Java I/O

Session Objectives

- Understand basic principles of Java I/O using streams
- Understand how the Decorator and Bridge design patterns are used in Java I/O libraries
- Understand the `File` class
- Introduce the Java logging API

Design Patterns: What Are They?

- A solution to a recurring problem
- Models or abstractions
- Design, not implementation
- Not primitive building blocks

Why Would I Use Them?

- Speed
- Quality
- Functionality
- Flexibility
- Extensibility
- Reusability

Classifications of Design Patterns

- **Creational Patterns**
  Patterns that help to abstract the construction of objects.
- **Structural Patterns**
  Patterns that define a specific structure.
- **Behavioral Patterns**
  Patterns that address solutions to specific behavior problems.
- **Reference:**
  Design Patterns; Gamma, Helm, Johnson, and Vlissides; Addison Wesley, 1995.

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural Patterns

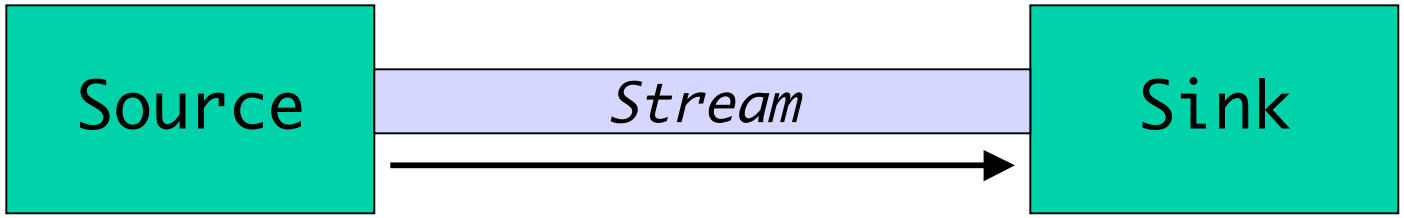
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

The Stream Model

- The stream model views all data as either a source or a sink



The Stream Model (cont'd.)

- When reading from a file, the file is the source, and the sink is wherever you're storing the data
- When writing to a file, the file is the sink, and the source is a data structure in your application
- In addition to files, what other things can serve as external sources/sinks?

The Stream Model (cont'd)

- Getting data from source to sink is the job of a stream
- Use different streams for doing different jobs
  - Write your own streams if needed
- Streams also appear in other APIs (RMI, java.net, servlets)
- Streams are the fundamental I/O paradigm in Java

OutputStream

- **When the sink is external to your application (e.g., you're writing to a file), use an** `OutputStream`
- **Abstract class**
- **Key methods:**

```
abstract void write() throws IOException
void write(byte[] b) throws IOException
void close() throws IOException
```

OutputStream (cont'd)

- **Subclasses differ in how they implement `write()` and in what kind of sink they deal with:**
  - `ByteArrayOutputStream`: **sink is a** `byte[]`
  - `FileOutputStream`: **sink is a file on disk**
  - `PipedOutputStream`: **source is a pipe from another thread**
  - `FilterOutputStream` **(we'll study this later in detail)**
  - `ObjectOutputStream` **(we'll study this later in detail)**

FilterOutputStream

- `FilterOutputStream` **is a common superclass for a set of streams that can be chained together**
- **Implementation of the Decorator design pattern**
- **Sink of one stream is the source of another**

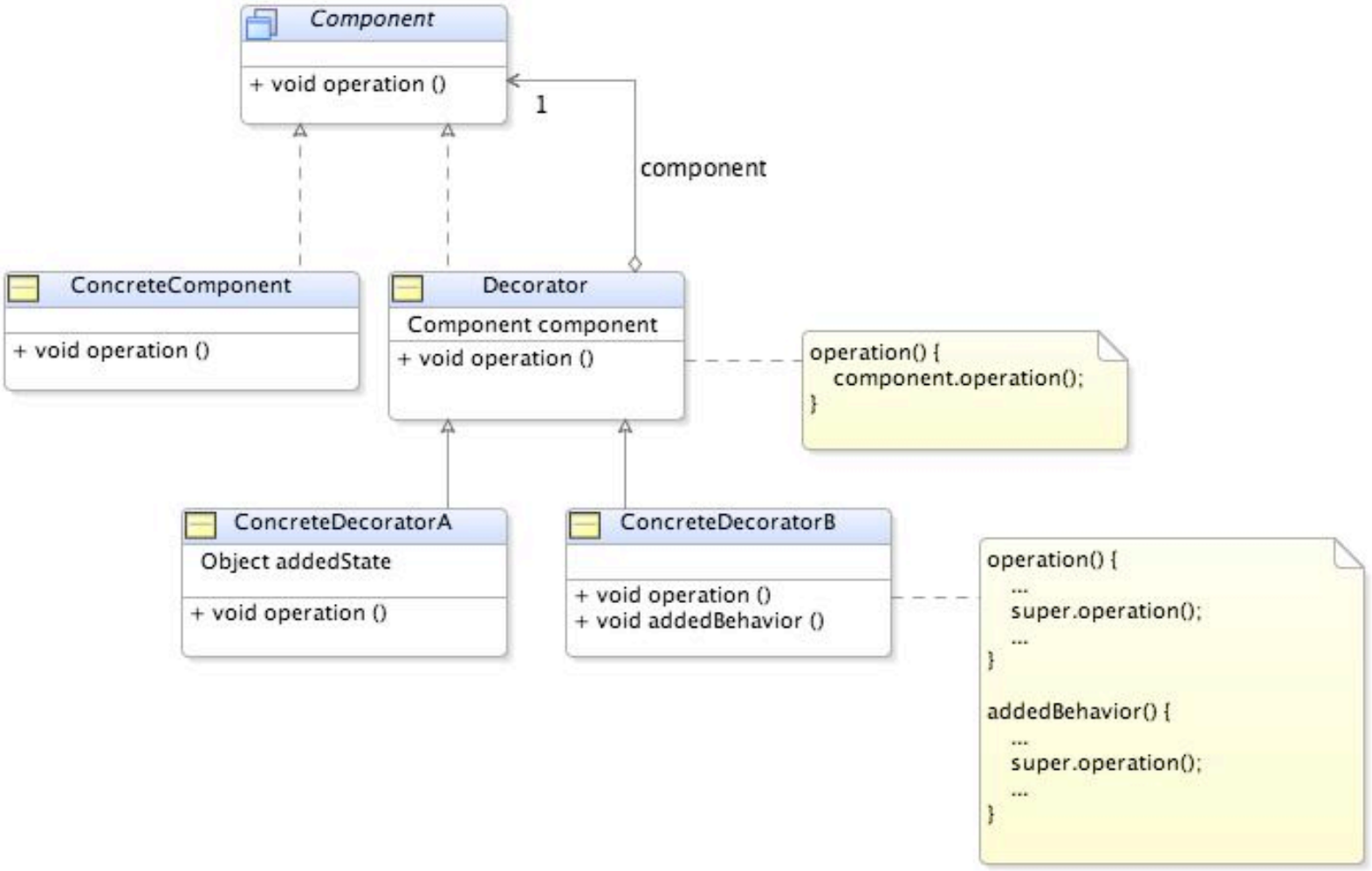
FilterOutputStream (cont'd)

- **Constructor takes an instance of** `OutputStream`
- **These classes decorate the basic** `OutputStream` **implementations with extra functionality**
- **Subclasses in java.io:**
  - `BufferedOutputStream`: **adds buffering for efficiency**
  - `PrintStream`: **supports display of data (in text form)**
  - `DataOutputStream`: **supports writing primitive data types and Strings (in binary form)**

Decorator

- Augments the functionality of an object.
- Decorator object wraps another object.
  - The Decorator has a similar interface
  - Calls are relayed to the wrapped object ...
  - ... but the Decorator can interpolate additional actions
- Example: `java.io.BufferedOutputStream`
  - Wraps and augments an unbuffered `OutputStream` object.

Decorator Structure



Decorator Example

InputStream

- **When the source is external to your application (e.g., you're reading from a file), use an** `InputStream`
- **Abstract class**
- **Key methods:**

```
abstract int read() throws IOException
int read(byte[] b) throws IOException
void close() throws IOException
```

InputStream (cont'd)

- **Subclasses differ in how they implement** `read()` **and in what kind of source they deal with:**
  - `ByteArrayInputStream`: **source is a byte[]**
  - `FileInputStream`: **source is a file on disk**
  - `PipedInputStream`: **source is a pipe from another thread**
  - `SequenceInputStream`: **source is multiple input streams**
  - `FilterInputStream` **(we'll study this later in detail)**
  - `ObjectInputStream` **(we'll study this later in detail)**

FilterInputStream

- `FilterInputStream` **is a common superclass for a set of streams that can be chained together**
- **Implementation of the Decorator design pattern**
- **Sink of one stream is the source of another**

FilterInputStream (cont'd)

- **Constructor takes an instance of** `InputStream`
- **These classes "decorate" the basic** `InputStream` **implementations with extra functionality**
- **Subclasses in java.io:**
  - `BufferedInputStream`: **adds buffering for efficiency**
  - `DataInputStream`: **supports reading primitive data types and Strings**
  - `PushbackInputStream`: **only compilers use it**

Detecting End of Stream

- There are two ways to detect when you have reached the end of the stream
- One is right, one is wrong!
- The right way is to test for null or -1 (depending on the situation) returned from a read method
- The wrong way is to catch an `EOFException`
- Test for null if the stream gets an `Object`; test for -1 if the stream returns a numeric primitive

The `File` class

- Does not represent a file, in the usual sense
- Really represents an entry in a directory
- Also can think of it as a path to a file
- Constructor is overloaded to take:
  - String, the name of the file (full path)
  - String, the directory; `String`, the name of the file
  - File, the directory; `String`, the name of the file

The `File` class (continued)

- **Examples:**

```
File startUp = new File("c:\autoexec.bat");
File mydir = new File("c:\mydir");
File myfile = new File(mydir, "myfile");
File myfile2 = new File("c:\mydir", "myfile");
```

The `File` class (continued)

- Use the File class for:
  - Listing the contents of a directory
  - Determining whether a file exists
  - Getting file info: name, size, last-modified-date, etc.
  - Getting the parent directory of a file or subdirectory
  - Renaming or deleting a file
  - Creating a directory
  - You cannot create a file using the `File` class!!! To create a file, use `OutputStream` with a new file as sink

Reader and Writer

- In Java 1.0, there was no distinction made between text streams and data/binary streams
- `PrintStream` was as close as Java got to providing text support
- Now `Reader` and `Writer` allow specific handling of text. `PrintStream` is deprecated in Java 1.1.x
- `Reader` and `Writer` automatically handle local unicode encodings

Reader and Writer (cont'd)

- Reader and Writer are abstract classes like InputStream and OutputStream, and are analogous
- Instead of ByteArrayInputStream and ByteArrayOutputStream, we have CharArrayReader and CharArrayWriter
- Reader has a StringReader subclass; Writer has a StringWriter subclass--for using a String as a source or sink

Reader and Writer (cont'd)

- `InputStreamReader` **allows you to create a** `Reader` **from an** `InputStream`
- `OutputStreamWriter` **allows you to create a** `Writer` **from an** `OutputStream`
- **These classes continue the model of chaining classes together to achieve desired functionality**

Reader and Writer (cont'd)

- **General rules:**
  - **If you're working with text (Strings and chars), use** `Reader` **and** `Writer`
  - **If you're working with primitive data types or raw bytes, use** `InputStream` **and** `OutputStream`
  - **If you get an** `InputStream` **or** `OutputStream` **from somewhere else, you can convert to** `Reader`/`Writer` **if needed**

System.in, System.out

- System.in **is a predefined** InputStream
- **You can convert it to a** Reader **like this:**

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

- System.out **is a predefined** OutputStream
- **You can convert to a** Writer **like this:**

```
Writer out =
  new OutputStreamWriter(System.out));
```

java.util.Properties

- **Represents a set of persistent string properties**
  - **String keys and values**
- **Key methods:**
  - `String getProperty(String key)`
  - `String getProperty(String key, String default)`
  - `Object setProperty(String key, String value)`
  - `void load(InputStream in)`
  - `void store(OutputStream out, String comment)`
  - `Enumeration<?> propertyNames()`

The Java Logging API

- Provides a standard way to log information about a running program to the console or log file(s)
- Classes are contained in the `java.util.logging` package

Using the Logging API

- **To use the logging API:**
  - **Instantiate a** `java.util.Logger` **object by using the** `Logger.getLogger(…)` **method**

```
Logger log = Logger.getLogger("classname");
```

  - **Simplest use is to just call the** `Logger` **methods with strings to be written to the log (the console, by default)**

```
log.info("This is a log message");
```

  - **For casual development use, replace**

```
System.out.println("debug message");
```

  - **with**

```
Logger.global.info("debug message");
```