Lecture 8

Object Oriented Design

The Challenge of Making Software

Use the four buttons to direct J*Man (the star-like piece) to capture the other colored pieces.
J*Man can capture:
   a green piece if he is yellow,
   a yellow piece if he is red,
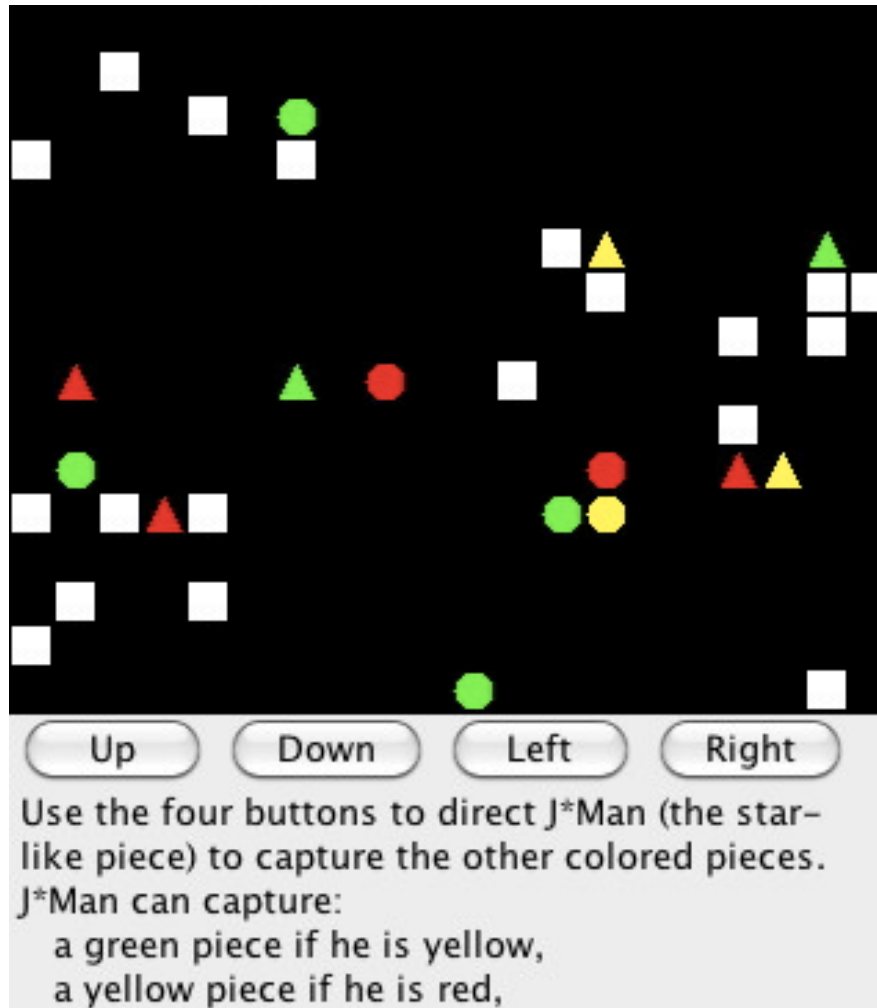
- Did a lot of JMan for you
  - Classes already completed
  - Detailed specifications
  - Lengthy instructions
  - You just "fill in blanks"
- The "Real World"
  - Vague specifications
  - Unknown # of classes
  - Everything from scratch
- Where do you start?

Software Patterns

- **Pattern**: reusable solution to a common problem
  - Template, not a single program
  - Tells you how to design your code
  - Made by someone who ran into problem first
- In many cases, a pattern gives you the interface
  - List of headers for the public methods
  - Specification for these public methods
  - Only thing missing is the implementation

Just like
this course!

Example Pattern: I/O Streams

Challenge: want to get input from somewhere
- Are these cases different?
- Or do they have a pattern?
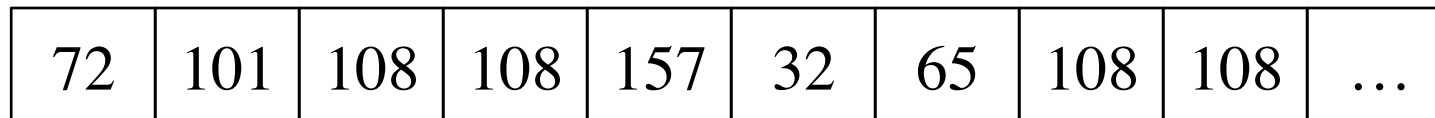
- From a file:

- From the keyboard:

- From the network

Example Pattern: I/O Streams

- **InputStream**: Read-only list of bytes (0..255)
  - Like an array, but can only read once
  - Once you read a byte, go to the next one

72	101	108	108	157	32	65	108	108	…

Read

- **OutputStream**: Like InputStream, but write-only

Example Pattern: I/O Streams

```
public class InputStream {
  /** Yields: next byte (0..255)
   *   in stream or -1 if empty */
  public int read() throws IOE{

    ...

  }
  /** Shuts the input stream
   *   down (close file, disconnect
   *   network, etc.) */
  public void close() throws IOE{

    ...

  }
}
```

```
public class OutputStream {
  /** Writes a byte to the stream
   *    Pre: b is in range 0..255 */
  public void write(int c) {

    ...

  }
  /** Shuts the input stream
   *   down (close file, disconnect
   *   network, etc.) */
  public void close() throws IOE{

    ...

  }
}
```
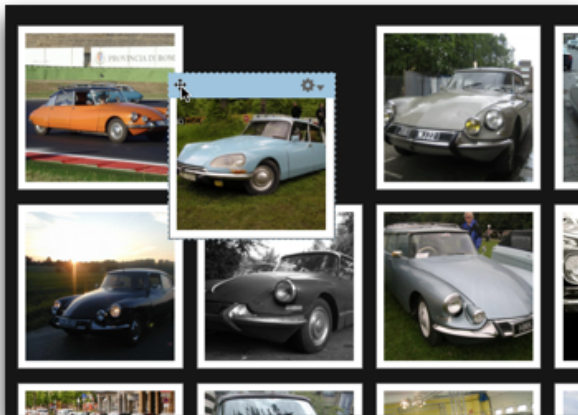
Example Pattern: I/O Streams

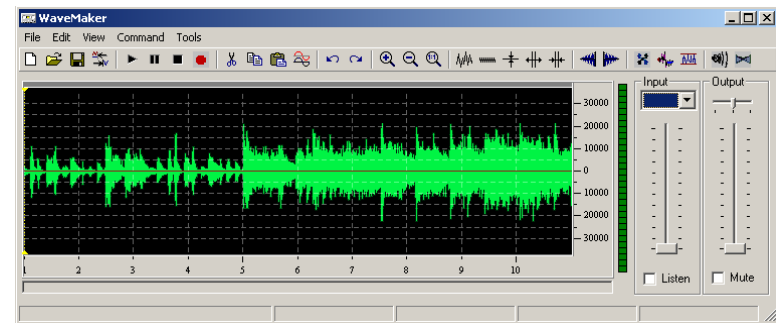**Challenge**: want I/O stream for data other than bytes

- Text:

- Images

- Sound:

- General Objects

How Many Classes Do We Need?

- ## Source:
  - Keyboard
  - File
  - Network
- ## Data Type:
  - Text
  - Images
  - Sound
  - Objects

3x4 = **12 Classes**!
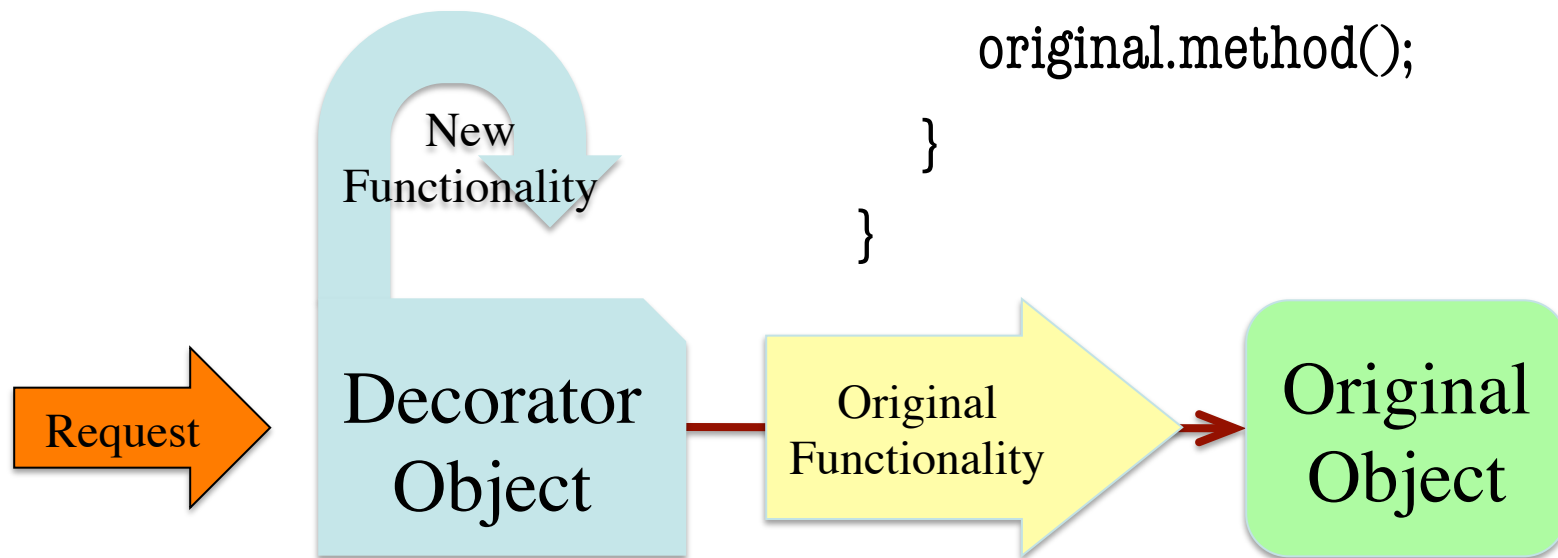
Need 3 more every time we add a new data type

Must be a better way!

Example Pattern: Decorators

```
public class Decorator {
    private Object original;
    public void method() {
        doSomethingNew();
        original.method();
    }
}
```

New Functionality

Request

Decorator Object

Original Functionality

Original Object
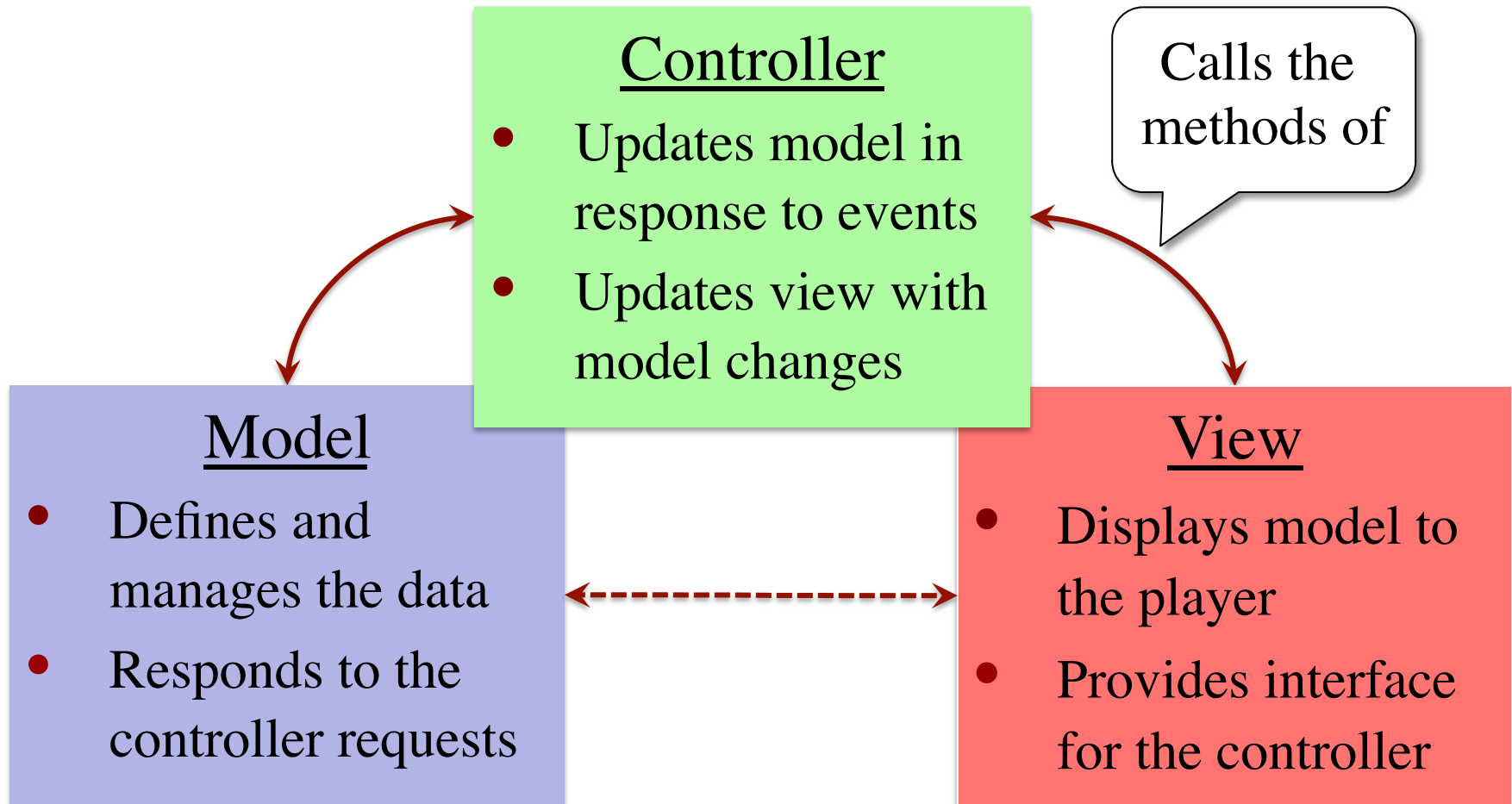
Decorators and Java I/O

- Java I/O works this way.
  - Start with basic Input/OutputStream
  - Determined by source (keyboard, file, etc.)
  - Add decorator for type (text, images, etc.)

- You did this in the lab on File I/O

```
FileInputStream input = new FileInputStream("myfile.txt");
BufferedReader reader = new BufferedReader(input);


// Read a line of text
String line = reader.readLine()
```

Architecture Patterns

- Essentially same idea as **software pattern**
  - Template showing how to organize code
  - But does not contain any code itself
- Only difference is **scope**
  - **Software pattern**: simple functionality
  - **Architecture pattern**: complete application
- Large part of the job of a **software architect**
  - Know the best patterns to use in each case
  - Use these patterns to distribute work to your team
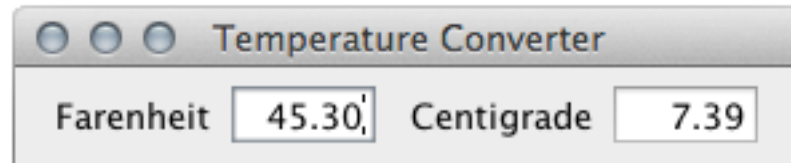
Model-View-Controller Pattern

Controller
- Updates model in response to events
- Updates view with model changes

Calls the methods of

Model
- Defines and manages the data
- Responds to the controller requests

View
- Displays model to the player
- Provides interface for the controller

TemperatureConverter Example

- Model: (TemperatureModel.java)
  - Stores one value: fahrenheit
  - But the methods present two values
- View: (TemperatureView.java)
  - Constructor creates GUI components
  - Recieves user input but does not "do anything"
- Controller: (TemperatureConverter.java)
  - **Main class**: instantiates all of the objects
  - "Communicates" between model and view
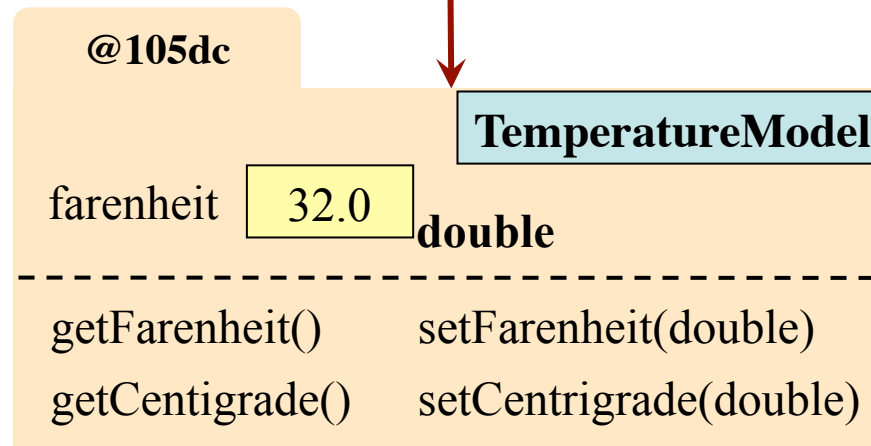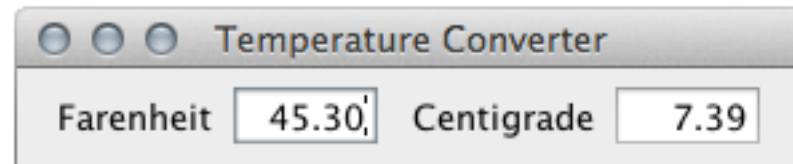
TemperatureConverter Example

View

TemperatureConverter

Controller

Model

@105dc

TemperatureModel

farenheit   32.0   **double**

getFarenheit()   setFarenheit(double)

getCentigrade()   setCentrigrade(double)

TemperatureConverter Revisited

Layout

Listener

Controller

Model

Temperature Converter

Farenheit [45.30]   Centigrade [7.39]

TemperatureConverter

@105dc

TemperatureModel

farenheit [32.0] **double**

- -

getFarenheit()   setFarenheit(double)

getCentigrade()   setCentrigrade(double)
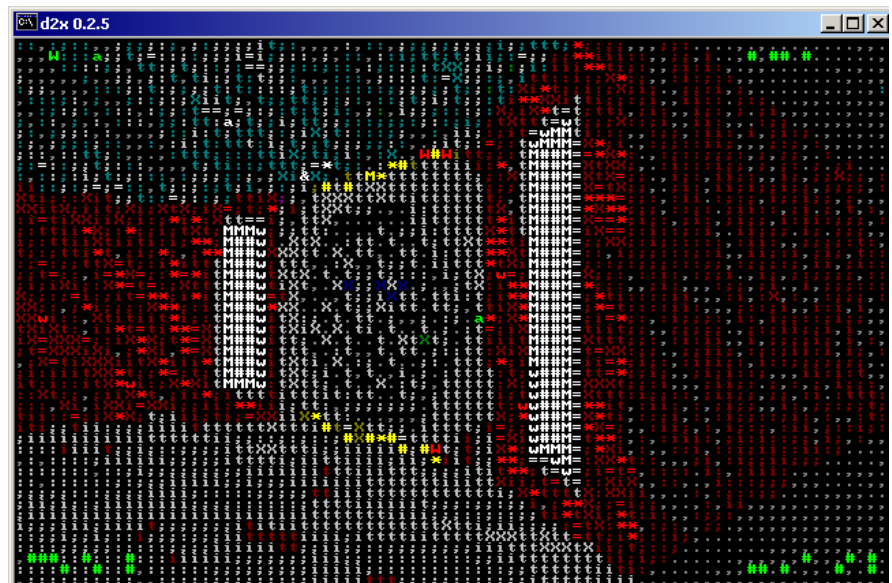
Advantages of This Approach

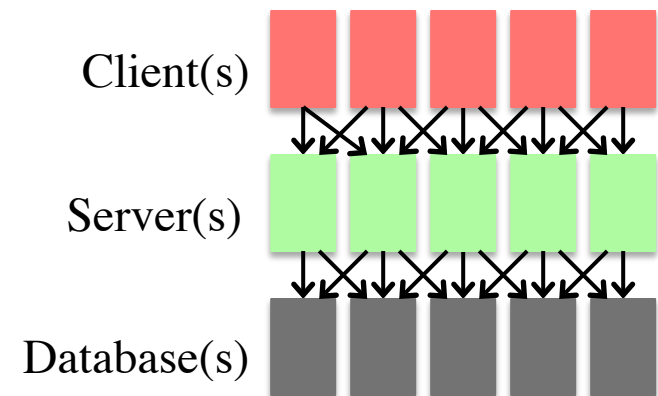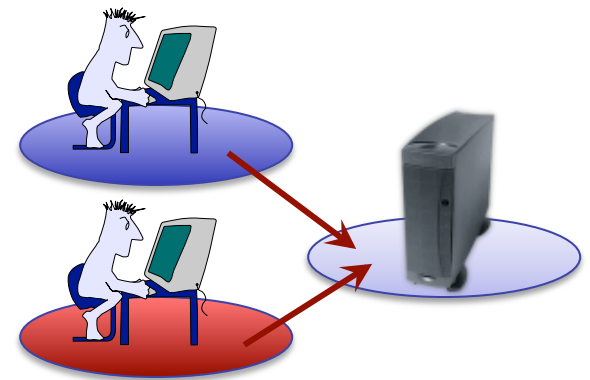View   Another View

Beyond Model-View-Controller

- MVC is best pattern for offline programs
  - Networked get more complex
- Client-Server
  - Client runs on your computer
  - Client connects to remoter server
- Three-Tier Applications
  - Client-Server-Database
  - Standard for web applications
- … and many others

Client(s)

Server(s)

Database(s)

You Can Even Mix and Match

Client

Controller

Model   View

Server

Controller

Model