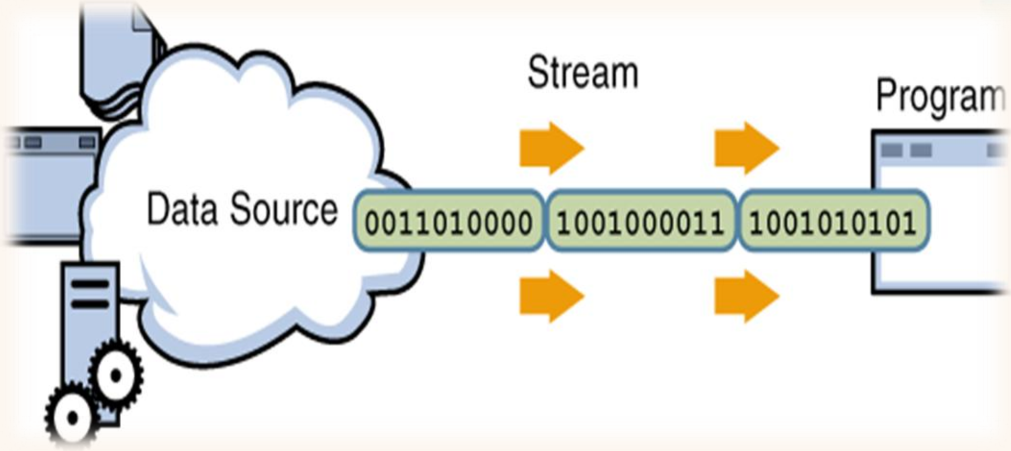
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 9

Decorator Design Pattern

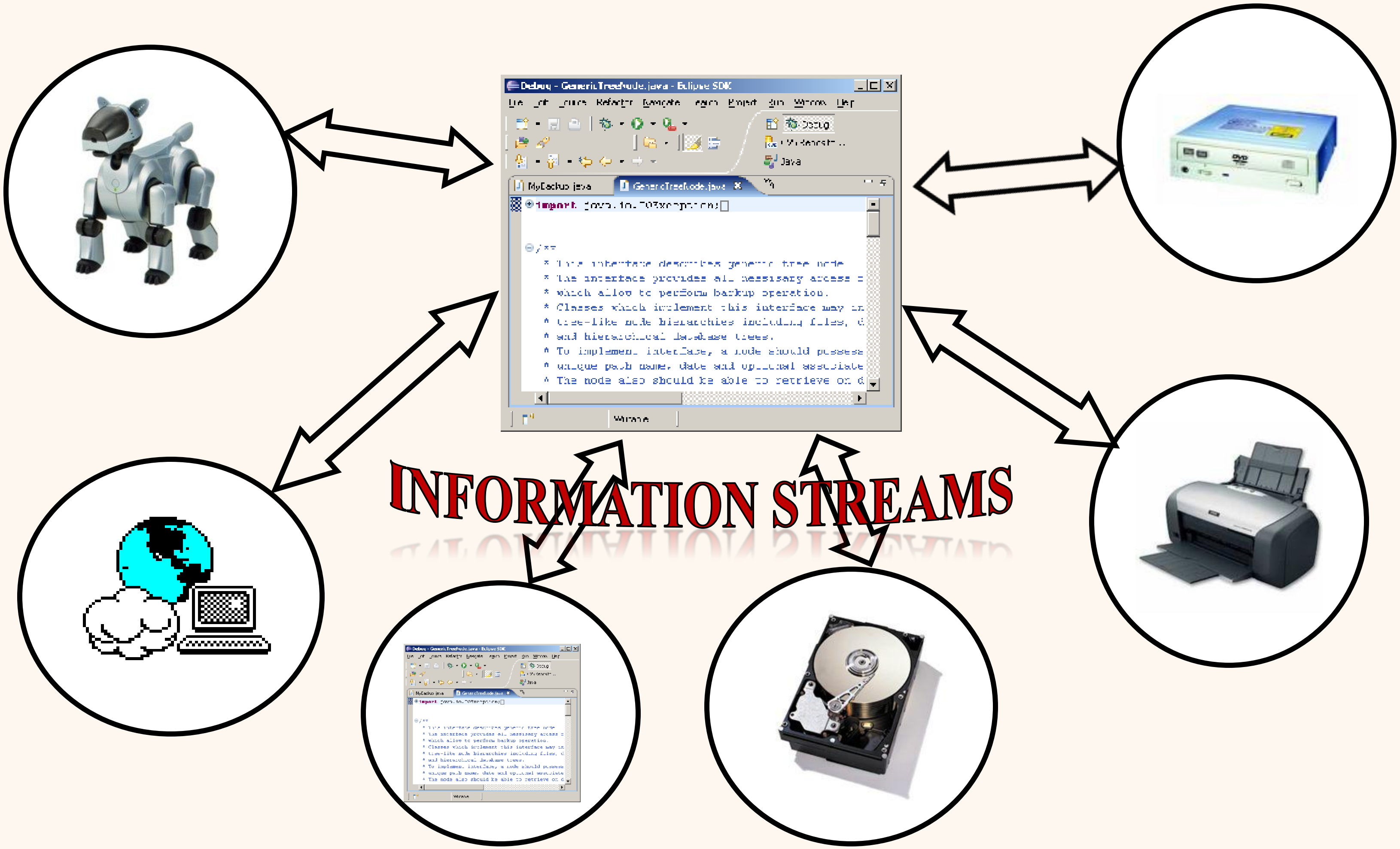
Streams



Files

Stream Concept

Program **sends** and **receives** data



Different Interface for Each Device?

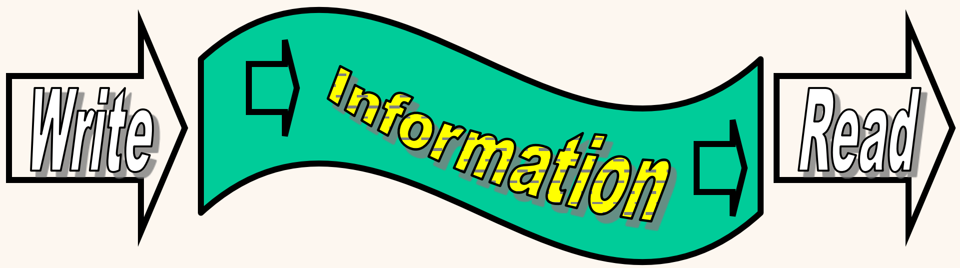
PutStringToDisk(String str)

MoveDogsLeftLeg(Speed s)

SendPageToPrinter(Page p)

GetInternetPage(Address a)

What is common?



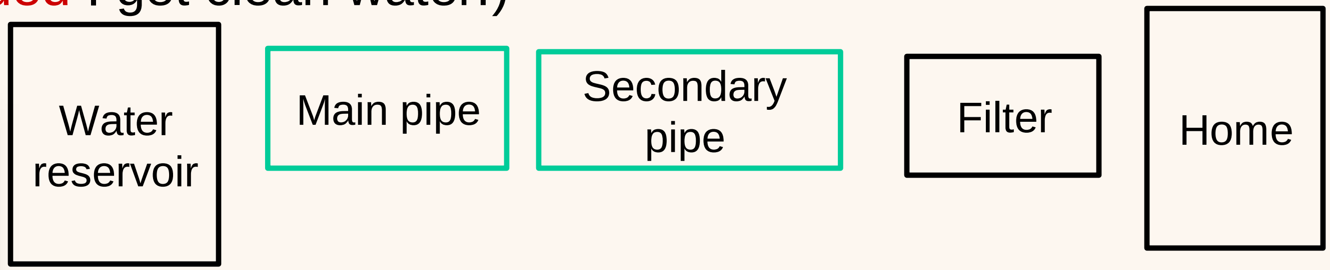
Put information into stream;
Get information from stream

The Java Stream Library

- Provides a common abstraction / set of services for stream processing

- Hides as much as possible the details of the actual sources / sinks
  - **Encapsulation**!

- Compare to: A water supply system
  - I do not care about the kinds of pipes, reservoirs and filters (provided I get clean water!)



Which Methods Do We Need?

- **Create** Stream (to whom? read or write?)
- **Write** data to stream (which? where to get it?)
- **Read** data from stream (which? where to put it?)
- **Delete** Stream (Second side should know!)
- Get **information** about streams

Basic Reading / Writing Procedure

```
1) Open a stream to (File ,Internet ,Other program…)

2) while (more data)
   2.1) Read/Write data

3) Close the stream
```

What is Data?

- **textual** data vs. **other kinds** of data
  - ‘text files’ ⇔ ‘binary files’
  - Binary files store their information in various formats
    - A reader must “understand” the format of the file
  - The structure of text files is simpler
    - It uses **unicode representation** that gives a numeric code for each symbol, and the text is stored as a list of numbers
    - Text files are binary files (**<u>not necessarily</u>** the other way around)
- Each operating system has its own (potentially non-standard) way of representing text
  - Java uses the standard unicode representation

Binary Data Encoding

- When using binary data representation, both sides need to agree on the **encoding**
  - I.e., what is the structure of the data

- There are various ways to encode the data

- There are standard representation to common data types
  - Integer – 32/64 bits
  - Boolean – 1 bit
  - …

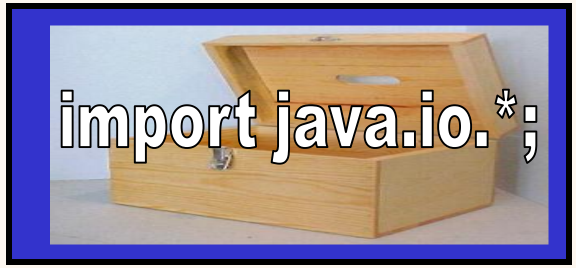
Binary Data Encoding

When using a binary data stream, it is the **responsibility of both sides** to know "what language they are talking"

- I.e., how the data is encoded

Streams in Java

Package:



Files:

Text files

Binary files

Reader

Writer

Abstract

InputStream

OutputStream

Character (Text) Streams

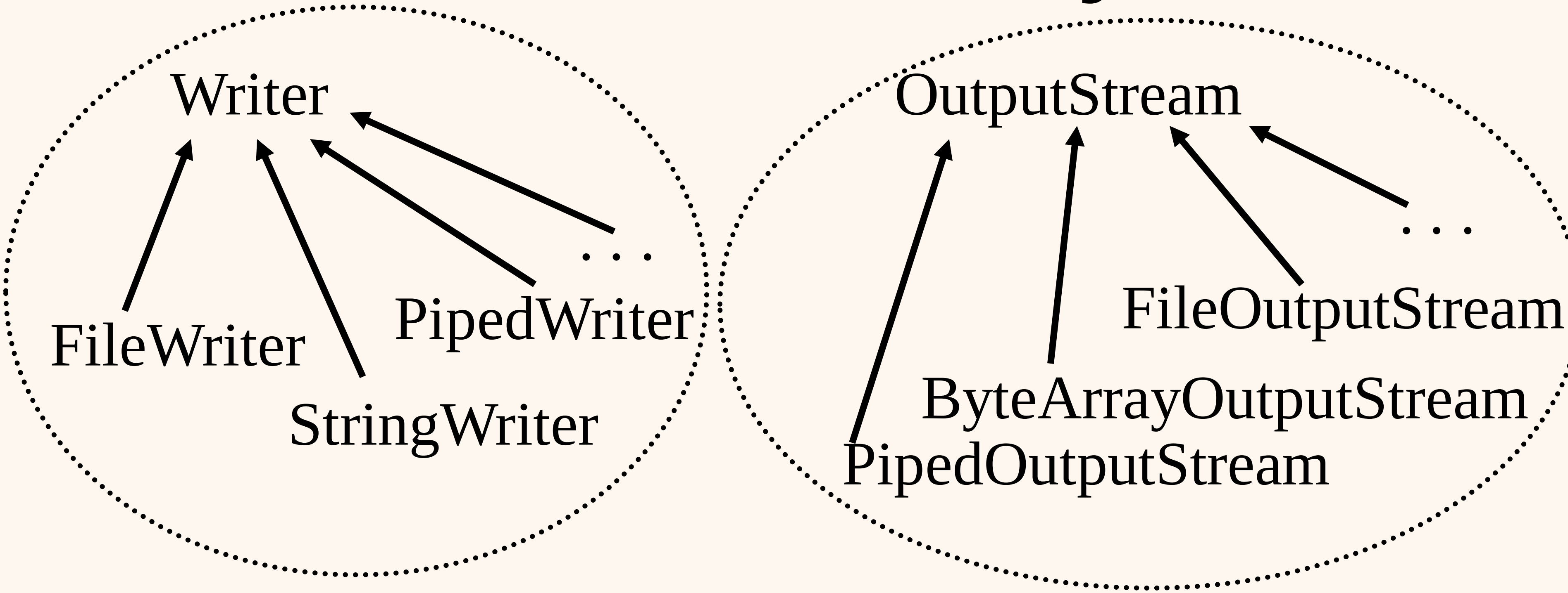
- *Reader and Writer* are the **abstract** super classes for character streams in java.io

- **Reader** provides the API and a partial implementation for readers — streams that read characters

- **Writer** provides the API and a partial implementation for writers — streams that write characters

- **InputStream** / **OutputStream** – same for binary data

Java Hierarchy



Choose class: Which device to use for I/O

```
Writer writer = new FileWriter("mail.txt");
writer.write('a');
```

File suffix is just a convention. The Writer class determines the file type

Stream Overview

I/O Type	Streams
Memory	*CharArrayReader/Writer* *ByteArrayInput/OutputStream*
Files	*FileReader/Writer* *FileInput/OutputStream*
Buffering	*BufferedReader/Writer* *BufferedInput/OutputStream*
Data Conversion	*DataInput/OutputStream*
Object Serialization	*ObjectInput/OutputStream*
Filtering	*FilterReader/Writer* *FilterInput/OutputStream*
Converting between Bytes and Characters	*InputStream/OutputReader*

Example: Copy a File

```
import java.io.*;
class CopyFile {
  public static void main(String[] args) {
    checkArgs(args); // Checking arguments length and a validity
    try {
        InputStream input = new FileInputStream(args[0]);
        OutputStream output = new FileOutputStream(args[1]);
        int result;
        // Reading the file
        while ((result = input.read()) != -1)  {
               output.write(result);
        }
        //Cleanup
        output.close();
        input.close();
    } catch (IOException ioErrorHandler) {
       System.err.println(“Couldn’t copy file”);
    }
  }
}
```

read() returns the next byte of data, or -1 if the end of the stream is reached

Typical I/O error handler

Safe Copy

java 7 only

```
try (OutputStream output = new FileOutputStream(args[1]);
        InputStream input = new FileInputStream(args[0]);) {
        int result;
        while ((result = input.read()) != -1)  {
                output.write(result);
        }
} catch (IOException ioe) {
        System.err.println("Couldn't copy file");
}  // No need to close streams! (AutoClosable interface rocks!)
```

Use this or write bad code!

So Far…

- Streams can be used for sequential data transfer
  - Open → Read/Write → Close
  - Different types for text and binary

- Further Reading
  - http://docs.oracle.com/javase/tutorial/essential/io/streams.html

Case Study 1:
Problem: Compressing a File

- **Problem**: when writing to a stream, we often want to write as little as possible
  - Save disk space
  - Network bandwidth is expensive

- It is useful to compress the data, so the same data takes less space

- **Problem 2**: We would like to be able compress data when working with **various** **input** and **output** devices

Compressing a File

A straightforward solution would be... (?)

Create I/O class for every type & device?

CompressedFileOutputStream ......

CompressedPrinterOutputStream

CompressedWebOutputStream

Case Study 2:
Efficiently Reading Bytes From a Large File

- Suppose we are given a **very large** file, which we want to read byte-by-byte

- This is **very inefficient**
  - Disk read / write operations usually involve the **Operating System (OS)** and are therefore very time consuming
    - The OS blocks the application until I/O operation is performed, which takes relatively long time
  - The basic reading mechanism of the OS is built on reading much bigger chunks of data from the disk at once
    - Reading 1000 bytes at once ≈ reading a single byte!

Efficiently Reading Bytes From a Large File

- Solution: read a big chunk of data into a buffer (in the local program memory)
  - Instead of reading the data byte by byte
  - Each time we want to read a byte, read it from the buffer instead of the actual file
  - Much more efficient
- Problem: We would like all our streams to have this functionality (not only files)

Case Study 1+2+…:
Efficiently Read Compressed Data

Write **less data** (*compressed* data), and do it **faster** (*buffered* writing)!

Efficiently Read Compressed Data

Once more: a bad solution would be…?

Extending each class?

CompressedBufferedFileOutputStream

……

BufferedPrinterOutputStream

CompressedWebOutputStream

The Design Problem

- **Objective:** Enhance streams with additional abilities
- **Problems:**
  - There are **many possible enhancements** for reading/writing data
  - There are **many types of input/output streams**
  - If we would include all enhancements in all types of streams we will end up with a **lot of duplicated** code
    - It would be hard to add new enhancements or new types of streams

Analogy

Electrical Plugs and Sockets

- There are many sockets and plugs in our world
  - All use **the same API**
- Occasionally we want to extend the functionality of the socket
  - Split one socket to many sockets
  - Extend it to reach plugs that are far away
  - Split one socket to many sockets **and** extend it to reach plugs that are far away
- We want this functionality to apply to **all sockets**

Solution:

“Decorator Design Pattern”

In order to enhance functionality of a socket*:*

- Build a **decorator** component (**extension cord**) that is also a socket (shares **the same API**) and can connect to **any socket**
- The extension cord does **not generate electricity** on its own, but gets its electricity **from the basic socket**
- The transparency allows decorators to be nested **recursively**, thereby allowing an unlimited number combinations!
  - You can put an electric splitter over an extension cord over …

What is the Analogy?

- Socket = data source InputStream
  - FileInputStream, ByteArrayInputStream, …
- Extension Cord = possible enhancement
  - Compressed reading/writing, efficient reading/writing

Recall

- Let A,B be 2 classes
  - A Composes B if
    - A **holds an instance** of B (as a member or a local variable)
  - A Delegates B if
    - A **composes** B and **forwards requests** to the composed instance (of type B)'s **methods**

Solution:
"Decorator Design Pattern"

In order to enhance functionality of class **A** (*InputStream*)*:*

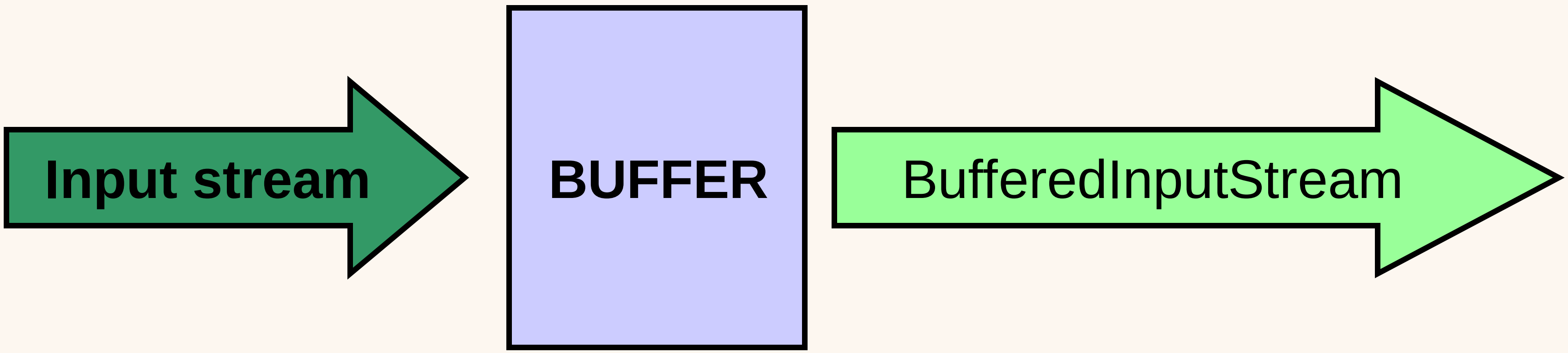
- Build class **B** (*BufferedInputStream*) that
  - **Extends A** (shares its API)
  - **Delegates its requests to a component of type A**
    - Constructor of **B** receives an object of type **A** and remembers it

- **B forwards** all requests to the **A** component and may perform **additional actions** before or after forwarding
  - Is not a data source by itself, but uses A as a data source

- The transparency allows the decorators to be nested recursively, thereby allowing an unlimited number combinations!

Buffered Streams

Reading and Writing with a Buffer



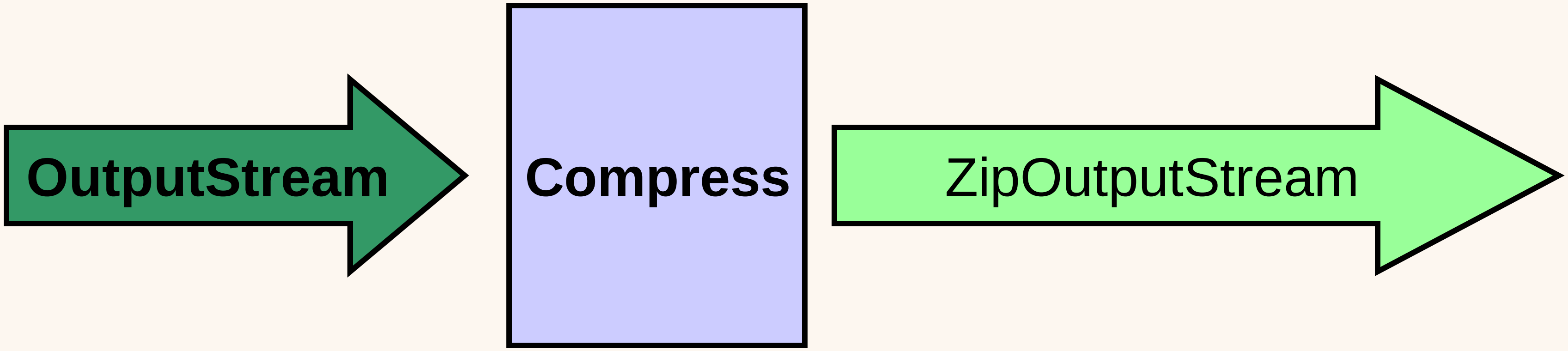

```
InputStream inFile = new FileInputStream("my_file");
InputStream inBuffer = new BufferedInputStream(inFile);

OutputStream outFile = new FileOutputStream("my_file");
OutputStream outBuffer = new BufferedOutputStream(outFile);
```

Decorator classes

Source classes

Reading/Writing Compressed Data

class **ZipOutputStream**




```
OutputStream basic = new FileOutputStream("myfile.dat");
ZipOutputStream advanced = new ZipOutputStream(basic);
```

Recursion

```
// Base stream – a FileInputStream
InputStream basic = new FileInputStream("myfile.dat");

// Efficient reading enhancement - BufferedInputStream
InputStream inBuffer = new BufferedInputStream(basic);

// Compressed reading enhancement - ZipInputStream
ZipInputStream advanced = new ZipInputStream(inBuffer);

// Now – advanced is both efficient and can read zip files
```

Decorator Notes

- Decorator classes **do not** have **their own data source**
  - They forward the read / write request to the Input/OutputStream they get in the constructor

- Similarly, the different device classes (e.g. *File* streams, *Communication* streams, etc.) are **not** decorators
  - **Conceptually** (they do not represent a functionality, but a data source)
  - **Practically** (they do not have a constructor that receives an InputStream)

To Summarize

- Let A,B be 2 classes
  - A Composes B if
    - A **holds an instance** of B (as a member or a local variable)
  - A Delegates B if
    - A **composes** B and **forwards requests** to the composed instance (of type B)’s **methods**
  - A Decorates B if
    - A **delegates** B and **extends** B

Scanner

- java.util.Scanner is a class that contains a component of type InputStream
  - It forwards reading requests to this components
  - In addition, it allows the parsing of the input text
- Scanner is useful when we want to analyze the text
  - Read text fields using delimiters, etc.
- Scanner uses a small buffer
  - Smaller than BufferedReader. Mainly affects very large files

Scanner
Design Patterns

- Scanner uses **delegation**
  - It composes a component of type InputStream, and forwards requests to that component

- Scanner is **not** a decorating class
  - It does **not extend** InputStream
  - As a result, other decorating classes cannot be nested over it

So Far…

- Decorator design pattern
  - Buffered streams and Zip streams use this pattern