

Chapter 17 Binary I/O



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.



Objectives

- ❑ To discover how I/O is processed in Java (§17.2).
- ❑ To distinguish between text I/O and binary I/O (§17.3).
- ❑ To read and write bytes using `FileInputStream` and `FileOutputStream` (§17.4.1).
- ❑ To read and write primitive values and strings using `DataInputStream/DataOutputStream` (§17.4.3).
- ❑ To store and restore objects using `ObjectOutputStream` and `ObjectInputStream`, and to understand how objects are serialized and what kind of objects can be serialized (§17.6).
- ❑ To implement the `Serializable` interface to make objects serializable (§17.6.1).
- ❑ To serialize arrays (§17.6.2).
- ❑ To read and write the same file using the `RandomAccessFile` class (§17.7).



package app;

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Date;
```

```
public class TestObjectStream {
    public static void main(String[] args) {
        demo1();
        demo2();
    }
}
```

```
public static void demo1() {
    File f = new File("TestObjectStream.dat");
    try {
        FileOutputStream fos = new FileOutputStream(f);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(new Date());
        oos.close();
    } catch (IOException ioe) {
        System.out.println(ioe.toString());
    }
    try {
        FileInputStream fis = new FileInputStream(f);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Date d = (Date)ois.readObject();
        System.out.println(d.toString());
        ois.close();
    } catch (Exception e){
        System.out.println(e.toString());
    }
}
```

```

public static void demo2() {
    File f = new File("TestObjectStream.dat");
    try {
        FileOutputStream fos = new FileOutputStream(f);
        ObjectOutputStream oos = new
ObjectOutputStream(fos);
        Circle c = new Circle(10.0);
        oos.writeObject(c);
        oos.close();
    } catch (IOException ioe) {
        System.out.println(ioe.toString());
    }
    try {
        FileInputStream fis = new FileInputStream(f);
        ObjectInputStream ois = new ObjectInputStream(fis);
        Circle c2 = (Circle)ois.readObject();
        System.out.println(c2.toString());
        /////System.out.println(ois.readObject().toString());
        ///// find : what will happen
        ois.close();
    }catch(Exception e){
        System.out.println(e.toString());
    }
}
}

```

```

//standard are writeable
//user owner must mplement Serializable or transient
class B { // not Seriallizable

}

class Circle implements Serializable{
    private transient B b = new B(); // make b not be Serializable
    and b wont be write in file
    private double radius;
    ////static int numberOfObject = 0;
    Circle(){
        this(1.0);
    }
    Circle(double radius){
        setRadius(radius);
        ////numberOfObject++;
    }
    public void setRadius(double radius){
        this.radius = radius;
    }
    public double getRadius(){
        return this.radius;
    }
    @Override
    public String toString(){
        return "radius = " + Double.toString(getRadius());
    }
    /*
    public String toString(){
        return "radius = " + Double.toString(getRadius()); + "
        NumberOFObject " + Integer.toString(numberOfObject) ;
    }

    */
}

```

รู้จัก Java Stream กันก่อน

Stream คือ ลำดับของ byte ที่ไหลจากจุดหนึ่งไปจุดหนึ่ง, จุดที่เข้าคือ I/O (input, output)

เช่น

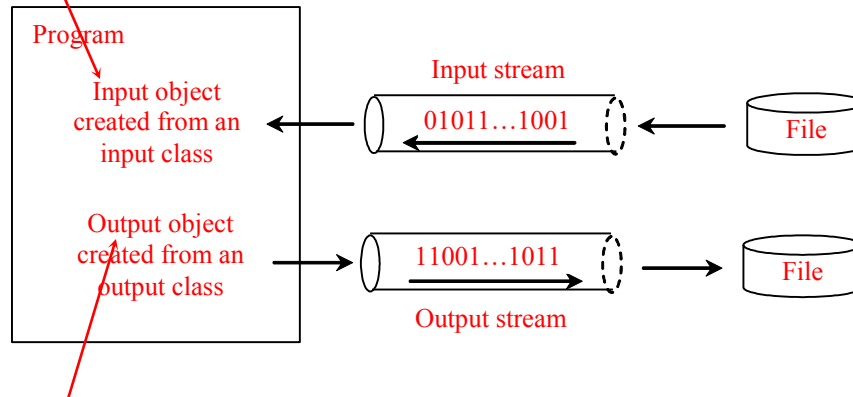


↳ การอ่านข้อมูล → เราทำการอ่านเป็น inputStream byte ที่ไหลเข้า = อ่าน
โปรแกรมของเรา

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.

```
Scanner input = new Scanner(new File("temp.txt"));  
System.out.println(input.nextLine());
```



```
PrintWriter output = new PrintWriter("temp.txt");  
output.println("Java 101");  
output.close();
```



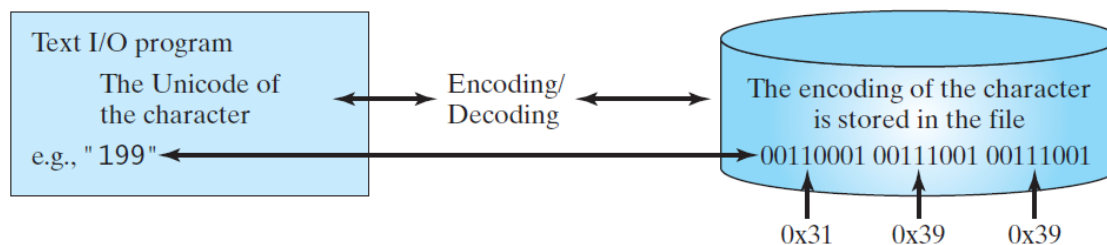
Text File vs. Binary File

- ❑ Data stored in a text file are represented in human-readable form. Data stored in a binary file are represented in binary form. You cannot read binary files. Binary files are designed to be read by programs. For example, the Java source programs are stored in text files and can be read by a text editor, but the 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.
- ❑ Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters and 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.

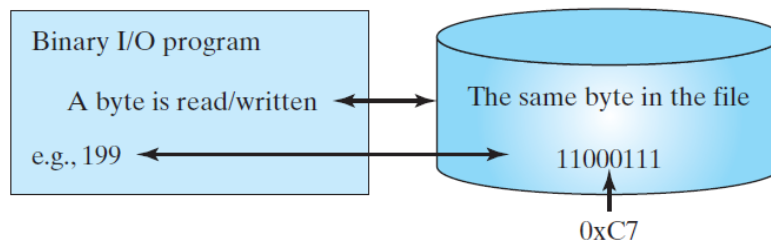
Binary I/O

อ่านต๊อบสี่

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.



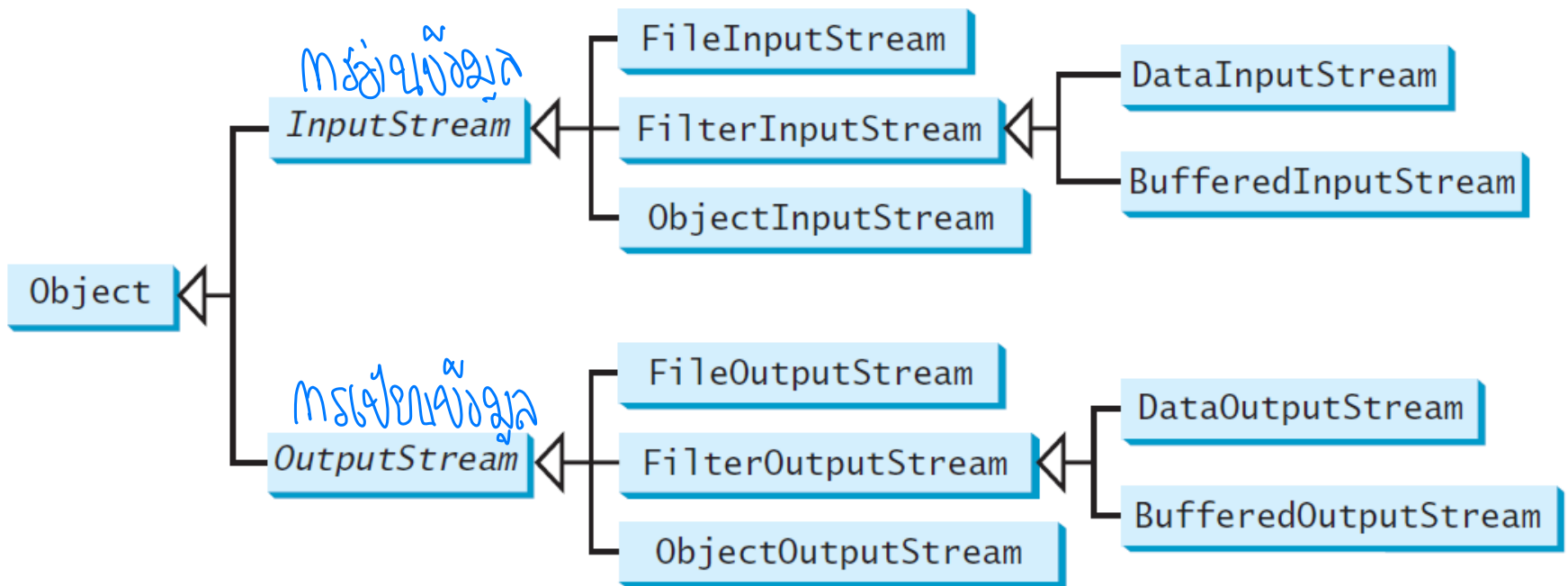
(a)



(b)



Binary I/O Classes



InputStream

The value returned is a byte as an int type.

java.io.InputStream

+read(): int

(1 byte = 8 bit)

+read(b: byte[]): int

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

+available(): int

+close(): void

+skip(n: long): long

+markSupported(): boolean

+mark(readlimit: int): void

+reset(): void

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.

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.

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.

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

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

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

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

Marks the current position in this input stream.

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*

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

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

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

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

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

+*close(): void*

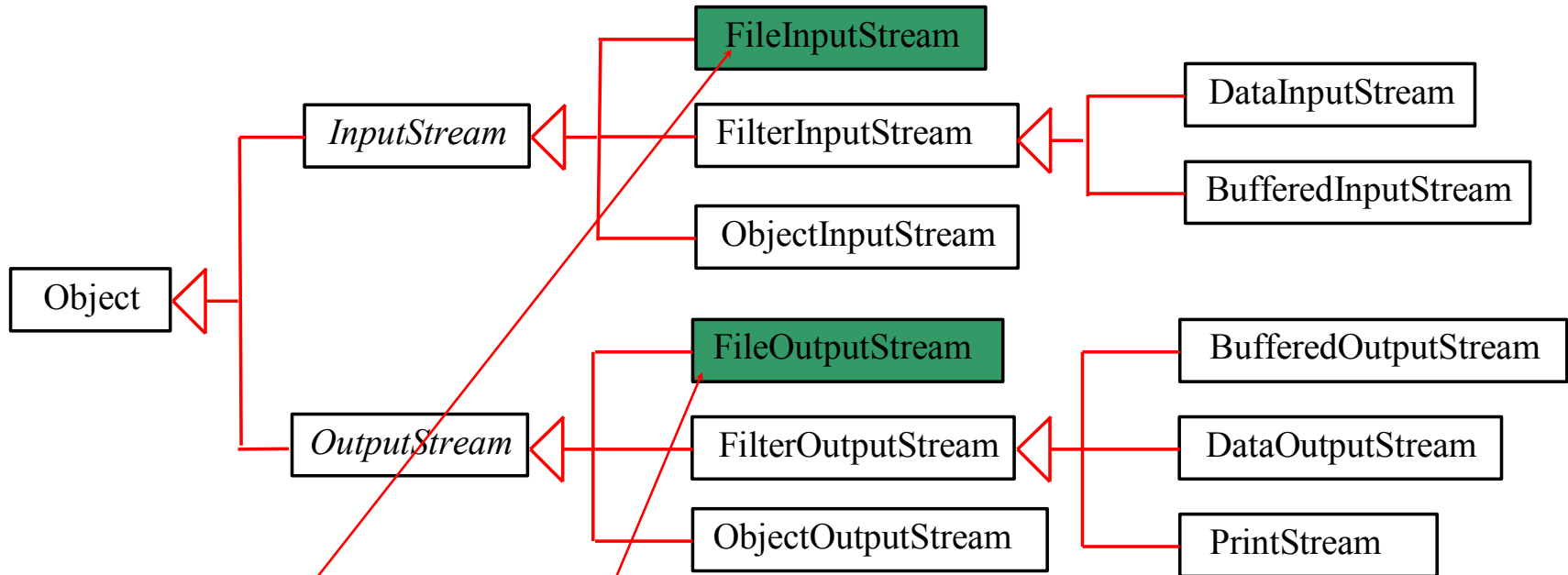
Closes this output stream and releases any system resources associated with the stream.

+*flush(): void*

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/FileOutputStream` are inherited from its superclasses.



FileInputStream

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

```
public FileInputStream(String filename) // ၂၀၀၀ String ဖိုင်အ  
public FileInputStream(File file) // ၂၀၀၀ ၂၀၀၀ File ဖိုင်အ
```

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

(ဖိုင် error ချိတ်) / ၂၀၀၀ / ၂၀၀၀ နံပါတ် - ဖိုင်မရှိပါ



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)
```

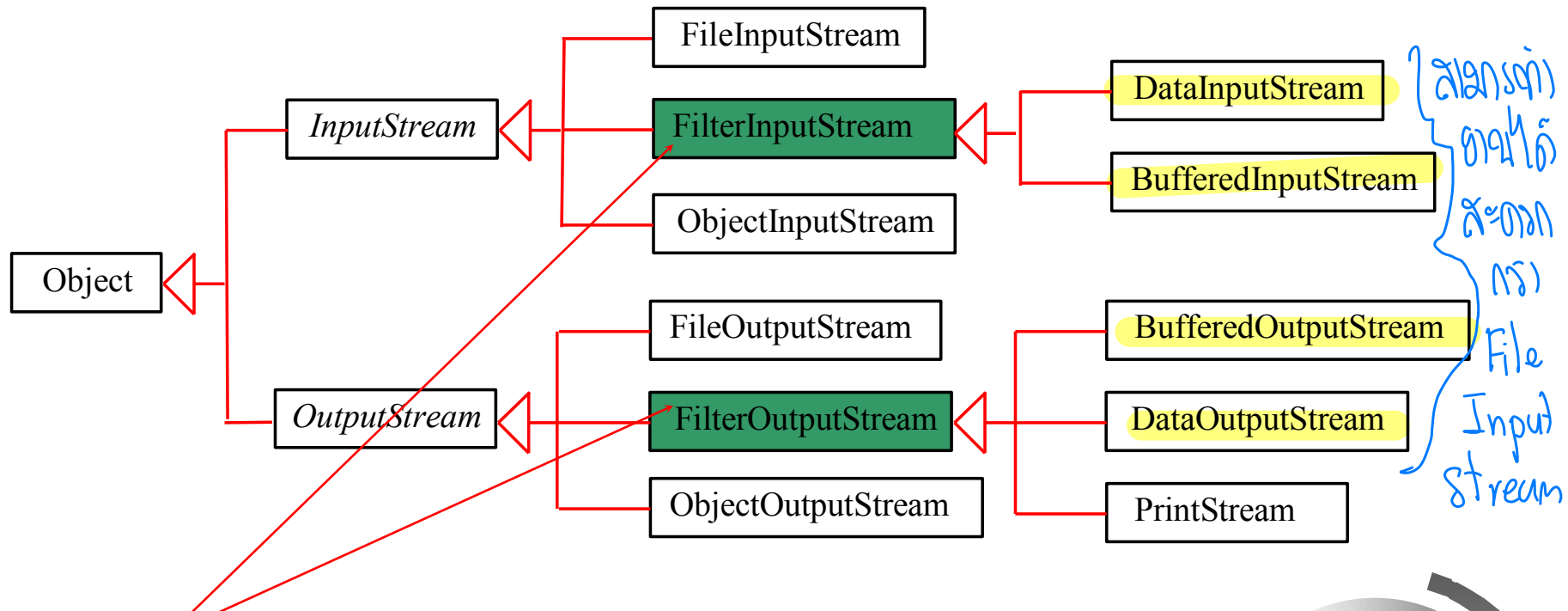
Handwritten notes:
A blue bracket groups the last two constructors with the note "append".
A red line points from the word "append" in the last constructor to the text below.
A blue bracket points to the word "true" in the text below, with the note "true" and "append" written next to it.

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

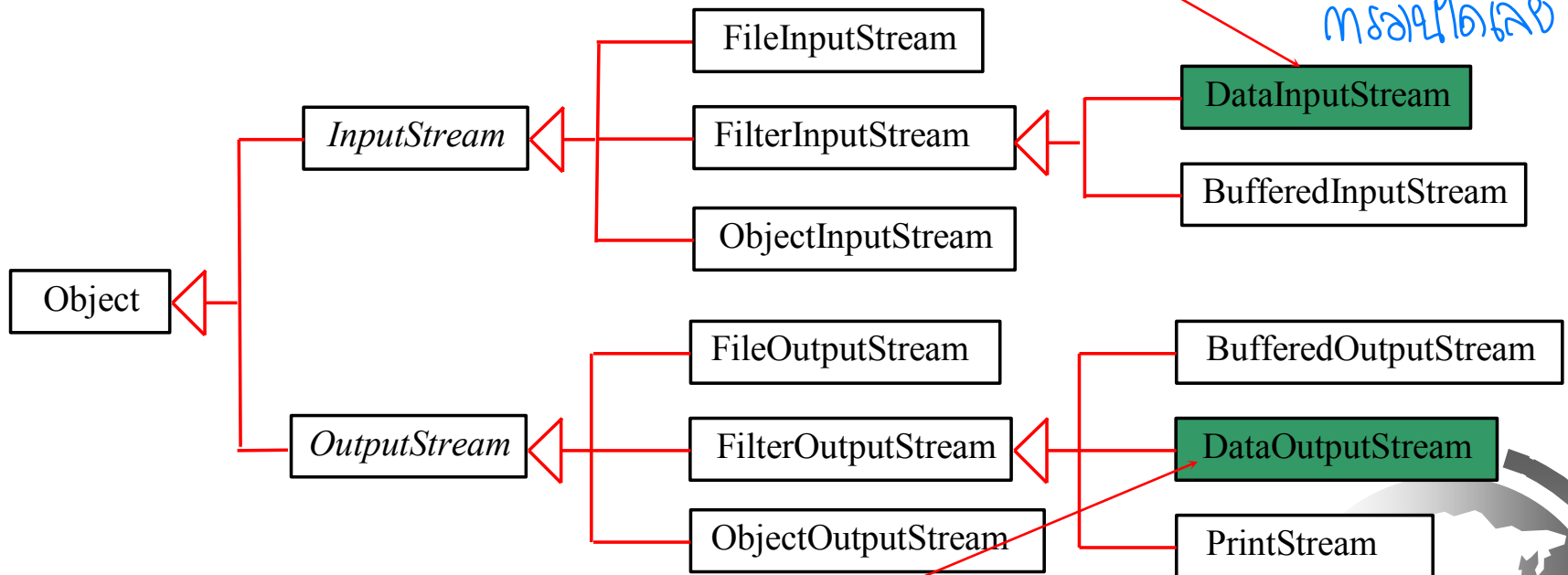


Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream. Using a filter class enables you to read 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.

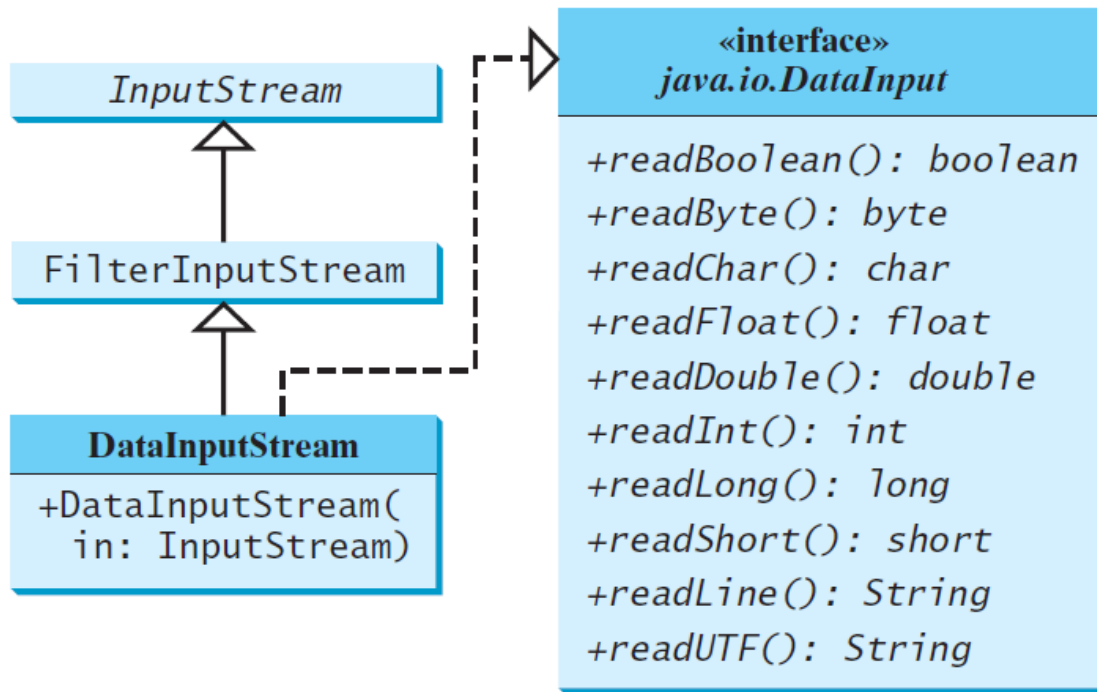
เลือก type ที่ต้องการให้คืนค่า
มาส่งให้โปรแกรม



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

DataInputStream

`DataInputStream` extends `FilterInputStream` and implements the `DataInput` interface. (ແຕ່ບ່ອນລາຍລະອຽດໄວ້ຍັງຕອນຕໍ່ໜ້າ)



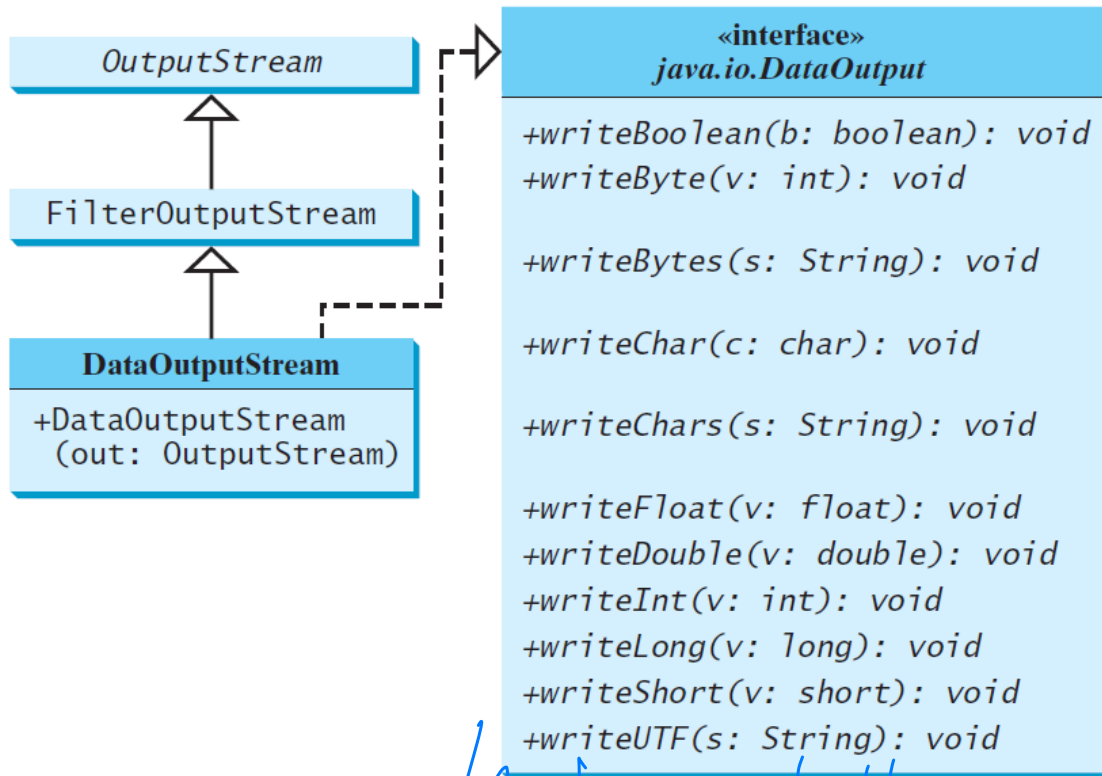
ລາຍລະອຽດ ວ່າແນວໃດໄດ້ເວົ້າທຸກບ່ອນ

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.

DataOutputStream

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.

บ่งบอกว่า unicode ต้องใช้ 2 byte
เขียนแล้วไม่สบายตา



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.

ใช้ intellisense กัน(งงๆ)

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)
```

FileInputStream
FileOutputStream

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

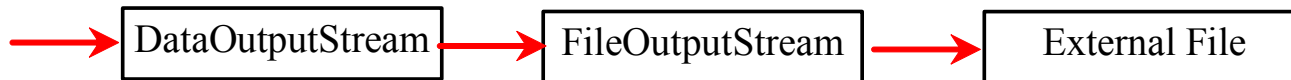
Concept of pipe line

พื้ทางข้อมูล (ข้อมูลนี้) is FileInput
Stream และ FileOutput
Stream (อ่าวอ่าว)



int, double, string ...

01000110011 ...



int, double, string ...

01000110011 ...



Order and Format

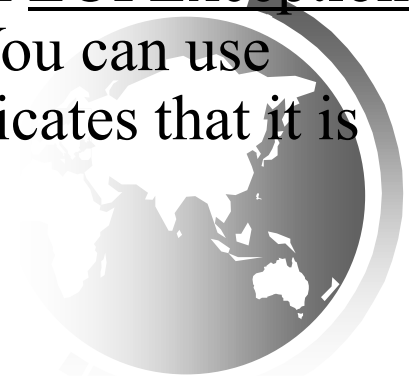
— เขียนด้วยรูปแบบใด
รูปแบบไหน

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.

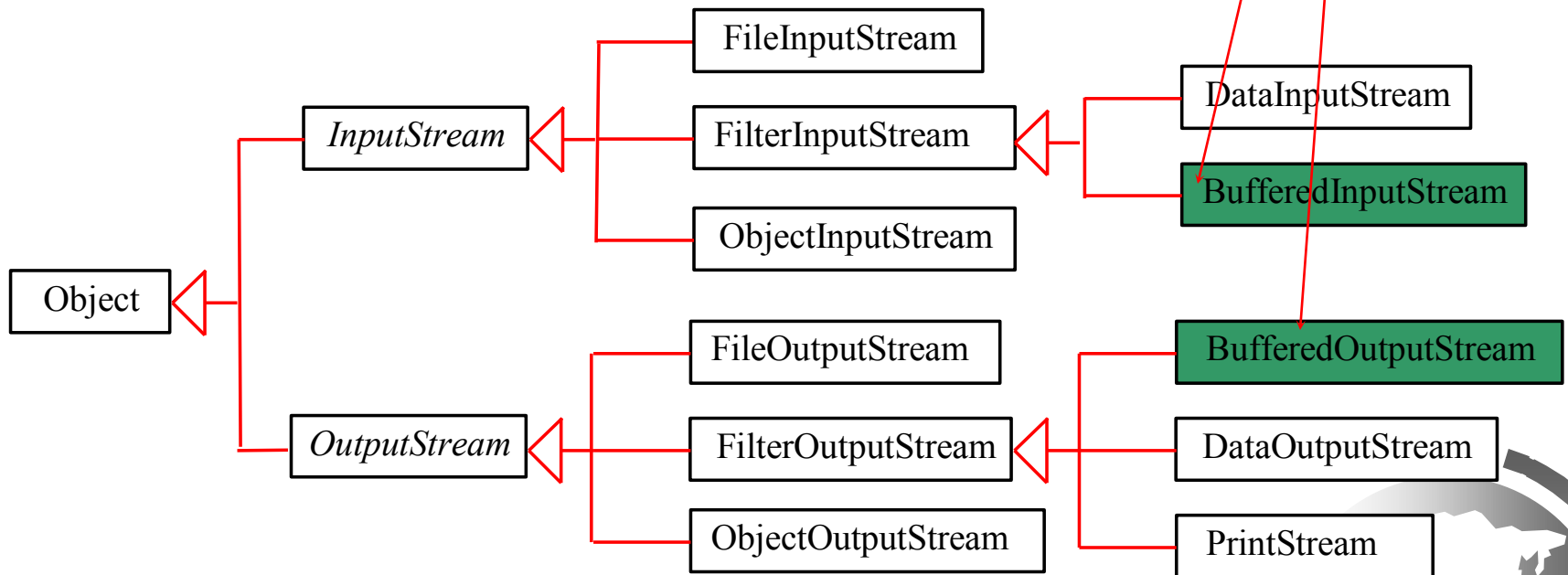
while(input.available() != 0) ทำอย่างนี้



BufferedInputStream/ BufferedOutputStream

Data Input Stream.
Data Output Stream.

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.

Data

Constructing BufferedInputStream/BufferedOutputStream

// Create a BufferedInputStream

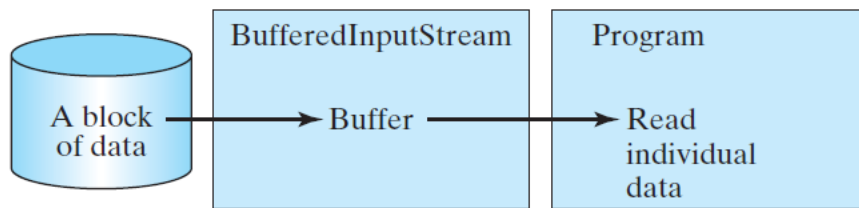
public BufferedInputStream(InputStream in)

public BufferedInputStream(InputStream in, int bufferSize)

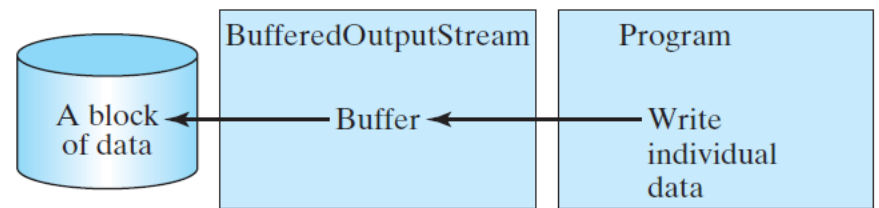
// Create a BufferedOutputStream

public BufferedOutputStream(OutputStream out)

public BufferedOutputStream(OutputStreamr out, int bufferSize)



(a)

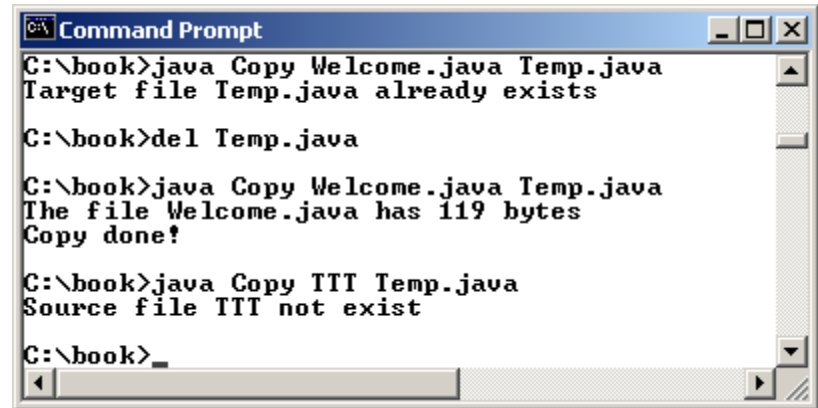


(b)

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.

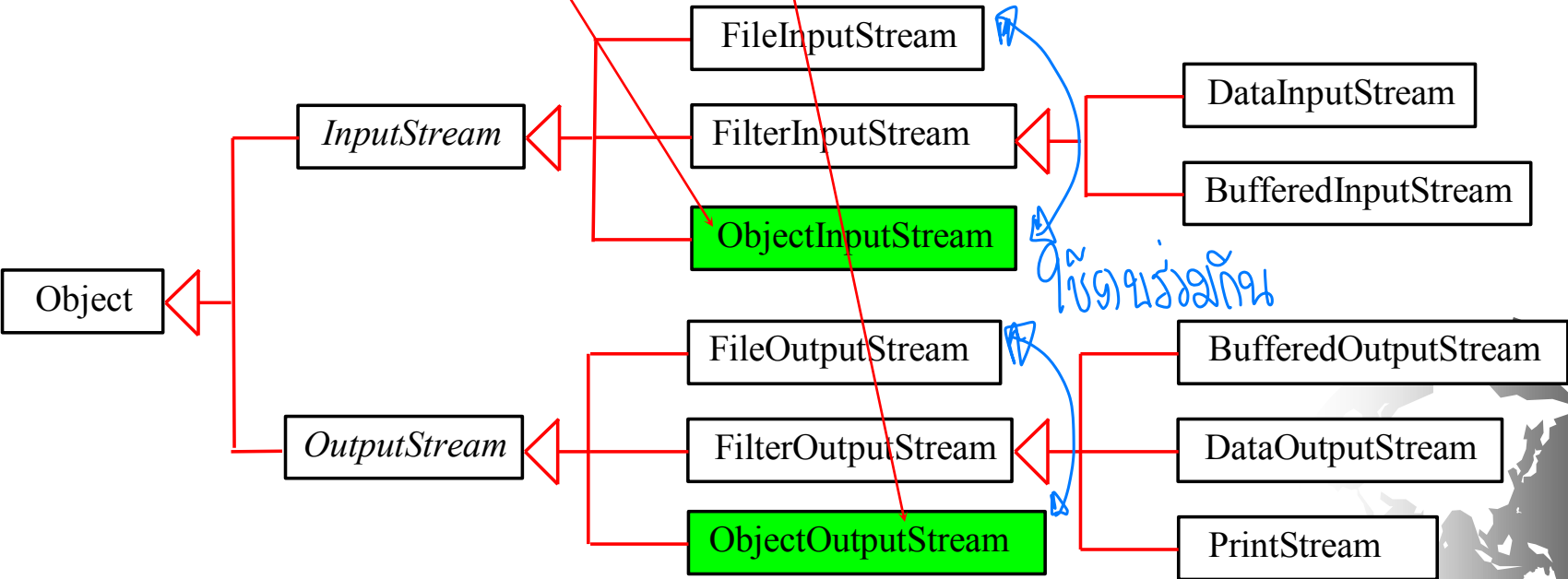
Copy

Run

Object I/O

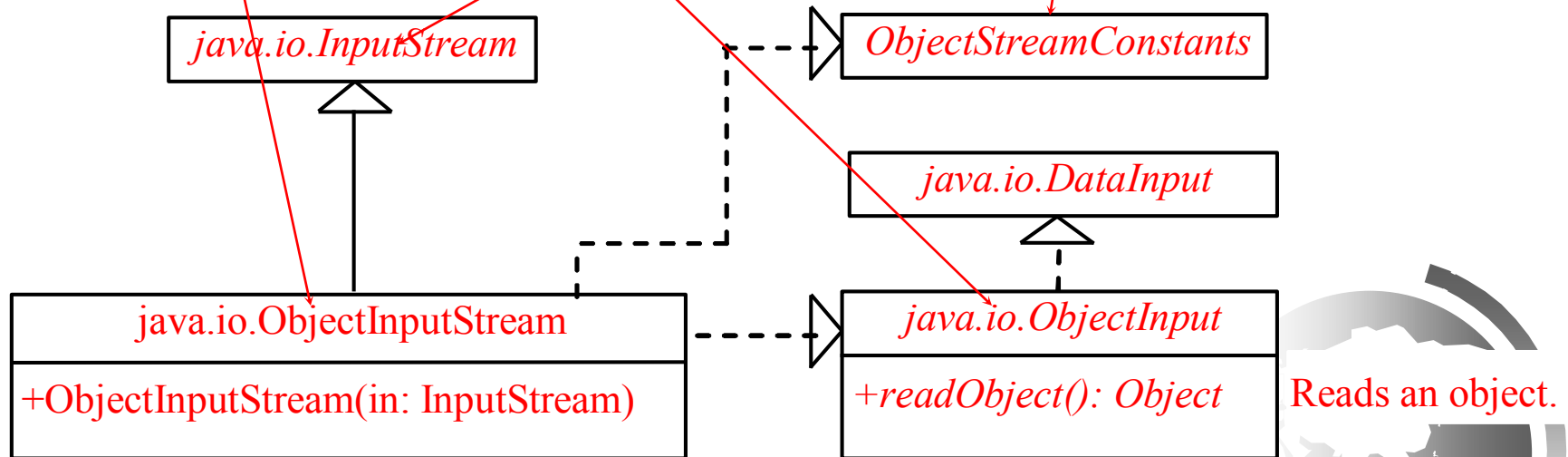
DataInputStream/DataOutputStream enables you to perform I/O for primitive type values and strings.

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



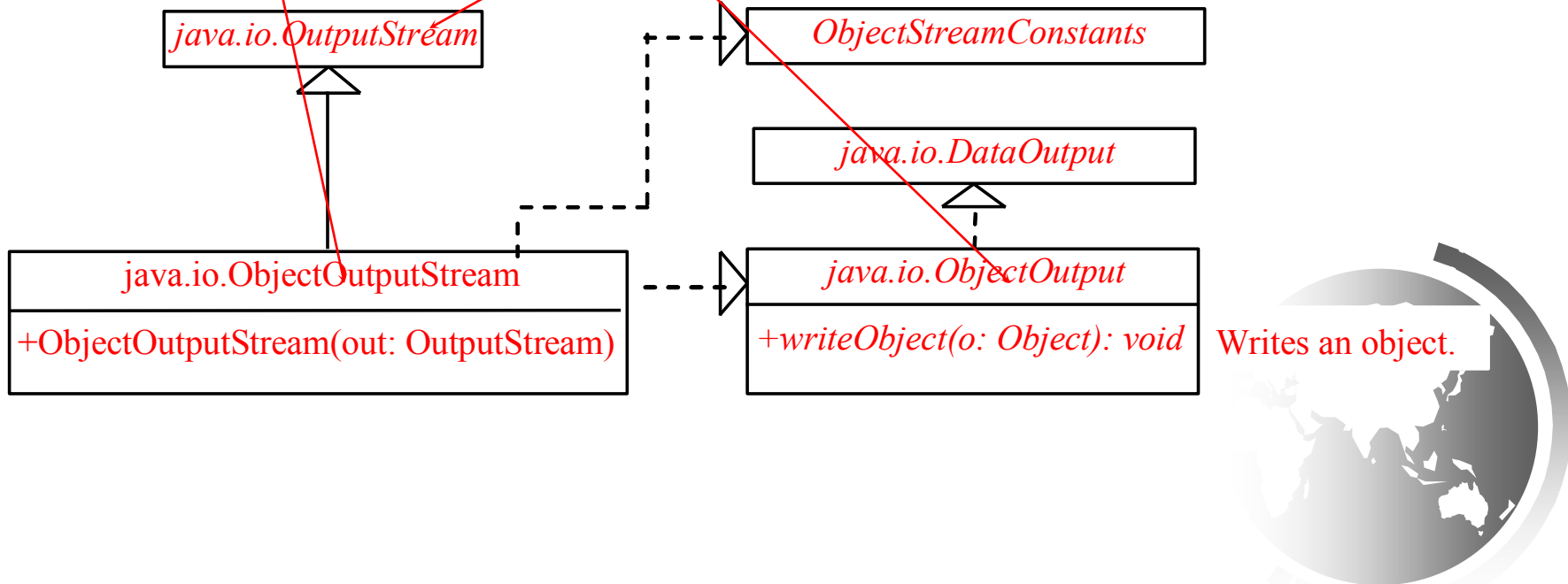
ObjectInputStream

ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.



ObjectOutputStream

ObjectOutputStream extends OutputStream and implements ObjectOutputStreamConstants.



Using Object Streams

You may wrap an `ObjectInputStream/ObjectOutputStream` on any `InputStream/OutputStream` using the following constructors:

```
// Create an ObjectInputStream
```

```
public ObjectInputStream(InputStream in)
```

```
// Create an ObjectOutputStream
```

```
public ObjectOutputStream(OutputStream out)
```



TestObjectOutputStream

Run

TestObjectInputStream

Run

The `Serializable` Interface

Not all objects can be written to an output stream. Objects that can be written to an object stream is said to be *serializable*. A serializable object is an instance of the `java.io.Serializable` interface. So the class of a serializable object must implement `Serializable`.

The `Serializable` interface is a marker interface. It has no methods, so you don't need to add additional code in your class that implements `Serializable`.

Implementing this interface enables the Java serialization mechanism to automate the process of storing the objects and arrays.



The transient Keyword

If an object is an instance of Serializable, but it contains non-serializable instance data fields, can the object be serialized? The answer is no. To enable the object to be serialized, you can use the transient keyword to mark these data fields to tell the JVM to ignore these fields when writing the object to an object stream.

```
private transient Date date = new Date();  
// date is Serializable → transient output stream
```



The transient Keyword, cont.

Consider the following class:

```
public class Foo implements java.io.Serializable {  
    private int v1;  
    private static double v2;  
    private transient A v3 = new A();  
}  
class A { } // A is not serializable
```

— static ကို serialize Output-Stream မှာ

When an object of the Foo class is serialized, only variable v1 is serialized. Variable v2 is not serialized because it is a static variable, and variable v3 is not serialized because it is marked transient. If v3 were not marked transient, a `java.io.NotSerializableException` would occur.

— အဲဒါက A ကို Serializable မဟုတ်ဘူး

Serializing Arrays

An array is serializable if all its elements are serializable. So an entire array can be saved using `writeObject` into a file and later restored using `readObject`. Here is an example that stores an array of five int values and an array of three strings, and reads them back to display on the console.



TestObjectStreamForArray

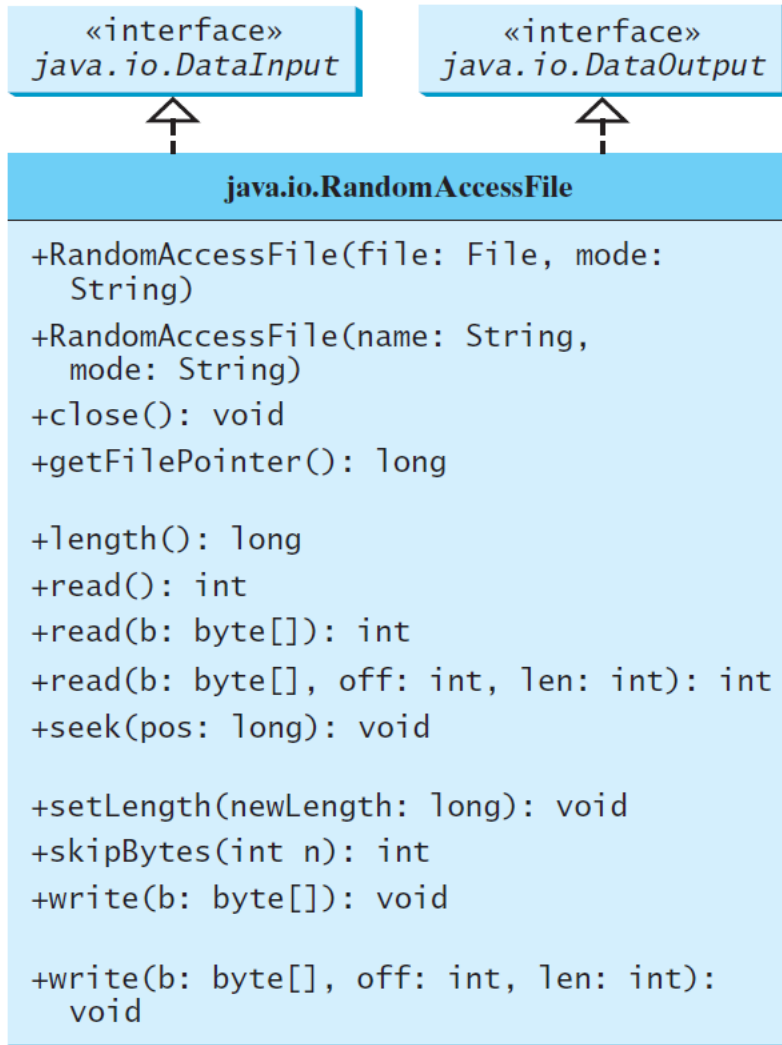
Run

Random Access Files

All of the streams you have used so far are known as *read-only* or *write-only* streams. The external files of these streams are *sequential* files that cannot be updated without creating a new file. It is often necessary to modify files or to insert new records into files. Java provides the `RandomAccessFile` class to allow a file to be read from and write to at random locations.



RandomAccessFile



Creates a `RandomAccessFile` stream with the specified `File` object and mode.

Creates a `RandomAccessFile` stream with the specified file name string and mode.

Closes the stream and releases the resource associated with it.

Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.

Returns the number of bytes in this file.

Reads a byte of data from this file and returns `-1` at the end of stream.

Reads up to `b.length` bytes of data from this file into an array of bytes.

Reads up to `len` bytes of data from this file into an array of bytes.

Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.

Sets a new length for this file.

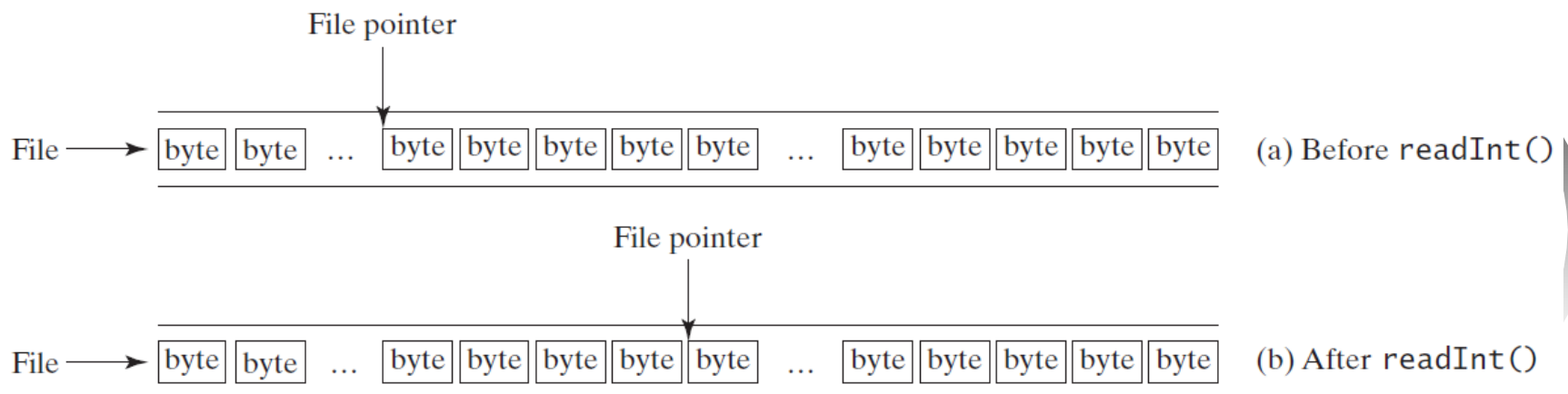
Skips over `n` bytes of input.

Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.

Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

File Pointer

A random access file consists of a sequence of bytes. There is a special marker called *file pointer* that is positioned at one of these bytes. A read or write operation takes place at the location of the file pointer. When a file is opened, the file pointer sets at the beginning of the file. When you read or write data to the file, the file pointer moves forward to the next data. For example, if you read an int value using `readInt()`, the JVM reads four bytes from the file pointer and now the file pointer is four bytes ahead of the previous location.



RandomAccessFile Methods

Many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`. For example, `readInt()`, `readLong()`, `writeDouble()`, `readLine()`, `writeInt()`, and `writeLong()` can be used in data input stream or data output stream as well as in `RandomAccessFile` streams.



RandomAccessFile Methods, cont.

`void seek(long pos) throws IOException;`

Sets the offset from the beginning of the RandomAccessFile stream to where the next read or write occurs.

`long getFilePointer() throws IOException;`

Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.



RandomAccessFile Methods, cont.

`long length() IOException`

Returns the length of the file.

`final void writeChar(int v) throws
IOException`

(เขียนตัวเดียว)

Writes a character to the file as a two-byte Unicode, with the high byte written first.

`final void writeChars(String s)
throws IOException`

(เขียนหลายตัว)

Writes a string to the file as a sequence of characters.



RandomAccessFile Constructor

```
RandomAccessFile raf =  
    new RandomAccessFile("test.dat", "rw");  
// allows read and write
```

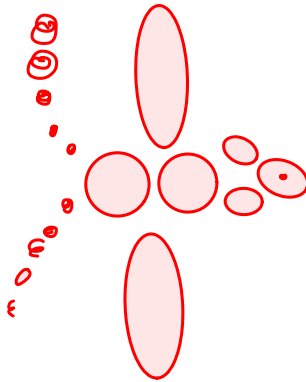
read write

```
RandomAccessFile raf =  
    new RandomAccessFile("test.dat", "r");  
// read only
```

read



A Short Example on RandomAccessFile



TestRandomAccessFile

Run