Introduction to Java for C++ Programmers

Java Input / Output
Segment 1- Basics

Professor: Mahboob Ali

Objectives

Upon completion of this lecture, you should be able to:

- Examine Input / Output classes in Java
- Create and Use I/O Streams in Java
- Distinguish Byte, Character, and Buffered Stream
- Design and Develop File I/O programs

Input / Output

In this lesson you will be learning about:

- Types of Input / Output Stream
- Typical use of IO Streams
- Character and Byte Streams.

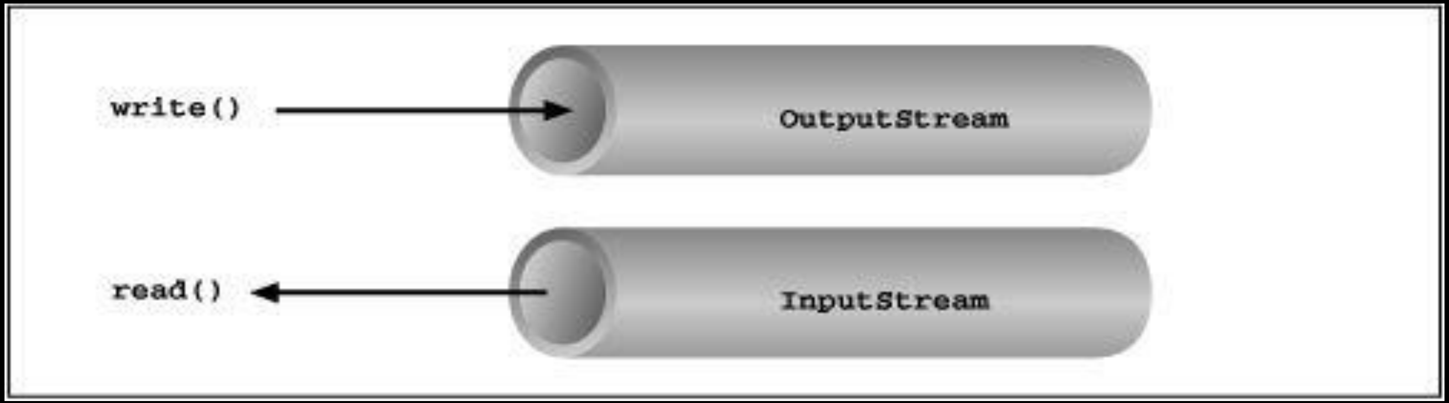
Reading / Writing Data

Reading

```
Open a stream
while more information
    read information
close the stream
```

Writing

```
open a stream
while more information
    write information
close the stream
```



Java I/O

- **Java I/O** (Input and Output) is used
  - *to process the input* and
  - *produce the output*.

- Java uses the concept of **stream** to make I/O operation fast.

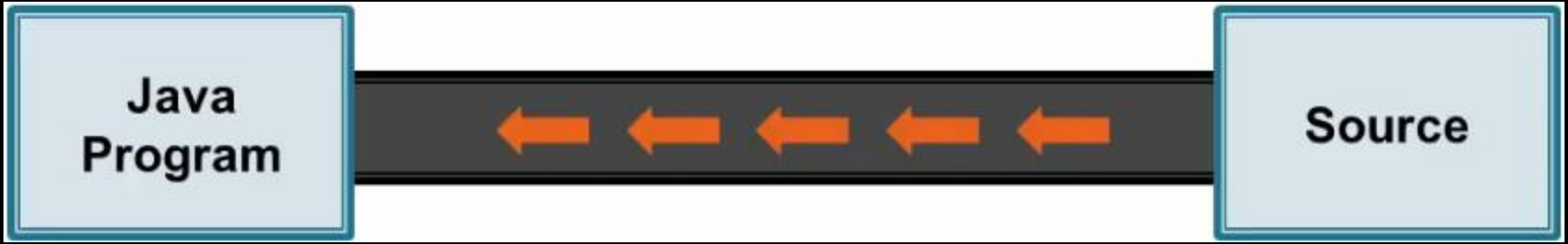
- Java treats all the data sources or sinks (like, file or network) in the same manner as sequential flow of data.
  - Receive data from a source by opening an input stream
  - Send data to sink by opening an output stream.

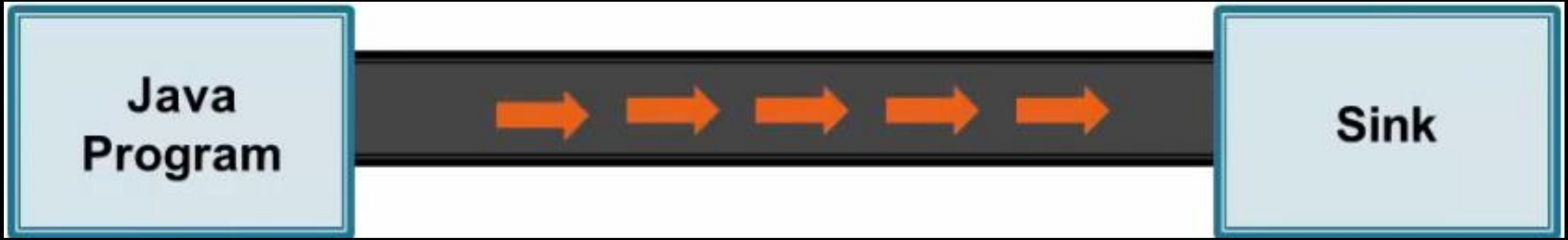
- The **java.io** package contains all the classes required for input and output operations.

Stream

- A stream is a sequence of data and is composed of **bytes**.
- Streams support different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- In simple words a stream is a *link* between a java program and a source/ sink.
    - A source is an input to the program, like file, console input, etc.
    - A sink can be any external format to which data can be written out. Like databases, files, console output, etc.

Input & Output Streams





Stream IO Operations

1. Open stream
2. Read/ Write
3. Close stream

Stream

- In java, 3 streams are created for us automatically also called **Standard Streams**.
  - **System.out:** standard output stream
  - **System.in:** standard input stream
  - **System.err:** standard error stream

(Out stream) `System.out.println("Some Message");`

normally outputs the data you write

(Error Stream) `System.err.println("Error message");`

normally only used to output error texts.

(In Stream)

```
   int i=System.in.read();//returns ASCII code of 1st character
   System.out.println((char)i);//will print the character
```

Standard Template

```
FileInputStream in = null;
try{
      in = new FileInputStream(filename); //open stream
      // read data
      }catch (FileNotFoundException ){
            …
      } finally{
            try{
                  if(n != null)
                        in.close();
                  }catch (IOException e){ …}
      }
```

try – with – resources (Java 7)

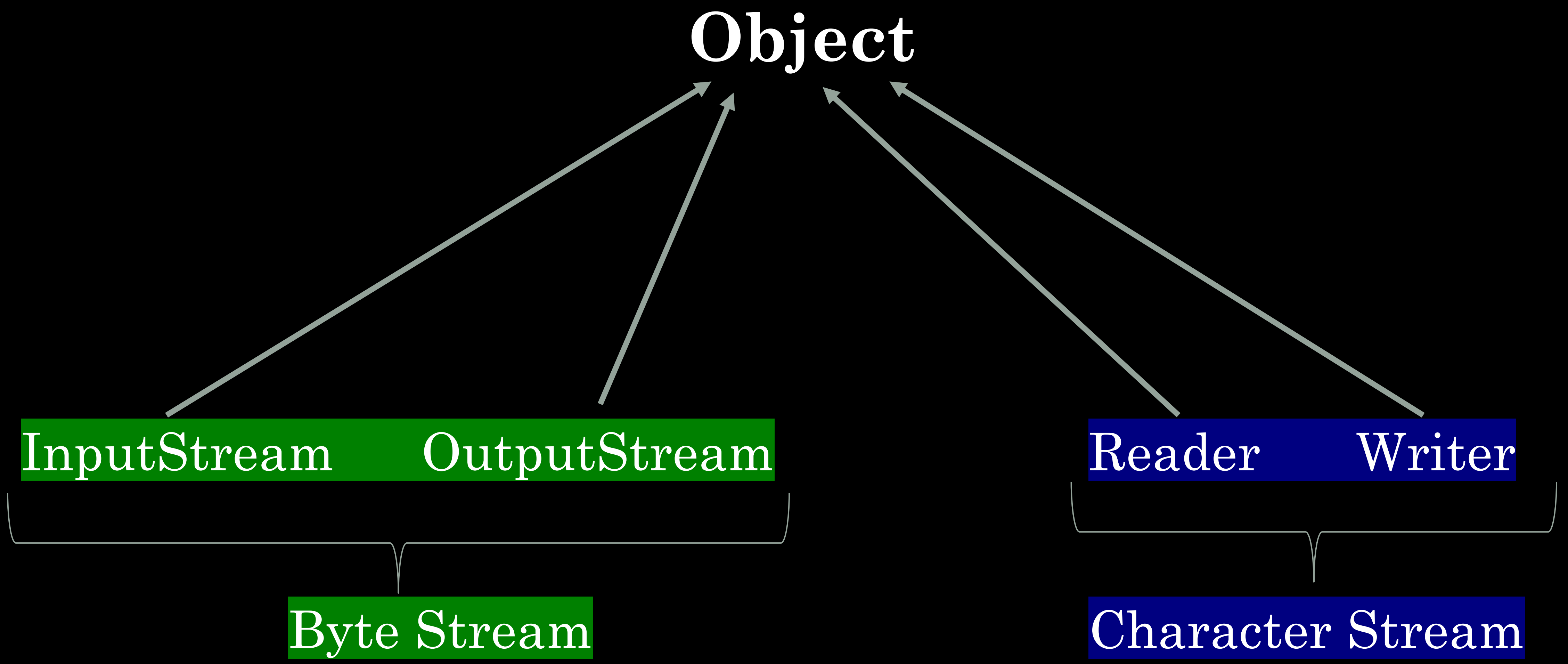
java.lang.Autocloseable

```
try(FileInputStream in = new FileInputStream(filename)){
      // Read Data
      }catch(FileNotFoundException e){
            … …
      }catch(IOException e){
            … …
      }
```

try – with – resources ~ Multiple Resources

```
try(FileInputStream in = new FileInputStream(filename);
    FileOutputStream out = new FileOutputStream(filename)){
     // Read Data
     }catch(FileNotFoundException e){
          … …
     }catch(IOException e){
          … …
     }
```

Stream Classification



Byte Stream

- Programs use byte streams to perform input and output of 8-bit bytes.

- **InputStream** and **OutputStream** are served as base classes for all the byte oriented data.
    - InputStream - Base abstract class for all byte input streams
    - OutputStream - Base abstract class for all byte output streams

- Most frequently used ones are **FileInputStream** and **FileOutputStream**

- Byte streams are used to read/write ***raw bytes*** serially from/to an external device.

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class TestBytes {
     public static void main(String[] args) throws IOException {
     FileInputStream in = null;
     FileOutputStream out = null;
     try {
             in = new FileInputStream("somefile.txt");
             out = new FileOutputStream("outagain.txt");
             int c;
             while ((c = in.read()) != -1) {
                              out.write(c);
              }
      }
     finally {
             if (in != null) {
                     in.close();
             }
             if (out != null) {
                      out.close();
             }
      }
  }
}
```

- **abstract** int **read()** throws IOException
  - **Reads 1 byte** & return as int between 0 & 255
- Returns ***-1*** if end-of-stream detected

- **abstract** void **write(int)** throws IOException
  - **Writes 1 byte** to the output stream

Character Stream

- The Java platform stores character values using Unicode conventions.

- Character stream I/O automatically translates this internal format to and from the local character set.

- The **Reader** class is the superclass of all the Java input streams.
- **Writer** is the superclass of all character output streams.

- Character-based I/O is used for processing **texts**.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class TestCharacters {
      public static void main(String[] args) throws IOException {
      FileReader inputStream = null;
      FileWriter outputStream = null;
      try {
             inputStream = new FileReader(“something.txt");
             outputStream = new FileWriter("characteroutput.txt");
             int c;
             while ((c = inputStream.read()) != -1) {
                          outputStream.write(c);
              }
       }
       finally {
                 if (inputStream != null) {
                               inputStream.close();
                 }
                 if (outputStream != null) {
                               outputStream.close();
                 }
         }
     }
}
```

Byte Stream vs Character Stream

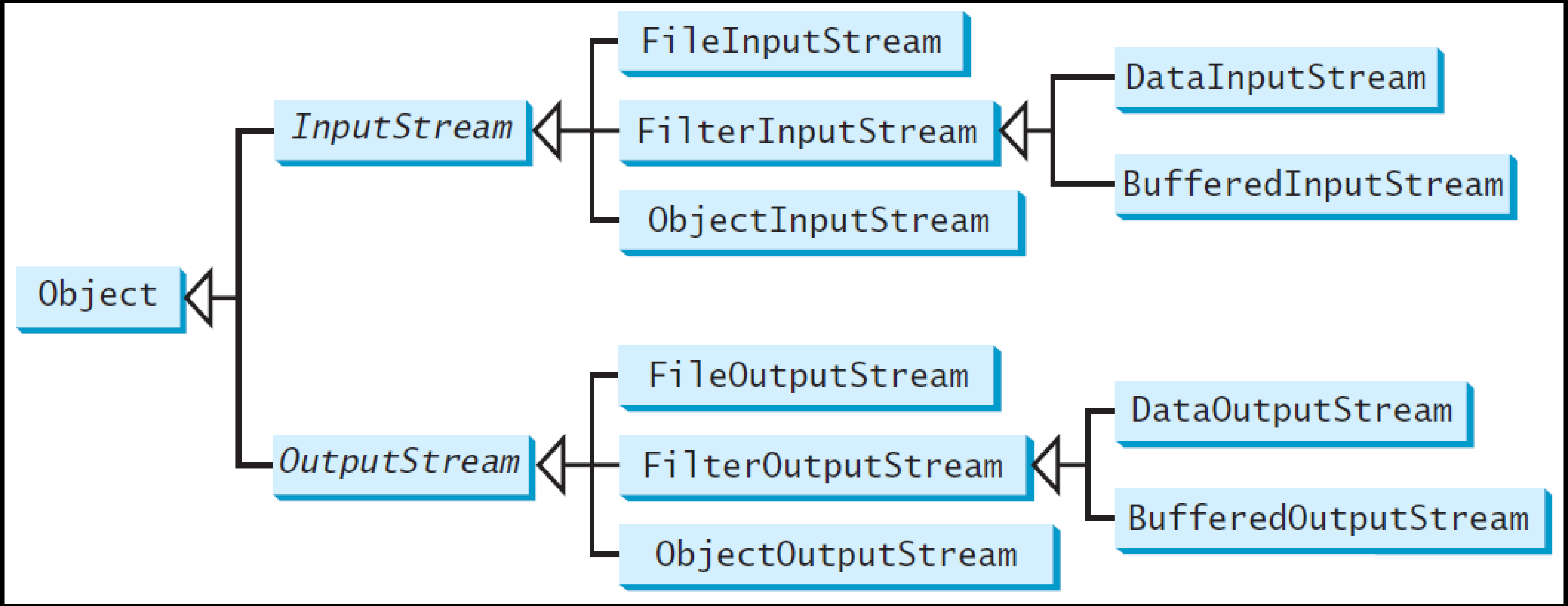
Byte Stream	**Character Stream**
Deal with "Raw Data"	Deal with "Character Data"
Byte by Byte (1 Byte -- 8-bits)	Character by Character (1 char – 16 bits)
Coder responsibility to convert bytes to character	No worries for the coder
Does not always handle Unicode correctly	Handle Unicode appropriately

Which one is more efficient?

Binary I/O does not involve encoding or decoding and is more efficient then text I/O.

Binary I/O Classes

- Text I/O required encoding and decoding.
- Binary I/O doesn’t required any conversion of Unicode.



FileInputStream Class

- **FileInputStream** is for reading bytes-oriented data from files like image data, audio etc.

- All the methods in this class is inherited from **InputStream**.

- A **FileNotFoundException** will occur if you attempt to create a **FileInputStream** with a nonexistent file.

- To construct a FileInputStream, use the following constructors:

    - public FileInputStream(String filename)
    - public FileInputStream(File file)

FileOutputStream Class

- **FileOutputStream** is used for writing data to a file.
- All the methods in this class is inherited from **OutputStream**.
- To construct a **FileOutputStream**, use the following constructors.

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

If the file doesn't exit it will be created and if the file exits then the contents will be overridden.

To avoid overriding the contents we can use the other two constructors, where we have to pass true value.

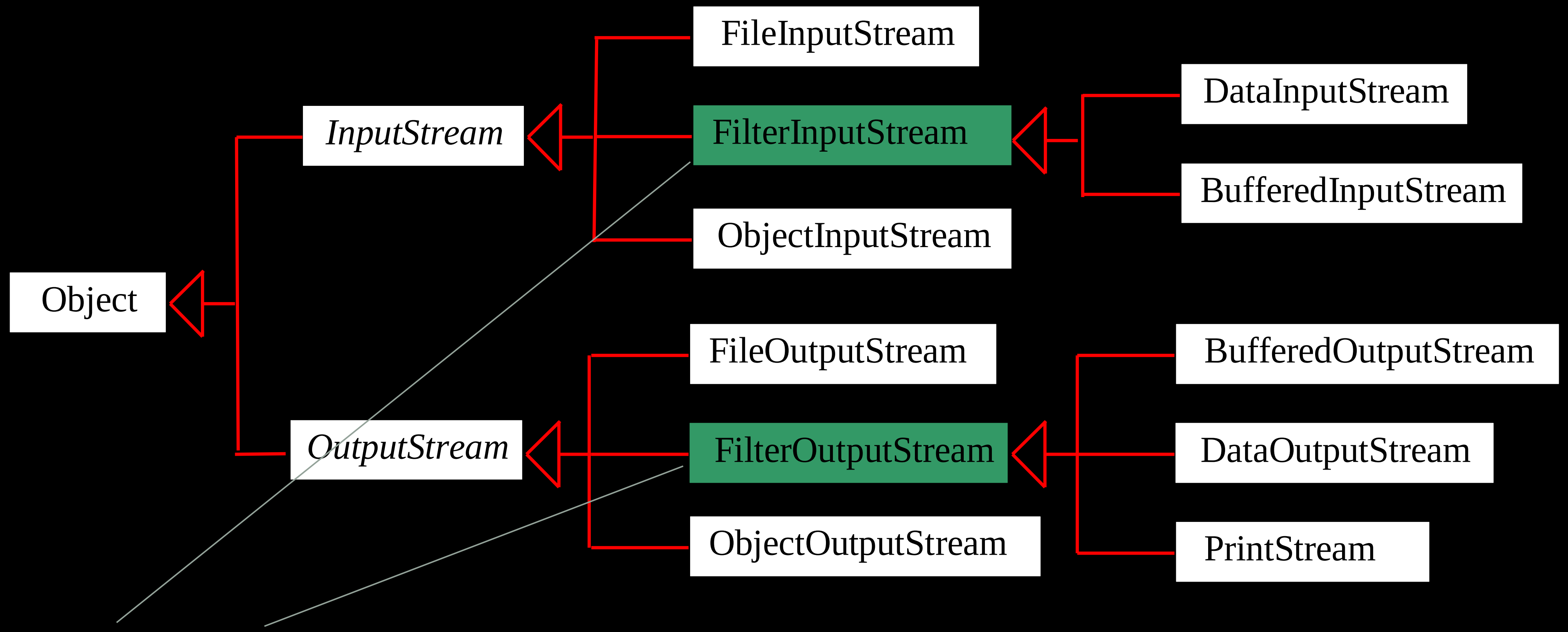
Writing a byte

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
            try{
              FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
              fout.write(65);
              fout.close();
              System.out.println("success...");
             }catch(Exception e){System.out.println(e);}
      }
}
```

Writing string

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
            try{
              FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
              String s="Welcome to JAC-444.";
              byte b[]=s.getBytes();//converting string into byte array
              fout.write(b);
              fout.close();
              System.out.println("success...");
             }catch(Exception e){System.out.println(e);}
      }
}
```

FilterInputStream/ FilterOutputStream



- *Filter streams* are streams that filter bytes for some purpose.
- If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream.
- When you need to process primitive numeric types, use DatInputStream and DataOutputStream to filter bytes.

BufferedInputStream/ BufferedOutputStream

- The read()/write() method in InputStream/OutputStream are designed to read/write a single byte of data on each call.

- Overall inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations).

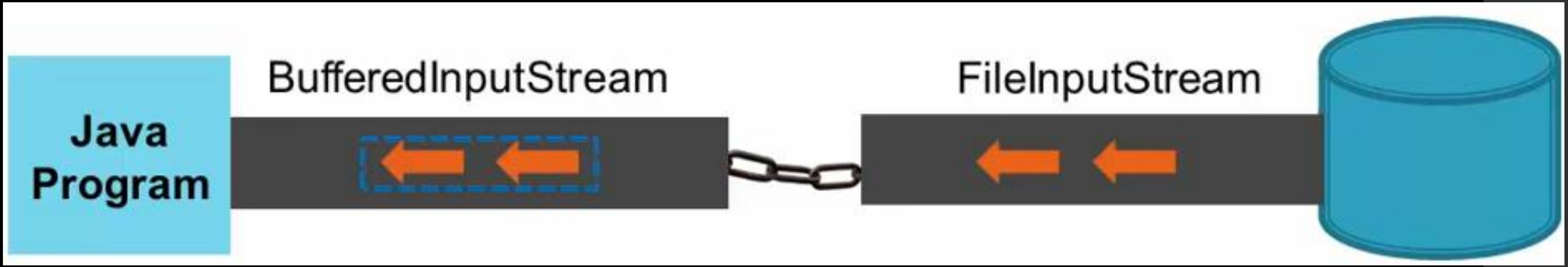
- ***Buffering***, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

Buffering

- Read/Write *blocks of bytes* into memory buffer
- Buffer ~ byte array
- Default buffer size ~ 8192bytes

Chained Streams

- Buffering only provides buffering as its core functionality.
- Doesn’t take responsibility of dealing with the file on the system.
- Works with or chained with another Stream.



- FileInputStream/ FileOutputStream is often chained to a BufferedInputStream or BufferedOutputStream, which provides the buffering.

```
BufferedInputStream in = new BufferedInputStream(new FileInputStream("my.jpg"));
```

- To chain the streams together, simply pass an instance of one stream into the constructor of another stream.

- For example, the following codes chain a FileInputStream to a BufferedInputStream, and finally, a DataInputStream.

```
FileInputStream fileIn = new FileInputStream("in.dat");
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);
DataInputStream dataIn = new DataInputStream(bufferIn);

// or

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

Copy file byte-by-byte: No Buffering (try-with-resources)

```
import java.io.*;
public class FileCopyNoBuffer {
    public static void main(String[] args) throws IOException{
    String inFileStr = "test-in.jpg";
    String outFileStr = "test-out.jpg";
    long startTime, elapsedTime; // for speed benchmarking

    // Print file length
    File fileIn = new File(inFileStr);
    System.out.println("File size is " + fileIn.length() + " bytes");

    try(FileInputStream in = new FileInputStream(inFileStr);
            FileOutputStream out = new FileOutputStream(outFileStr)) {
        startTime = System.nanoTime();
        int byteRead;
        // Read a raw byte, returns an int of 0 to 255.
        while ((byteRead = in.read()) != -1) {
          // Write the least-significant byte of int, drop the upper 3 bytes
            out.write(byteRead);
        }
        elapsedTime = System.nanoTime() - startTime;
        System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0)
                                                      + " msec");
    }  }   }
```

```
File size is 11119 bytes
Elapsed Time is 192.181881 msec
```

Copy File: Programmer Managed Buffer

```
import java.io.*;
public class FileCopyUserBuffer {
   public static void main(String[] args)throws IOException {
   String inFileStr = "test-in.jpg";
   String outFileStr = "test-out.jpg";
   long startTime, elapsedTime; // for speed benchmarking
   // Check file length File
   File fileIn = new File(inFileStr);
   System.out.println("File size is " + fileIn.length() + " bytes");

   try (FileInputStream in = new FileInputStream(inFileStr);
        FileOutputStream out = new FileOutputStream(outFileStr)) {
      startTime = System.nanoTime();
      byte[] byteBuf = new byte[4096]; // 4K byte-buffer
      int numBytesRead;
      while ((numBytesRead = in.read(byteBuf)) != -1) {
         out.write(byteBuf, 0, numBytesRead); }
      elapsedTime = System.nanoTime() - startTime;
      System.out.println("Elapsed Time is " +
             (elapsedTime / 1000000.0) + " msec");
    }
  }
}
```

```
File size is 11119 bytes
Elapsed Time is 0.330524 msec
```

Copy: Various Buffer Sizes

```
import java.io.*;
public class FileCopyUserBuffer {
    public static void main(String[] args)throws IOException {
    String inFileStr = "test-in.jpg";
    String outFileStr = "test-out.jpg";
    long startTime, elapsedTime; // for speed benchmarking
    // Check file length File
    File fileIn = new File(inFileStr);
    System.out.println("File size is " + fileIn.length() + " bytes");
    int[] bufSizeKB = {1, 2, 4, 8, 16, 32, 64, 256, 1024}; // in KB
    int bufSize; // in bytes
    for (int run = 0; run < bufSizeKB.length; ++run) {
        bufSize = bufSizeKB[run] * 1024;
        try (FileInputStream in = new FileInputStream(inFileStr);
               FileOutputStream out = new FileOutputStream(outFileStr)) {
            startTime = System.nanoTime();
            byte[] byteBuf = new byte[bufSize]; int numBytesRead;
            while ((numBytesRead = in.read(byteBuf)) != -1) {
                out.write(byteBuf, 0, numBytesRead); }
            elapsedTime = System.nanoTime() - startTime;
    System.out.printf("%4dKB: %6.2fmsec%n", bufSizeKB[run], (elapsedTime / 1000000.0));
        }
      }
}
```

```
Output:
File size is 11119 bytes
   1KB:   0.47msec
   2KB:   0.49msec
   4KB:   0.37msec
   8KB:   0.30msec
  16KB:   0.22msec
  32KB:   0.25msec
  64KB:   0.32msec
 256KB:   0.65msec
1024KB:   2.55msec
```

Increasing buffer size helps only up to a certain point.

Copying file byte-by-byte: Using Buffered Streams

```
import java.io.*;
public class FileCopyUsingBufferedStream {
   public static void main(String[] args)throws IOException {
   String inFileStr = "test-in.jpg";
   String outFileStr = "test-out.jpg";
   long startTime, elapsedTime;
   File fileIn = new File(inFileStr);
   System.out.println("File size is " + fileIn.length() + " bytes");
try (BufferedInputStream in = new BufferedInputStream(new FileInputStream(inFileStr));
  BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(oFileStr)))
   {
      startTime = System.nanoTime();
      int byteRead;
      while ((byteRead = in.read()) != -1) {
      out.write(byteRead); // Read byte-by-byte from buffer
         }
      elapsedTime = System.nanoTime() - startTime;
      System.out.println("Elapsed Time is " + (elapsedTime /
            1000000.0)    + " msec");
      }
   }
}
```

```
File size is 11119 bytes
Elapsed Time is 2.337563 msec
```

Character Based I/O

- Character-based I/O is almost identical to byte-based I/O.
- Instead of InputStream and OutputStream, we use Reader and Writer for character-based I/O.
- Built on top of Byte stream. (Everything is binary essentially)

Reader

Writer

Reader

- The Java Reader class (java.io.Reader) is the base abstract class for all character input streams.
- Read 16-bit char data in UTF-16 format.
- The Reader class is capable of decoding bytes into characters.

```
int read() throws IOException
```

- Reads 1 character and returns as int between 0 & $2^{16} – 1$ (65535)
- Returns *-1* if end-of-stream is detected
- Java IO contains a lot of Reader subclasses. **InputStreamReader**, **CharArrayReader**, **FileReader** etc.

FileReader Class

- Java FileReader class is used to read data from the file.
- It returns data in byte format like FileInputStream class.
- It is character-oriented class which is used for file handling in java.

```
import java.io.FileReader;
public class FileReaderExample {
    public static void main(String args[])throws Exception{
            FileReader fr = new FileReader("D:\\testout.txt");    //reader
            int i;
            while((i=fr.read())!=-1)
            System.out.print((char)i);
            fr.close();
    }
}
```

FileWriter Class

- Java FileWriter class is used to write character-oriented data to a file.
- Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

```
import java.io.FileWriter;
public class FileWriterExample {
    public static void main(String args[]){
         try{
           FileWriter fw = new FileWriter("D:\\testout.txt");   //writer
           fw.write("Welcome to java Class.");
           fw.close();
          }catch(Exception e){System.out.println(e);}
          System.out.println("Success...");
     }
}
```

BufferedReader Class

- Java BufferedReader class is used to read the text from a character-based input stream.
- It can be used to read data line by line by readLine() method.
- It makes the performance fast.
- It inherits Reader class.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class BufferedReadFile {
      private static final String FILENAME = "C:\\bufferReadfile.txt";
      public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader(FILENAME))){
         String sCurrentLine;
         while ((sCurrentLine = br.readLine()) != null) {
                  System.out.println(sCurrentLine); }
          } catch (IOException e) { e.printStackTrace(); }
   }    }
```

// Reading Text from a file

```
    StringBuilder text = new StringBuilder();

try(BufferedReader in = new BufferedReader(new InputStreamReader(
                              new FileInputStream("go.txt"), "UTF-8 "))){

       String line;

       while((line = in.readLine()) != null){

          text.append(line).append("\n");
       }
    }catch(IOException e){
       e.printStackTrace();
    }
```

- FileInputStream reads bytes from the file
- InputStreamReader translates bytes into characters using UTF-8
- BufferedReader will just buffer those characters.
- Buffer ~ char array ~ 8192 character

BufferedWriter Class

- Java BufferedWriter class is used to provide buffering for Writer instances.
- Makes the performance fast.
- Inherits Writer class.
- The buffering characters are used for providing the efficient writing of single arrays, characters, and strings.

```
import java.io.*;
public class CopyCharacterExample {
public static void main(String[] args) throws IOException {
try(BufferReader bf = new BufferReader(new FileReader(“in.txt”));
    BufferWriter bw = new BufferedWriter(new FileWriter("Buff.txt"))
    {
       String line;
       while((line = bf.readLine()) != null) {
       bw.write(line);
       bw.newLine();
    }
       bw.flush();
    }
 }
}
```

- The existing contents will be overridden.
- Use BufferedWriter – Append

```
// append to end of file
FileWriter fw = new FileWriter(FILENAME, true);
BufferedWriter bw = new BufferedWriter(fw);
```

Line-oriented I/O

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;
public class TestLinesCopy {
    public static void main(String[] args) throws IOException {
       BufferedReader inputStream = null;
       PrintWriter outputStream = null;
       try {
      inputStream = new BufferedReader(new FileReader("something.txt"));
      outputStream = new PrintWriter(new FileWriter("charoutput.txt"));
          String l;
          while ((l = inputStream.readLine()) != null) {
              outputStream.println(l);
          }
        }
       finally {
            if (inputStream != null) {
                  inputStream.close();
            }
            if (outputStream != null) {
                  outputStream.close(); }
           }
    } }
```

Invoking readLine returns a line of text with the line.