# Chapter 10 I/O Operation

# Objects Transmission by Streams



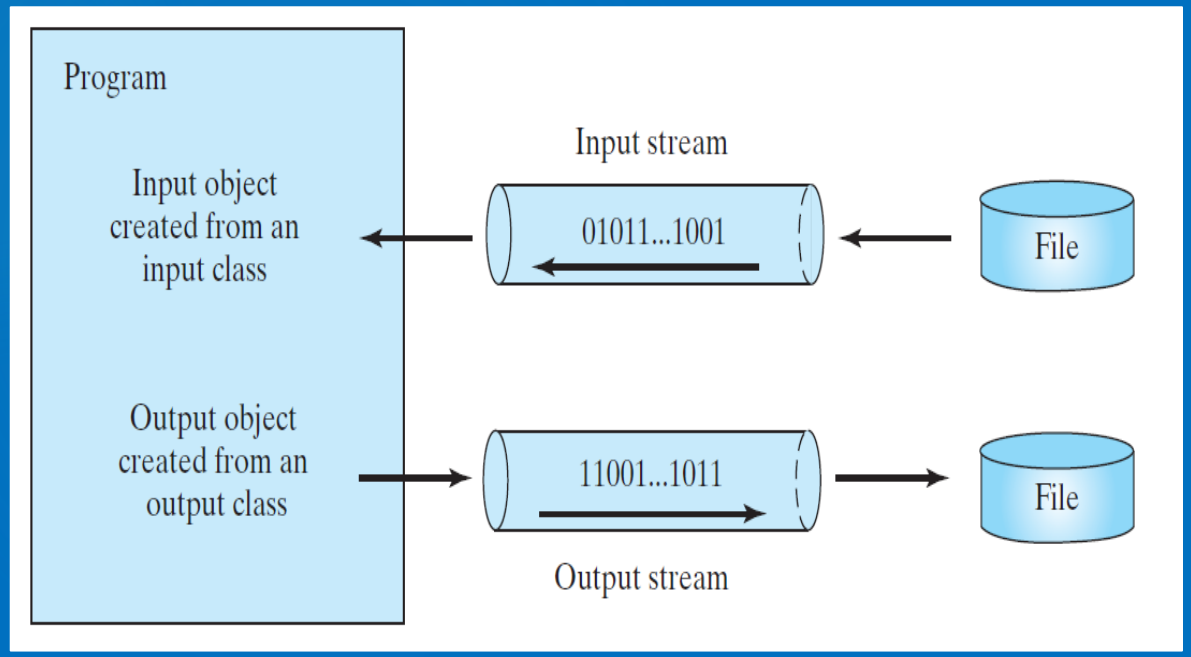
## Oil Output Stream

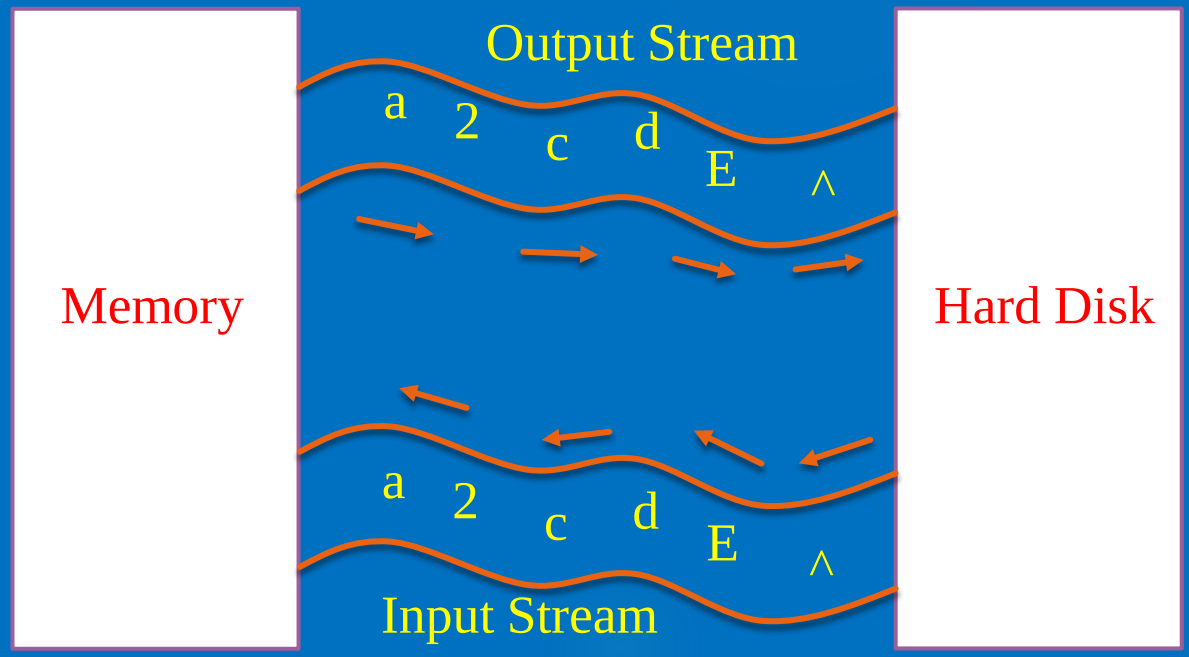
## Oil Input Stream

# How is I/O Handled in Java?



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. In order to perform I/O, you need to create *objects using appropriate Java I/O classes (InputStream and OutputStream)*.



# Text and Binary Files

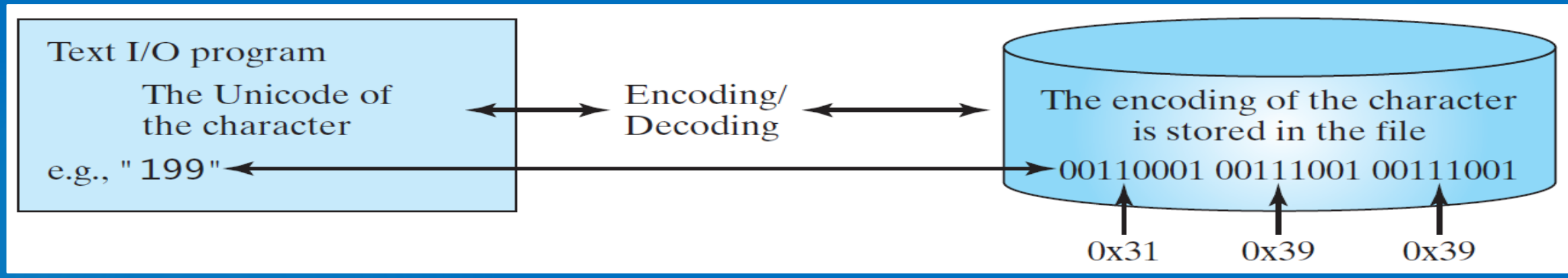


Files can be classified as either ***text*** or ***binary***. A file that can be processed (read, created, or modified) using a text editor such as Notepad on Windows or vi on UNIX is called a text file. All the other files are called binary files. You cannot read binary files using a text editor—they are designed to be read by programs. For example, *Java source* programs are *text files* and can be read by a text editor, but *Java class* files are *binary files* and are read by the JVM.

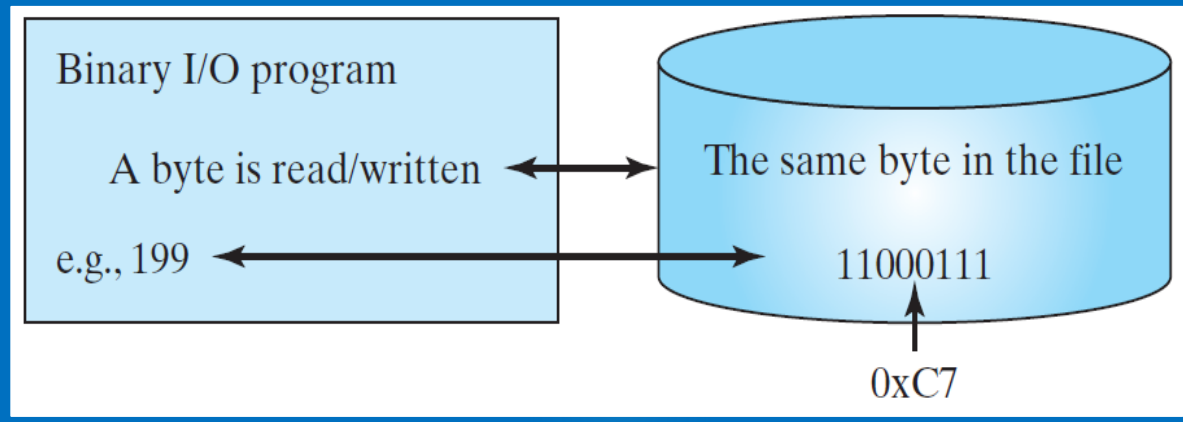
# Binary I/O vs. Text I/O



*Text I/O requires encoding and decoding*. The JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character.

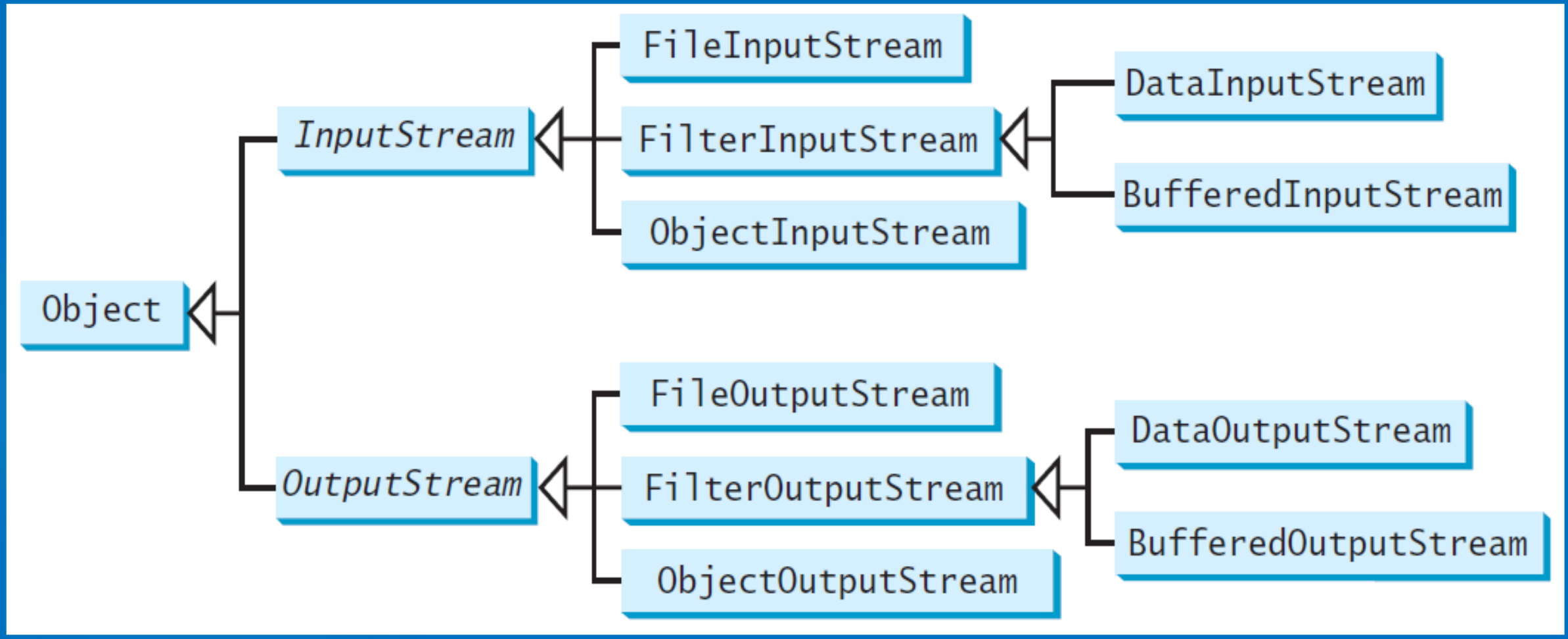



*Binary I/O does not require conversions*. When you write a *byte* to a file, the original byte is copied into the file. When you read a *byte* from a file, the exact byte in the file is returned.



# Binary I/O Classes





# The Abstract `InputStream` Class



```
public abstract class InputStream implements Closeable {}
```

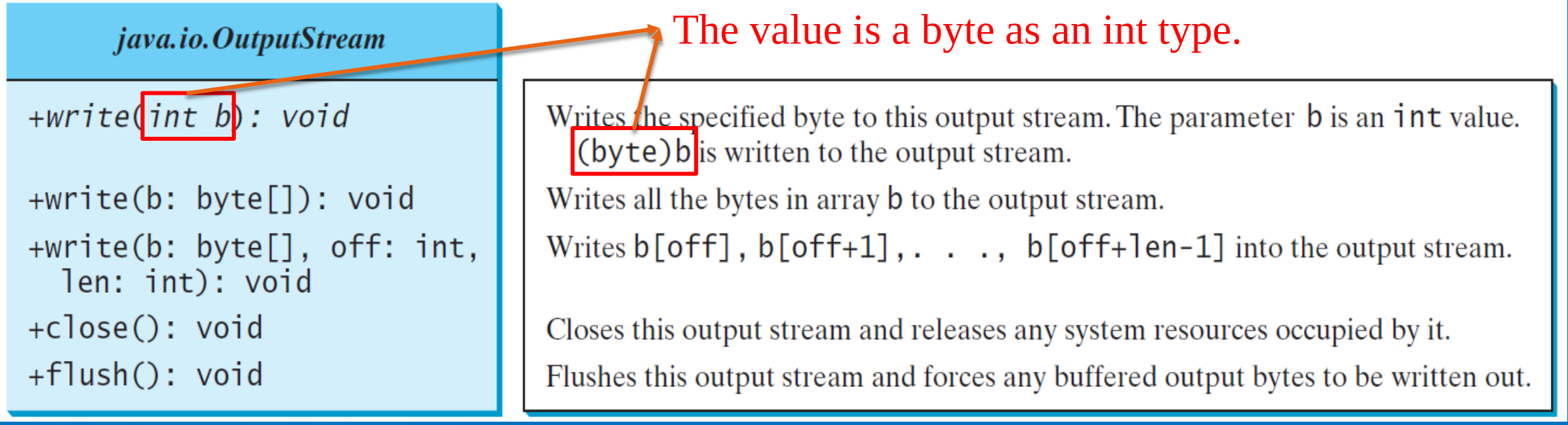
| *java.io.InputStream* | |
|---|---|
| `+read(): int` | Reads the next byte of data from the input stream. The value byte is returned as an `int` value in the range `0` to `255`. If no byte is available because the end of the stream has been reached, the value –1 is returned. |
| `+read(b: byte[]): 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 them in `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 an estimate of the number of bytes that can be read from the input stream. |
| `+close(): void` | Closes this input stream and releases any system resources occupied by it. |
| `+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 whether 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. |

The value returned is a byte as an int type.

# The Abstract `OutputStream` Class

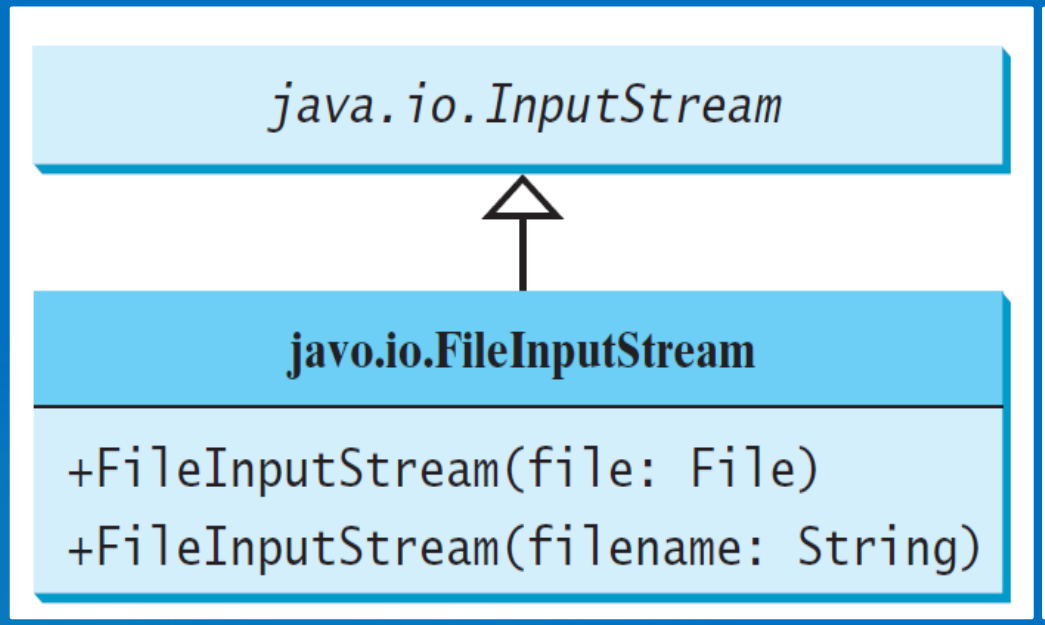


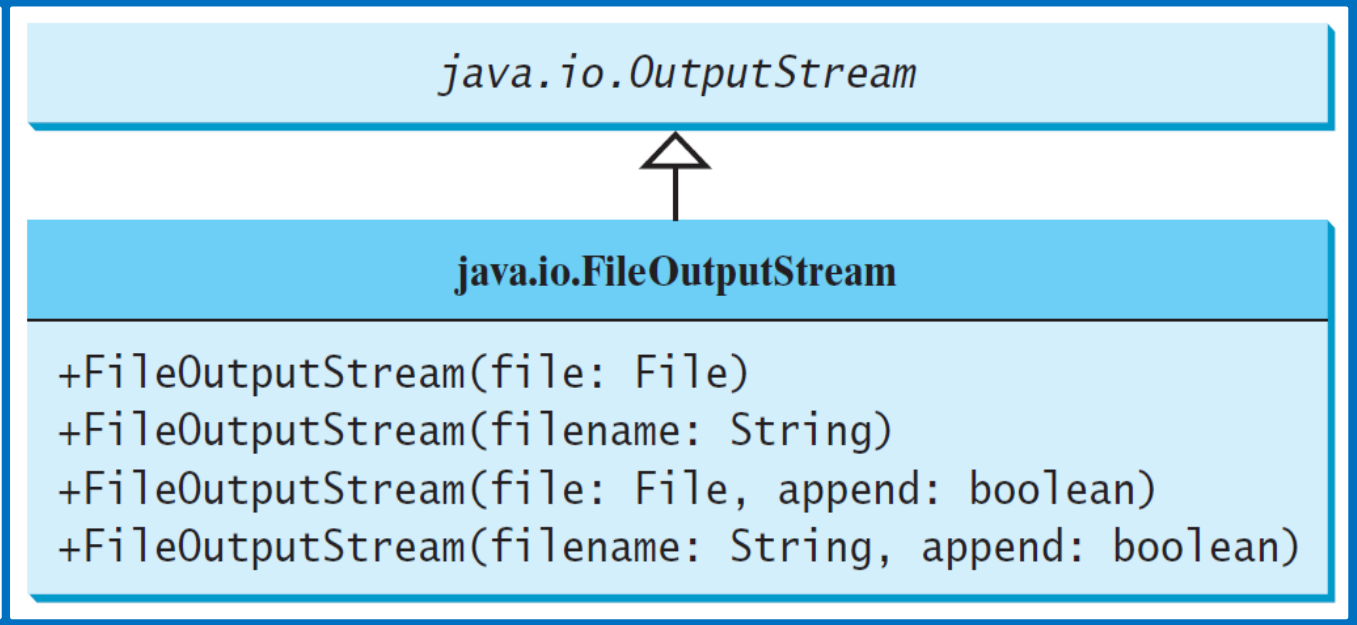
```
public abstract class OutputStream implements Closeable, Flushable {}
```



# I/O Operation to Files






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

# The FileInputStream Class



```
public class FileInputStream extends InputStream{
    public FileInputStream(String name) throws FileNotFoundException {
        this(name != null ? new File(name) : null);
    }
    public FileInputStream(File file) throws FileNotFoundException {
        String name = (file != null ? file.getPath() : null);
        //...
        if (name == null) {
            throw new NullPointerException();
        }
        if (file.isInvalid()) {
            throw new FileNotFoundException("Invalid file path");
        }
        //...
    }
}
```

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

# The `FileOutputStream` Class



To construct a `FileOutputStream`, use the following constructors.

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

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*.

# FilterInputStream/FilterOutputStream

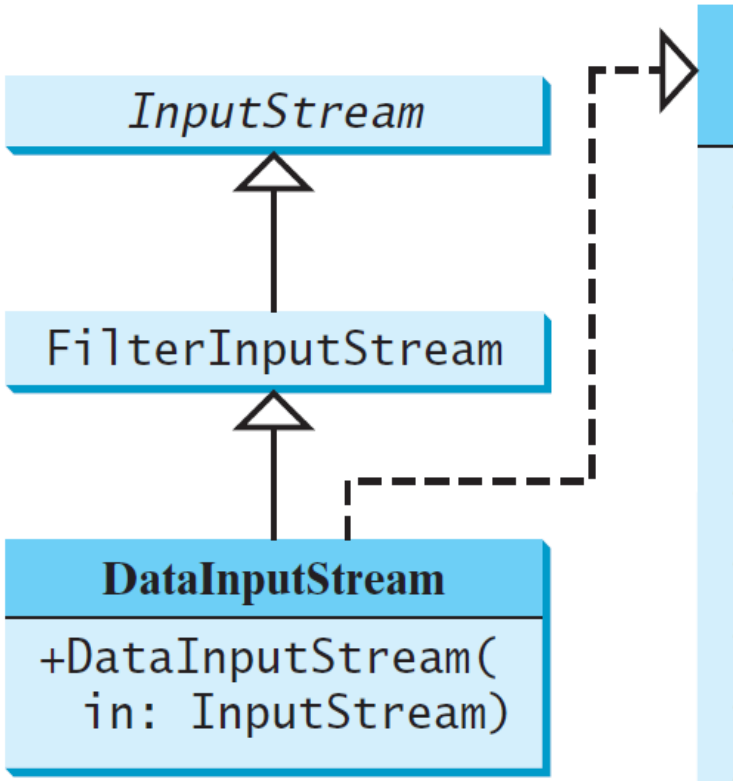


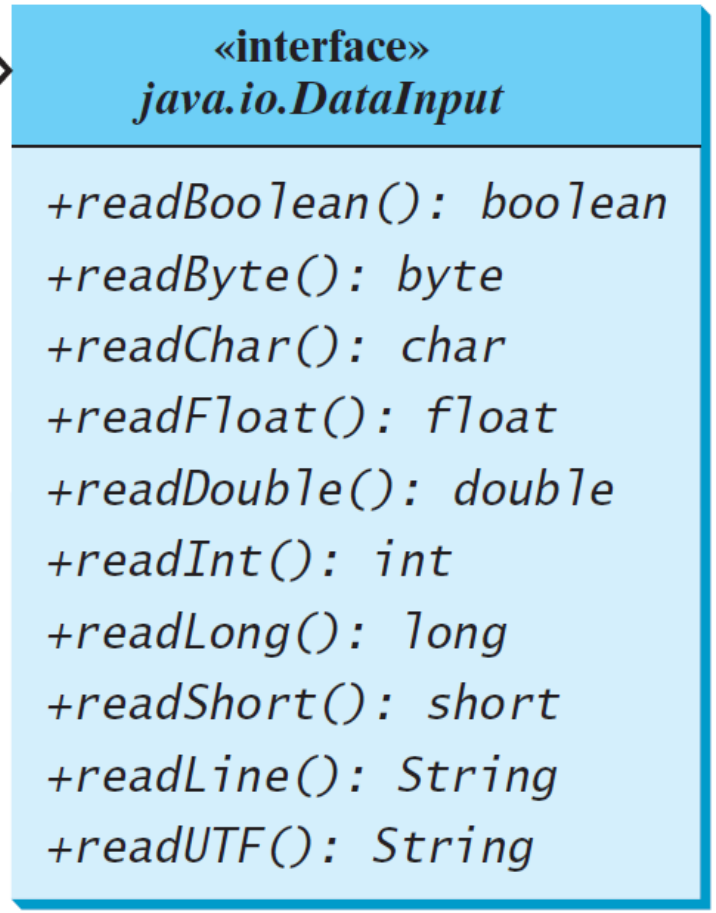
A `FilterInputStream` contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or *providing additional functionality*. The class `FilterInputStream` itself *simply overrides* all methods of `InputStream` with versions that pass all requests to the contained input stream. *Subclasses* of `FilterInputStream` may further override some of these methods and may also provide additional methods and fields.

The class `FilterOutputStream` itself *simply overrides* all methods of `OutputStream` with versions that pass all requests to the underlying output stream. *Subclasses* of `FilterOutputStream` may further override some of these methods as well as provide *additional methods* and fields.

# The DataInputStream Class






| | |
|---|---|
| Reads a `Boolean` from the input stream. | |
| Reads a `byte` from the input stream. | |
| Reads a character from the input stream. | |
| Reads a `float` from the input stream. | |
| Reads a `double` from the input stream. | |
| Reads an `int` from the input stream. | |
| Reads a `long` from the input stream. | |
| Reads a `short` from the input stream. | |
| Reads a `line` of characters from input. | |
| Reads a `string` in UTF format. | |

# Source Code of `DataInputStream`



```
public class DataInputStream
extends FilterInputStream implements DataInput {
    public final byte readByte() throws IOException {
        int ch = in.read();
        if (ch < 0) throw new EOFException();
        return (byte)(ch);
    }
    public final short readShort() throws IOException {
        int ch1 = in.read();
        int ch2 = in.read();
        if ((ch1 | ch2) < 0) throw new EOFException();
        return (short)((ch1 << 8) + (ch2 << 0));
    }
}
```

# Checking End of File

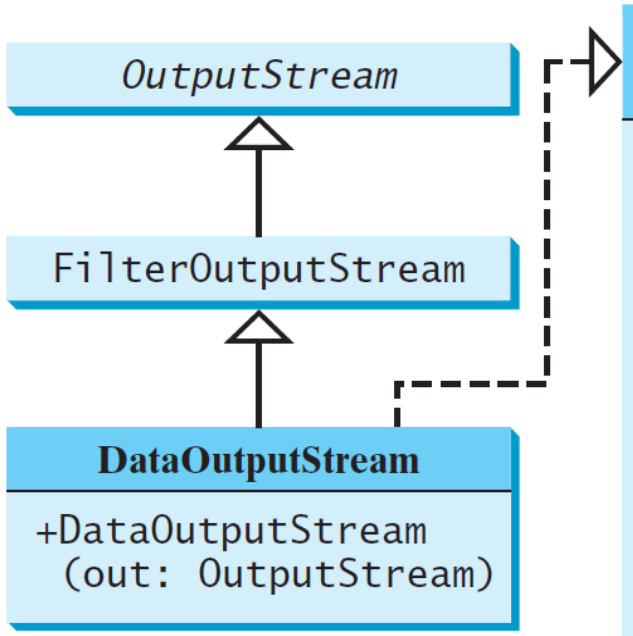


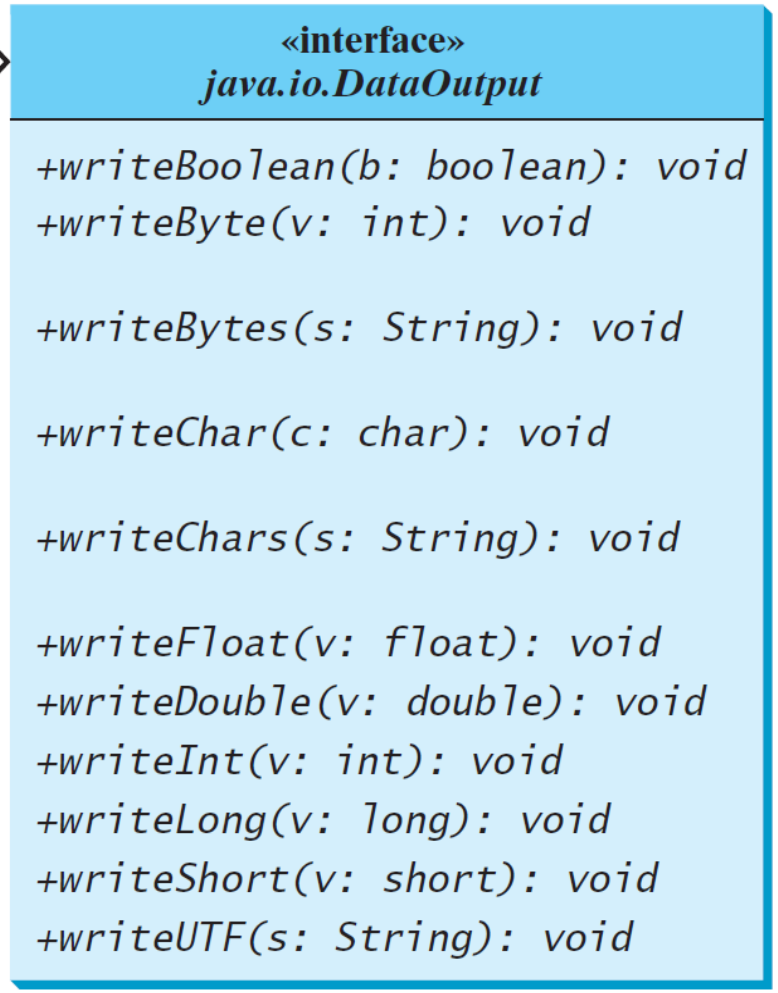
The `read()` method read the next byte of data, or `-1` if the end of the stream is reached. In `DataInputStream` class, read methods will throw `EOFException` if this input stream has reached the end. For example:

```
public final int readInt() throws IOException {
    int ch1 = in.read();
    int ch2 = in.read();
    int ch3 = in.read();
    int ch4 = in.read();
    if ((ch1 | ch2 | ch3 | ch4) < 0)
        throw new EOFException();
    return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) + (ch4 << 0));
}
```

# The DataOutputStream Class






Writes a Boolean to the output stream.

Writes the eight low-order bits of the argument `v` to the output stream.

Writes the lower byte of the characters in a string to the output stream.

Writes a character (composed of 2 bytes) to the output stream.

Writes every character in the string `s` to the output stream, in order, 2 bytes per character.

Writes a `float` value to the output stream.

Writes a `double` value to the output stream.

Writes an `int` value to the output stream.

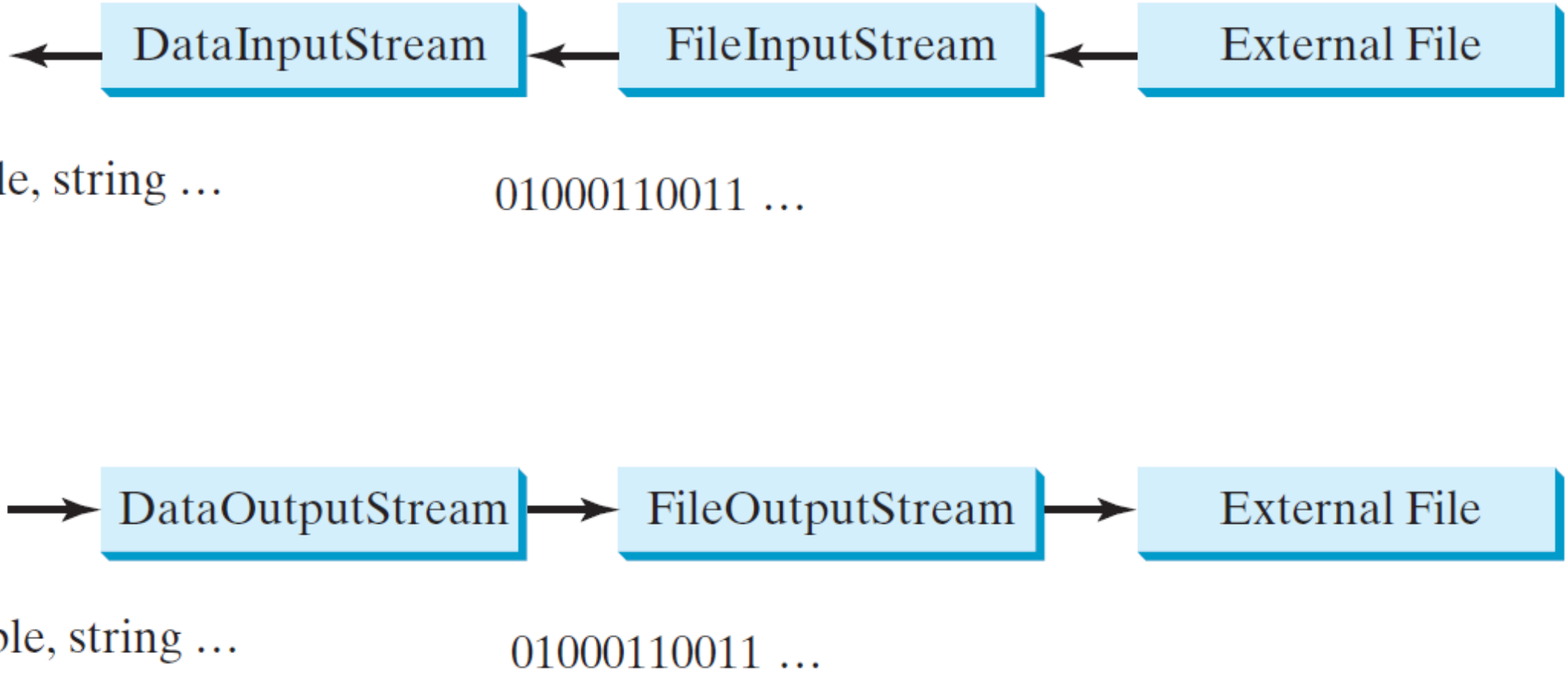
Writes a `long` value to the output stream.

Writes a `short` value to the output stream.

Writes `s` string in UTF format.

# Concept of Pipe Line





# Source Code of `DataOutputStream`



```
public class DataOutputStream
extends FilterOutputStream implements DataOutput {
    public final void writeByte(int v) throws IOException {
        out.write(v);
        incCount(1);
    }
    public final void writeInt(int v) throws IOException {
        out.write((v >>> 24) & 0xFF);
        out.write((v >>> 16) & 0xFF);
        out.write((v >>>  8) & 0xFF);
        out.write((v >>>  0) & 0xFF);
        incCount(4);
    }
}
```

# Order and Format



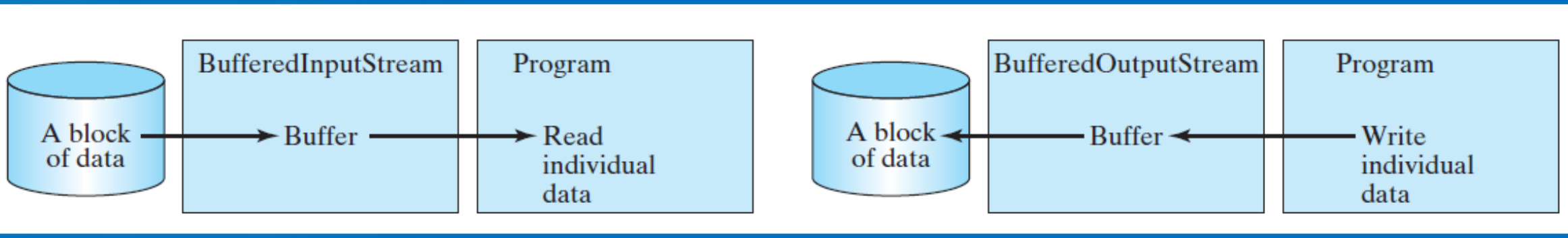
CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using `writeUTF()`, you must read names using `readUTF()`. For example:

```
public static void main(String args[]) throws IOException{
    DataOutputStream output =
            new DataOutputStream(new FileOutputStream("temp.dat"));
    output.writeUTF("John");
    output.writeDouble(85.5);
    DataInputStream input =
            new DataInputStream(new FileInputStream("temp.dat"));
    System.out.println(input.readUTF() + " " + input.readDouble());
    input.close();
    output.close();
}
```

# BufferedInputStream and BufferedOutputStream



BufferedInputStream/BufferedOutputStream can be used to *speed up* input and output by reducing the number of disk reads and writes. Using BufferedInputStream, the whole block of data on the disk is read into the buffer in the memory once. The individual data are then delivered to your program from the buffer. Using BufferedOutputStream, the individual data are first written to the buffer in the memory. When the buffer is full, all data in the buffer are written to the disk once.



# Source Code of BufferedInputStream



```
public class BufferedInputStream extends FilterInputStream {
    protected volatile byte buf[];
    private void fill() throws IOException {
        //Fills the buffer with more data,
    }
    public synchronized int read() throws IOException {
        if (pos >= count) {
            fill();
            if (pos >= count)
                return -1;
        }
        return getBufIfOpen()[pos++] & 0xff;
    }
}
```

# Source Code of BufferedOutputStream



```
public class BufferedOutputStream extends FilterOutputStream {
    protected volatile byte buf[];
    private void flushBuffer() throws IOException {
        if (count > 0) {
            out.write(buf, 0, count);
            count = 0;
        }
    }
    public synchronized void write(int b) throws IOException {
        if (count >= buf.length)
            flushBuffer();
        buf[count++] = (byte)b;
    }
}
```

# Copy a File



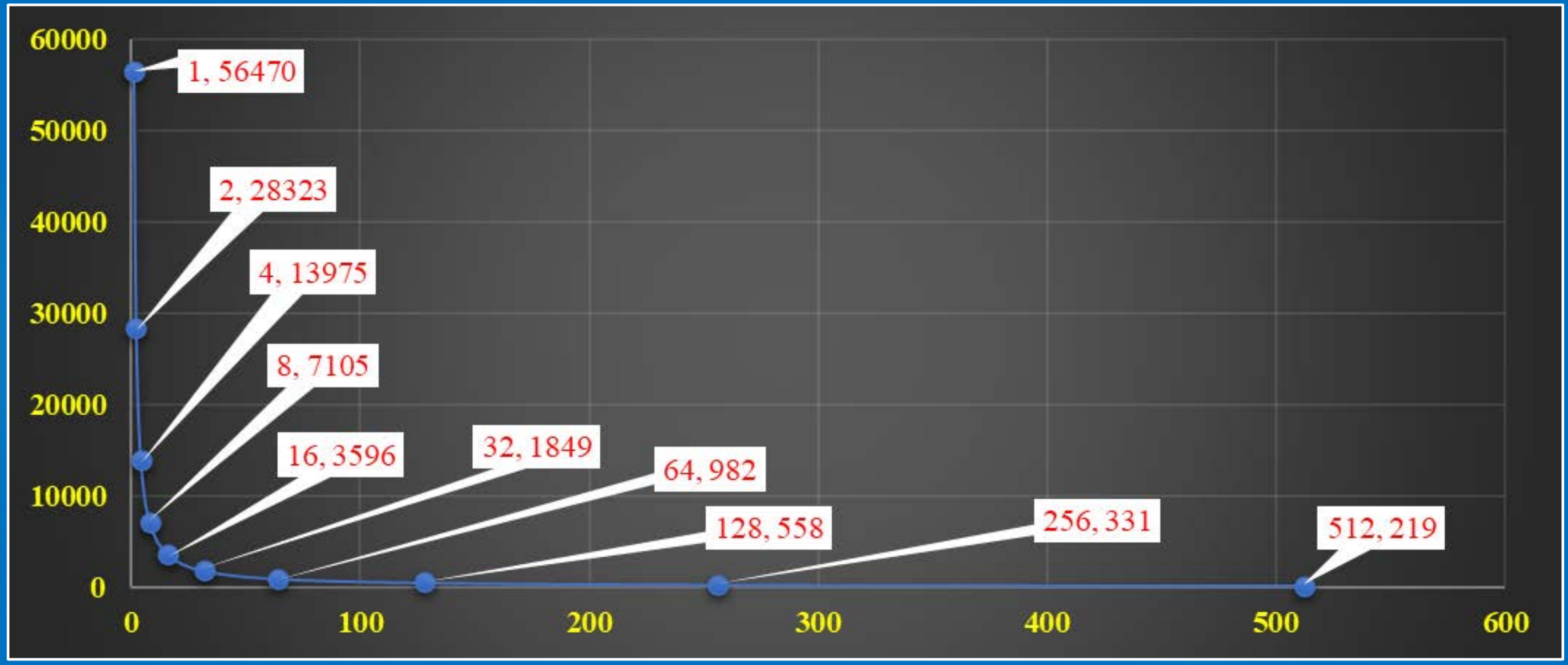
This case study develops a program that copy a file with different buffer strategy.

```
public static void copy(String src, String tar, int s)
throws IOException {
    long startTime = System.currentTimeMillis();
    InputStream is = new BufferedInputStream(new FileInputStream(src), s);
    OutputStream os = new BufferedOutputStream(new FileOutputStream(tar), s);
    int c;
    while((c = is.read()) != -1) { os.write(c);}
    is.close();
    os.close();
    long endTime = System.currentTimeMillis();
    System.out.printf("[%d]The cost is %d\n", s, endTime - startTime);
}
```

# The Cost of the Copy Method



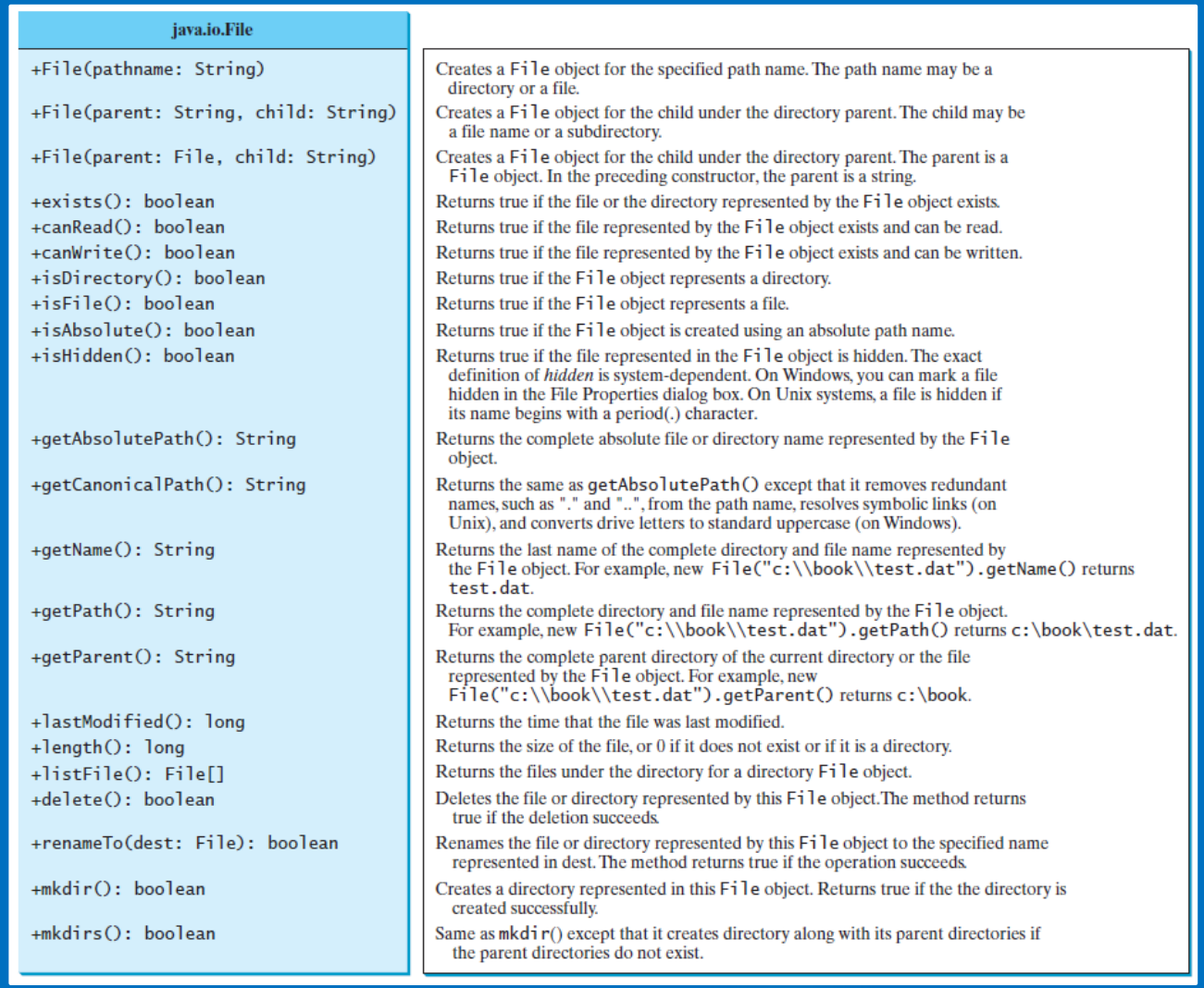
The cost to copy a file of size 14.8 MB



# The File Class



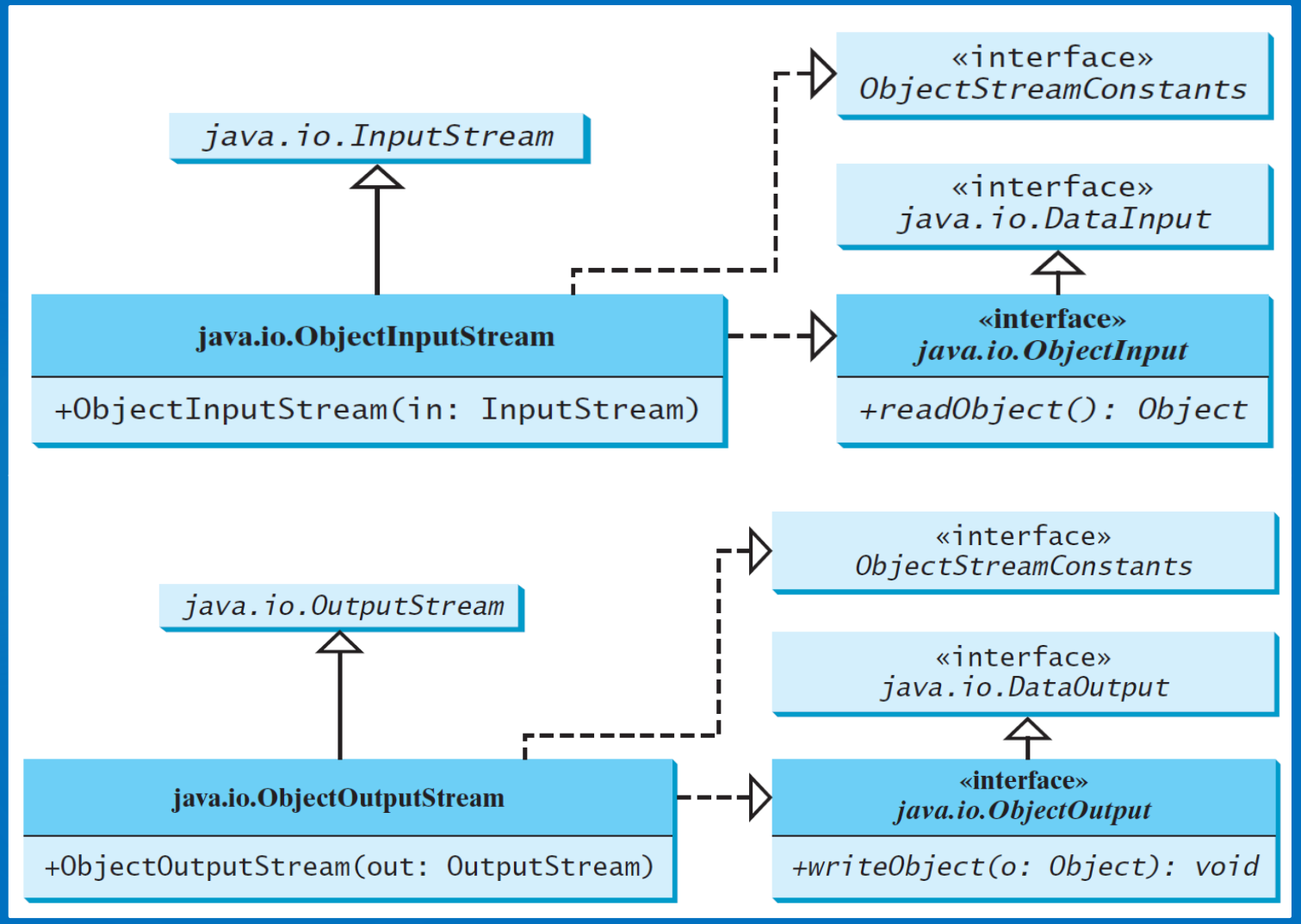
The **File** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The filename is a string. The **File** class is a **wrapper class** for the **file name and its directory path**.



| java.io.File | |
|---|---|
| `+File(pathname: String)` | Creates a `File` object for the specified path name. The path name may be a directory or a file. |
| `+File(parent: String, child: String)` | Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory. |
| `+File(parent: File, child: String)` | Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string. |
| `+exists(): boolean` | Returns true if the file or the directory represented by the `File` object exists. |
| `+canRead(): boolean` | Returns true if the file represented by the `File` object exists and can be read. |
| `+canWrite(): boolean` | Returns true if the file represented by the `File` object exists and can be written. |
| `+isDirectory(): boolean` | Returns true if the `File` object represents a directory. |
| `+isFile(): boolean` | Returns true if the `File` object represents a file. |
| `+isAbsolute(): boolean` | Returns true if the `File` object is created using an absolute path name. |
| `+isHidden(): boolean` | Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character. |
| `+getAbsolutePath(): String` | Returns the complete absolute file or directory name represented by the `File` object. |
| `+getCanonicalPath(): String` | Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows). |
| `+getName(): String` | Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`. |
| `+getPath(): String` | Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`. |
| `+getParent(): String` | Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`. |
| `+lastModified(): long` | Returns the time that the file was last modified. |
| `+length(): long` | Returns the size of the file, or 0 if it does not exist or if it is a directory. |
| `+listFile(): File[]` | Returns the files under the directory for a directory `File` object. |
| `+delete(): boolean` | Deletes the file or directory represented by this `File` object. The method returns true if the deletion succeeds. |
| `+renameTo(dest: File): boolean` | Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds. |
| `+mkdir(): boolean` | Creates a directory represented in this `File` object. Returns true if the the directory is created successfully. |
| `+mkdirs(): boolean` | Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist. |

# Read and Write an Object






Read Object

Write Object

# Write and Read a `Date` Object



```
public static void main(String[] args)
            throws IOException, ClassNotFoundException {
    ObjectOutputStream oos =
            new ObjectOutputStream(new FileOutputStream("date.dat"));
    Date d1 = new Date();
    d1.setTime(900000000);
    oos.writeObject(d1);

    ObjectInputStream ois =
            new ObjectInputStream(new FileInputStream("date.dat"));
    Date d2 = (Date) ois.readObject();
    System.out.println(d2.getTime());

    oos.close();
    ois.close();
}
```

# The `Serializable` Interface



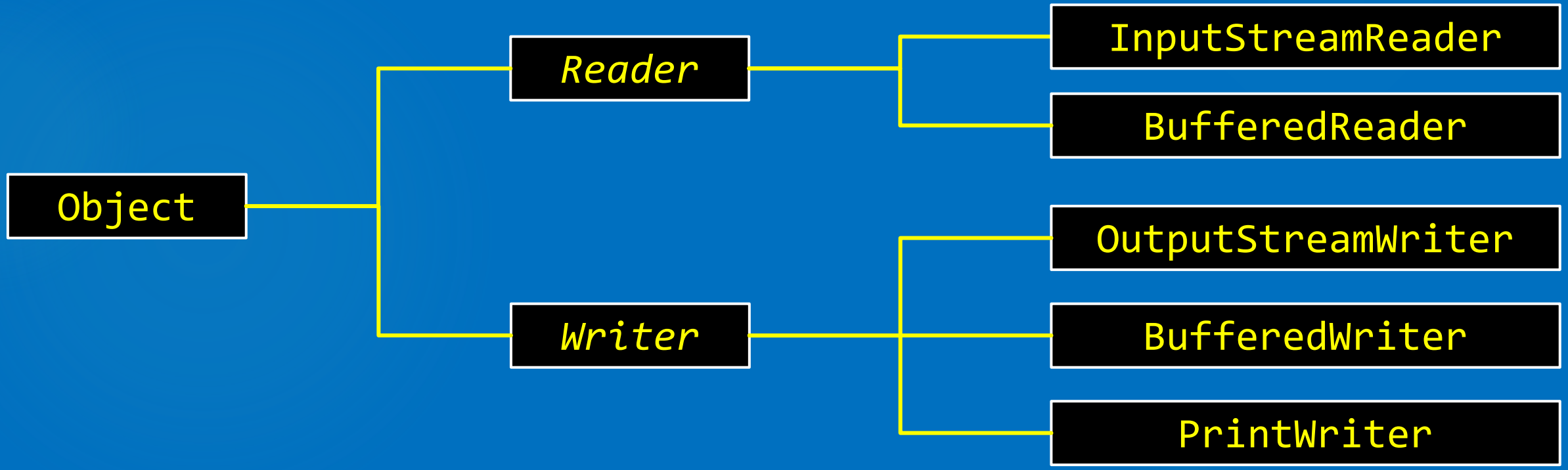
Not every object can be written to an output stream. Objects that can be so written are said to be *serializable*. A serializable object is an instance of the `java.io.Serializable` interface, so the object's class must implement Serializable. The `Serializable` interface is a *marker interface*. Since it has no methods, you don't need to add additional code in your class that implements Serializable.

```
class SerObject implements Serializable{
    private static final long serialVersionUID = -3288399016693459819L;
    private int d = 10;
    public int getD() {
        return d;
    }
    public void setD(int d) {
        this.d = d;
    }
}
```

# Text I/O



Text is composed of characters, so *Text I/O* are directly oriented to character streams. In java, `Reader` and `Writer` classes are designed for reading and writing to character streams.



# The `Writer` Class



```
public abstract class Writer implements Appendable, Closeable, Flushable {
    public void write(int c) throws IOException {
        synchronized (lock) {
            if (writeBuffer == null){
                writeBuffer = new char[WRITE_BUFFER_SIZE];
            }
            writeBuffer[0] = (char) c;
            write(writeBuffer, 0, 1);
        }
    }
    public void write(String str) throws IOException {
        write(str, 0, str.length());
    }
}
```

# The `write()` method in `OutputStreamWriter`



```
public class OutputStreamWriter extends Writer {
    private final StreamEncoder se;
    public OutputStreamWriter(OutputStream out, Charset cs) {
        super(out);
        if (cs == null) throw new NullPointerException("charset");
        se = StreamEncoder.forOutputStreamWriter(out, this, cs);
    }
    public void write(char cbuf[], int off, int len) throws IOException {
        se.write(cbuf, off, len);
    }
}
```

# An Example of Writing



```
public static void main(String[] args) throws IOException {
    FileOutputStream fos = new FileOutputStream("out.txt");
    BufferedWriter bw =  new BufferedWriter(
            new OutputStreamWriter(fos, Charset.forName("UTF-8")));
    bw.write("This is an apple!\n");
    bw.write("This is an apple!\n");
    bw.close();
}
```

# The `PrintWriter` Class



| java.io.PrintWriter | |
|---|---|
| `+PrintWriter(file: File)` | Creates a `PrintWriter` object for the specified file object. |
| `+PrintWriter(filename: String)` | Creates a `PrintWriter` object for the specified file-name string. |
| `+print(s: String): void` | Writes a string to the file. |
| `+print(c: char): void` | Writes a character to the file. |
| `+print(cArray: char[]): void` | Writes an array of characters to the file. |
| `+print(i: int): void` | Writes an `int` value to the file. |
| `+print(l: long): void` | Writes a `long` value to the file. |
| `+print(f: float): void` | Writes a `float` value to the file. |
| `+print(d: double): void` | Writes a `double` value to the file. |
| `+print(b: boolean): void` | Writes a `boolean` value to the file. |
| Also contains the overloaded `println` methods. | 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. |
| Also contains the overloaded `printf` methods. | The `printf` method was introduced in §4.6, "Formatting Console Output." |

# PrintWriter Constructors

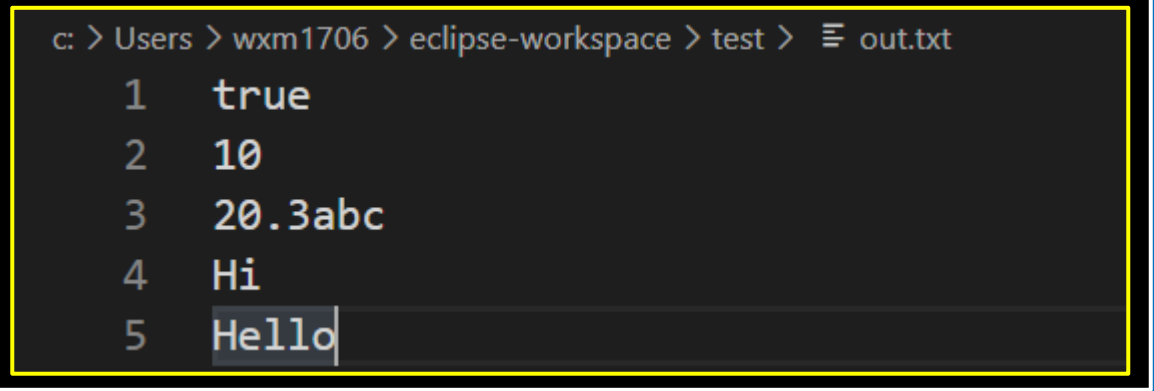


```
public class PrintWriter extends Writer {
    public PrintWriter (Writer out) {
        this(out, false);
    }
    public PrintWriter(Writer out, boolean autoFlush) {
            super(out);
            this.out = out;
            this.autoFlush = autoFlush;
            lineSeparator = java.security.AccessController.doPrivileged(
                new sun.security.action.GetPropertyAction("line.separator"));
    }
    public PrintWriter(String fileName) throws FileNotFoundException {
        this(new BufferedWriter
            (new OutputStreamWriter(new FileOutputStream(fileName))),false);
    }
}
```

# Writing Data Using PrintWriter



```
public static void main(String[] args) throws IOException {
    PrintWriter pw = new PrintWriter("out.txt");
    pw.println(true);
    pw.println(10);
    pw.print(20.3);
    pw.println(new char[]{'a','b','c'});
    pw.println("Hi");
    pw.print("Hello");
    pw.close();
}
```



# The Reader Class



```
public abstract class Reader implements Readable, Closeable {
    public int read() throws IOException {
        char cb[] = new char[1];
        if (read(cb, 0, 1) == -1)
            return -1;
        else
            return cb[0];
    }
    abstract public int read(char cbuf[], int off, int len)
        throws IOException;
}
```

# The `read()` Method in InputStreamReader



```
public class InputStreamReader extends Reader {
    private final StreamDecoder sd;
    public InputStreamReader(InputStream in, Charset cs) {
        super(in);
        if (cs == null) throw new NullPointerException("charset");
        sd = StreamDecoder.forInputStreamReader(in, this, cs);
    }
    public int read() throws IOException {
        return sd.read();
    }
}
```

# The `readLine()` Method in BufferedReader



```
public class BufferedReader extends Reader {
    private Reader in;
    /**
     * Reads a line of text.  A line is considered to be terminated by any one
     * of a line feed ('\n'), a carriage return ('\r'), or a carriage return
     * followed immediately by a linefeed.
     */
    String readLine(boolean ignoreLF) throws IOException {}
    public String readLine() throws IOException {
        return readLine(false);
    }
}
```

# An Example of Reading



```
public static void main(String[] args) throws IOException {
    FileInputStream fis = new FileInputStream("out.txt");
    BufferedReader br =  new BufferedReader(
            new InputStreamReader(fis, Charset.forName("UTF-8")));
    String line = null;
    while((line = br.readLine()) != null)
        System.out.println(line);
    br.close();
}
```

# Reading Data Using Scanner



| java.util.Scanner | |
|---|---|
| +Scanner(source: File) | Creates a Scanner that scans tokens from the specified file. |
| +Scanner(source: String) | Creates a Scanner that scans tokens from the specified string. |
| +close() | Closes this scanner. |
| +hasNext(): boolean | Returns true if this scanner has more data to be read. |
| +next(): String | Returns next token as a string from this scanner. |
| +nextLine(): String | Returns a line ending with the line separator from this scanner. |
| +nextByte(): byte | Returns next token as a byte from this scanner. |
| +nextShort(): short | Returns next token as a short from this scanner. |
| +nextInt(): int | Returns next token as an int from this scanner. |
| +nextLong(): long | Returns next token as a long from this scanner. |
| +nextFloat(): float | Returns next token as a float from this scanner. |
| +nextDouble(): double | Returns next token as a double from this scanner. |
| +useDelimiter(pattern: String): Scanner | Sets this scanner's delimiting pattern and returns this scanner. |

# Read Data from File Using Scanner



```
public static void main(String[] args) throws IOException {
    Scanner input = new Scanner(new FileInputStream("out.txt"));
    while(input.hasNext()) {
        System.out.println(input.nextLine());
    }
    input.close();
}
```

# Access System.in Using InputStreamReader



```
public static void main(String[] args)
        throws IOException, InterruptedException {
    BufferedReader isw =
            new BufferedReader(new InputStreamReader(System.in));
    while(true) {
        System.out.print("Enter a string:");
        System.out.println("The string is: " + isw.readLine());
    }
}
```

# Automatically Closing Using `try-with-resources`



```
public static void main(String[] args) {
    try(BufferedReader isw
            = new BufferedReader(new InputStreamReader(System.in))){
        while(true) {
            System.out.print("Enter a string:");
            System.out.println("The string is: " + isw.readLine());
        }
    } catch(IOException e) {
        e.printStackTrace();
    }
}
```

# Automatically Closing Using `try-with-resources`



```
public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
public abstract class Reader implements Readable, Closeable {}
public class BufferedReader extends Reader {
    private Reader in;
    public void close() throws IOException {
        synchronized (lock) {
            if (in == null) return;
            try {   in.close();}
            finally { in = null; cb = null;}
        }
    }
}
```

# Problem: Replacing Text



Write a method named `ReplaceText` that replaces a string in a text file with a new string. For example:

Software is all around you, even in devices that you might not think would need it. Of course, you expect to find and use software on a personal computer, but software also plays a role in running airplanes, cars, cell phones, and even toasters.

Software is all around you, even in devices that you might not think would need it. Of course, you expect to find and use software on a PC, but software also plays a role in running airplanes, cars, cell phones, and even toasters.

# Read From Web



```
public static void main(String[] args) {
    URL url;
    try {
        url = new URL("http://www.baidu.com");
        try(Scanner input = new Scanner(url.openStream(), "UTF-8");) {
            while(input.hasNext()) {
                System.out.println(input.next());
            }
            input.close();
        }catch (IOException e) {
            e.printStackTrace();
        }
    } catch (MalformedURLException e1) {
        e1.printStackTrace();
    }
}
```

# Exercises



What is wrong in the following code?

```
public class Test {
    public static void main(String[] args) {
        try (
            FileInputStream fis = new FileInputStream("test.dat");) {
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
        catch (FileNotFoundException ex) {
            ex.printStackTrace();
        }
    }
}
```

# Exercises



Suppose you run the following program on Windows using the default ASCII encoding after the program is finished, how many bytes are there in the file t.txt? Show the contents of each byte.

```
public class Test {
    public static void main(String[] args)
            throws java.io.IOException {
        try (java.io.PrintWriter output
                = new java.io.PrintWriter("t.txt"); ) {
            output.printf("%s", "1234");
            output.printf("%s", "5678");
            output.close();
        }
    }
}
```

# Exercises



After the following program is finished, how many bytes are there in the file t.dat? Show the contents of each byte.

```
public class Test {
    public static void main(String[] args)
            throws IOException {
        try (DataOutputStream output
            = new DataOutputStream(new FileOutputStream("t.dat")); ) {
            output.writeInt(1234);
            output.writeInt(5678);
            output.close();
        }
    }
}
```

# Exercises



What will happen when you attempt to run the following code?

```
import java.io.*;
public class Test {
    public static void main(String[] args) throws IOException {
        try ( ObjectOutputStream output = new
            ObjectOutputStream(new FileOutputStream("object.dat")); ) {
            output.writeObject(new A());
        }
    }
}
class A implements Serializable { B b = new B();}
class B {}
```