CS 2113
Software Engineering

Lecture 11: File and Network IO

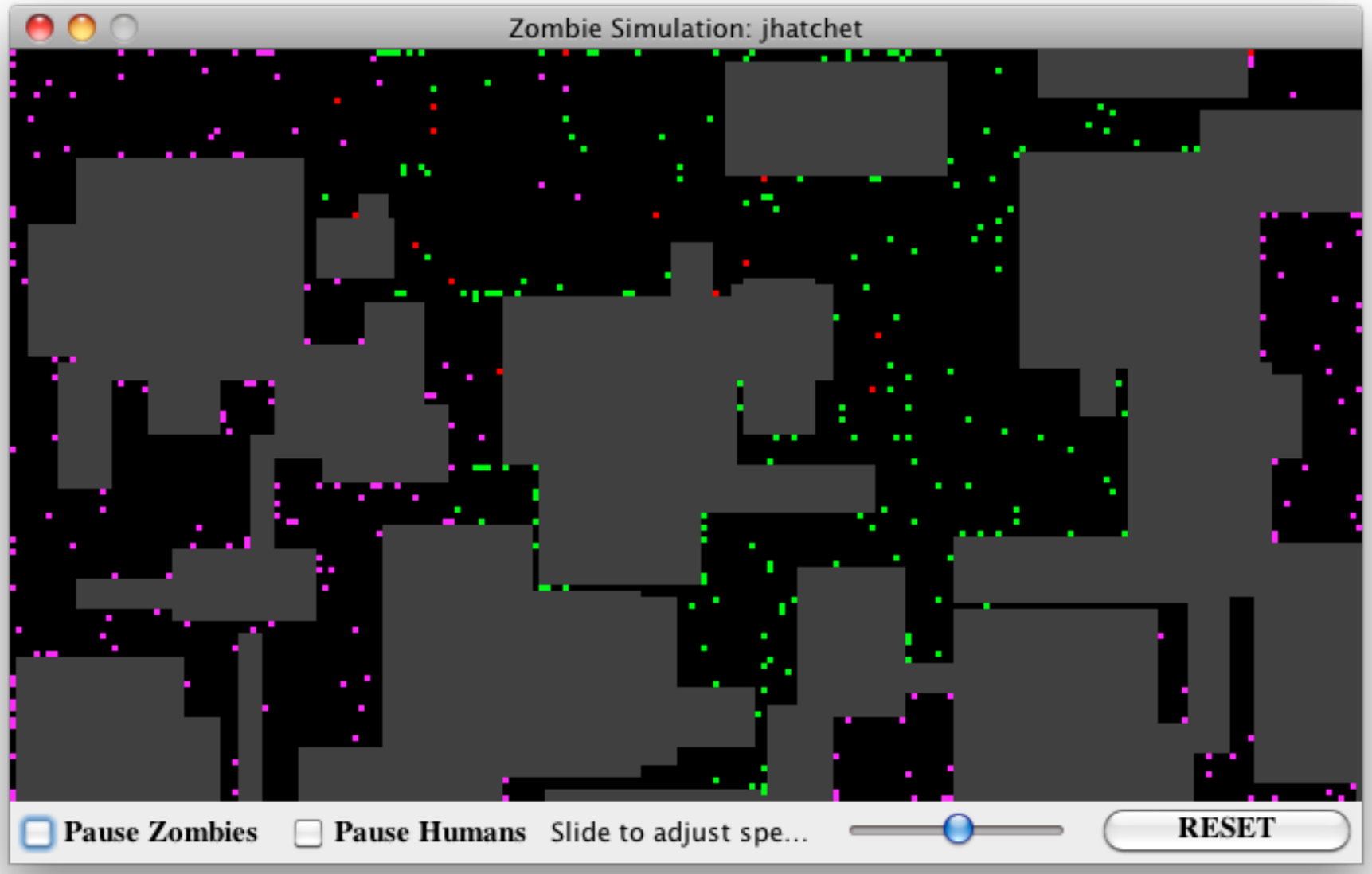
Get: https://github.com/cs2113f16/lec-11

Professor Tim Wood - The George Washington University

Project 2

- Zombies
- Basic GUI interactions

Due on Sunday!



Keyboard input

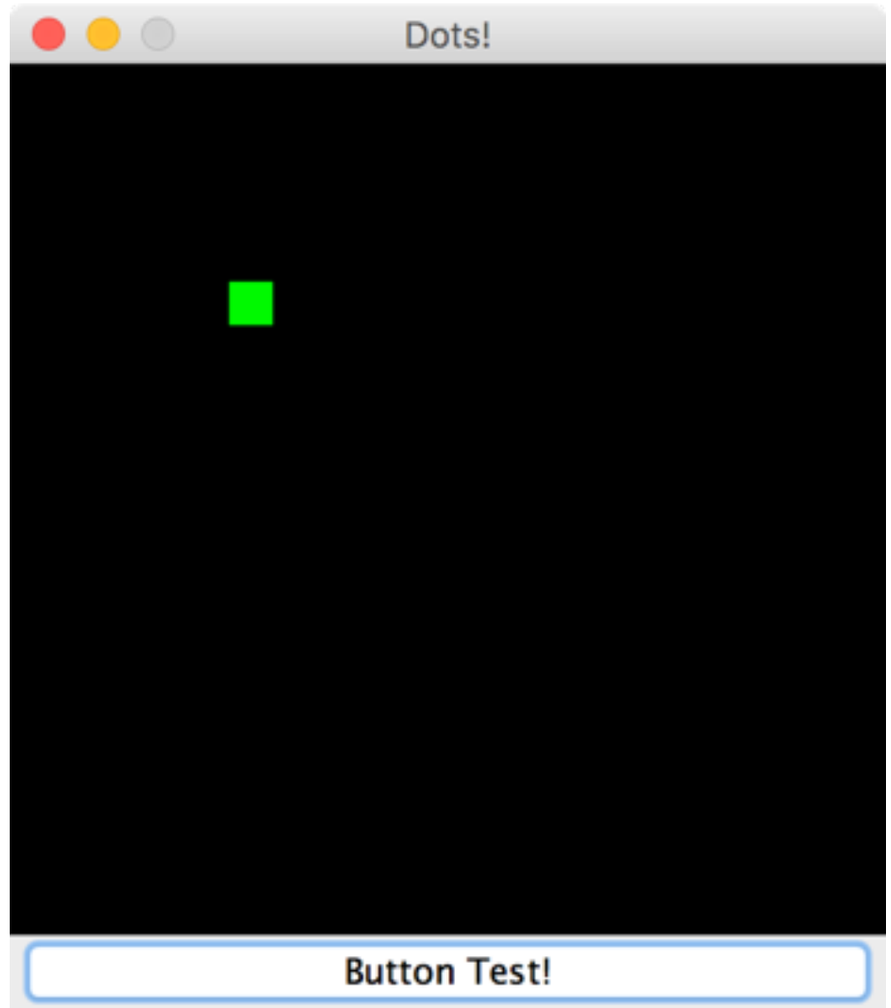
- See updated lec-10-guis for keyboard example

```
package dots;

public class DotKeys extends JFrame implements ActionListener, KeyListener {
{
  public DotKeys()
  {
    // ... in constructor add:
    this.addKeyListener(this);
  }

  @Override
  public void keyTyped(KeyEvent keyEvent) {
   System.out.println(keyEvent.getKeyChar());
  }
  // also need to implement keyPressed() and keyReleased()
}
```

https://github.com/cs2113f16/lec-10-guis

Keyboard input

- Be careful with buttons:
  - Keyboard input will go to UI widget currently in focus
  - If you have a button in your window, it will be focused and may block events from reaching the JFrame
- Solutions:
  - Prevent button from gaining focus: `button.setFocusable(false);`
  - or Use KeyBindings class instead of KeyListener
  - or Add KeyListener to the Button as well



This Week

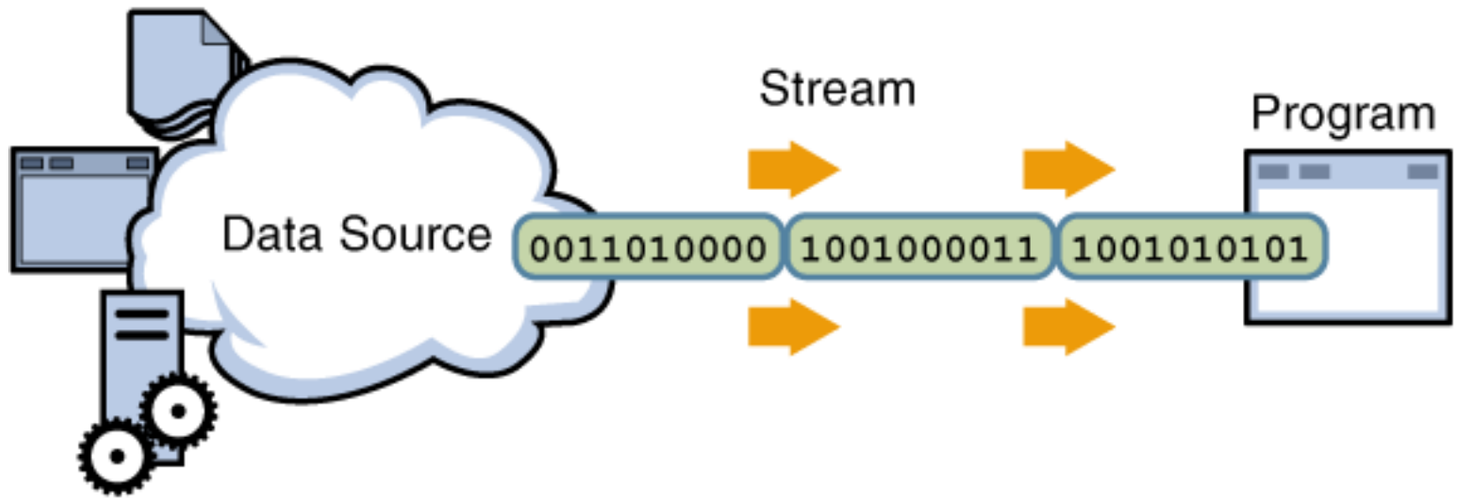
- Input and Output
  - Briefly: working with files
  - Readers, Writers, and streams
- Networking
  - Connecting with sockets
  - Sending and Receiving

Input and Output

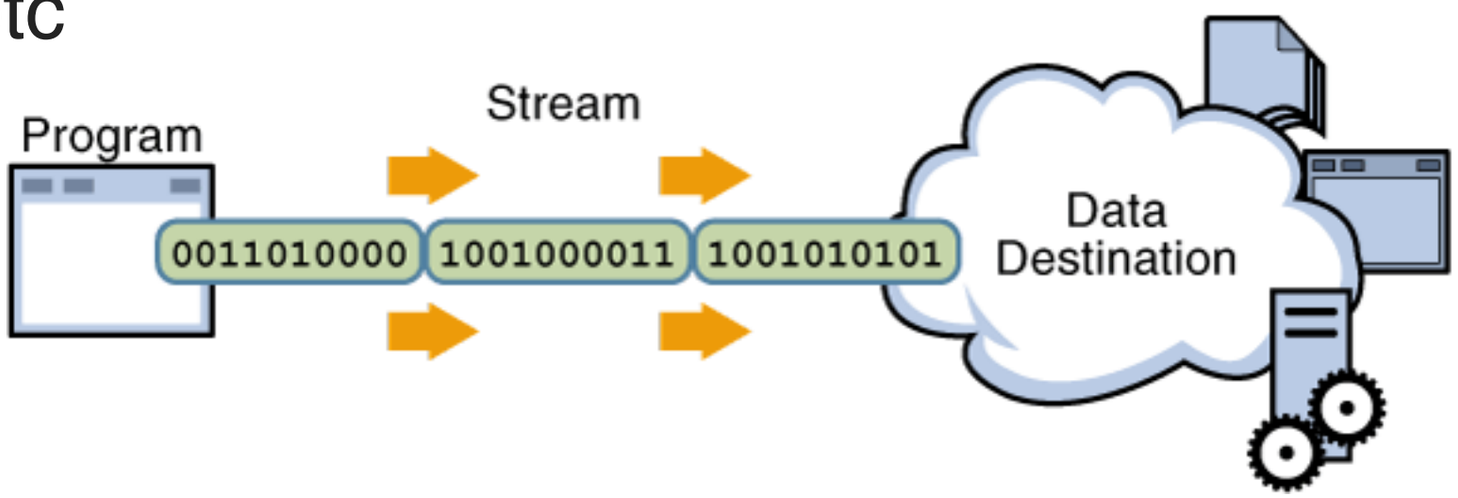
- What are examples of:
- Input?
- Output?

Input and Output

- Inputs
  - command line arguments, files, network, gamepads, keyboard, mouse, temperature sensor, webcam, other processes, etc



- Outputs
  - files, network, gamepad rumble, monitor, LEDs, speakers, robot motor, etc



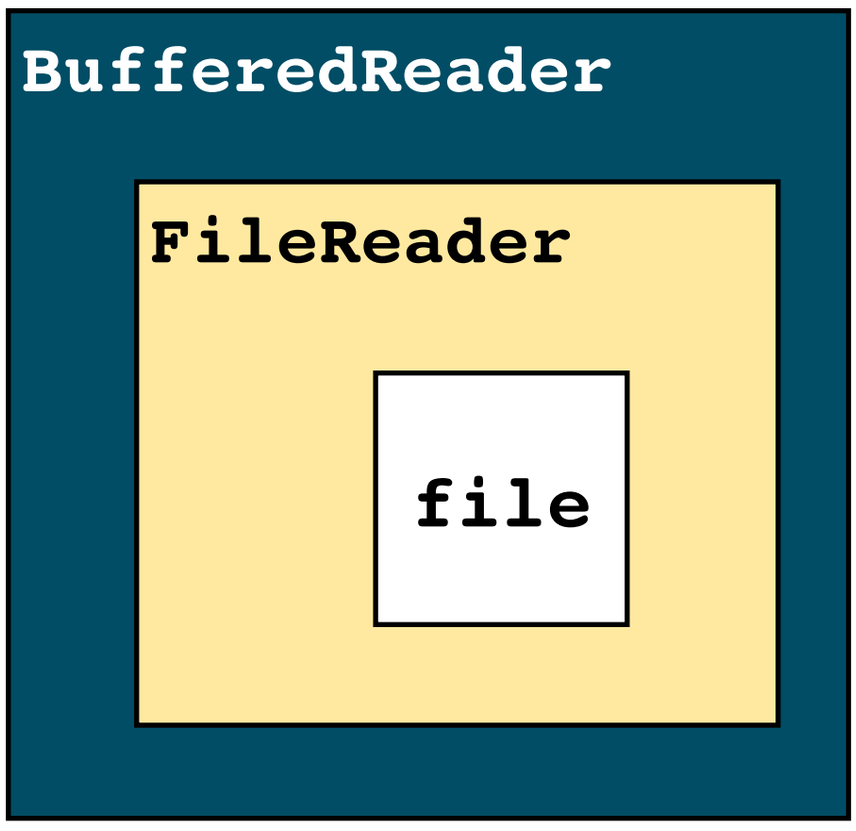
Reminder: Reading a File

```
BufferedReader freader
  = new BufferedReader(new FileReader("data.txt"));

String line = freader.readLine();
while(line != null) {
   System.out.println(line);
   line = freader.readLine();
}
```

Readers and Streams

- We prepared to read a file with:

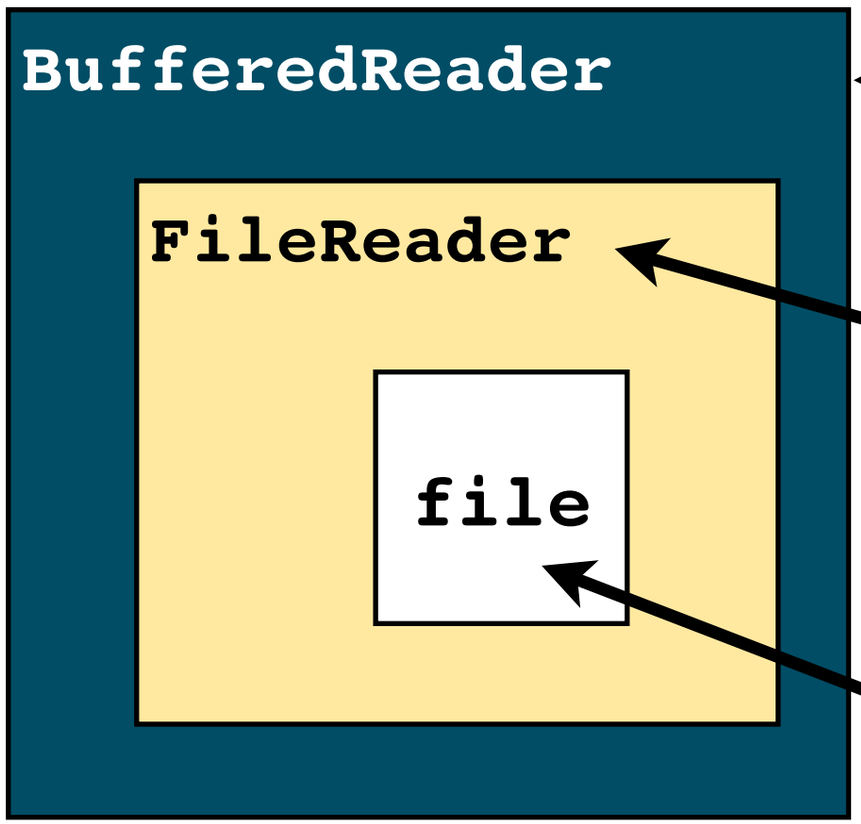
```
BufferedReader freader
   = new BufferedReader(new FileReader("data.txt"));
```



Readers and Streams

- We prepared to read a file with:

```
BufferedReader freader
   = new BufferedReader(new FileReader("data.txt"));
```



Reads lines from an input stream

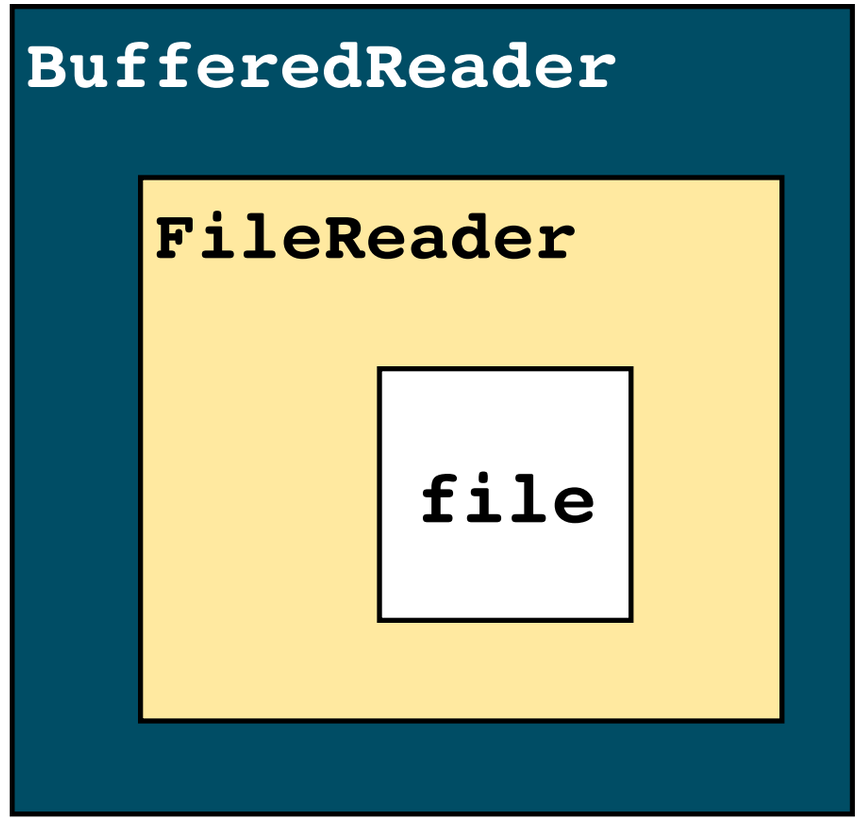
Turns bytes in a file into an input stream

Holds data or text as a series of bytes

Design Patterns

- Basic **principle**: wrapping one class inside another to provide additional functionality
  - This applies to lots of situations!
- We call principles like this Design Patterns
  - Here we have an example of the Decorator design pattern

- BufferedReader is taking a simple data stream and "decorating" it with more advanced functionality

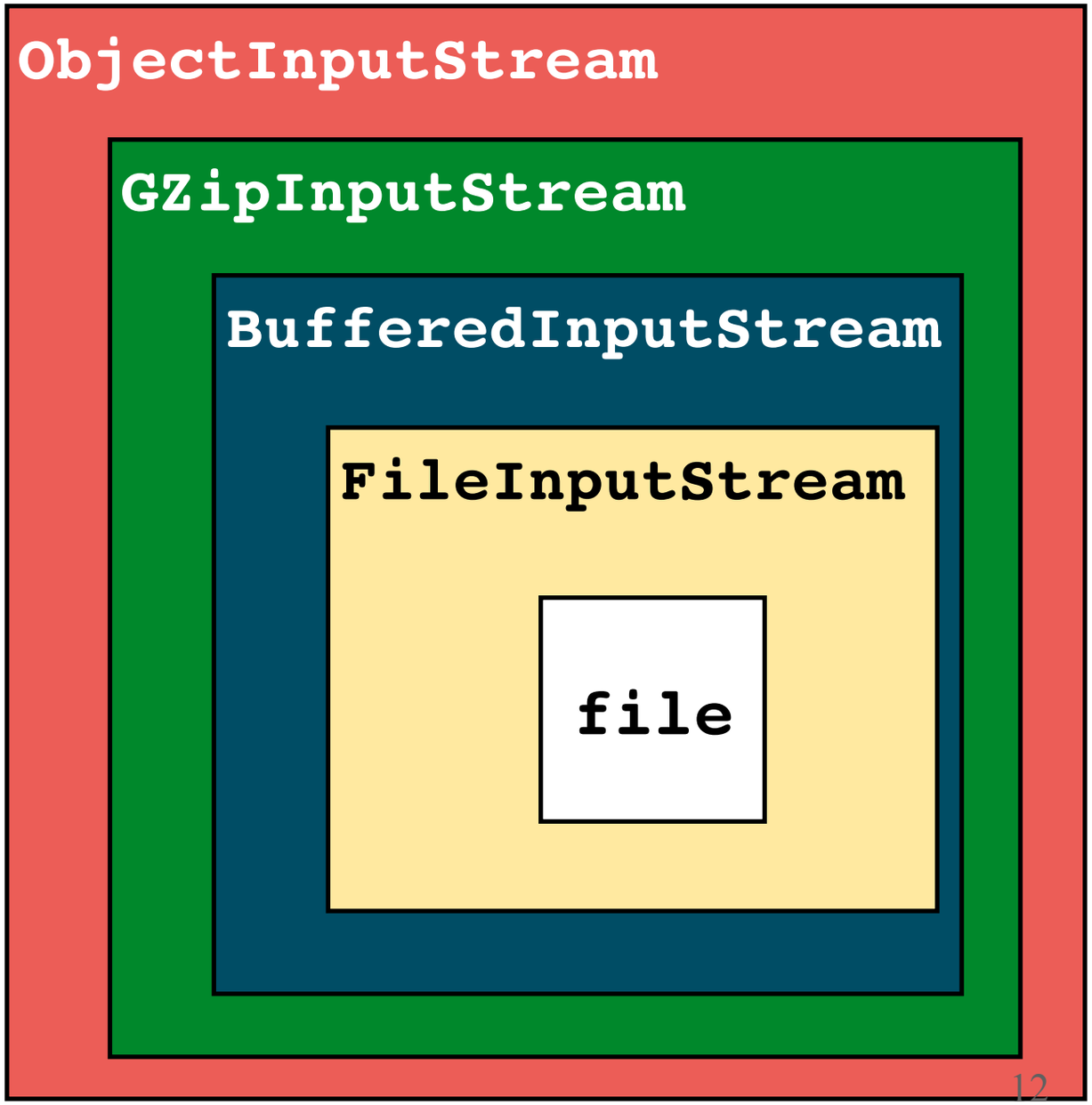


Decorator Pattern

- Can take this principle even further to flexibly add more functionality

- This combination:
  - gets 1 byte input from file
  - buffers bytes for efficiency
  - uncompresses zipped bytes
  - converts raw bytes into objects of a particular class

Learn more about design patterns this Friday 5pm! Free pizza!



Finding a random line

- Start with **`fileio.RandomLine.java`**

- Goal: store all the lines into an ArrayList and then print out a random entry
  - **Ignore lines with zero length**

- You do not know ahead of time how many lines are in the file

```
"Out, you green-sickness
 carrion..."
```

Try / Catch

- IO is unpredictable
  - What if the file is not there or the disk is full?
  - What if the server crashes?
- Java supports exception handling with **try / catch**
  - Code inside the **`try`** block is run
    - Java run time monitors for errors
  - If something goes wrong, runs the **`catch`** block
    - Can have multiple catch blocks, one for each exception type
  - Optional: run a **`finally`** block at end
    - Happens whether or not an error occurred

Writing to files

- I'll bet you can figure it out...

Files

- What does this code do?

```
//imports

public class Mystery {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;
        try {
            inputStream =
                new BufferedReader(new FileReader("file1.txt"));
            outputStream =
                new PrintWriter(new FileWriter("file2.txt"));

            String line;
            while ((line = inputStream.readLine()) != null) {
                outputStream.println(line);
            }
        }
    }
}
```

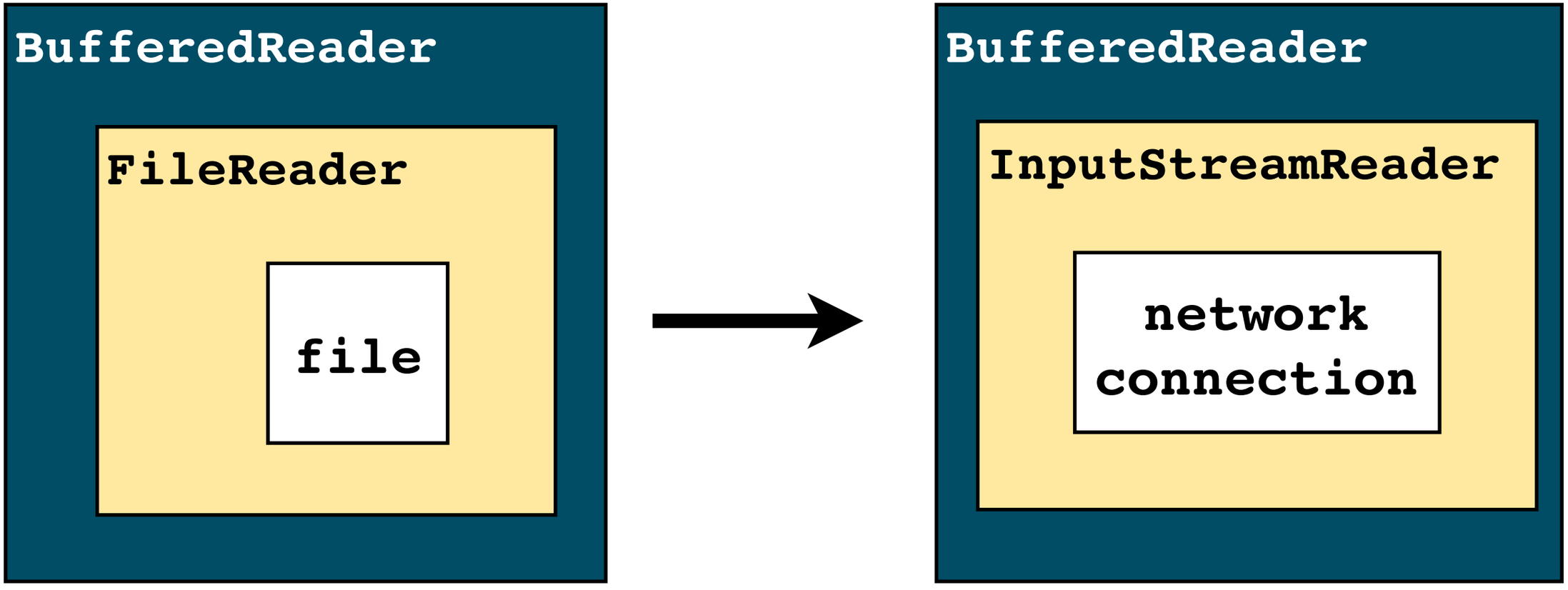
Line Reader + Writer

- Read in a file, then write it back out to a second

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

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =
                new BufferedReader(new FileReader("file1.txt"));
            outputStream =
                new PrintWriter(new FileWriter("file2.txt"));

            String line;
            while ((line = inputStream.readLine()) != null) {
                outputStream.println(line);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

From Files to Networking

- What if we want to read data over the network instead of from a file?
- We need a different data source
  - But we are still just trying to read lines



Networking Basics

- **Clients** and **servers**
  - Client initiates communication with a server
  - Server listens for incoming requests
- Who is the client/server in....?
  - Browser connecting to a web site
  - Database returning a result to an application
  - Bit torrent file sharing
- Networking is done with **sockets**
  - An endpoint of the communication channel between the client and server
  - Allows two way communication
  - Can also be used for applications running on same computer

Network Protocols

- Socket represents one end of a **TCP** connection
  - TCP = Transmission Control Protocol
  - TCP makes sending messages reliable, ordered, and fair
- Alternative: **UDP** = User Datagram Protocol
  - Does not provide reliability or ordering guarantees
  - Has lower overhead, so can make network sends faster
- What protocol would you use for?
  - Connecting to a web site?
  - A multiplayer shooter game?
  - Making a voice call over the Internet?
  - Accessing a database?

Network Protocols

- What protocol would you use for?
  - Connecting to a web site?
    - TCP: want to guarantee that client requests reach the server and client gets whatever response it produces
  - A multiplayer shooter game?
    - UDP: minimizing latency is more important than being sure that game clients get all updates
  - Streaming online video/audio?
    - UDP: missing every other frame of video or audio is better than having every frame take twice as long to be displayed
  - Accessing a database?
    - TCP: need to guarantee that connections are reliable and messages reach the server in order

In general, TCP is the most popular protocol

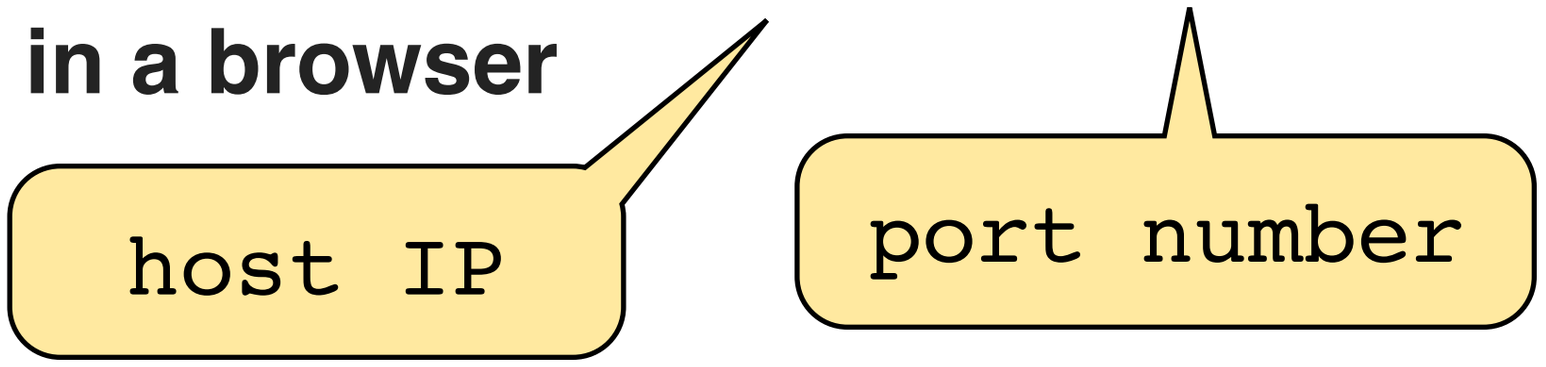
Opening a Connection

- What do you need to know to make a connection?

Opening a Connection

- What do you need to know to make a connection?
  - address of server
    - hostname (google.com) or IP address
  - port number to connect to
    - common ports: 80 for web, 22 for ssh, 3306 for mysql database

Try going to: `http://209.85.201.102:80`
in a browser



Opening a Connection

- What do you need to know to make a connection?
  - address of server
    - hostname (google.com) or IP address
  - port number to connect to
    - common ports: 80 for web, 22 for ssh, 3306 for mysql database
- Create a new socket using the host and port

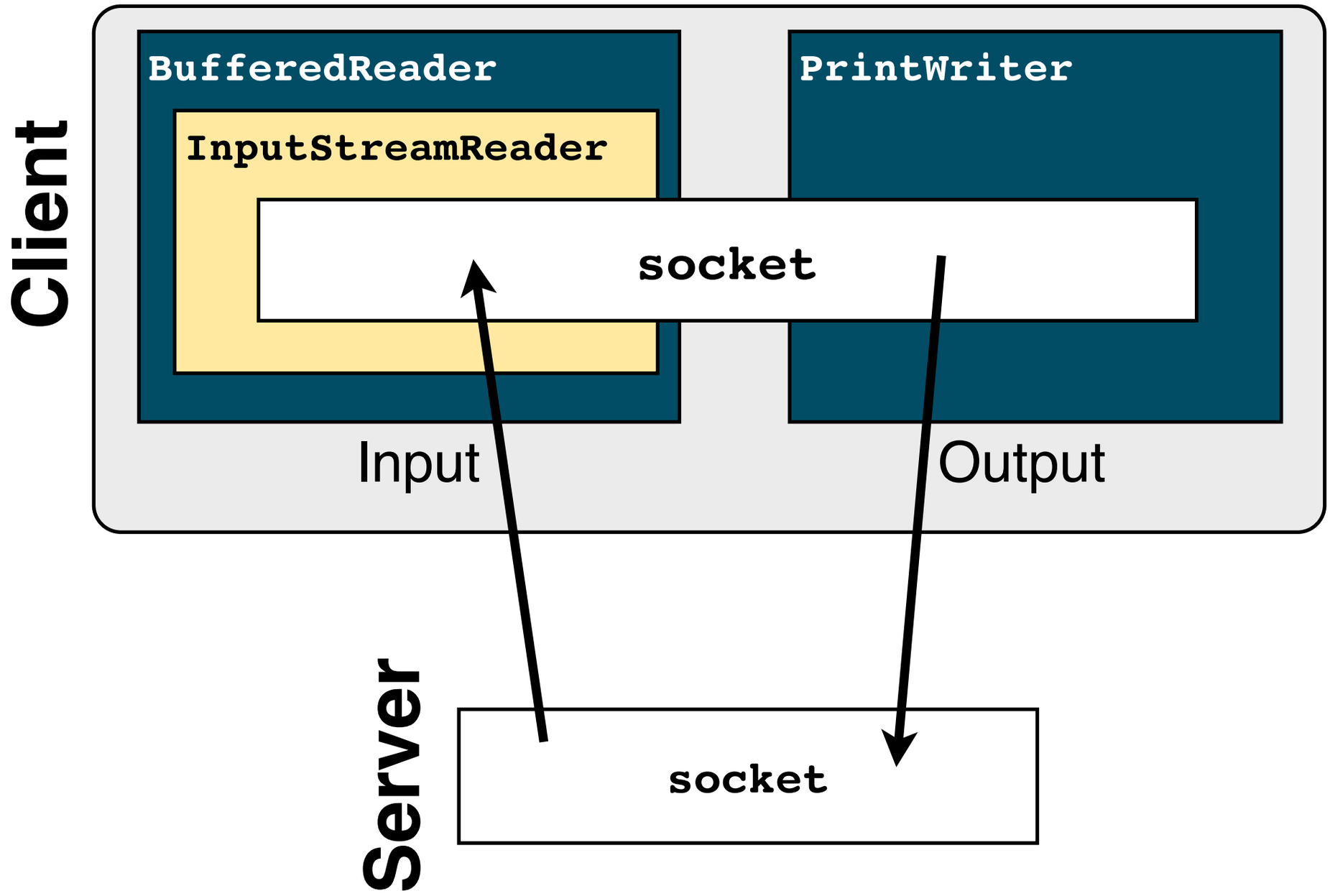
```
Socket s = new Socket(host, portnum);
```

- Work with its input and output streams:

```
BufferedReader in = new BufferedReader(new InputStreamReader
        (s.getInputStream()));
PrintWriter out = new PrintWriter(s.getOutputStream(), true);
```

Readers and Writers

```
BufferedReader in = new BufferedReader(new InputStreamReader
                        (s.getInputStream()));
PrintWriter out = new PrintWriter(s.getOutputStream(), true);
```



Read and Write

- Just like working with files!

```
// Set things up
Socket s = new Socket(host, portnum);
BufferedReader in =
     new BufferedReader(new InputStreamReader(s.getInputStream()));
PrintWriter out = new PrintWriter(s.getOutputStream(), true);

// Receive a line
String s = in.readLine();
// Send a line
out.println(s.toUpperCase());
```

Election Day!

- Oops, we are a bit late
- #1: Connect to my server
  - Create a new Socket, Buffered Writer, and PrintWriter
- #2: Vote for your favorite language:
  - Send a string: "C", "Java", "Python", "PHP", or "Assembly"
- #3: Read a confirmation message from server
  - Print it to the console

```
Socket s = new Socket(host, portnum);
BufferedReader in = new BufferedReader(new InputStreamReader
(s.getInputStream()));
PrintWriter out = new PrintWriter(s.getOutputStream(), true);
```

PrintWriters

- Why did we use the **`true`**?

```
PrintWriter out = new PrintWriter(s.getOutputStream(), true);
```

- The second argument sets whether **autoflushing** is enabled

`Enables autoflush`

- If **`autoflush = true`**, then calling **`println()`** will immediately send the line over data stream
- If **`autoflush = false`**, then it will wait until you call **`out.flush()`** or it runs out of buffer space

When to autoflush?

- If you are writing War and Peace to a file?
- If you are sending messages over the network and want an immediate response?
- If you are writing out entries to a database file?
- What is the drawback of autoflush?

When to autoflush?

- If you are writing War and Peace to a file?
  - nope: grouping lines together makes the writes more efficient
- If you are sending messages over the network and want an immediate response?
  - yes: we want a message to be sent immediately
- If you are writing out entries to a database file?
  - yes: we want to be sure that if we print out a record that it will immediately be written to the database
- What is the drawback of autoflush?
  - Autoflush can be inefficient if it leads to many small writes. This is true for both network data streams and file writers

The Server

- The basic server loop:

```
ServerSocket server = new ServerSocket(portnum); // needs try/catch block

while (true) {
  try {
    Socket sock = server.accept(); // wait for a call
    BufferedReader in = new BufferedReader(new InputStreamReader
              (sock.getInputStream()));
    PrintWriter out = new PrintWriter(sock.getOutputStream(), true);

    String input = in.readLine(); // read a message
    out.println("Message received");
    out.close();
    in.close();
    sock.close(); // hang up

  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Server Steps

- Create a ServerSocket on a specific port

```
serverSocket server = new ServerSocket(portnum);
```

- Call accept on the socket to wait for a connection
  - This creates a new socket, specifically for this client

```
Socket sock = serverSocket.accept();
```

- Setup reader and writer streams using the new client specific socket

```
BufferedReader in = new BufferedReader(new InputStreamReader
                        (sock.getInputStream()));
PrintWriter out = new PrintWriter(sock.getOutputStream(), true);
```

Knock Knock

- Work with a neighbor or two
  - in the **knockknock** package
  - one group writes client, the other writes the server
- Write a Knock-Knock joke server and client
  - The client says: "Knock Knock"
  - The server says: "Who is there?"
  - The client says: "**Something**"
  - The server says: "**Something** who?"
  - The client says: "Something wittier than this"
  - (print all messages to screen at both client and server)
- You can run **`netclient.FindMyIP.java`** to get your own IP

Client Server Protocol

- The client and server must agree on a set ordering of how they will exchange information

- What happens if **client** calls **`readLine()`** but the **server** doesn't call **`println()`**?

- What about the reverse?

Blocking Calls

- Receive calls such as **`readLine()`** are **blocking**

- The function call will not return until something is read from the data source (file or network)

- If you are writing network code and your program freezes, it is probably because of this kind of issue
  - Or your PrintWriter isn't flushing!

Mixed Up

- Split into pairs and look at **`netmismatch`** package
  - One of you will be client, one will be server

- Edit the client file to have the IP of the server
  - You can run **`netclient.FindMyIP.java`** to get your own IP

- What happens when you start the server and then run the client?

- How can you fix this?

Sending something else

- What if we want to send something more interesting?
  - An int?
  - A float?
- Use DataOutputStream and DataInputStream

http://download.oracle.com/javase/6/docs/api/java/io/DataOutputStream.html

```
DataOutputStream out=new DataOutputStream(sock.getOutputStream());
DataInputStream in =new DataInputStream(sock.getInputStream());

out.writeFloat(Math.pi);
out.writeInt(42);

int x = in.readInt();
long y = in.readLong();
```

Summary

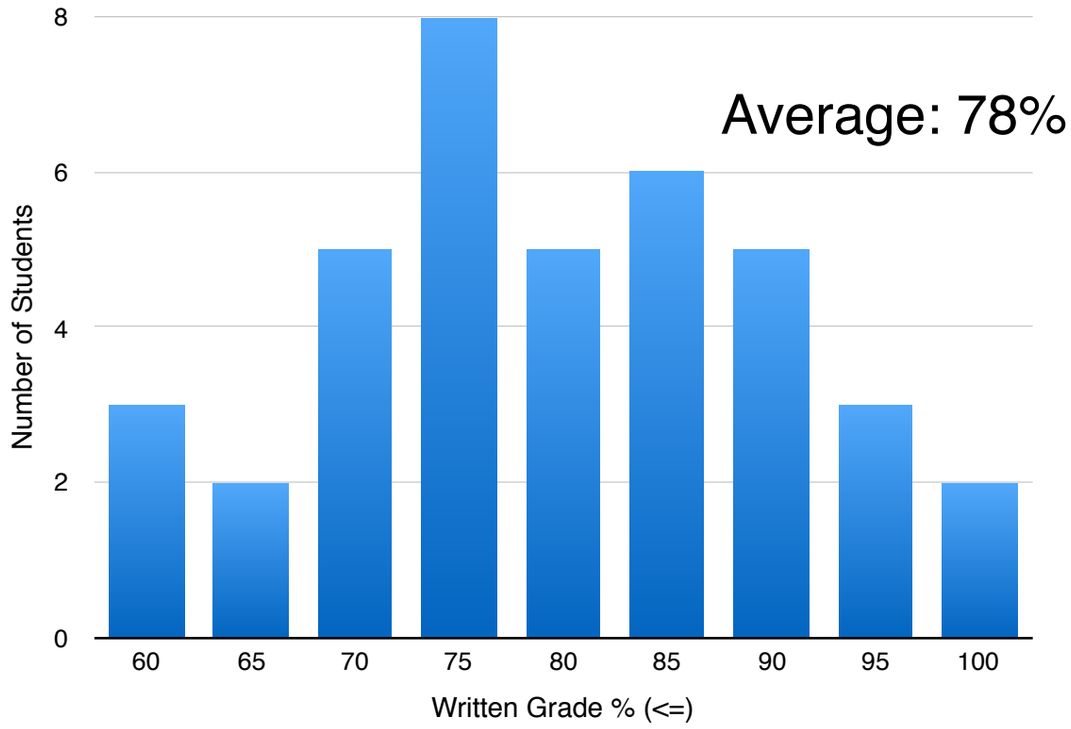
- File and Network IO are very similar in Java
  - Abstraction! Code Reuse!

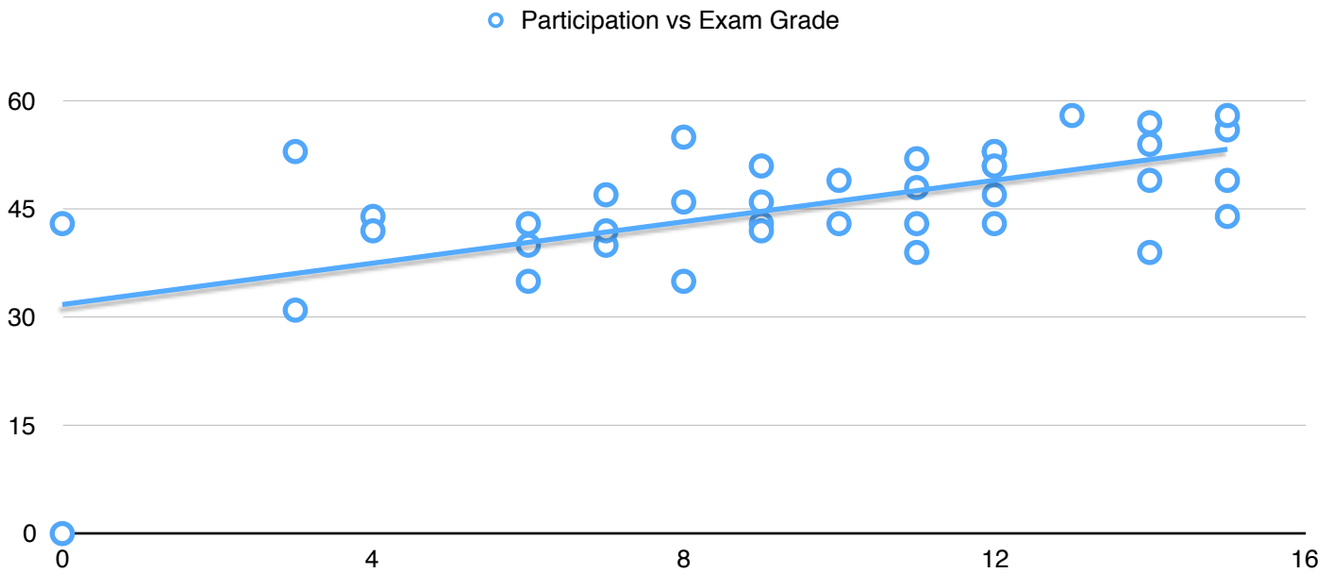
- Use different types of input and output streams depending on what you need to send

- Clients and Servers need to agree ahead of time on the protocol
  - Be careful of unmatched sends and receives!

Exam

- Written portion only (60 points)
  - Everyone got 2 points extra
- Coding is 40 points
  - Should be done soon





Sending something **else**

- ints and floats aren't interesting enough...

- I want to send a Zombie

Zombie
int x int y Color c int direction

Sending Objects

- We just need a different type of data stream*!
- To send an object:

```
ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream());

Zombie z = new Zombie();
out.writeObject(z);
```

a connected socket

- To receive an object:

```
ObjectInputStream in = new ObjectInputStream(s.getInputStream());

Object o = in.readObject();
Zombie z = (Zombie) o;
```

*and to do what is on the next slide

Serializable

- Java can read and write objects over the network or to disk using Object*putStreams

- **But**, first the object class needs to tell the compiler that it is allowed to be sent in this way!

- Need to make the class implement Serializable

```
public class Zombie implements Serializable
```

- What is inside the interface?
  - Nothing! It only acts as a marker for the compiler

Bonus!

- Object streams and serializable can also be used to write or read objects to disk!

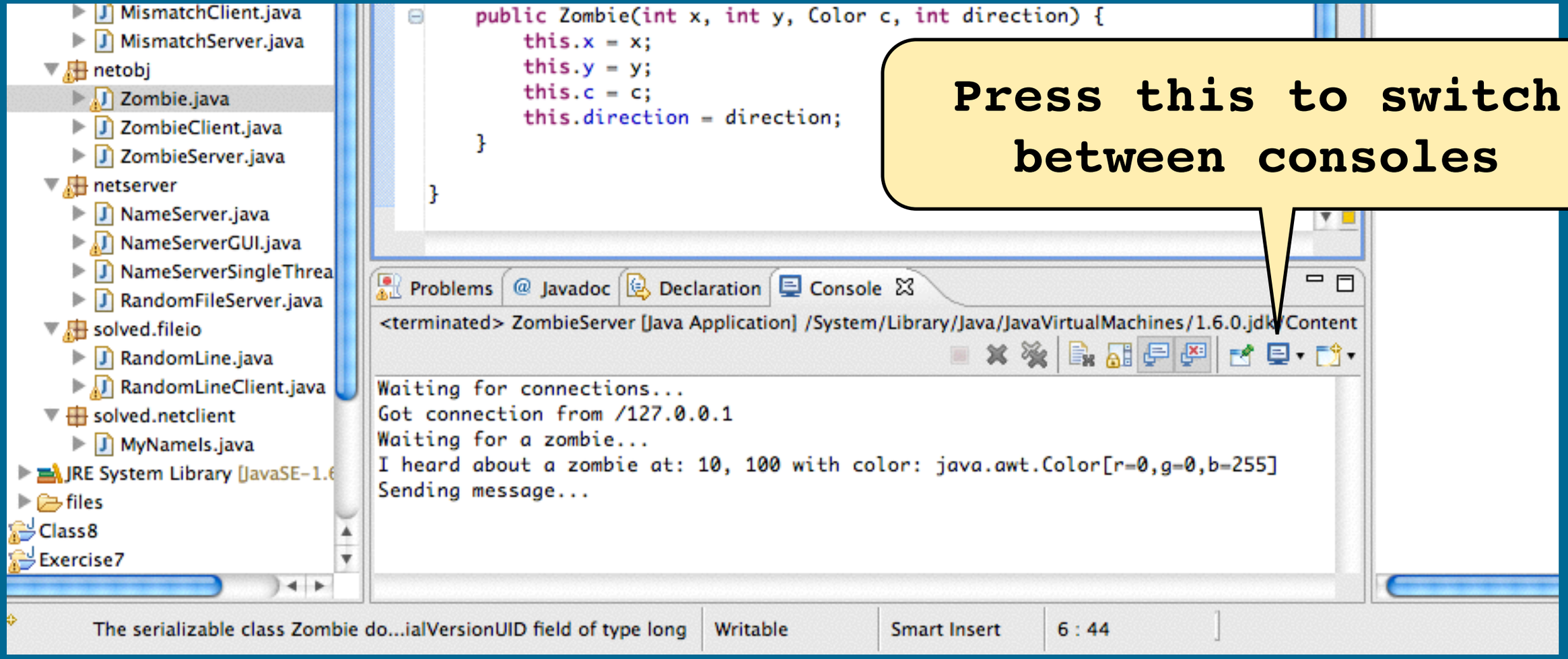
```
FileInputStream freader = new FileInputStream("date.out");
ObjectInputStream in = new ObjectInputStream(freader);

Object o = in.readObject();
Zombie z = (Zombie) o;
```

- This is why Java uses streams wrapped around streams!

- Hooray object oriented programming!

Object Sends

- Look at the **`netobj`** package
- Try running both the server and client locally



It seems so easy...

- But it's actually pretty complicated

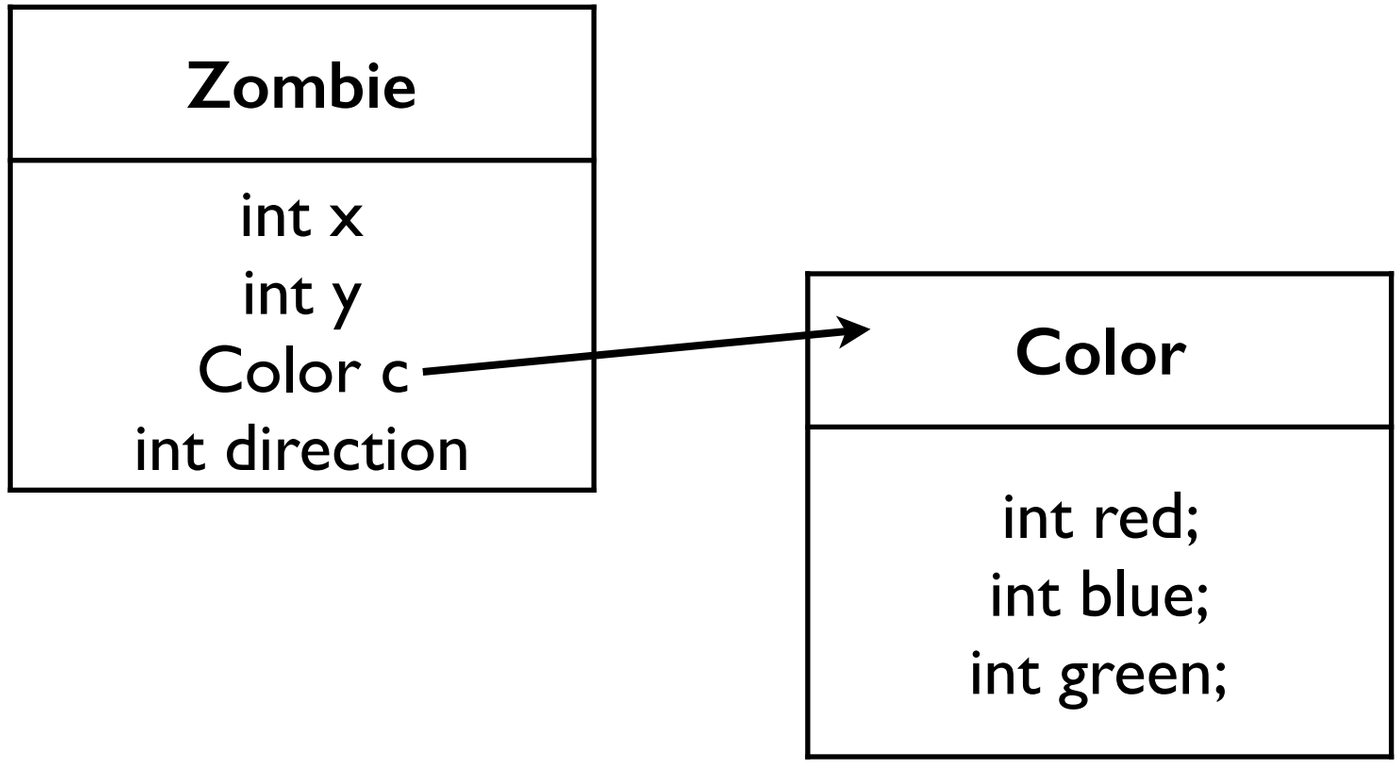
- What are we really sending with a Zombie?

Zombie
int x int y Color c int direction

It seems so easy...

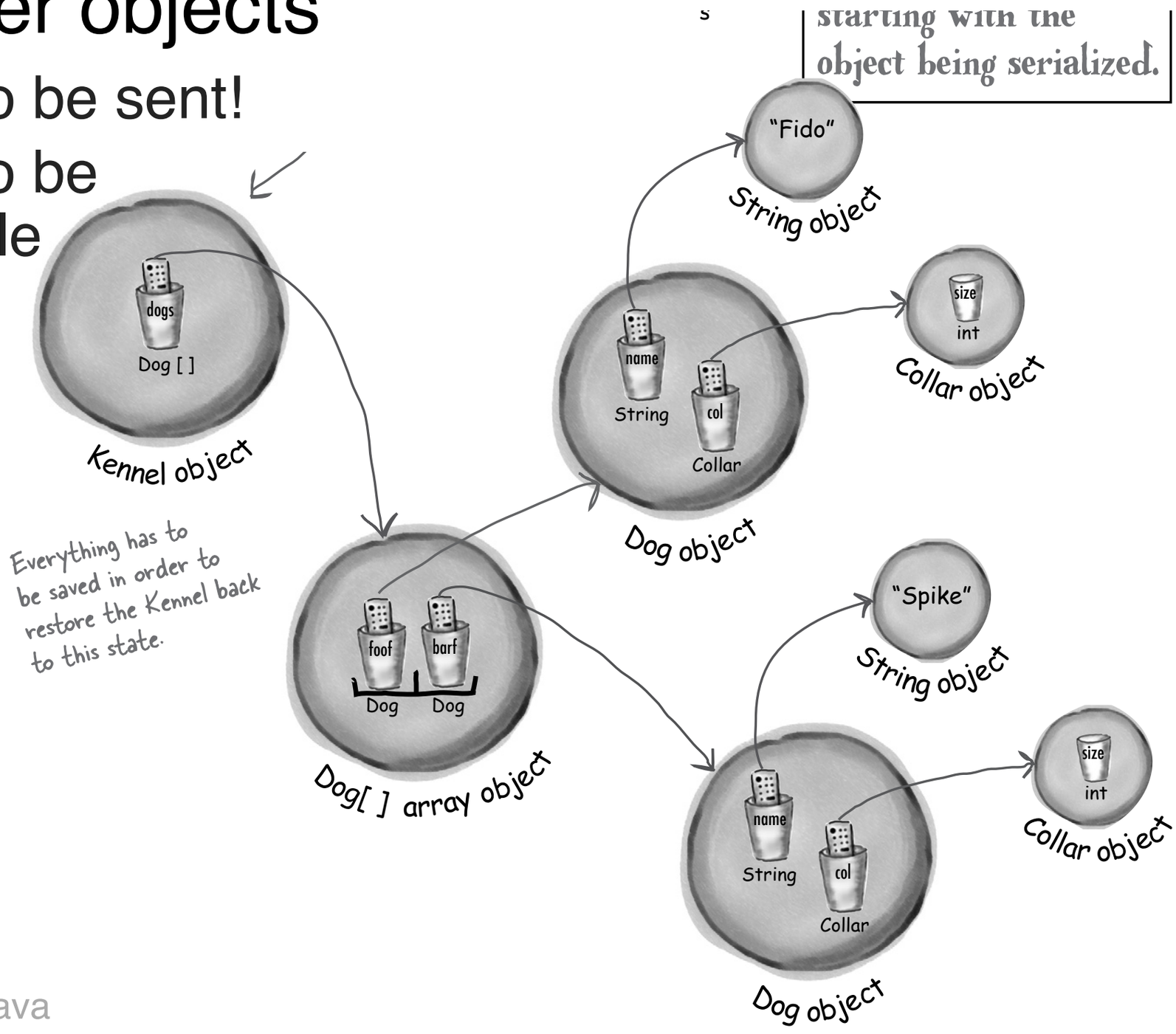
- But it's actually pretty complicated

- What are we really sending with a Zombie?



Sending Object Graphs

- The object being sent may have references to many other objects
  - All need to be sent!
  - All need to be Serializable



Example from Head First Java

Serialization Challenges

- What happens if the server is running a newer version of the code than the client
  - The fields inside a Zombie may have changed
  - The compiler assigns a version number to each class and runtime will detect if different

- What if an object has a reference to a class which does not implement serializable?
  - May cause a **java.io.NotSerializableException**
  - Solution: mark the variable as transient (will be treated as null)

```
public class Zombie implements Serializable {
   public int x, y;
   public transient DotPanel dp; // do not send
```