Chapter - 2
Java Input/output

Outline

- Introduction
- Streams
- Types of Streams
  - Byte based Streams
  - Character based streams
  - Predefined streams
- Reading Console Input
- Reading and Writing **text files**
- Reading and Writing **binary files**
- Java Serialization
  - Serializing objects

Introduction

- Whenever programs produce output, the output may be sent to either
  - the Java console, a text area, or some other GUI component.
    - ✓These destinations are transitory, in the sense that the output resides in the computer's primary memory and exists only as long as the program is running.
  - Or to **relatively permanent storage medium**
- A **file** is a collection of data stored on a disk or on some other relatively permanent storage medium.
- A file's existence **does not depend** on a running program.

Introduction…

- I/O = Input/output
- In this context, it is input to and output from programs
- **Input** refers to information or data read from some external source into a running program.
- **Output** refers to information or data written from a running program to some external destination.
- Input can be from keyboard, a file, or network connection etc.
- Output can be to display (screen) or a file
- Advantages of file I/O:
  - Permanent copy
  - Output from one program can be input to another
  - Input can be automated (rather than entered manually)

STREAMS

STREAM: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)

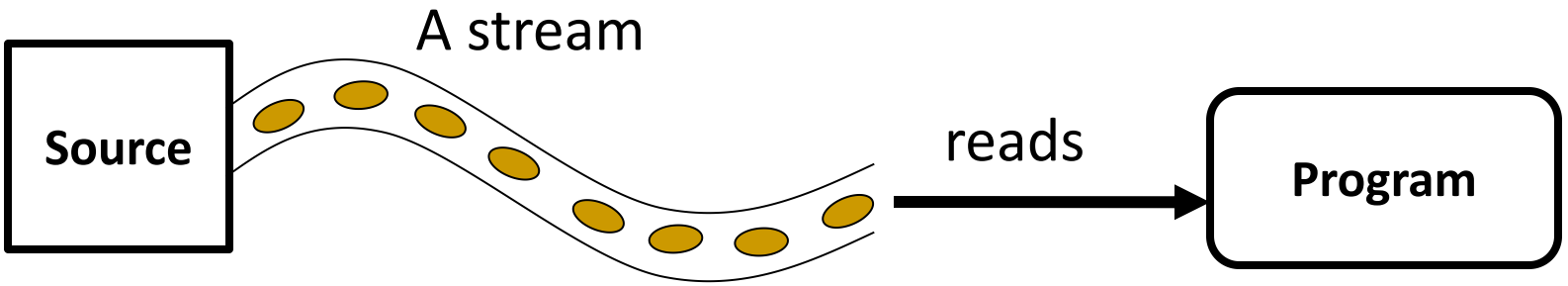
- It acts as a buffer between the data source and destination

- **Input stream**: a stream that provides input to a program
  - `System.in` is a standard input stream
- **Output stream**: a stream that accepts output from a program
  - `System.out` is a standard output stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

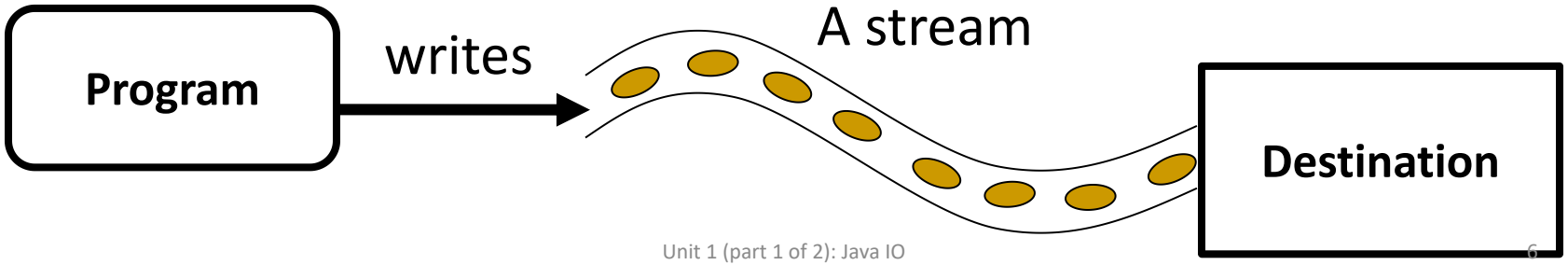
Streams

- Java input and output is based on the use of streams.

Reading from a stream



Writing to a stream



Types of Streams

- Java defines two types of streams: **byte based** and **character based**.

Byte Based

- Streams provide convenient way for handling input and output of **8-bit bytes** and are used for input/output of binary data.
- Files created by byte based streams are called **binary files**. These files are easily read by the computer but not humans.
- Binary data are not very portable (platform dependent).
- On some systems an integer might be 16 bits, and on others it might be 32 bits, so even if you know that a Macintosh binary file contains integers, that still won't make it readable by Windows/Intel programs.

Types of Streams…

Character Based

- Streams provide a convenient way for handling input and output of characters (**text files**).
- Files created using character based streams are called **text file.**
  - A **text file** is processed as a sequence of characters.
  - A text file created by a program on a Windows/Intel computer can be read by a Macintosh program.
    - Text files are portable

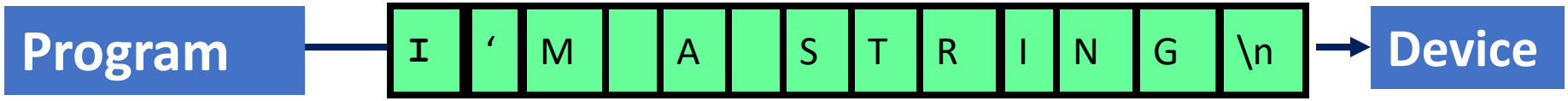
Note that: Text files are human readable files.

- They are universal and can be edited with many different programs such as NOTEPAD.

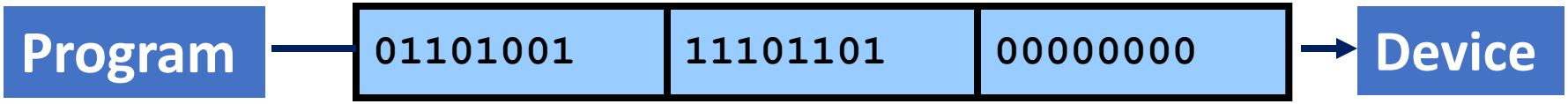
Types of Streams…

- JAVA distinguishes between 2 types of streams:

Text – streams, containing ‘**characters**‘



Binary Streams, containing **8bit** information



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.

Java IO and Streams

- The Java I/O package gives classes support for reading and writing data to and from different input and output sources including Arrays, files, strings, sockets, memory and other data sources.
- The **java.io** package provides more than 60 input/output classes (stream).
- These classes are used to manipulate **binary** and **text** files.
- Here, we will learn how to create files and how to perform input and output operations on their data using the Java classes designed specifically for this purpose.

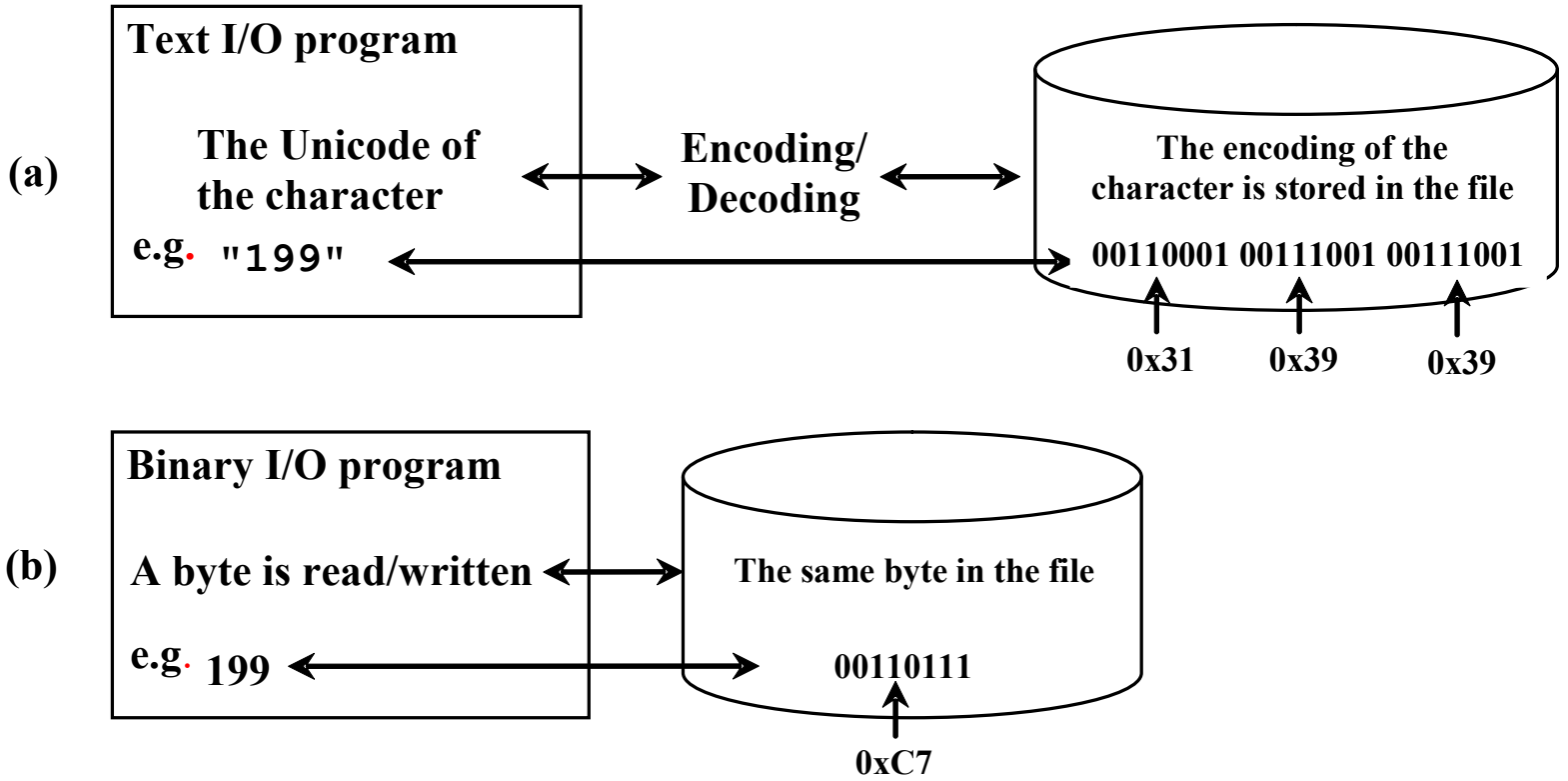
Text File vs. Binary File

- Although it is not technically precise and correct, you can imagine that a text file consists of a sequence of characters 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.
- You can check https://www.rapidtables.com/convert/number/decimal-to-hex.html for easy conversion of the numbers.

Binary I/O


(a)
Text I/O program
The Unicode of the character
e.g. "199"
Encoding/ Decoding
The encoding of the character is stored in the file
00110001 00111001 00111001
0x31
0x39
0x39
(b)
Binary I/O program
A byte is read/written
e.g. 199
The same byte in the file
00110111
0xC7

Java IO and Streams

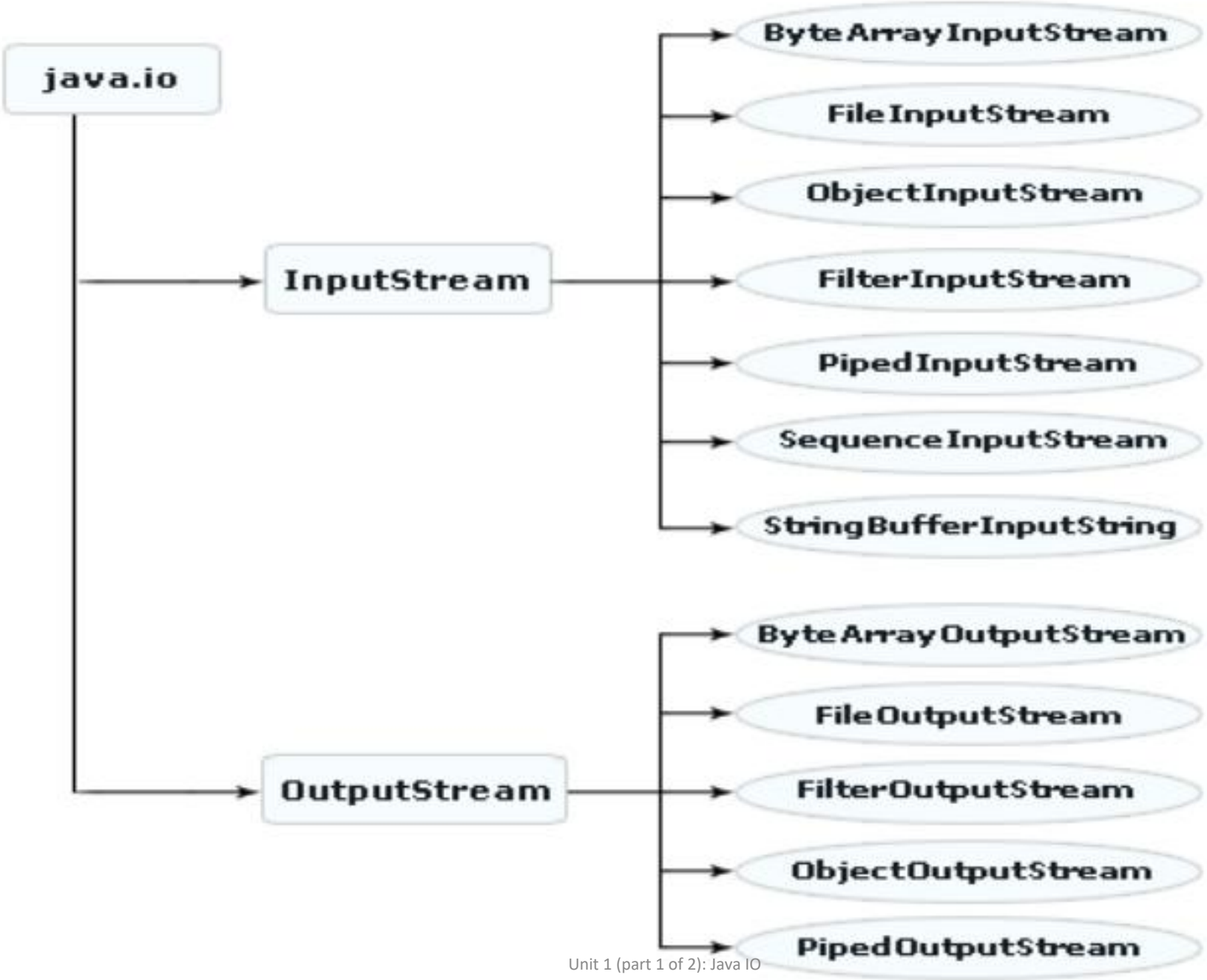
- Streams in JAVA are Objects.
- Having
  - ✓ 2 types of streams (text / binary) and
  - ✓ 2 directions (input / output)
- Results in 4 base-classes dealing with I/O:
  1. **InputStream:** byte-input
  2. **OutputStream:** byte-output
  3. **Reader:** text-input
  4. **Writer:** text-output

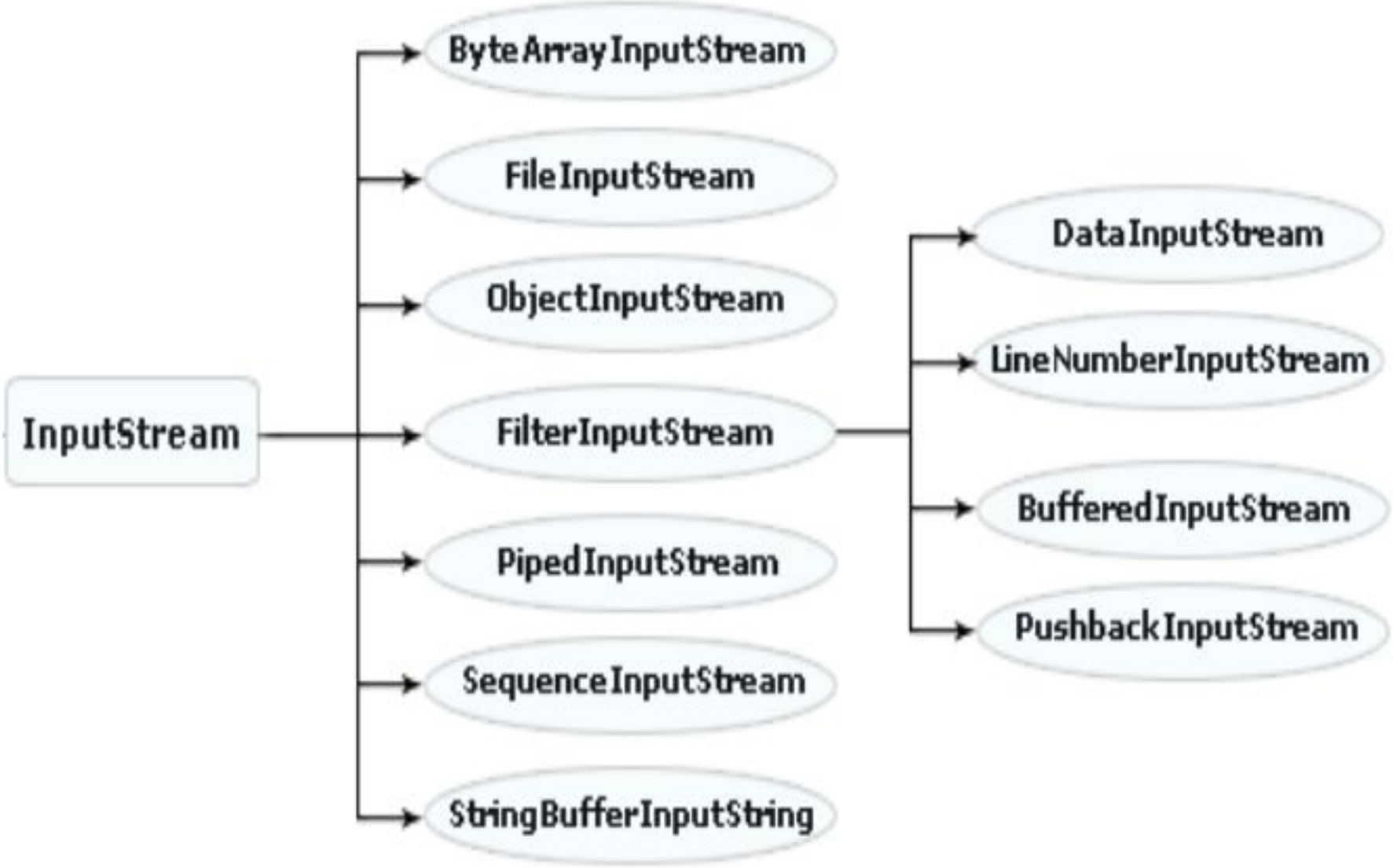
Java IO and Streams

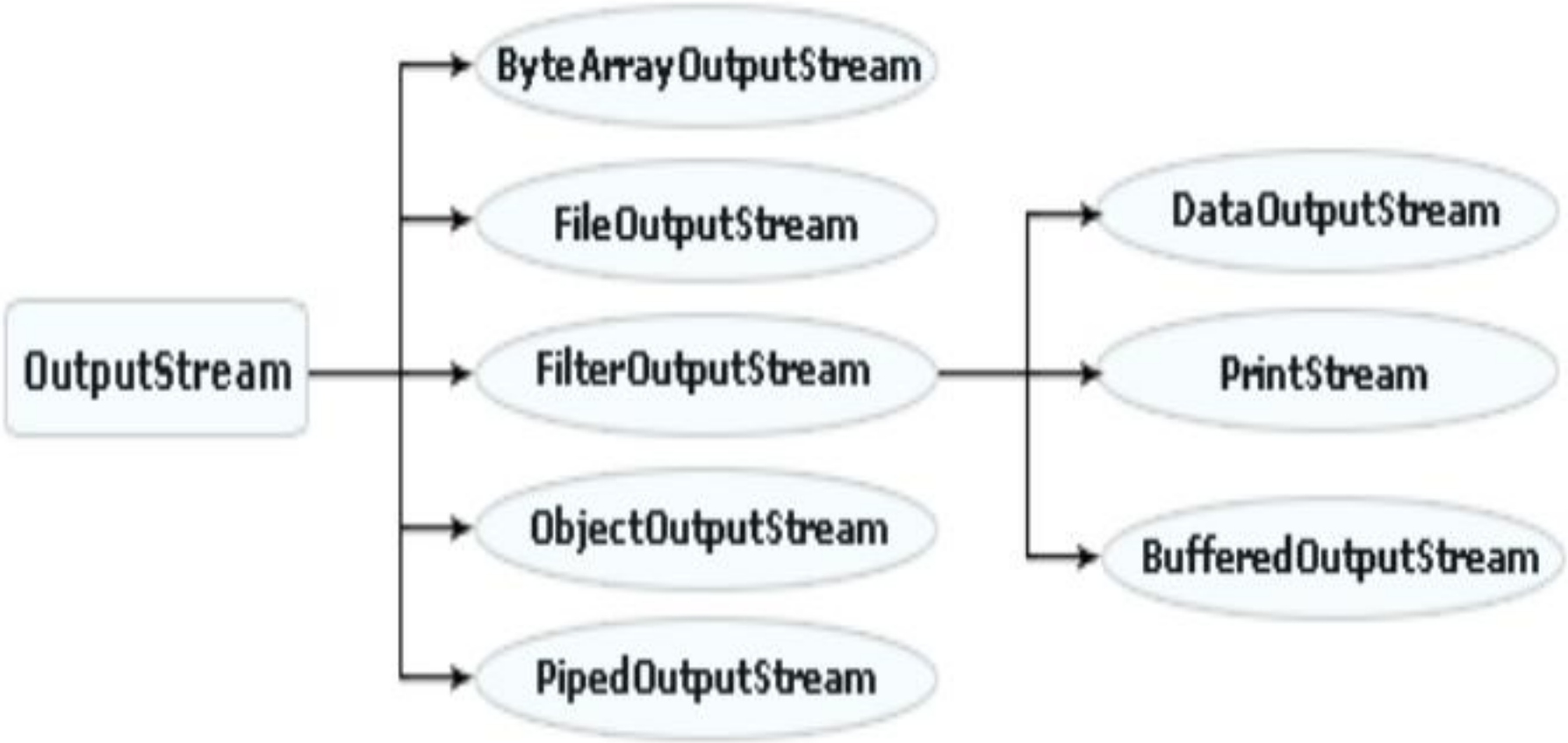
- Generally speaking,
  - **Binary files** are processed by ***subclasses*** *of* **InputStream** *and* **OutputStream**.
  - **Text files** are processed by ***subclasses*** of **Reader** and **Writer**
- Both of which are **streams** despite their names.
- Base classes are used to describe the basic functionality required.
- Derived classes provided the functionality for specific kinds of I/O environments.
  - Files, Pipes, Networks, IPC, etc.

Byte based Streams

- Byte based streams classes are defined by two class hierarchies. At the top are two **abstract classes**:
    1. **InputStream**
    2. **OutputStream**
- Both of these classes are **abstract** and have several **concrete subclasses**, that handle the difference between various sources, such as disk, files or sockets.
- The abstract classes **InputStream** and **OutputStream** declare several **abstract methods** that all subclasses implement.
- Two of the most important are **read( )** and **write( )**, which respectively read and write single byte.


java.io
InputStream
ByteArrayInputStream
FileInputStream
ObjectInputStream
FilterInputStream
PipedInputStream
SequenceInputStream
StringBufferInputString
OutputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream


InputStream
ByteArrayInputStream
FileInputStream
ObjectInputStream
FilterInputStream
PipedInputStream
SequenceInputStream
StringBufferInputString
DataInputStream
LineNumberInputStream
BufferedInputStream
PushbackInputStream


OutputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
DataOutputStream
PrintStream
BufferedOutputStream

Reading/Writing to a Binary File

Writing data to a file requires **three steps**:

1. Connect an output stream to the file.
2. Write text data into the stream, possibly using a loop.
3. Close the stream.

Step 1:

- The output stream serves as a channel between the program and a named file.
- The output stream opens the file and gets it ready to accept data from the program.
- If the file already exists, then opening the file will destroy any data it previously contained.
- If the file doesn't yet exist, then it will be created from scratch.

Reading/Writing to a Binary File…

- Step 2
  - Once the file is open, the next step is to write the text to the stream, which passes the text on to the file.
  - This step may require a loop that outputs one line of data on each iteration.
- Step 3
  - Finally, once all the data have been written to the file, the stream should be closed. This also has the effect of closing the file.
  - Even though Java will close any open files and streams when a program terminates normally, it is good programming practice to close the file yourself with a close() statement.
  - This reduces the chances of damaging the file if the program terminates abnormally.

Reading/Writing from a Text File

- Three steps to reading data from a file:
  1. Connect an input stream to the file.
  2. Read the text data using a loop.
  3. Close the stream.

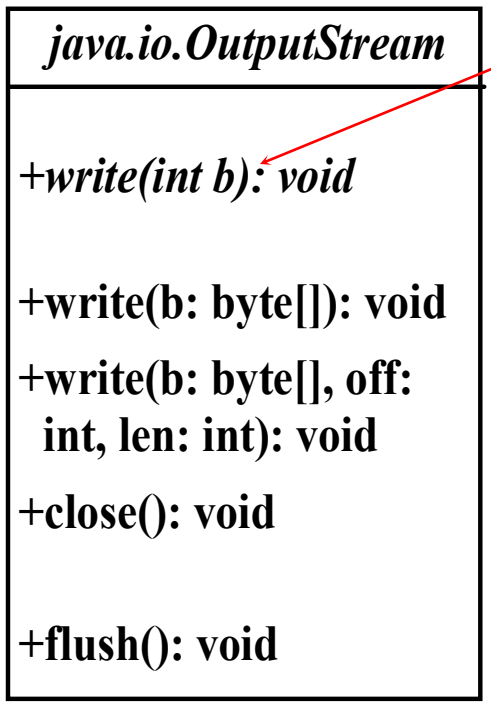
InputStream

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.
	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.



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

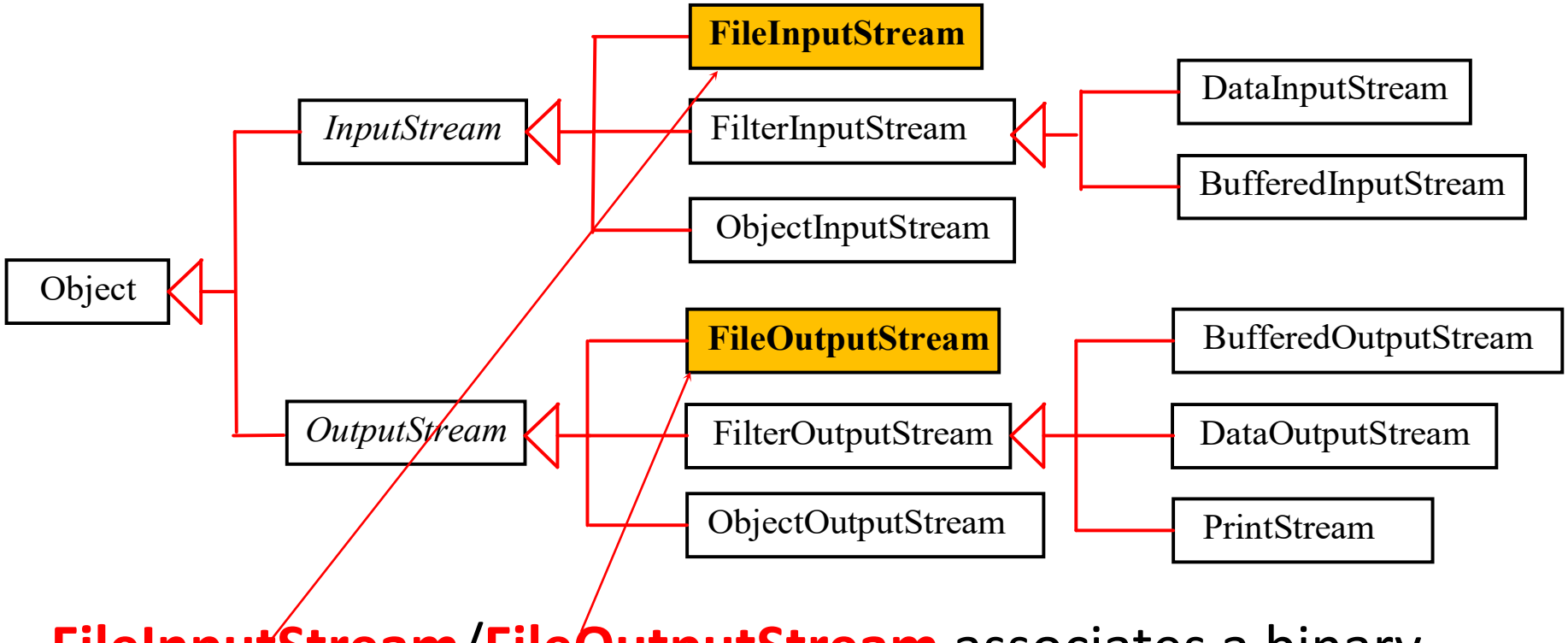
Writes all the bytes in array b to the output stream.

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

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

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

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/FileOutputStream

- FileInputStream/FileOuptputStream is for reading/writing bytes from/to files .

- All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses.

- FileInputStream/FileOuptputStream does not introduce new methods.

FileInputStream

- To construct a FileInputStream, use the following constructors:

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

- Creates a FileInputStream from a file name.

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

- Creates a FileInputStream from a File object.

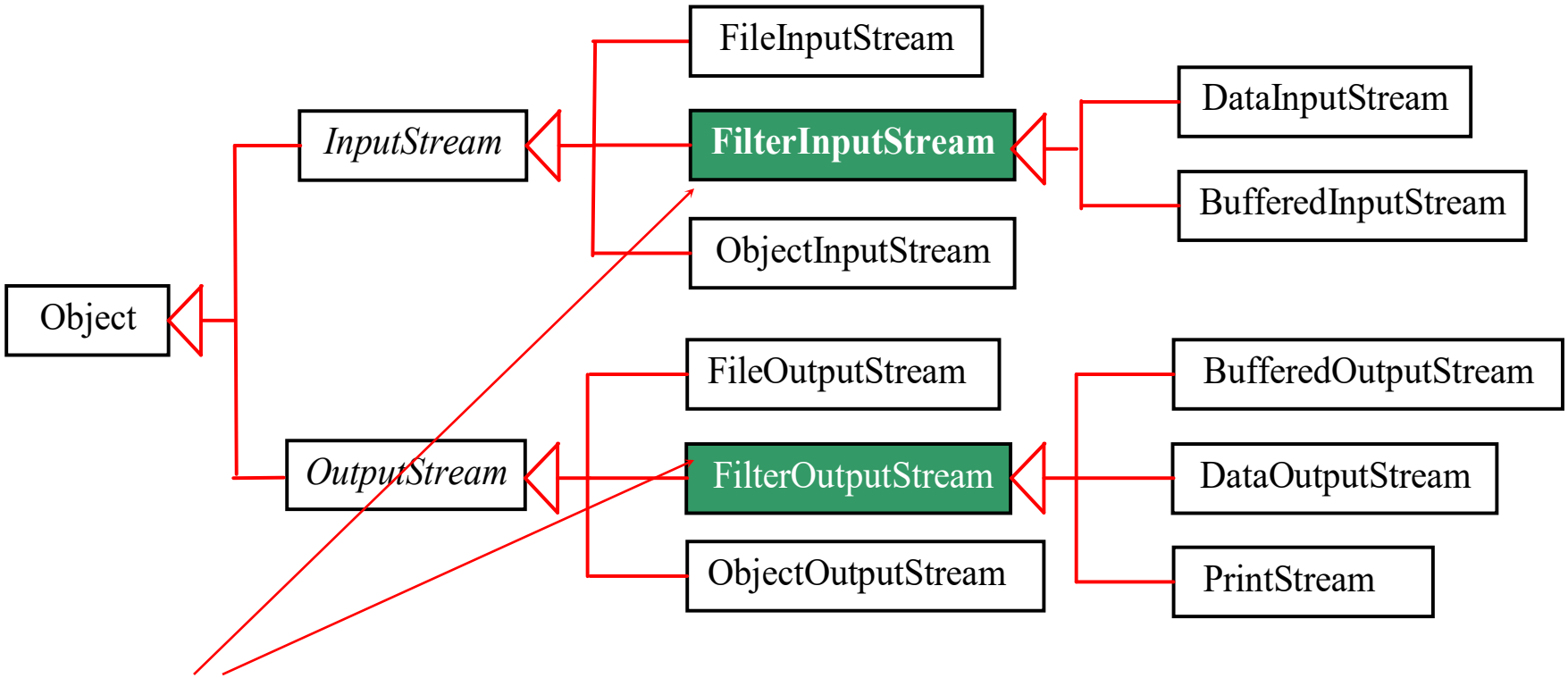
- A **`java.io.FileNotFoundException`** would occur if you attempt to create a FileInputStream with a nonexistent file.

FileOutputStream

- To construct a FileOutputStream, use the following constructors:
  - `public FileOutputStream(String filename)`
    - To Create a FileOutputStream from a file name:
  - `public FileOutputStream(File file)`
    - To create a FileOutputStream from a File object.
  - `public FileOutputStream(String filename,boolean append)`
    - If append is true, data is appended to the existing file.
  - `public FileOutputStream(File file, boolean append)`
    - If append is true, data is appended to the existing file.
- 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.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
       FileInputStream in = new
                          FileInputStream("SourceFile.txt"); ;
       FileOutputStream out = new
                          FileOutputStream("TargetFile.txt‟);
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
  }}
```

FilterInputStream/FilterOutputStream



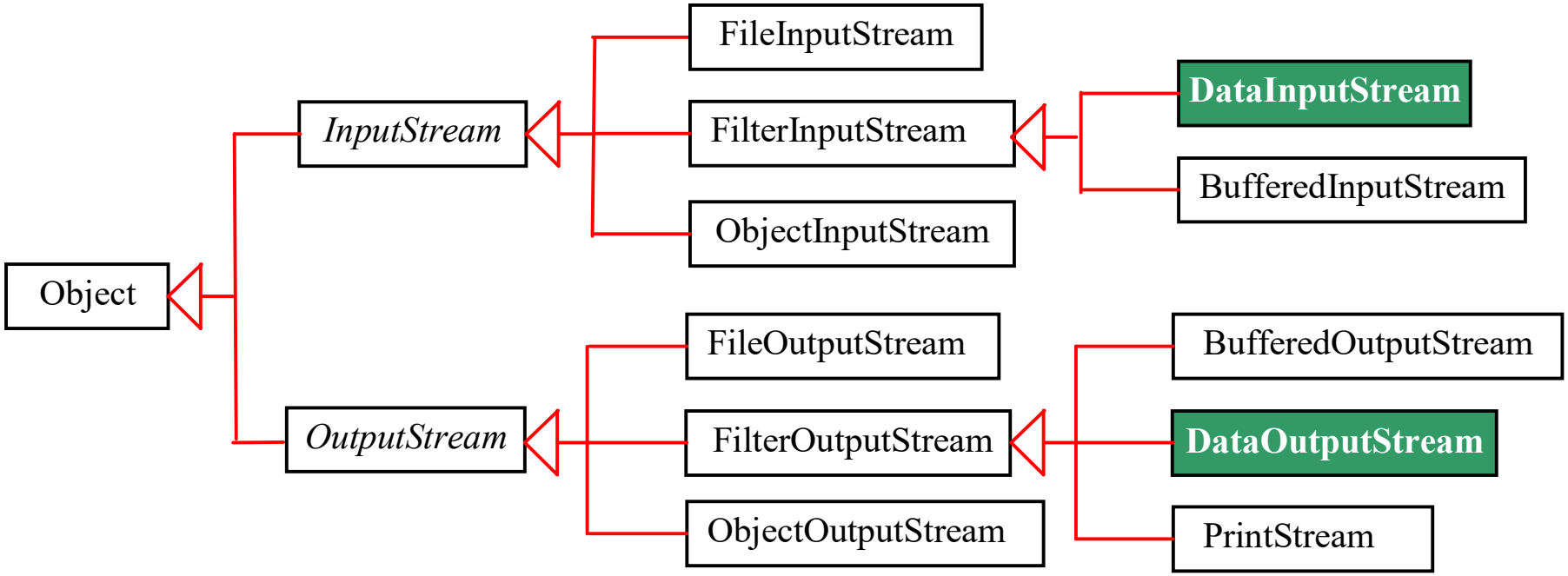
Filter streams are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes.

FilterInputStream/FilterOutputStream

- 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 **DatInputStream** and **DataOutputStream** to filter bytes.

DataInputStream/DataOutputStream

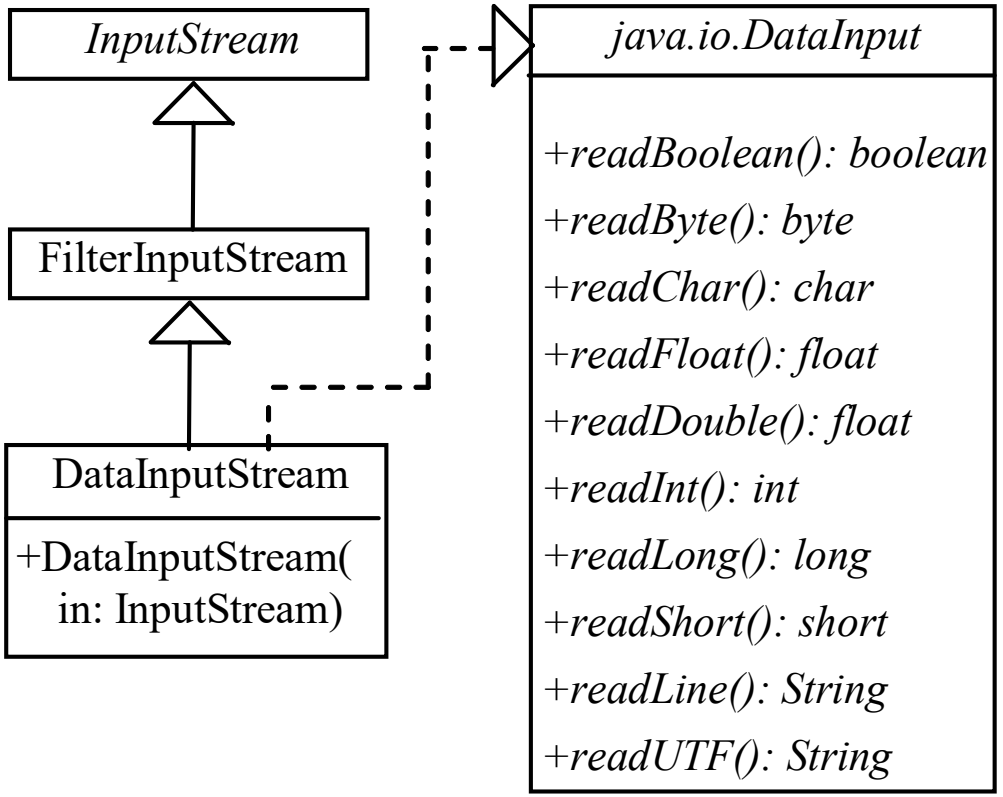
DataInputStream reads bytes from the stream and converts them into appropriate primitive type values



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

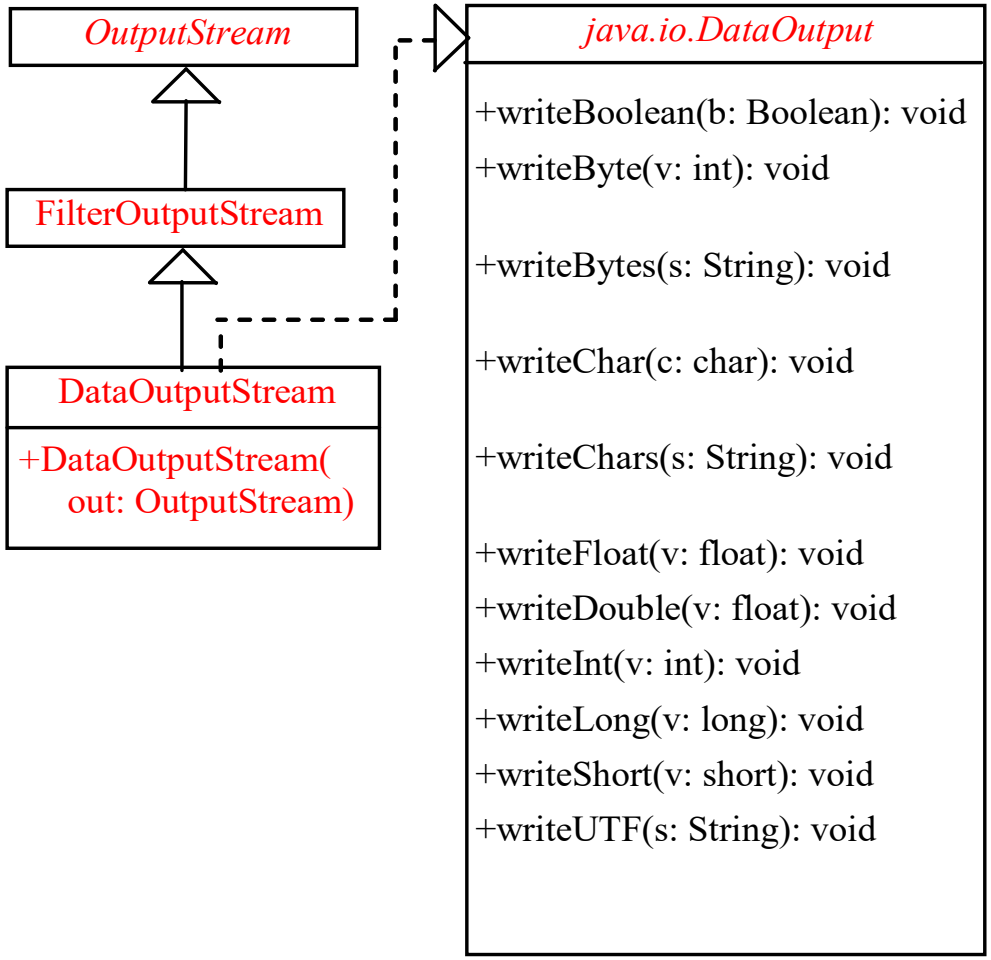
DataInputStream

DataInputStream extends FilterInputStream and implements the DataInput interface.



DataOutputStream

DataOutputStream extends FilterOutputStream and implements the DataOutput interface.



Writes a Boolean to the output stream.

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

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

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

Writes every character in the string s, to the output stream, in order, two 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 two bytes of length information to the output stream, followed by the UTF representation of every character in the string s.

Using DataInputStream/DataOutputStream

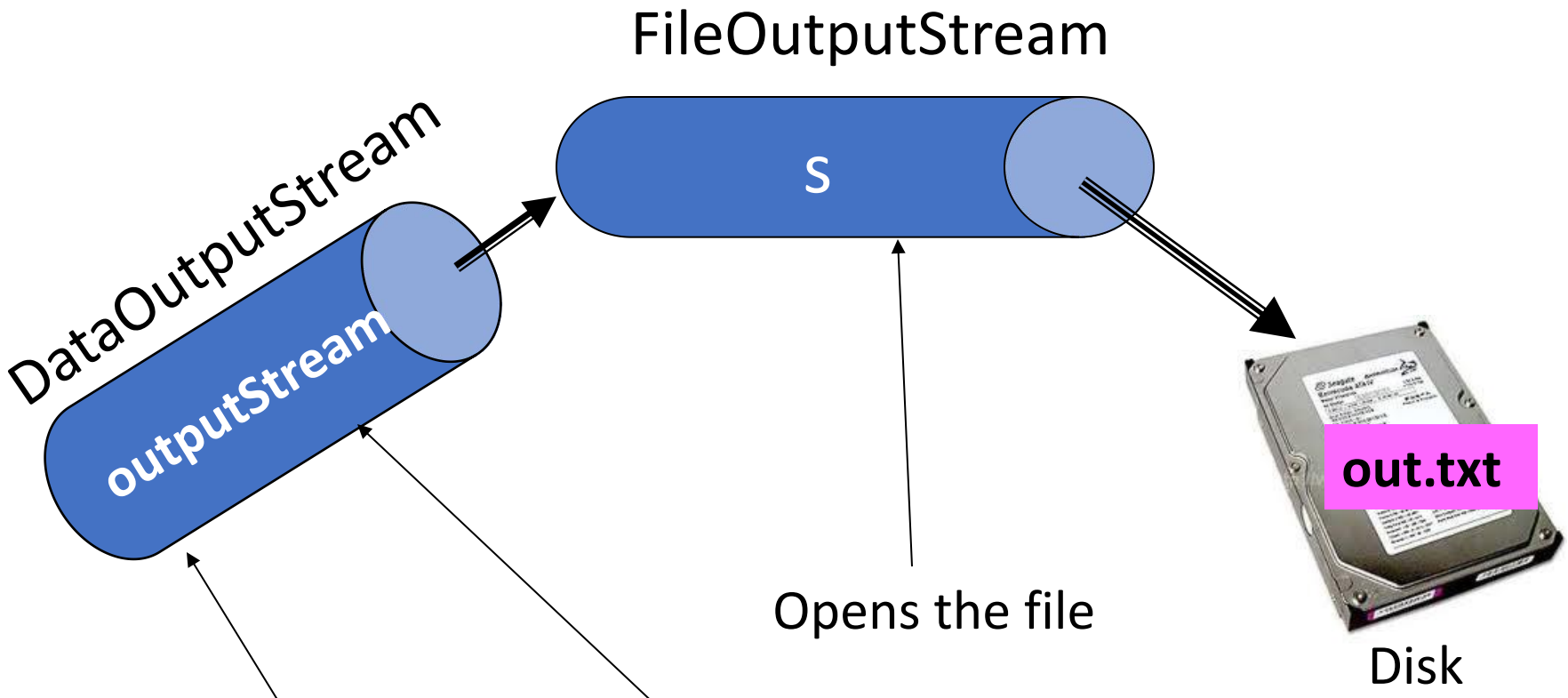
- Data streams are used as wrappers on existing input and output

  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**.

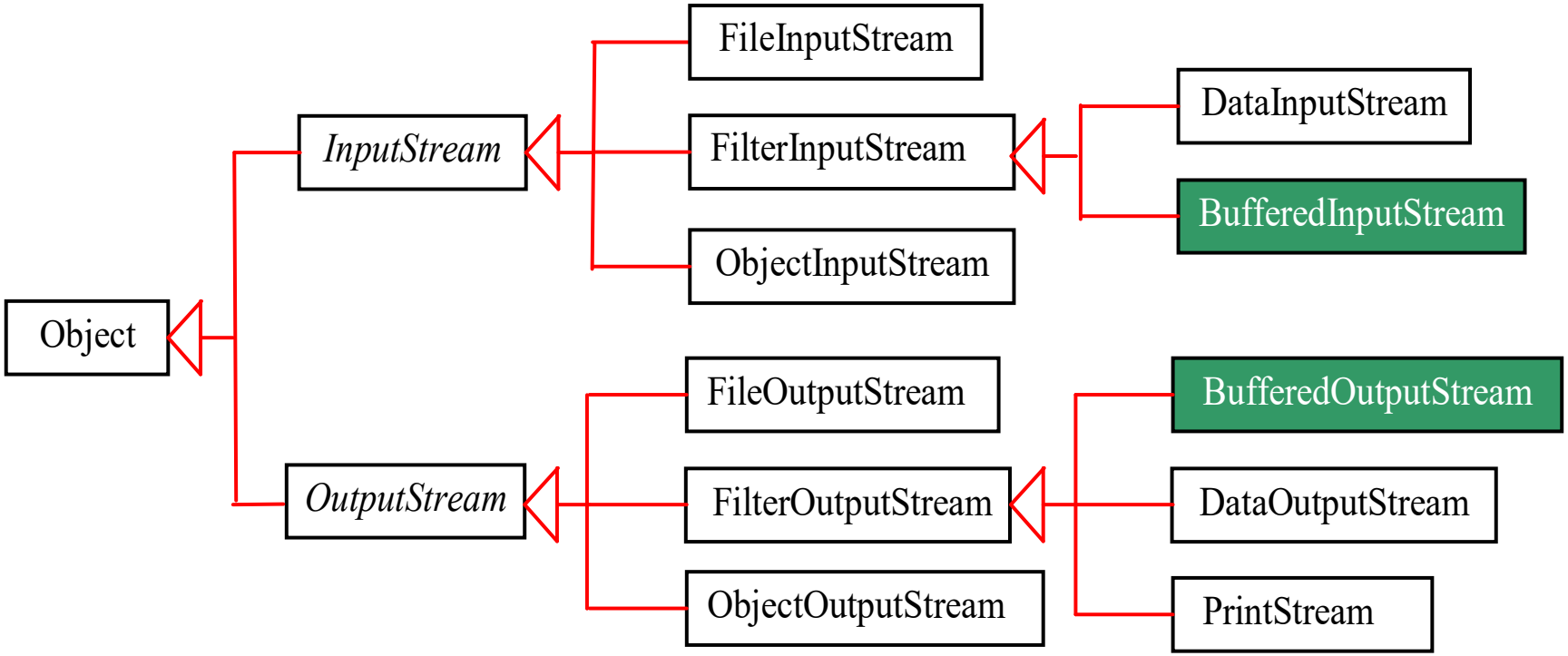
```
DataInputStream infile =
    new DataInputStream(new FileInputStream("in.txt"));
DataOutputStream outfile =
    new DataOutputStream(new FileOutputStream("out.txt"));
```


FileOutputStream
DataOutputStream
outputStream
s
out.txt
Opens the file
Disk
Has methods to access the opened file
Java uses this name to access out.txt

```
import java.io.*;

public class TestDataStream {
  public static void main(String[] args) throws
 IOException {
    // Create an output stream for file temp.txt
    DataOutputStream output = new DataOutputStream(
      new FileOutputStream("temp.txt"));

    // Write student test scores to the file
    output.writeUTF("John");
    output.writeDouble(85.5);
    output.writeUTF("Jim");
    output.writeDouble(185.5);
    output.writeUTF("George");
    output.writeDouble(105.25);

    // Close output stream
    output.close();
```

```
 // Create an input stream for file temp.txt
DataInputStream input = new DataInputStream(new
                    FileInputStream("temp.txt”));

    // Read student test scores from the file
   System.out.println(input.readUTF() + " " +
         input.readDouble());
   System.out.println(input.readUTF() + " " +
         input.readDouble());
   System.out.println(input.readUTF() + " " +
         input.readDouble());
  }
}
```

BufferedInputStream/BufferedOutputStream



BufferedInputStream/BufferedOutputStream

- Most of the examples we've seen so far use ***unbuffered* I/O**. This means each read or write request is handled directly by the underlying OS.
- This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- To reduce this kind of overhead, the Java platform implements ***buffered* I/O streams**. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty.
- Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full

BufferedInputStream/BufferedOutputStream

- BufferedInputStream/BufferedOutputStream Can be used to speed up input and output by reducing the number of reads and writes.

- BufferedInputStream/BufferedOutputStream does not contain new methods.

- All the methods of 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)
```

```
import java.io.*;

public class TestFIleStream {
  public static void main(String[] args) throws IOException
  {
    // Create an output stream to the file
 BufferedOutputStream output = new BufferedOutputStream(new
 FileOutputStream("temp.txt"));

    // Output values to the file
    for (int i = 1; i <= 10; i++)
      output.write(i);

    // Close the output stream
    output.close();
```

```
    // Create an input stream for the file
 BufferedInputStream input = new BufferedInputStream(new
                                    FileInputStream("temp.txt"));

    // Read values from the file
    int value;
    while ((value = input.read()) != -1)
      System.out.print(value + " ");

    // Close the output stream
    input.close();
  }
}
```

Character based Streams

- Character based streams are also defined by two class hierarchies. At the top are two **abstract** classes:
    1. **Reader**
    2. **Writer**
- These abstract classes handle Unicode characters.
- These abstract classes **Reader** and **Writer** declare many abstract methods which are implemented by other subclasses.
- Two most important methods are **read( )** and **write( )**, which respectively read and write one character.
- Both methods are declared as abstract inside **Reader** and **Writer**.

Using Character Streams

- All character stream classes are descended from **Reader** and **Writer**.

- As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWirter.

- The CopyCharacters example illustrates these classes.

Reading/Writing to a Text File…

- Constructors of FileWriter stream

FileWriter(File file)

Constructs a FileWriter object given a File object.

FileWriter (File file, boolean append**)**

Constructs a FileWriter object given a File object.

FileWriter(String fileName)

Constructs a FileWriter object given a file name.

FileWriter(String fileName, boolean append)

Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

Reading/Writing to a Text File…

- Constructors of FileReader stream

`FileReader``(File file)`

Constructs a FileReader object given a File object.

`FileReader``(String fileName)`

Constructs a FileReader object given a file name.

```
import java.io.*;
public class CopyCharacter {
    public static void main(String[] args) throws IOException
      {
       FileReader inputStream =new FileReader("File.txt");
       FileWriter outputStream = new FileWriter("File1.txt");

          int c;
         while ((c = inputStream.read())!= -1) {
              outputStream.write(c);
          }

         if (inputStream != null) {
             inputStream.close();
          }
         if (outputStream != null) {
             outputStream.close();
          }
      }
}
```

Reading/Writing to a Text File…

- Other classes which could be used for writing to a text file are: all classes are subclasses of Writer class:

  BufferedWritter, CharArrayWritter, PipedWritter, StringWritter, PrintWritter.

- Each of these classes have their own constructors and methods .

Reading/Writing to a Text File…

- Some of the classes which are used for reading text files are: all classes are subclasses of Reader class:

  - BufferedReader, LineNumberReader
  - CharArrayReader
  - InputStreamReader, FileReader
  - FilterReader
  - PushbackReader
  - PipedReader
  - StringReader

Reading/Writing to a Text File…

Line-Oriented I/O

- Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end.
- A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n").
- Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes **BufferedReader** and **PrinterWriter**
- The **CopyLines** example invokes `BufferedReader.readLine` and `PrintWriter.println` to do input and output one line at a time.

Reading from a Text File…

- Joining a BufferedReader and a FileReader.
- BufferedReader
  - Has readLine() method
  - But lacks a constructor that can take file name
- FileReader
  - Has constructor which can take file name
  - But lacks readLine() method
- Combine them together as follows:

```
BufferedReader inStream = new BufferedReader(new
                                FileReader(fileName));
```

Reading/Writing from a Text File…

- Here the **BufferedReader** will read from a file.
- Now it is possible to use: inStream.readLine() to read one line at a time from the file.
- An important fact about **readLine**() is that it will return null as its value when it reaches the end of the file.
  - That is readLine() does not return the end-of-line character as part of the text it returns.

PrintWritter…

- **PrintWriter:-** object takes Strings and data types from program and write in output stream.

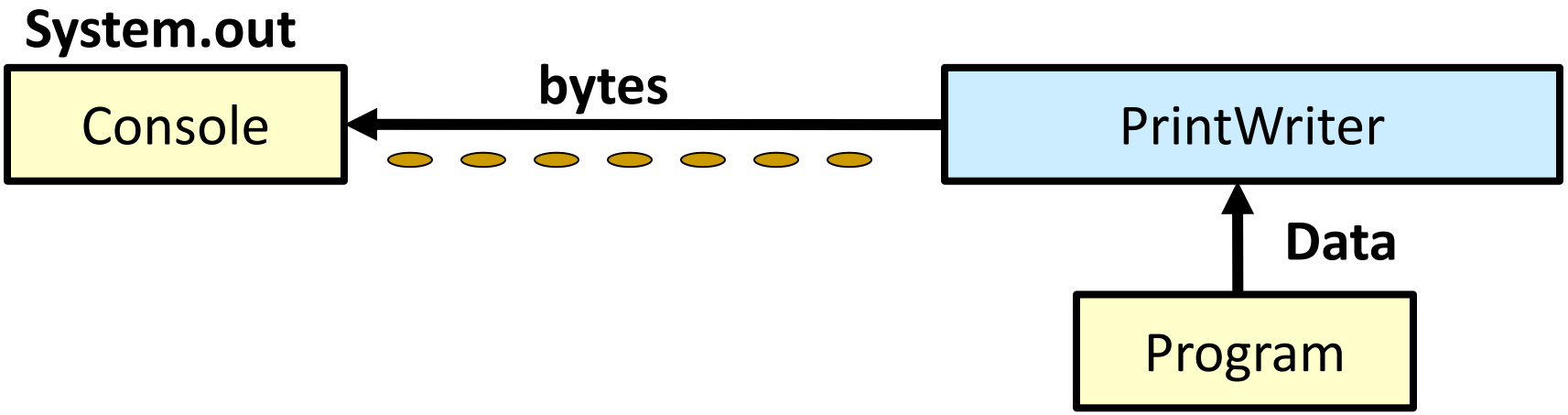
- We can use the print( ) and println( ) methods just like System.out.println( ) to write any type of data to the console.

```
String s = "Hello World";
p.println( s );

int i = 55;
p.print( i );
```

PrintWritter

- For real world programs the recommended way of writing to the console is **PrintWriter** class.
- To create the object we use the following constructor.

```
PrintWriter p = new PrintWriter(System.out);
```



```
import java.io.*;

public class CopyLines {
    public static void main(String[] args) throws IOException {
      BufferedReader inputStream = new BufferedReader(new
                FileReader("File.txt"));
      PrintWriter outputStream = new PrintWriter(new
                FileWriter("File1.txt"));

          String l;
        while ((l = inputStream.readLine()) != null) {
            outputStream.println(l);
          }

        if (inputStream != null) {
            inputStream.close();
          }
        if (outputStream != null) {
            outputStream.close();
        }
   }
}
```

Exercise

- Write a method that takes the file name and extracts each line of the files and displays the line to a console.

The File Class

- Before we see how to read and write binary and text files lets see File class.
- The **File** class does not permit any I/O, instead it provides a means of querying/modifying filename or pathnames (the class would be better termed FileName).
- java.io.File is the central class in working with files and directories.
- Files and directories are both represented by File objects.
- When a File object is created, the system doesn't test to see if a corresponding file/directory actually exists; you must call exists() to check.
- The constructors and methods of the File class are summarized as follows

The File Class…

- Constructors
  - File *f* = new File(String *path*);
    - Create File object for default directory (usually where program is located).
  - File *f* = new File(String *dirpath*, String *fname*);
    - Create File object for directory path given as string.
  - File *f* = new File(File *dir*, String *fname*);
    - Create File object for directory.
- public static constants
  - String *s* = File.separator;
    - Default path separator (eg, "/" in Unix, "\" in Windows).

The File Class…

- *Getting Attributes* (Assume File f)
  - boolean b = f.**exists**(); →true if file exists.
  - boolean b = f.**isFile**(); →true if this is a normal file.
  - boolean b = f.**isDirectory**();→true if f is a directory.
  - String s = f.**getName**(); →name of file or directory.
  - boolean b = f.**canRead**(); →true if can read file.
  - boolean b = f.**canWrite**(); →true if can write file.
  - boolean b = f.**isHidden**(); →true if file is hidden.
  - long l = f.**lastModified**(); →Time of last modification.
  - long l = f.**length**();→Number of bytes in the file.
- *Setting Attributes*
  - *f*.**setLastModified**(*t*); →Sets last modified time to long value *t*.
  - boolean *b* = *f*.**setReadOnly**();→Make file read only. Returns true if successful.

The File Class…

- Paths
  - String s = f.**getPath**();→path name.
  - String s = f.**getAbsolutePath**(); →path name (how is it different from above?).
  - String s = f.**getCanonicalPath**(); →path name. May throw IOException.
  - String s = f.**toURL**();& String s = f.**toURI**(); ; →path with "file:" prefix and /'s. Directory paths end with /.
- Creating and deleting files and directories
  - Boolean b = f.**delete**(); →Deletes the file.
  - boolean b = f.**createNewFile**(); →Create file, may throw IOException. true if OK; false if already exists.
  - boolean b = f.**renameTo**(f2);→Renames f to File f2. Returns true if successful.
  - boolean b = f.**mkdir**();→Creates a directory. Returns true if successful.
  - boolean b = f.**mkdirs**(); →Creates directory and all dirs in path. Returns true if successful.

The File Class… example

```
import java.io.File;
import java.io.IOException;
public class FileTest{
    public static void main(String[] args){
    File f =new File("D:/Documents and
    Settings/yozi/Desktop/JavaExamples/Buffered.java");
    System.out.println(f.exists());
    System.out.println(f.canRead());
    System.out.println(f.canWrite());
    System.out.println(f.getName());
    System.out.println(f.getParent());
    System.out.println(f.isFile());
     System.out.println(f.length());
     }
}
```

Output:

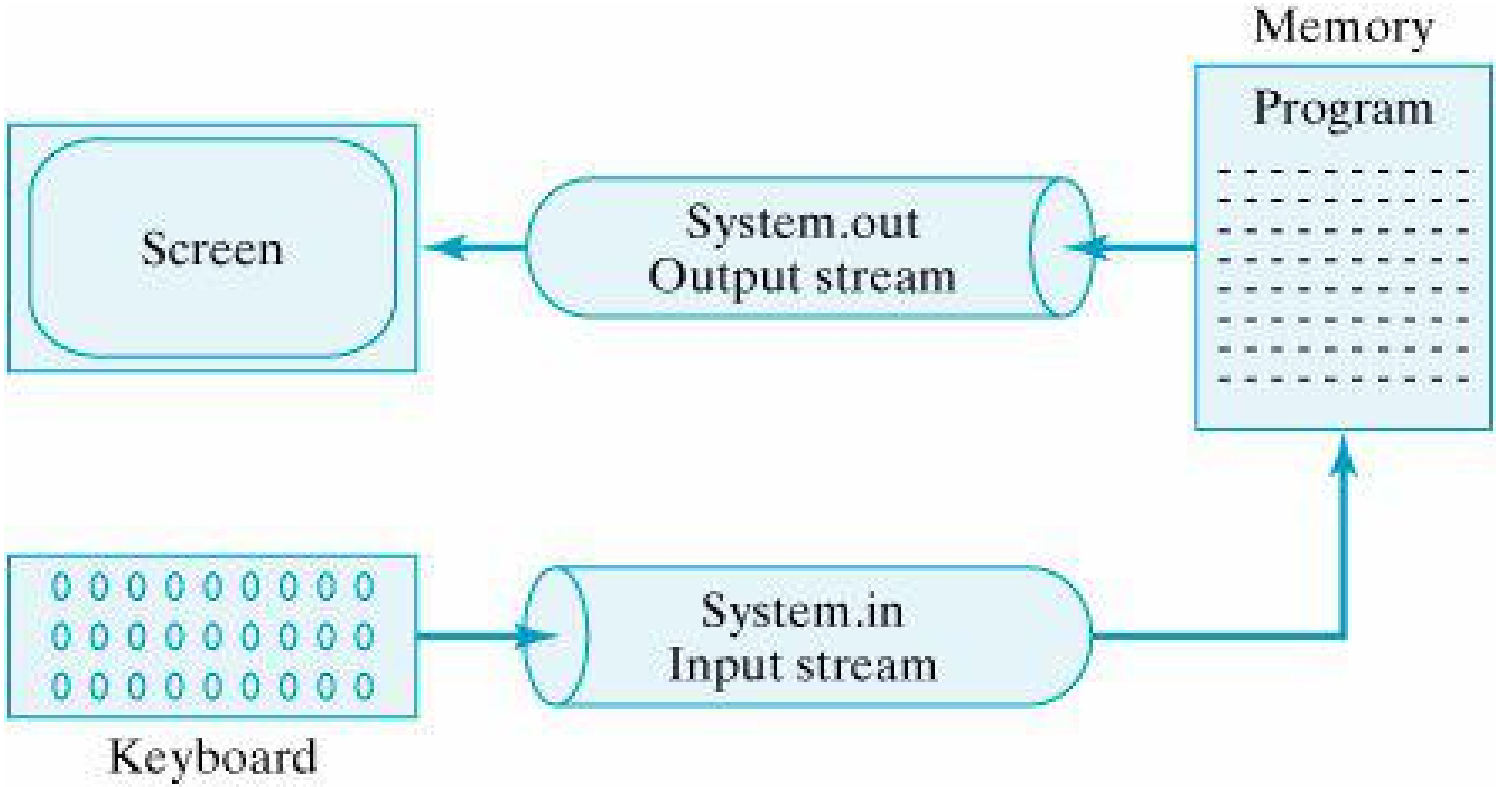
```
true
true
true
Buffered.java
D:\Documents and
Settings\esubalew\Desktop\JavaTests
false
true
739
```

Predefined Streams

- **java.lang** package defines a class called **System**, which encapsulates several aspects of the run time environment: **System.out, System.in, System.err**.
- System class contains three predefined streams, **in**, **out** and **err**.
- They are **public** and **static** fields defined inside the **static final** class **System**.
- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to the standard input stream, which is the keyboard by default.
- **System.err** refers to the standard error stream, which is also the console by default.

Predefined Streams…

- System.in is an object of **InputStream** class, and **System.out** and **System.err** are objects of **PrintStream** class.

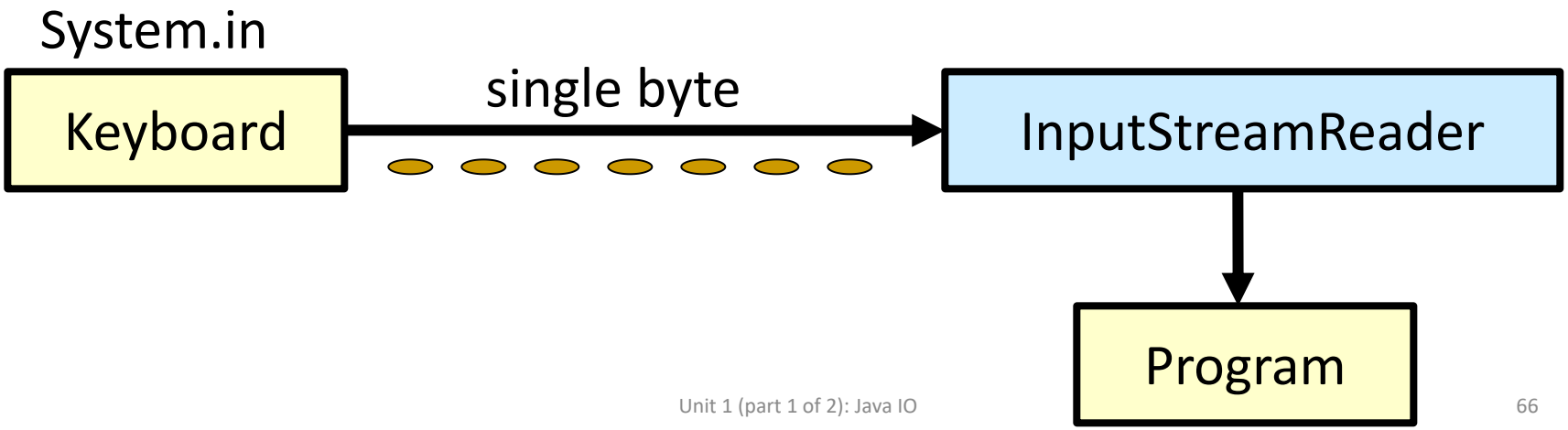


Reading Console Input

- In java, console input is accomplished by reading from **System.in**.
- To read an input we link **System.in** to a **InputStreamReader** class as follows

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

- As **InputStreamReader** read single byte at a time, this affects system performance.



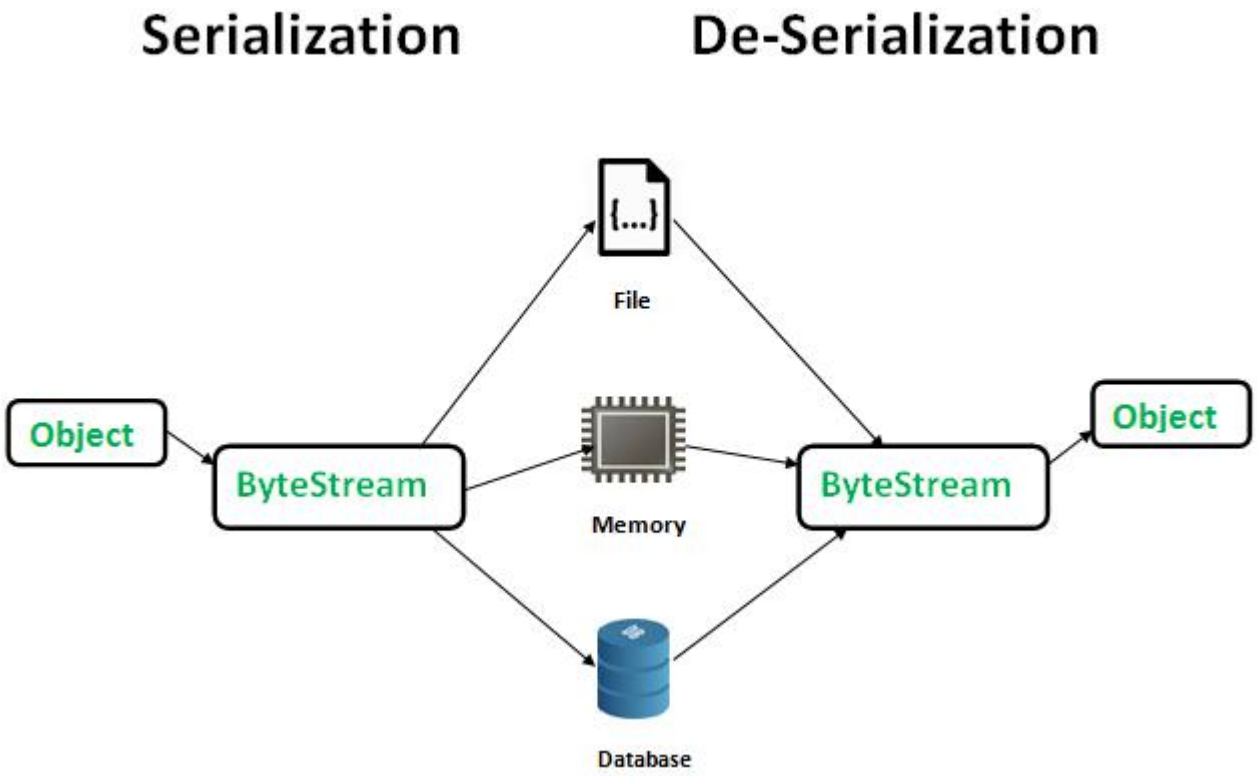
Reading Console Input…

- For this purpose we use **BufferReader** class to read number of bytes from input buffer.
- To read input from buffer we wrap InputStreamReader into BufferedReader as follows:
  - **`BufferedReader r = new BufferedReader(new InputStreamReader ( System.in));`**
- BufferedReader has a number of methods.
- By using this methods we can read an input from the keyboard.Some of these methods are:
  - **close()** →Closes the stream and releases any system resources associated with it.
  - **read()** →Reads a single character
  - **readLine()** → Reads a line of text.
  - **ready()** → Tells whether this stream is ready to be read.

Serialization/marshalling

- Serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted and reconstructed later.
- Why Serialization is important?
  - A method of storing data (in databases, on hard disk drives)
  - A method of transferring data through the wires
  - A method of remote procedure call, as in SOAP
- Object serialization is representation of an object as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object
- Object Serialization API: java.io. Serializable – provides a framework for encoding objects as byte streams.

Serialization/marshalling…



Java - Serialization

- Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
- After a serialized object has been written into a file, it can be read from the file and deserialized.
  - that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.
- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Java – Serialization…

- The Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.
- The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

  public final void writeObject(Object x) throws IOException

- The above method serializes an Object and sends it to the output stream.

Java – Serialization…

- Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject() throws IOException,
                                        ClassNotFoundException
```

- This method retrieves the next Object out of the stream and deserializes it.
- The return value is Object, so you will need to cast it to its appropriate data type.
- To demonstrate how serialization works in Java, suppose that we have the following Employee class, which implements the Serializable interface.

Java – Serialization…Example

- public class Employee implements java.io.Serializable {

  public String name;

  public String address;

  public int transient SSN;

  public int number;

  public void mailCheck() {

  System.out.println("Mailing a check to " + name+ " " + address); }}

- Notice that for a class to be serialized successfully, two conditions must be met:
  - The class must implement the java.io.Serializable interface.
  - All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

Serialization - Serializing an Object

- The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.
- When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.
- **Note:** When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

```
import java.io.*;
 public class SerializeDemo{
   public static void main(String [] args)  {
      Employee e = new Employee();
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
```

Serialization - Serializing an Object

```
e.SSN = 11122333;
e.number = 101;
try{
        FileOutputStream fileOut =new FileOutputStream("employee.ser");
        ObjectOutputStream out =new ObjectOutputStream(fileOut);
        out.writeObject(e);
        out.close();
        fileOut.close();
     }catch(IOException i){
         i.printStackTrace();
     }
   }
}
```

Serialization - DeSerializing an Object

- The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program.

```
import java.io.*;
public class DeserializeDemo{
   public static void main(String [] args){
      Employee e = null;
      try{
         FileInputStream fileIn = new FileInputStream("employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      }catch(IOException i){
         i.printStackTrace();
         return;
      }
```

Serialization - DeSerializing an Object…

```
catch(ClassNotFoundException c){
        System.out.println(.Employee class not found.);
        c.printStackTrace();
        return;
    }
    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name +“ , Address: " +
  e.address  + “, SSN: " + e.SSN + “,  Number: " + e.number);
}
}
```

Output

Deserialized Employee...

Name: Reyan Ali, Address:Phokka Kuan, Ambehta Peer, SSN: 0, Number:101

Serialization - DeSerializing an Object…

- Here are following important points to be noted:
  - The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
  - Notice that the return value of readObject() is cast to an Employee reference.
  - The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Important Points about Serialization

- If you declare a variable as **transient** it will not be saved during serialization.
- **Serializable** interface is an empty interface.
- If a class is serializable, then all the subclasses of this super class are implicitly serializable even if they don't explicitly implement the **Serializable** interface.
- If you are serializing an array or collection, each of its elements must be serializable.
- **static** variables are not saved as part of serialization.