UBC OKANAGAN

COSC 121
Computer Programming II

Binary I/O

Part 2/2

Dr. Mostafa Mohamed

Acknowledgement: The slides mainly rely on the textbook of Y. D. Liang titled “Introduction to Java Programming, 10th Ed.”, Pearson Edu. Inc.

Outline

Previously:

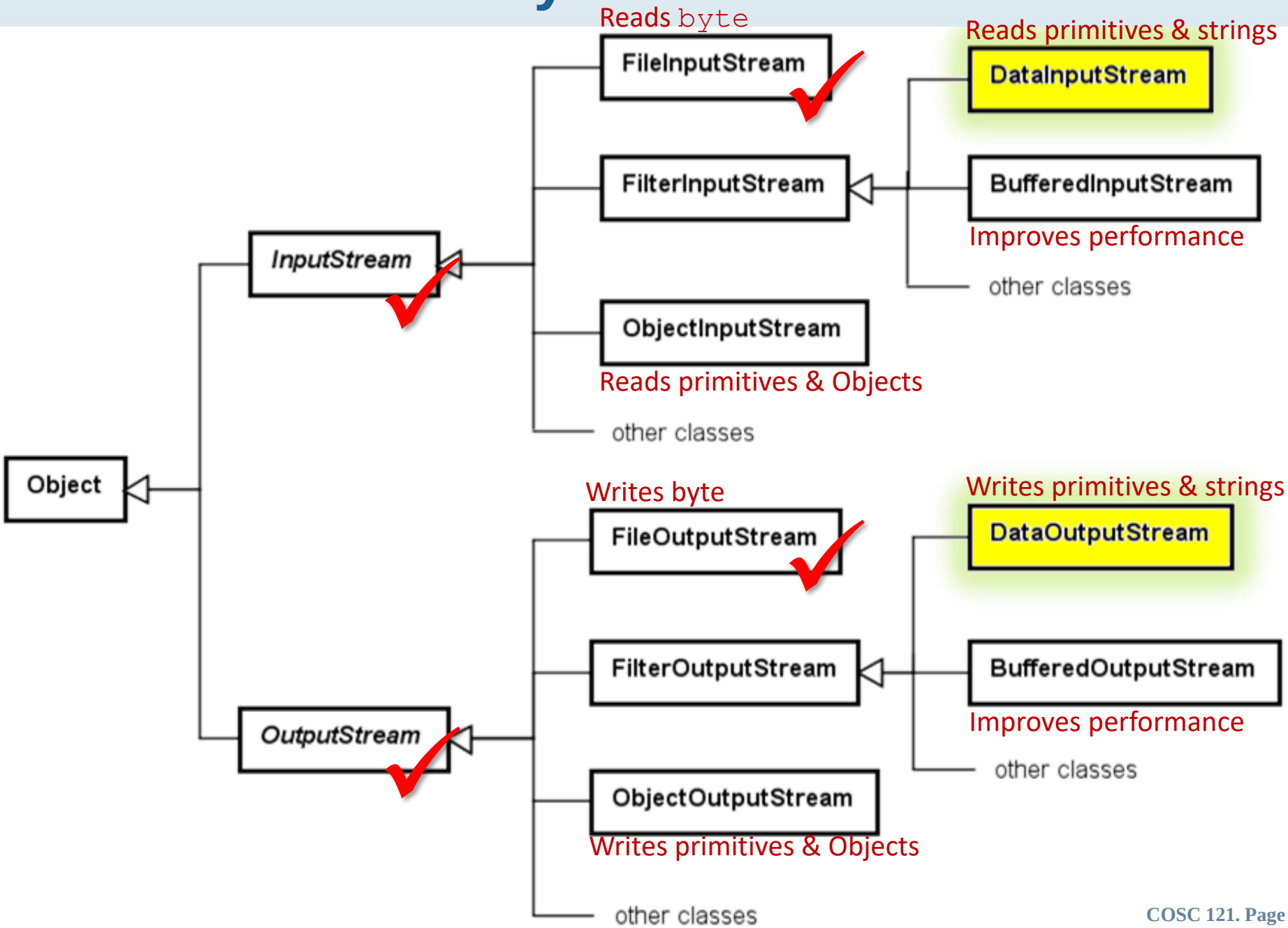
- Text I/O vs Binary I/O
- Binary I/O classes
  - `InputStream` / `OutputStream`
  - `FileInputStream` / `FileOutputStream`

Today:

- More Binary I/O classes:
  - `DataInputStream` / `DataOutputStream`
  - `ObjectInputStream` / `ObjectOutputStream`
    - `Serializable, transient.`
- Improving I/O Performance
  - `BufferedInputStream / BufferedOutputStream`

DataInputStream / DataOutputStream

Binary I/O Classes

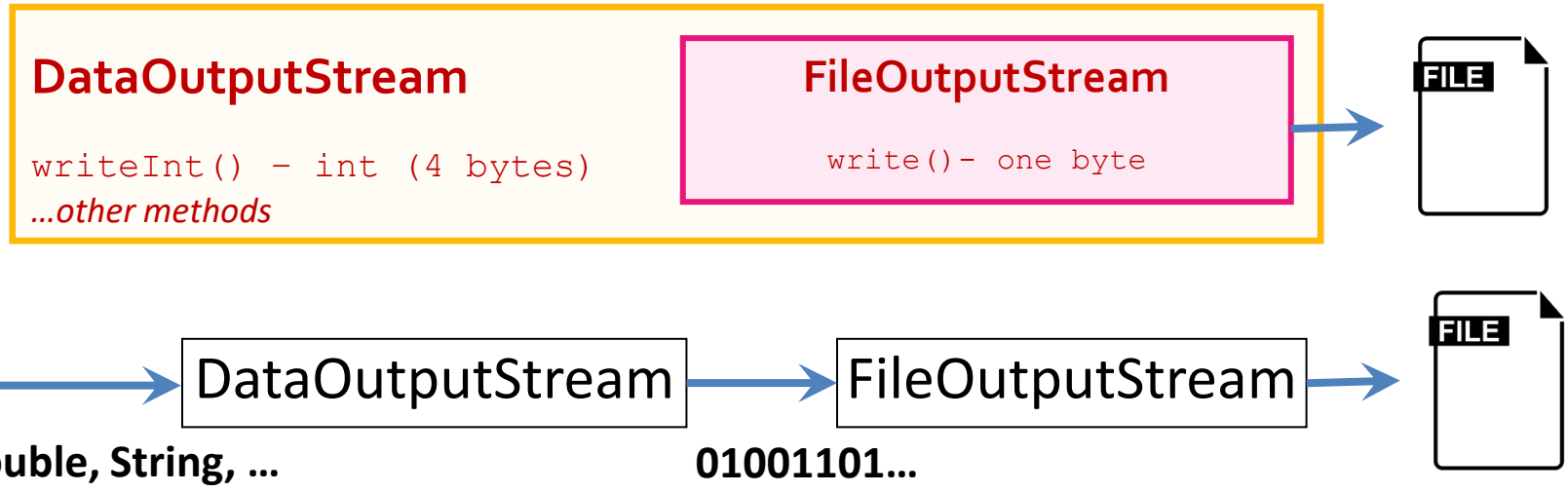


DataOutputStream

Wraps around FileOutputStream

- **Converts primitive types** or **strings → bytes and output the bytes to the stream**

```
DataOutputStream out =
    new DataOutputStream(new FileOutputStream("file.dat"));
```



https://docs.oracle.com/javase/8/docs/api/?java/io/FileOutputStream.html

DataOutputStream, cont.

Methods: All methods of `OutputStrem` in addition to:

- **`+writeBoolean(b:boolean): void`**

- **`+writeByte(v:int): void`**
- **`+writeShort(v:short): void`**
- **`+writeInt(v:int): void`**
- **`+writeLong(v:long): void`**
- **`+writeFloat(v:float): void`**
- **`+writeDouble(v:double): void`**

- **`+writeChar(c:char): void`**
  - Writes c character composed of 2 bytes (Unicode).

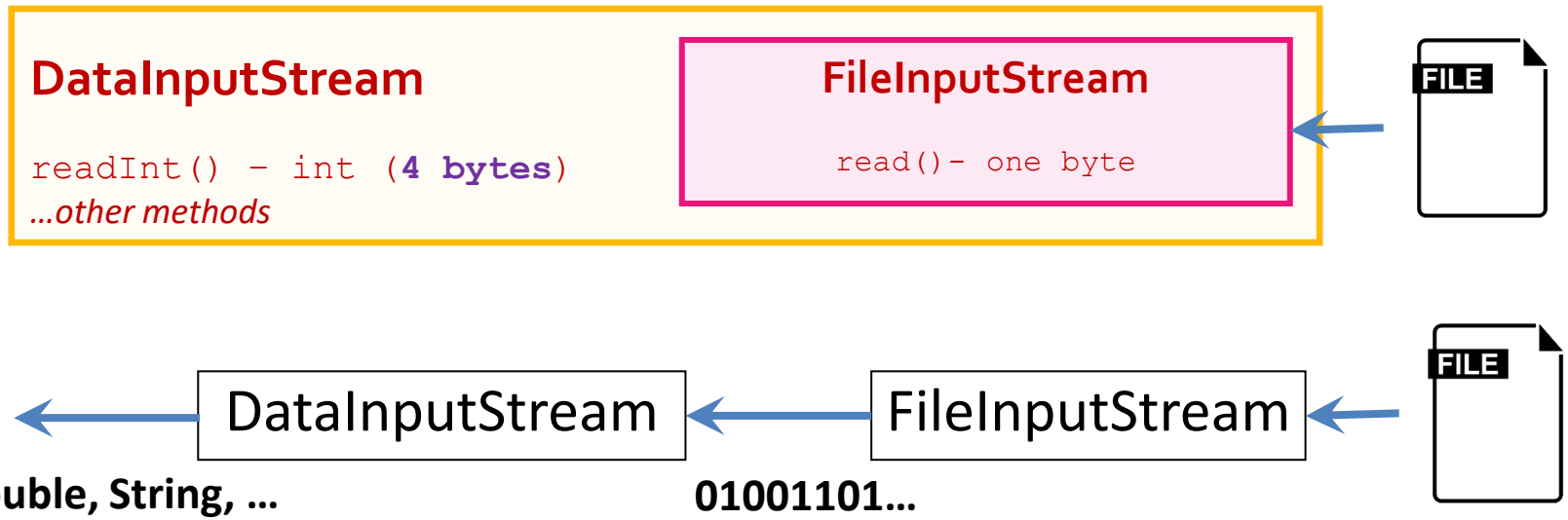
- **`+writeUTF(s:String): void`**
  - Writes the length of `s`, then `s` in modified-UTF format.
    - Must use readUTF to read.`
    - **`Unicode transformation format (UTF): Unicode to a unique byte sequence.`**

DataInputStream

Wraps around FileInputStream

- **It read bytes from the input stream** → interprets binary data as **primitive types** or **strings**.

```
DataInputStream in =
    new DataInputStream(new FileInputStream("file.dat"));
```



Stream chaining is a way of connecting several streams to get the data in the form required

DataInputStream, cont.

Methods: All methods of `InputStrem` in addition to:

- `+readBoolean():` Reads 1 byte, returns true if that byte is nonzero, false if it is zero.
- `+readByte(): byte` reads and returns 1 bytes
- `+readShort(): short` reads 2 bytes, returns a short value
- `+readInt(): int` reads 4 bytes, returns an int value
- `+readLong(): long` reads 8 bytes, returns a long value
- `+readFloat(): float` reads 4 bytes, returns a float
- `+readDouble():double` reads 8 bytes, returns a double
- `+readChar(): char` reads 2 bytes, returns a char in Unicode
- `+readUTF(): String` reads a string in modified-UTF format (UTF uses different lengths to represent different characters).

Exercise

Q1) Write Java code to write one int value into a file, f.dat, using binary I/O.

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("c:/1/f.dat"));
out.writeInt(222);
out.close();
```

Q2) Write code to display the contents of f.dat on the screen.

```
DataInputStream in = new DataInputStream(new FileInputStream("c:/1/f.dat"));
System.out.println(in.readInt());
in.close();
```

Q3) Try Q2 again, but when reading don't read data as integer! Instead, try reading data as bytes, etc. Will there be an error? What is the output?

- No errors if we read into byte, but output is not same as Q2
- Error if we read into double. WHY?

```
DataInputStream in = new DataInputStream(new FileInputStream("c:/2/f.dat"));
System.out.println(in.readByte());
System.out.println(in.readByte());
System.out.println(in.readByte());
System.out.println(in.readByte());
in.close();
```

Exercise

Q1) Write Java code to write 100 double values created randomly (from 0 to 100000) into a file, named numbers.dat, using binary I/O. If the file doesn't exist, you need to create it, otherwise append new data to it if it already exists.

```
DataOutputStream out = new DataOutputStream(
        new FileOutputStream("numbers.dat", true));
for (int i = 0; i < 100; i++)
    out.writeDouble(Math.random() * 100000);
out.close();
```

Q2) Write code to display 100 doubles from numbers.dat.

```
DataInputStream in = new DataInputStream(
        new FileInputStream("numbers.dat"));
for (int i = 0; i < 100; i++)
    System.out.println(in.readDouble());
in.close();
```

Q3) Try Q2 again, but when reading don't read data as doubles! Instead, try reading data as integers, bytes, etc. will there be an error? What is the output?

`DataOutputStream` and Strings

How do we know the end of a string?

`+writeUTF(s:String): void`

- Writes the number of bytes taken by `s` in the first two bytes, then `s` itself in modified-UTF8 format.
  - modified UTF8 stores a character using 1 or more bytes, depending on the character.
    - ASCII characters are stored in 1 byte. Other characters may require more than one byte.
    - The initial bits of a UTF-8 text indicate the length of the character.

- **Example**:

  `writeUTF("ABCDEF")`

  writes eight bytes (i.e., Hex value: **00 06 41 42 43 44 45 46**), because the first two bytes store the number of characters in the string.

Aside: modified UTF format (source: Java Doc)

- This first two bytes are called the *UTF length* and specify the number of additional bytes to be read. These bytes are then converted to characters by considering them in groups.
- The length of each group is computed from the value of the first byte of the group

Encoding of UTF-8 values

Value	Byte	Bit Values							
		7	6	5	4	3	2	1	0
\u0001 to \u007F	1	0	bits 6-0						
\u0000, \u0080 to \u07FF	1	1	1	0	bits 10-6				
	2	1	0	bits 5-0					
\u0800 to \uFFFF	1	1	1	1	0	bits 15-12			
	2	1	0	bits 11-6					
	3	1	0	bits 5-0					

- After every group has been converted to a character by this process, the characters are gathered, in the same order in which their corresponding groups were read from the input stream, to form a String, which is returned."

Exercise

Write a program that writes into a file the names of two students, John and Jim, and their scores, 85.5 and 185.5 and then read them back. Use Binary I/O.

- **CAUTION**: You have to read the data in the same order and format in which they are stored.

What happens if you use readDouble() to read all data (i.e., “John” 85.5 “Jim" 185.5)?

Checking End of File

So how do you check the end of a file?

- Remember that DataInputStream is a descendent of InputStream → use available() method to check the stream, input.available() == 0 indicates the end of file.

```
// WRITE student test scores to the file
DataOutputStream out =
        new DataOutputStream(new FileOutputStream("temp.dat"));
out.writeUTF("John"); out.writeDouble(85.5);
out.writeUTF("Jim");  out.writeDouble(185.5);
out.close();

// READ: must use the SAME ORDER
DataInputStream in =
        new DataInputStream(new FileInputStream("temp.dat"));
System.out.println(in.available());
System.out.println(in.readUTF() + " " + in.readDouble());
System.out.println(in.available());
System.out.println(in.readUTF() + " " + in.readDouble());
System.out.println(in.available());
in.close();
```

Output

```
27
John 85.5
13
Jim 185.5
0
```

Exercise

Write Java code to display the contents of a file named values.dat on the screen. The file contains only double values. The number of values is unknown.

```
DataInputStream in = new DataInputStream(
        new FileInputStream("values.dat"));
while(in.available() > 0)
    System.out.println(in.readDouble());
in.close();
```

Efficiency?

available() method is not very efficient method (i.e. takes some time to return result)

An alternative way is to keep reading until your reader object fails to get any data.

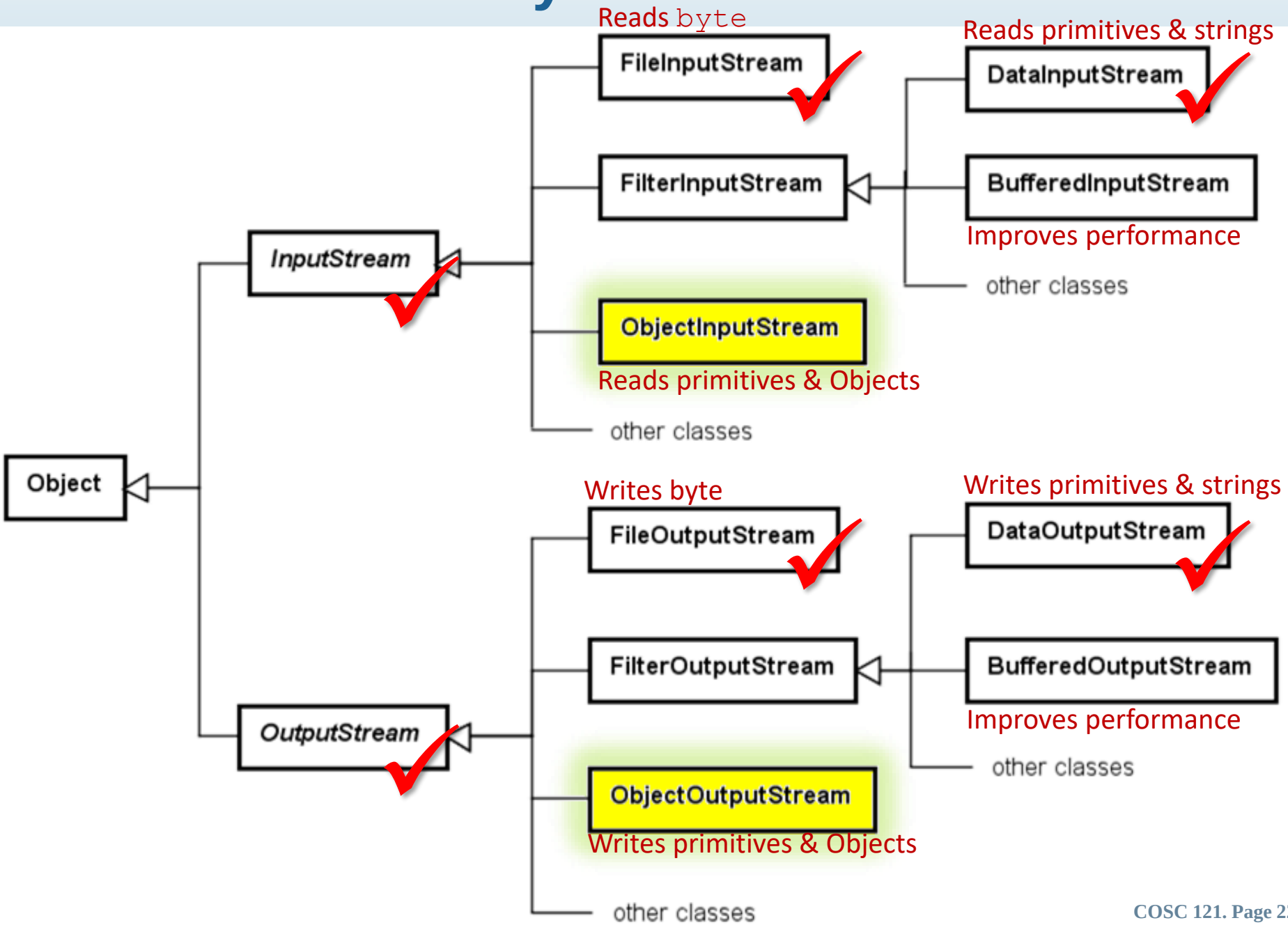
If you try to read past the end-of-file, DataInputStream will through an exception EOFException. We can use this in our code as follows:

```
DataInputStream in = new DataInputStream(new FileInputStream("t.dat"));
boolean EOF = false;
while (!EOF) {
    try {
        System.out.println(in.readDouble());
    } catch (EOFException e) {
        EOF = true;
    }
}
in.close();
```

ObjectInputStream / ObjectOutputStream

https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html
https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html

Binary I/O Classes



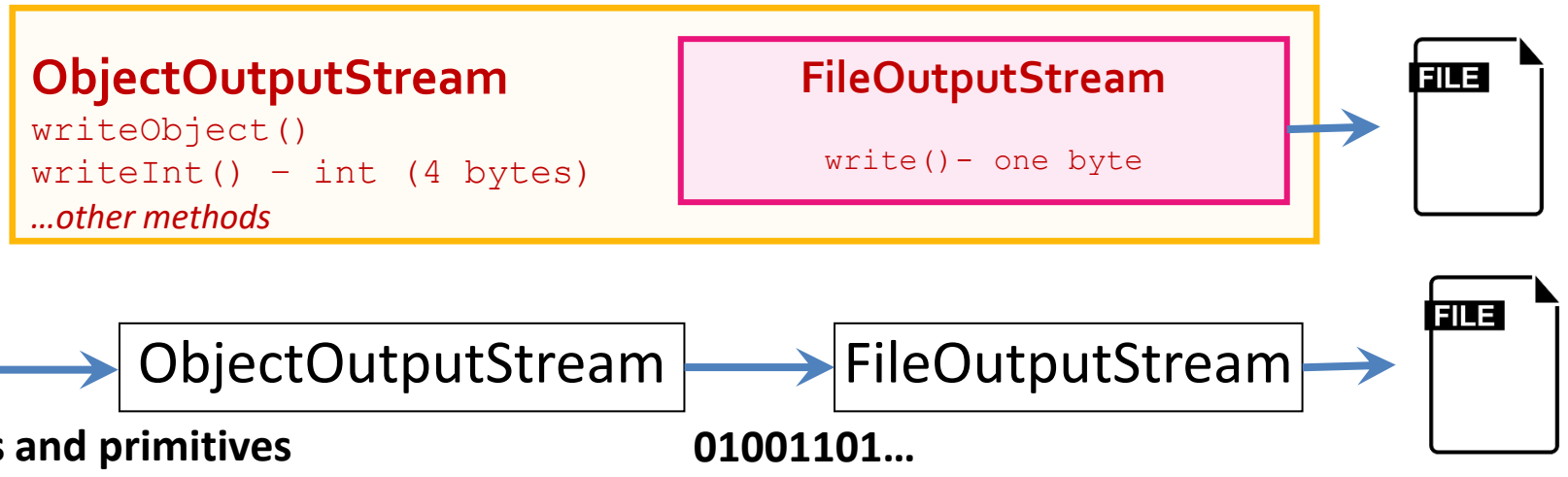
ObjectOutputStream

Can be used to write *serializable* objects and primitive data

Wraps FileOutputStream.

- **Converts primitive types** or **strings → bytes and output the bytes to the stream**

```
ObjectOutputStream out =
    new ObjectOutputStream(new FileOutputStream("file.dat"));
```



Methods: All methods of `DataOutputStrem` in addition to:

```
+writeObject(b:Object): void
```

ObjectInputStream

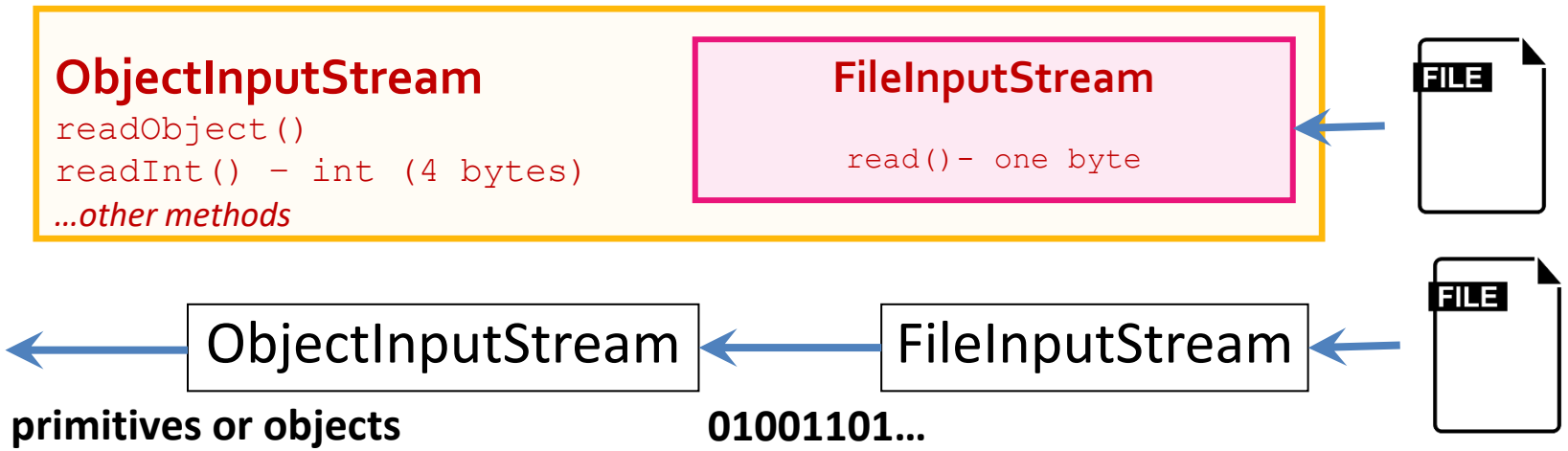
Can be used to read *serializable* objects and primitive data

Wraps FileInputStream

- **It read bytes from the input stream** → converts them to **primitive types**, **strings, objects**

ObjectInputStream in =

new ObjectInputStream(**new** FileInputStream(**"file.dat"**));



Methods: All methods of `DataInputStrem` in addition to:

`+readObject():Object` (Throws a checked exception: ClassNotFoundException)

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.

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.

https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html

Example1

```
import java.io.Serializable;
public class Robot implements Serializable{
    int x = 10, y = 20;
    String location = "Canada";
    public void amethod(){}
}
```

Output

```
My Robot, Canada, (10, 20)
```

```
import java.io.*;
public class TestObjStream {
    public static void main(String[] args) throws Exception {
        // WRITE student test scores to the file
        ObjectOutputStream out =
                new ObjectOutputStream(new FileOutputStream("temp.dat"));
        out.writeUTF("My Robot"); out.writeObject(new Robot());
        out.close();
        // READ: must use the SAME ORDER
        ObjectInputStream in =
                new ObjectInputStream(new FileInputStream("temp.dat"));
        String s = in.readUTF();
        Robot r = (Robot)in.readObject();
        System.out.println(s + ", " + r.location + ", (" + r.x + ", " + r.y + ")");
        in.close();
    }
}
```

Example2: Serializing Arrays

***An array is serializable if all its elements are serializable*.**

Example:

```
int[] numbers = { 1, 2, 3, 4, 5 };
String[] strings = { "John", "Susan", "Kim" };

// Write arrays to the object output stream
ObjectOutputStream out=new ObjectOutputStream(new FileOutputStream("array.dat", true));
out.writeObject(numbers);
out.writeObject(strings);
out.close();

//Read arrays from the object output stream
ObjectInputStream in = new ObjectInputStream(new FileInputStream("array.dat"));
int[] newNumbers = (int[]) (in.readObject()); //read an array-of-int object
String[] newStrings = (String[]) (in.readObject()); //read an array-of-string object
in.close();

// Display arrays
for (int i = 0; i < newNumbers.length; i++)
    System.out.print(newNumbers[i] + " ");
System.out.println();

for (int i = 0; i < newStrings.length; i++)
    System.out.print(newStrings[i] + " ");
```

The `transient` Keyword

If a serializable object contains non-serializable data fields, can the object be serialized?

Answer:

NO. To enable the object to be serialized, you have two options:

1. Make all data fields serializable

   i.e. their classes must implement serializable

2. Use the `transient` keyword with these fields so that the JVM **ignores** them when sending the object to the output stream.

Example

```
import java.io.Serializable;
public class Robot implements Serializable{
    int x = 10, y = 20;
    transient String location = "Canada";
    public void amethod(){}
}
```

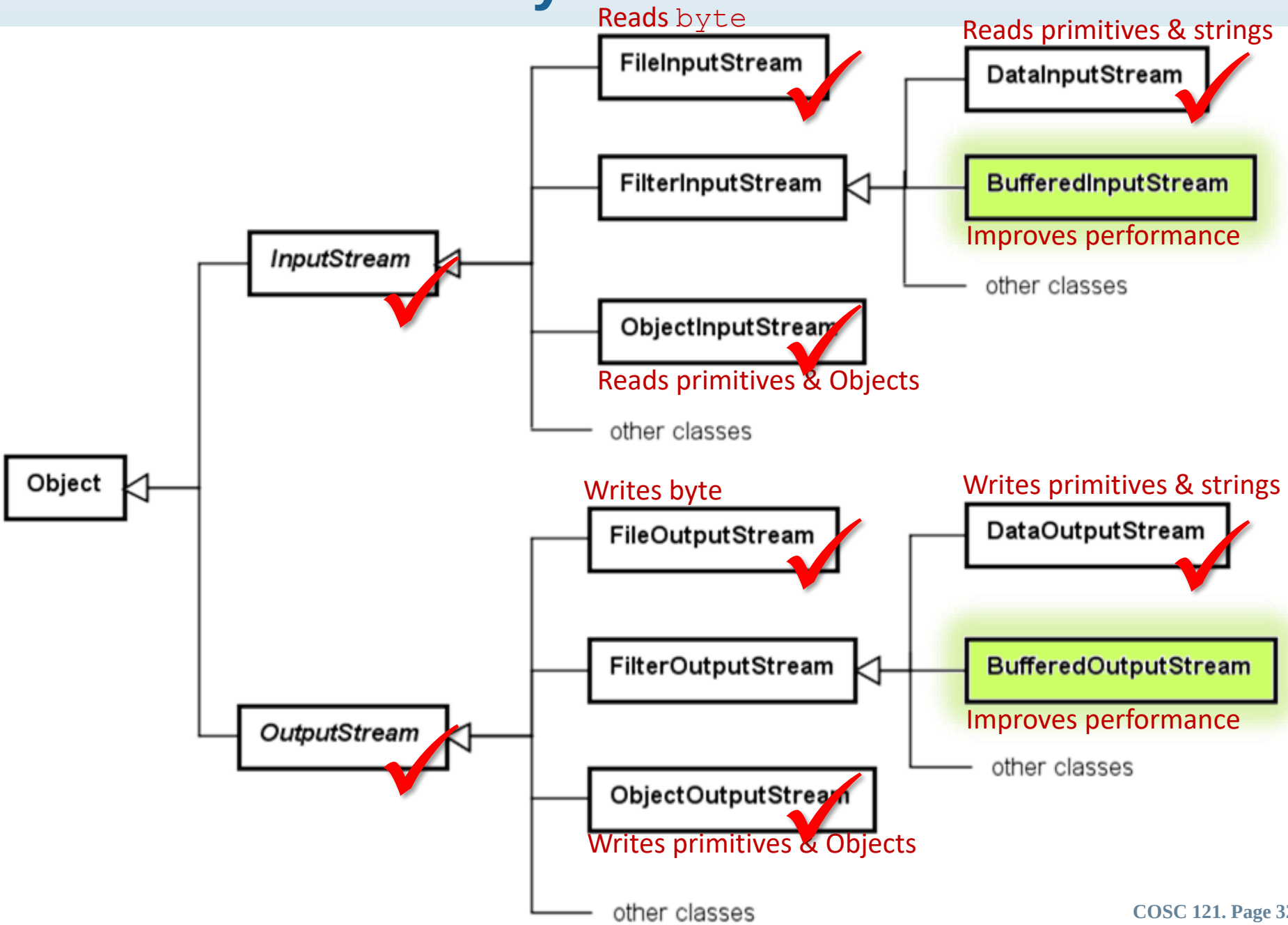
Output

```
My Robot, null, (10, 20)
```

```
import java.io.*;
public class TestObjStream {
    public static void main(String[] args) throws Exception {
        // WRITE student test scores to the file
        ObjectOutputStream out =
                new ObjectOutputStream(new FileOutputStream("temp.dat"));
        out.writeUTF("My Robot"); out.writeObject(new Robot());
        out.close();
        // READ: must use the SAME ORDER
        ObjectInputStream in =
                new ObjectInputStream(new FileInputStream("temp.dat"));
        String s = in.readUTF();
        Robot r = (Robot)in.readObject();
        System.out.println(s + ", " + r.location + ", (" + r.x + ", " + r.y + ")");
        in.close();
    }
}
```

Improving I/O Performance

Binary I/O Classes



BufferedInputStream / BufferedOutputStream

Used to speed up I/O by reducing the number of disk reads and writes.

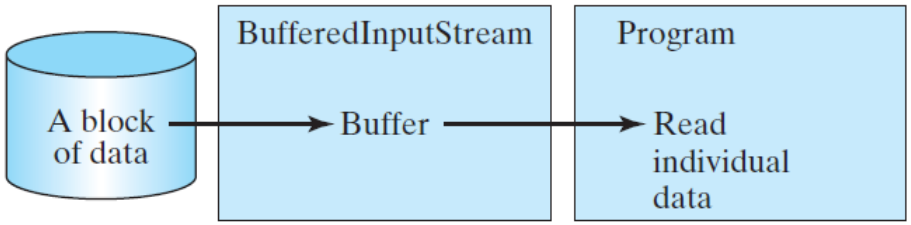
Methods:

- All methods from *InputStream* / *OutputStream*

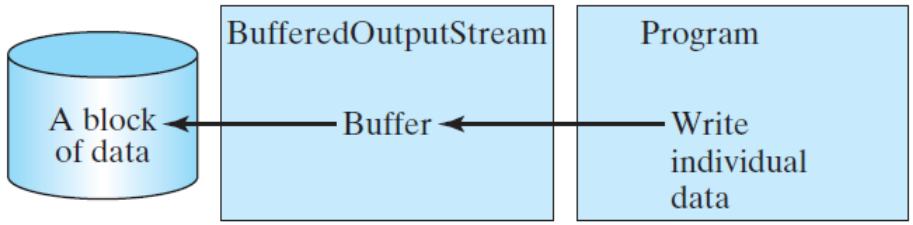
Constructors

- public BufferedInputStream(InputStream in) // bufferSize=512
- public BufferedInputStream(InputStream in, int bufferSize)

- public BufferedOutputStream(OutputStream in) //bufferSize=512
- public BufferedOutputStream(OutputStream in, int bufferSize)



(a)



(b)

Example: Using BufferedInputStream

```
BufferedInputStream input =                                    /*read byte*/
        new BufferedInputStream(new FileInputStream("temp.dat"));
```


```
DataInputStream input = new DataInputStream(          /*read primitive and string*/
        new BufferedInputStream(new FileInputStream("temp.dat")));
```


```
ObjectInputStream input = new ObjectInputStream(  /*read primitive and objects*/
        new BufferedInputStream(new FileInputStream("temp.dat")));
```

Let’s see if you got this right!

Exercise

Suppose you want to back up a huge file (e.g., a 10-GB AVI file) to a CD-R. You can achieve it by splitting the file into smaller pieces and backing up these pieces separately.

Write a utility program that splits a large file into smaller ones **SourceFile.1**, **SourceFile.2**, . . . , **SourceFile.n**, where **n** is **numberOfPieces** and the output files are about the same size. Use this header:

void splitFile(String sourceFile, int splitFileSize)

Tips

Using FileOutputStream with PrintWriter

An instance of **FileOutputStream** can be used as an argument to construct a **PrintWriter**. You can create a **PrintWriter** to append text into a file.

```
FileOutputStream f = new FileOutputStream("temp.dat", true);
PrintWriter out = new PrintWriter(f);
out.print("append this data");
out.close();
f.close();
```

Interested in more?

Java offers more functionality to programmers. For example:

`RandomAccessFile` class

- allows a file to be read from and write to at random locations.
  - Note that all of the streams you have used so far are read-only or write-only streams. The external files of these streams are sequential files that cannot be updated without creating a new file.

`CipherOutputStream` / `CipherInputStream` classes

- encrypt/decrypt data before writing/reading them

`ZipOutputStream` / `ZipInputStream` classes

- writes / reads files in the ZIP file format.

`CheckedOutputStream` / `CheckedInputStream` classes

- writes / reads files with checksum (to verify data integrity).

And more…!!!