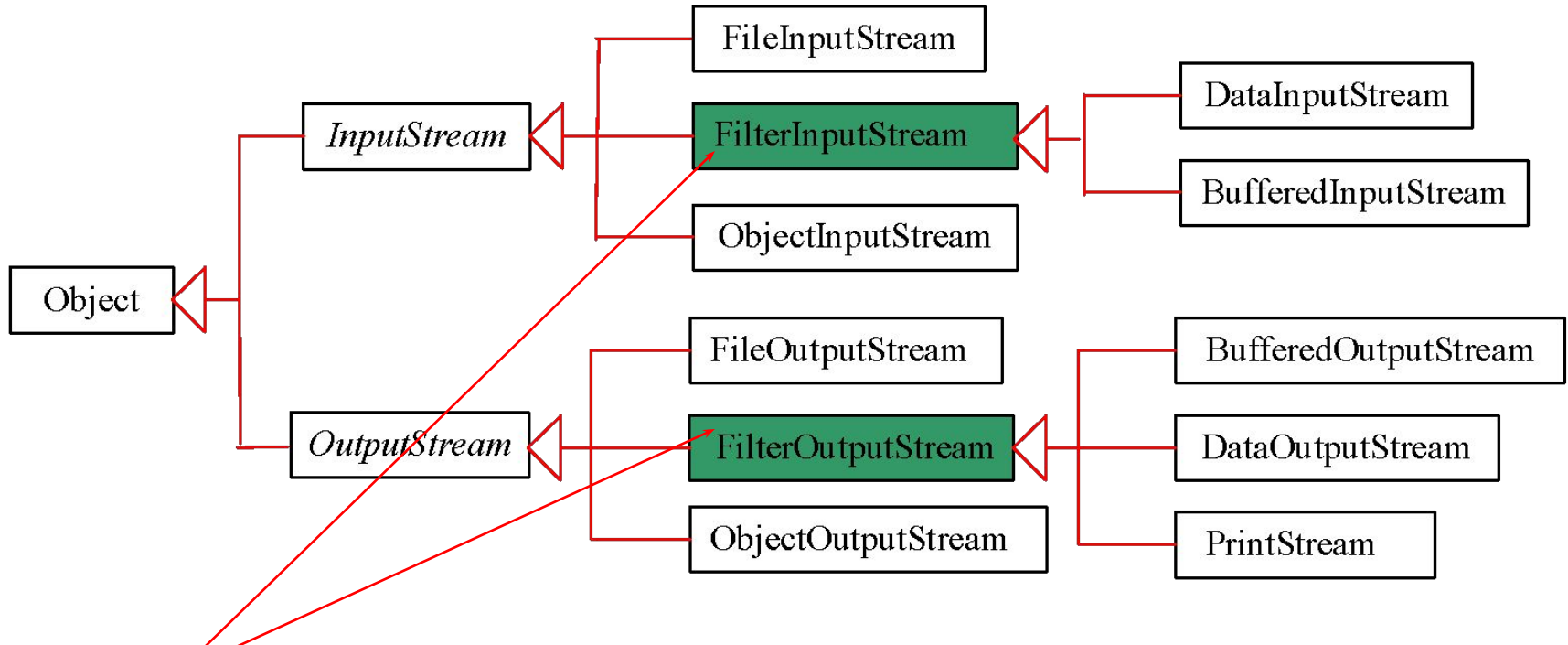
If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.



TestFileStream

Run

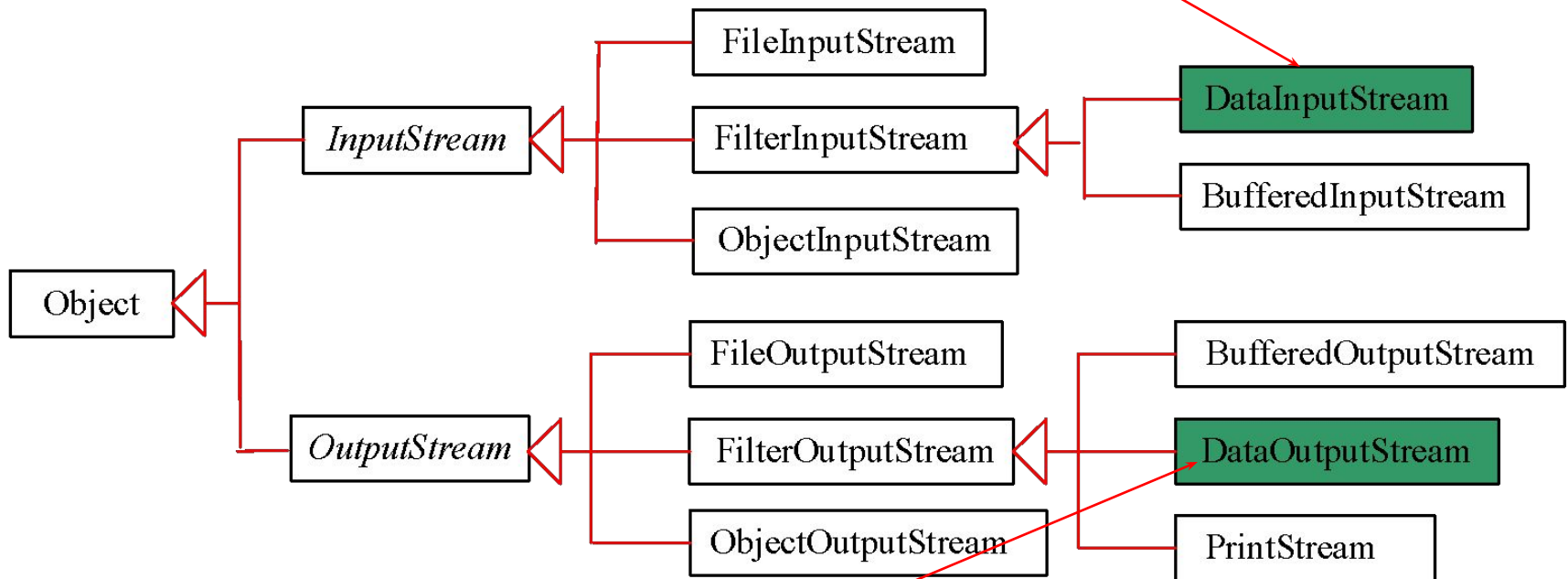
FilterInputStream/FilterOutputStream



Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. FilterInputStream and FilterOutputStream are the base classes for filtering data. When you need to process primitive numeric types, use DataInputStream and DataOutputStream to filter bytes.

DataInputStream/DataOutputStream

DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

Characters and Strings in Binary I/O

A Unicode consists of two bytes. The `writeChar(char c)` method writes the Unicode of character `c` to the output. The `writeChars(String s)` method writes the Unicode for each character in the string `s` to the output.

Why UTF-8? What is UTF-8?

UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently. Most operating systems use ASCII. Java uses Unicode. The ASCII character set is a subset of the Unicode character set. Since most applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character. The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes. ASCII values (less than 0x7F) are coded in one byte. Unicode values less than 0x7FF are coded in two bytes. Other Unicode values are coded in three bytes.

Using DataInputStream/DataOutputStream

Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

```
DataInputStream infile =
    new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =
    new DataOutputStream(new FileOutputStream("out.dat"));
```



[TestDataStream](#)

Run

LISTING 17.2 TestDataStream.java

```
1  import java.io.*;
2
3  public class TestDataStream {
4      public static void main(String[] args) throws IOException {
5          try ( // Create an output stream for file temp.dat
6              DataOutputStream output =
7                  new DataOutputStream(new FileOutputStream("temp.dat"));
8          ) {
9              // Write student test scores to the file
10             output.writeUTF("John");
11             output.writeDouble(85.5);
12             output.writeUTF("Jim");
13             output.writeDouble(185.5);
14             output.writeUTF("George");
15             output.writeDouble(105.25);
16         }
17
18         try ( // Create an input stream for file temp.dat
19             DataInputStream input =
20                 new DataInputStream(new FileInputStream("temp.dat"));
21         ) {
22             // Read student test scores from the file
23             System.out.println(input.readUTF() + " " + input.readDouble());
24             System.out.println(input.readUTF() + " " + input.readDouble());
25             System.out.println(input.readUTF() + " " + input.readDouble());
26         }
27     }
28 }
```

John 85.5
Susan 185.5
Kim 105.25

Order and Format

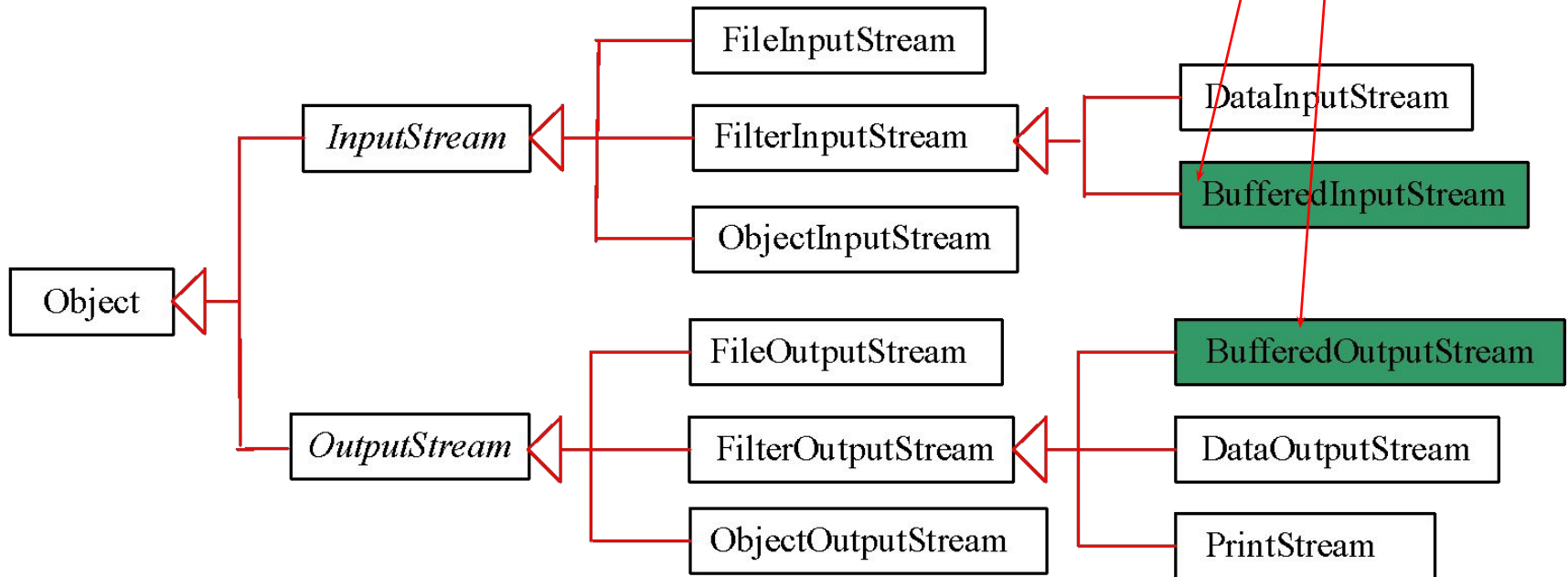
CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.

Checking End of File

TIP: If you keep reading data at the end of a stream, an EOFException would occur. So how do you check the end of a file? You can use input.available() to check it. input.available() == 0 indicates that it is the end of a file.

BufferedInputStream/ BufferedOutputStream

Using buffers to speed up I/O



BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.

Constructing BufferedInputStream/BufferedOutputStream

// Create a BufferedInputStream

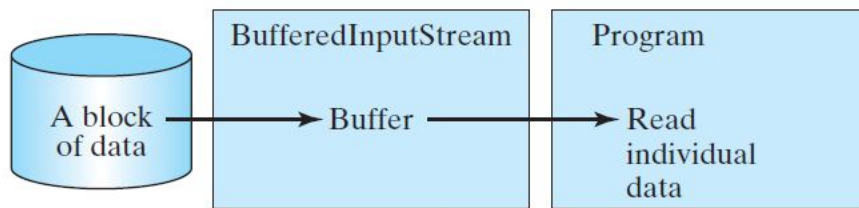
```
public BufferedInputStream(InputStream in)
```

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

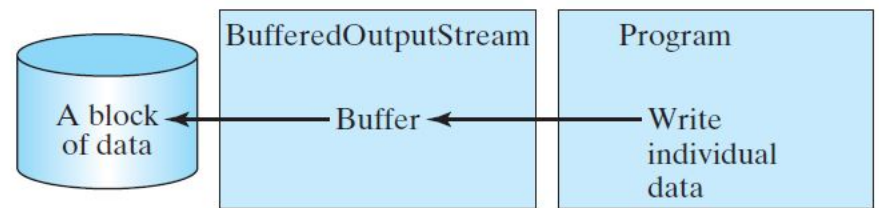
// Create a BufferedOutputStream

```
public BufferedOutputStream(OutputStream out)
```

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



(a)

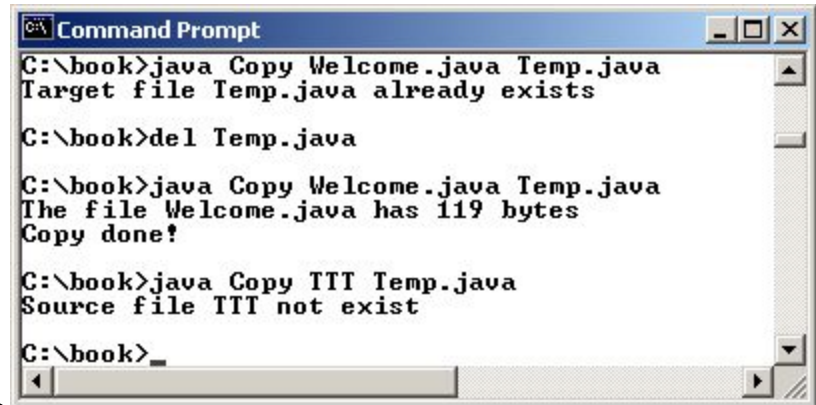


(b)

Case Studies: Copy File

This case study develops a program that copies files. The user needs to provide a source file and a target file as command-line arguments using the following command:

java Copy source target



```
Command Prompt
C:\book>java Copy Welcome.java Temp.java
Target file Temp.java already exists

C:\book>del Temp.java

C:\book>java Copy Welcome.java Temp.java
The file Welcome.java has 119 bytes
Copy done!

C:\book>java Copy TTT Temp.java
Source file TTT not exist

C:\book>
```

The program copies a source file to a target file and displays the number of bytes in the file. If the source does not exist, tell the user the file is not found. If the target file already exists, tell the user the file already exists.



Copy

Run

LISTING 17.4 Copy.java

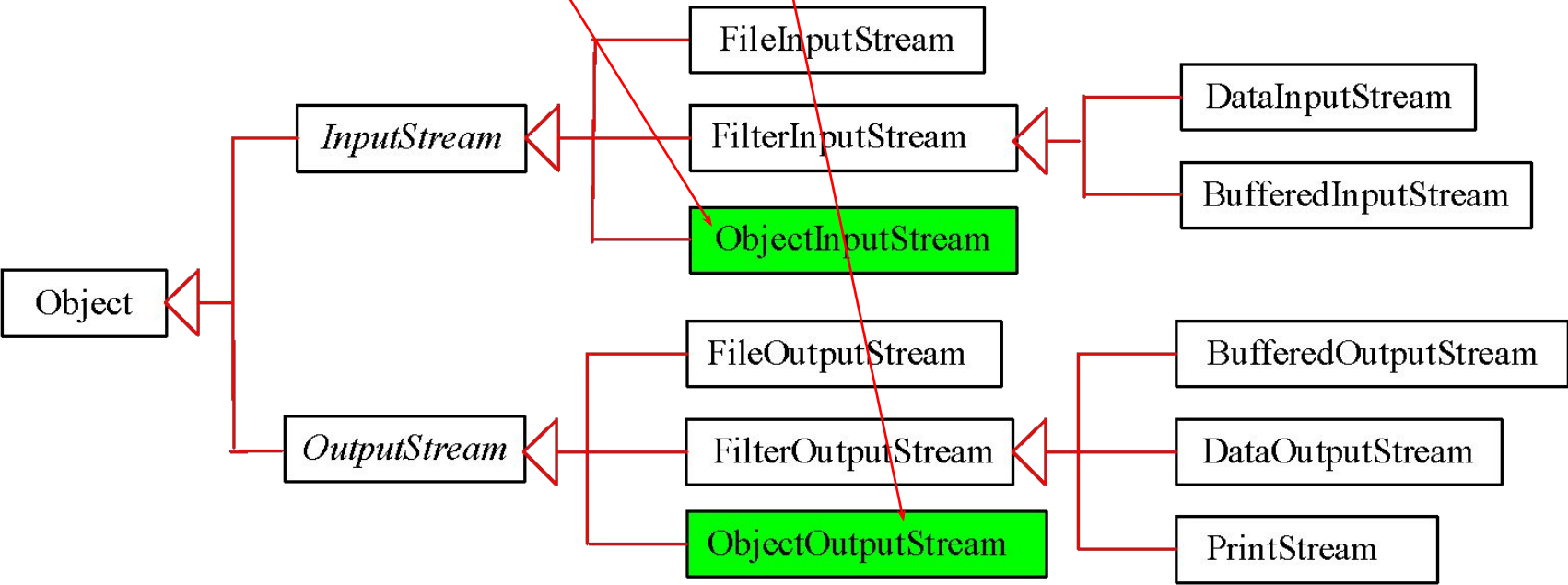
```
1  import java.io.*;
2
3  public class Copy {
4      /** Main method
5          @param args[0] for sourcefile
6          @param args[1] for target file
7      */
8      public static void main(String[] args) throws IOException {
9          // Check command-line parameter usage
10         if (args.length != 2) {
11             System.out.println(
12                 "Usage: java Copy sourceFile targetfile");
13             System.exit(1);
14         }
15
16         // Check if source file exists
17         File sourceFile = new File(args[0]);
18         if (!sourceFile.exists()) {
19             System.out.println("Source file " + args[0]
20                 + " does not exist");
21
22             System.exit(2);
23         }
24     }
25 }
```

```
24 // Check if target file exists
25 File targetFile = new File(args[1]);
26 if (targetFile.exists()) {
27     System.out.println("Target file " + args[1]
28         + " already exists");
29     System.exit(3);
30 }
31
32 try (
33     // Create an input stream
34     BufferedInputStream input =
35         new BufferedInputStream(new FileInputStream(sourceFile));
36
37     // Create an output stream
38     BufferedOutputStream output =
39         new BufferedOutputStream(new FileOutputStream(targetFile));
40 ) {
41     // Continuously read a byte from input and write it to output
42     int r, numberOfBytesCopied = 0;
43     while ((r = input.read()) != -1) {
44         output.write((byte)r);
45         numberOfBytesCopied++;
46     }
47
48     // Display the file size
49     System.out.println(numberOfBytesCopied + " bytes copied");
50 }
51 }
52 }
```

Object I/O

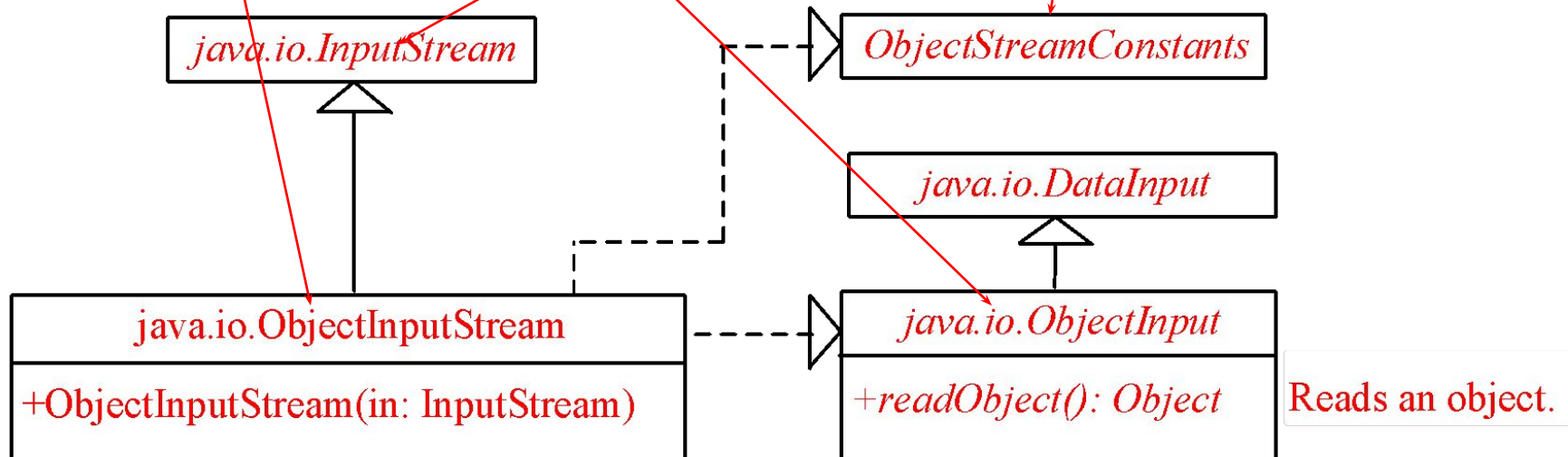
DataInputStream/DataOutputStream enables you to perform I/O for primitive type values and strings.

ObjectInputStream/ObjectOutputStream enables you to perform I/O for objects in addition for primitive type values and strings.



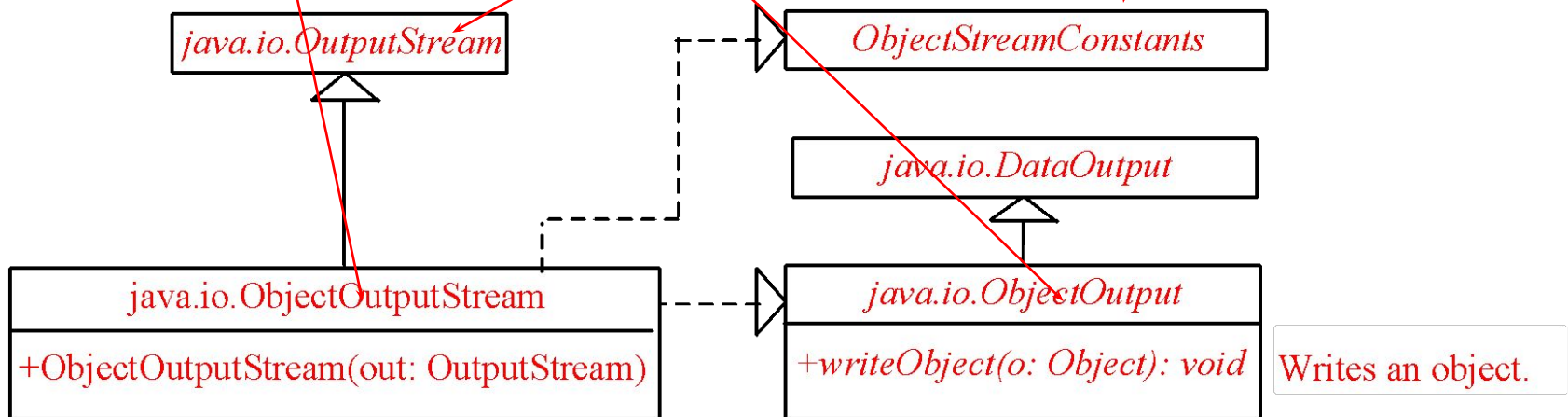
ObjectInputStream

ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.



ObjectOutputStream

ObjectOutputStream extends OutputStream and implements ObjectOutputStreamConstants.



LISTING 17.5 TestObjectOutputStream.java

```
1  import java.io.*;
2
3  public class TestObjectOutputStream {
4      public static void main(String[] args) throws IOException {
5          try ( // Create an output stream for file object.dat
6              ObjectOutputStream output =
7                  new ObjectOutputStream(new FileOutputStream("object.dat"));
8              ) {
9              // Write a string, double value, and object to the file
10             output.writeUTF("John");
11             output.writeDouble(85.5);
12             output.writeObject(new java.util.Date());
13         }
14     }
15 }
```

LISTING 17.6 TestObjectInputStream.java

```
1  import java.io.*;
2
3  public class TestObjectInputStream {
4      public static void main(String[] args)
5          throws ClassNotFoundException, IOException {
6          try ( // Create an input stream for file object.dat
7              ObjectInputStream input =
8                  new ObjectInputStream(new FileInputStream("object.dat"));
9          ) {
10             // Read a string, double value, and object from the file
11             String name = input.readUTF();
12             double score = input.readDouble();
13             java.util.Date date = (java.util.Date)(input.readObject());
14             System.out.println(name + " " + score + " " + date);
15         }
16     }
17 }
```