Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 9:

Streams and Decorator

Last Week

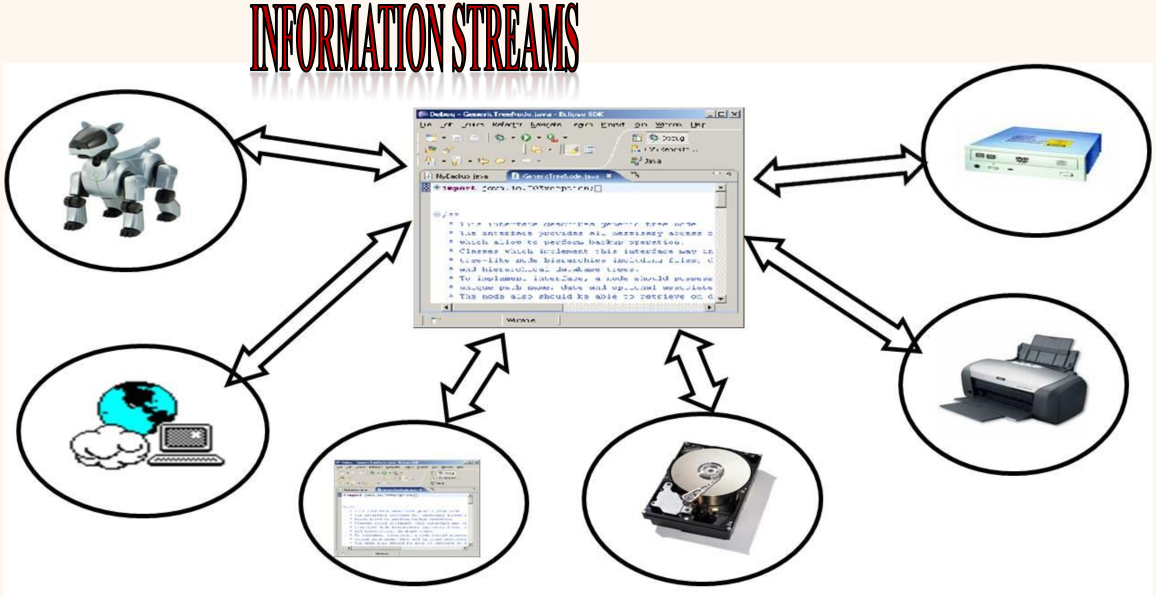
- Modularity
- More Design Patterns

Lecture 9a: Overview

- Intro to Streams
- Streams in java
- Decorator: Motivation
- Decorator: The Solution

Stream Concept

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



Different Interface for Each Device?

PutStringToDisk(String str)

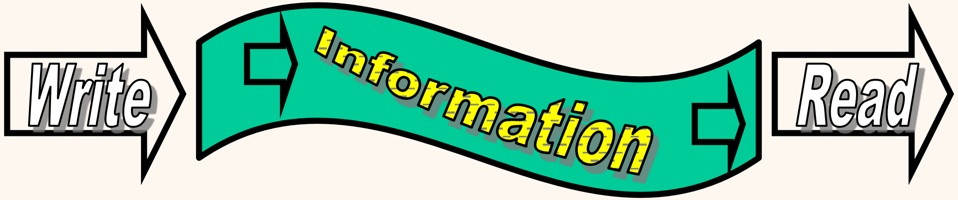
MoveDogsLeftLeg(Speed s)

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 many of the details of the actual sources / sinks
  - **Encapsulation**!
- Analogy: 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 data?)
- **Read** data from stream (where do you put the data?)
- **Delete** Stream (second side should know!)

Basic Reading / Writing Procedure

1) open a stream to (file ,internet ,another program…)

2) while (more data)
    I. Read/Write data

3) close the stream

Textual Data vs. Binary Data

- Textual files
  - Contain sequences of characters (human-readable text)
  - There are various types of text files
  - Examples: plain text files, XML files, .java / .py files
- Binary files
  - Composed of sequences of bytes
  - Not (necessarily) interpreted as text
  - Examples: jpeg, mp3, .class, .zip

Data Encoding

- Transmitting data to/from a program is a type of communication
- As with any communication, both sides need to agree on the **encoding**
  - I.e., what is the structure of the data
    - Whether textual or not
    - What each sequence of characters/bytes represent
    - …
  - Example: html files start with the following line: **<html>**
  - Counter example: trying to load an mp3 file, which is in fact a jpg file

Data Encoding

When forming stream communication, it is the **responsibility of both sides** to know “what language they are speaking”

– *I.e., how the data is encoded*

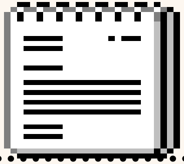
Lecture 9b: Overview

- Intro to Streams
- Streams in java
- Decorator: Motivation
- Decorator: The Solution

Streams in Java

Package:

Files:



Text files

Binary files

Reader

Writer

InputStream

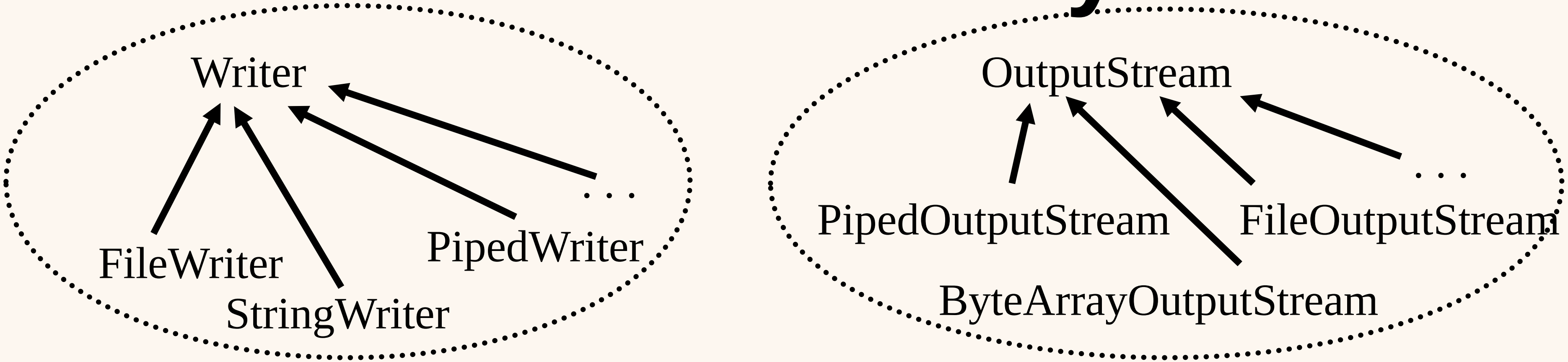
OutputStream

Abstract

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: read and write **bytes**

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.
Using a Writer class makes it a text file.

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

```
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);
        }
        output.close(); input.close(); //Cleanup
} catch (IOException ioErrorHandler) {
        System.err.println("Couldn't copy file");
}
```

But what if an error occurs before the streams are closed?
This code is never 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! (AutoCloseable interface rocks!)
```

This is the recommended way to work with streams

Lecture 9c: Overview

- Intro to Streams
- Streams in java
- Decorator: Motivation
- Decorator: The Solution

Case Study 1:

Problem: Compressing a File

- **Problem 1a**: when writing to a stream, we often want to write as little as possible
  - Save disk space
  - Network bandwidth is expensive
- **Solution 1a**: compress the data, so the **same data** takes less space
- **Problem 1b**: We would like to be able to compress data when working with **various** **input** and **output** devices

Compressing a File

A straightforward solution would be... (?)

extend each IO class?

CompressedFileOutputStream

Code repetition, hard to maintain, ...

CompressedPrinterOutputStream

CompressedWebOutputStream

......

Case Study 2:

Efficiently Reading Bytes From a Large File

- **Problem 2a**: reading a **very large** file byte-by-byte is **very inefficient**
  - Disk read / write operations are very time consuming
  - 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 2a**: 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 2b**: 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…?

extend each IO class?

CompressedBufferedFileOutputStream

……

Exponentially large number of classes!

BufferedPrinterOutputStream

CompressedWebOutputStream

The Design Problem

- **Objective:** Enhance streams with additional abilities
- **Issues:**
  - **Many possible enhancements** for reading/writing data
    - Compression, buffering, coding / encryption, etc.
  - **Many types of input/output streams**
    - Files, printers, web, etc..
  - 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 a socket
  - Split one socket to many sockets
  - Extend it to reach plugs that are far away
  - Both split it to many sockets **and** extend it to reach far away plugs
- We want this functionality to apply to **all sockets**

Lecture 9d: Overview

- Intro to Streams
- Streams in java
- Decorator: Motivation
- Decorator: The Solution

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**
  - This transparency allows decorators to be nested **recursively**, thereby allowing an unlimited number of 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
    - A **holds an instance** of B (as a data member or a local variable)
  - A Delegates B
    - A **composes** B and **forwards requests** to the composed instance's **methods**
    - **Code reuse** alternative to **inheritance**

Solution:

“Decorator Design Pattern”

- In order to enhance functionality of class **B** (*InputStream*), build class **A** (*BufferedInputStream*) that
  - **extends B** (shares its API)
  - **Delegates its requests** to a component of type **B**
    - **A**‘s constructor receives an object of type **B** and composes it
    - **A forwards** all requests to the **B** component and may perform **additional actions** before or after forwarding
- Sharing a common API allows decorators to be nested recursively
  - This allows an exponential number of combinations

Buffered Streams

Reading and Writing with a Buffer

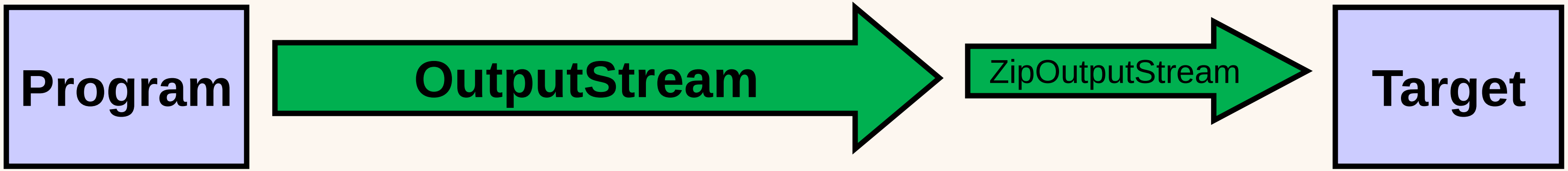

```
Reader inFile = new FileReader("my_file.txt");
Reader inBuffer = new BufferedReader(inFile);

Writer outFile = new FileWriter("my_file.txt");
Writer outBuffer = new BufferedWriter(outFile);
```

Decorator classes

Source classes

Reading/Writing Compressed Data




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

Recursion

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

// Efficient reading enhancement - BufferedInputStream
InputStream inBuffered = new BufferedInputStream(inStream);

// Compressed reading enhancement - ZipInputStream
ZipInputStream inZipped = new ZipInputStream(inBuffered);

// Now – inZipped 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 device classes (e.g. *File* streams, *Communication* streams, etc.) are **<u>not</u>** decorators in (at least) two senses:
  - **Conceptually** (they represent a data source, <u>not</u> a functionality)
  - **Practically** (they have no constructor that receives an InputStream)

To Summarize

- Let A,B be 2 classes
  - A Composes B
    - A **holds an instance** of B (as a data member or a local variable)
  - A Delegates B
    - A **composes** B and **forwards requests** to the composed instance's **methods**
    - **Code reuse** alternative to **inheritance**
  - A Decorates B
    - A **delegates** B and **extends** B
    - Add a set of **functionalities** to a set of classes

Scanner

- java.util.Scanner is a class that contains a component of type InputStream
  - It delegates reading requests to this components
  - In addition, it provides API for parsing 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 (this size difference only 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, it cannot be nested inside decorating InputStream classes

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

Next Week

- Generics
- Wildcards