

Objectives

- Standard Streams
- Object Oriented Programming
 - OOP review
 - Black-box programming
 - Creating classes in Java
 - State
 - Constructor
 - Methods
- Testing classes

Review

- What is the Java syntax for
 - Conditionals?
 - Loops? (two different loops)
- How do you create a new array?
 - provide a few variations
- What are command-line arguments?
 - How do we access/use them in Java?
- What does `==` check in Java? What is its Python equivalent?
- 111/112 Review object-oriented programming terminology

Methods and Arrays

- You cannot call a method on an array because arrays are not objects
- You **can** *pass in* an array to a method
- Example:
 - `Arrays.sort(args)`
 - `args` is a being passed into the method
 - You could not compile `args.method()`

Arrays

- Assigning one array variable to another → both variables refer to the same array

➤ Similar to Python

Arrays

- Assigning one array variable to another → both variables refer to the same array

➤ Similar to Python

- Draw picture of code:

```
int [] fibNums = {1, 1, 2, 3, 5, 8, 13};  
int [] otherFibNums;  
  
otherFibNums = fibNums;  
otherFibNums[2] = 99;  
  
System.out.println(otherFibNums[2]);  
System.out.println(fibNums[2]);
```

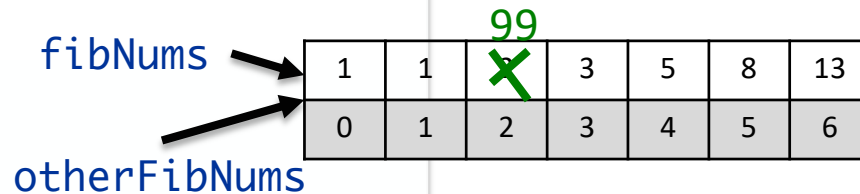
Arrays

- Assigning one array variable to another → both variables refer to the same array
 - Similar to Python
- Draw picture of code:

```
int [] fibNums = {1, 1, 2, 3, 5, 8, 13};  
int [] otherFibNums;
```

```
otherFibNums = fibNums;  
otherFibNums[2] = 99;
```

```
System.out.println(otherFibNums[2]);  
System.out.println(fibNums[2]);
```



Displays:

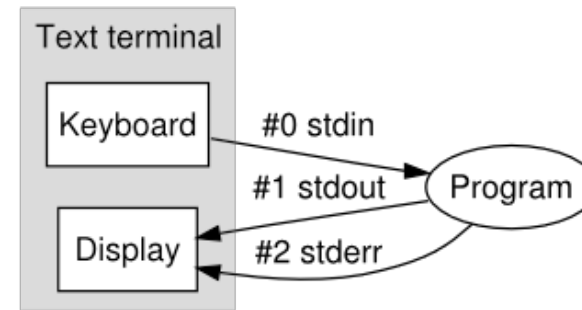
99

99

STANDARD ERROR

Standard Streams

- Preconnected streams
 - Standard Out: stdout
 - Standard In: stdin



Standard Streams

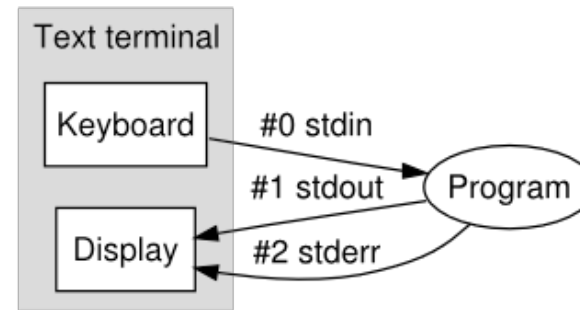
- Preconnected streams

- Standard Out: `stdout`

- Standard In: `stdin`

- *Standard Error: `stderr`*

- For error messages and diagnostics



Start thinking about the benefits
of two output streams (out and err)

Standard Streams: Java

- How we've been printing (to standard out)

```
System.out.println("Hello!");
```

- To print to standard error

```
System.err.println("Error Hello!");
```

Standard Streams: Python!

- Documentation for Python's `print` function:

```
print(...)  
    print(value, ..., sep=' ', end='\n',  
          file=sys.stdout, flush=False)
```

- file parameter says where to direct output

➤ Default is to standard out

How could you print to standard error?

Standard Streams: Python!

- Documentation for Python's `print` function:

```
print(...)  
    print(value, ..., sep=' ', end='\n',  
          file=sys.stdout, flush=False)
```

- file parameter says where to direct output

```
import sys  
print("Hello!")  
print("Error Hello!", file=sys.stderr)
```

Redirecting Output

What is the benefit of having two output streams – output and error?

- Recall `$ java Assign1 > debugged.out`
 - redirected `stdout` to `debugged.out`
 - `stderr` would still go to terminal
- To redirect `stderr` to same file as well:
- To send `stderr` to another file:

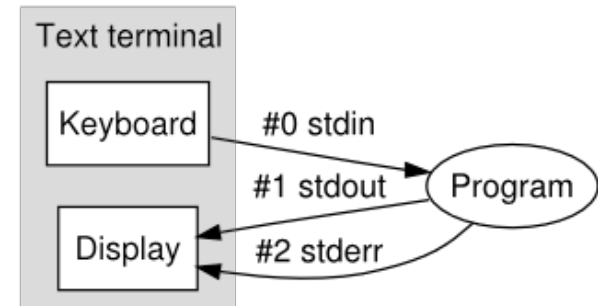
```
$ java Assign1 1> debugged.out 2>&1
```

```
$ java Assign1 1> debugged.out 2> debugged.err
```

Benefits of Separate Output and Error Stream

- Separate *output* vs *error* messages!

- Can save outputs in two different files, e.g., error.log vs output.log
- IDEs (e.g., IDLE) differentiate between output (black text) and error (red text)

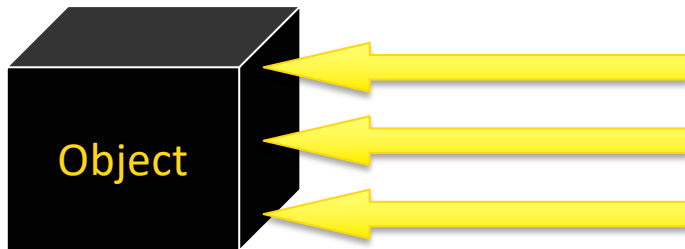


Review: Classes & Objects

- **Classes** define template from which objects are made
 - “Cookie cutters”
 - Define **state** (aka *fields* or *attributes*)
 - Define **behavior**
- Many objects can be created of a class
 - Object: the cookie!
 - Ex: Many Mustangs created from Ford’s “blueprint”
 - Object is an **instance** of the class
- **Constructor**: a special method that constructs and initializes an object
 - After construction, can call methods on object

Black-Box Programming

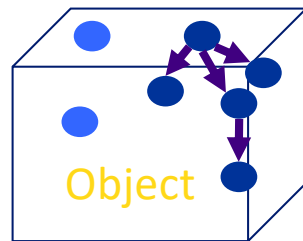
- **How** object does something doesn't matter
 - Example: if object *sorts*, does not matter to API user if implements merge or quick sort
- **What** object does (its **functionality**) matters
 - What object *exposes* to other objects
 - Referred to as **black-box programming** or **encapsulation**



- Has public **interface** that others can use
- Hides state from others

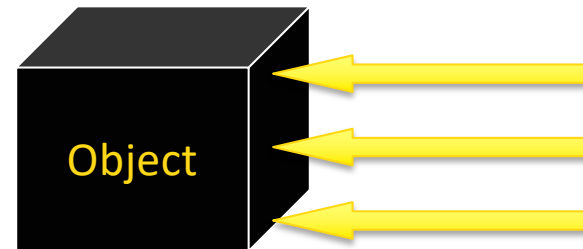
Discussion

What is the problem with white-box programming?



Others can see and manipulate object's internals

- May have unintended consequences



Java's structure helps us enforce black-box programming

Java Access Modifiers

- A **public** method (or instance field) means that any object *of any class* can *directly call* the method (or access the field)
 - Least restrictive
- A **private** method (or instance field) means that any object *of the same class* can directly call this method (or access the field)
 - Most restrictive
- Additional access modifiers will be discussed with inheritance

In general, what access modifiers will we use for instance fields? For methods?

CREATING CLASSES

Classes and Objects

- Java is **pure object-oriented programming**
 - All data and methods in a program must be contained *within a class*
- But, for *data*, can use objects (e.g., Strings or Scanners) as well as primitive types (e.g., **int**, **double**, **char**)

Java Class Structure

```
/**
 * Description ...
 */
public class ClassName {

    // ----- INSTANCE VARIABLES -----
    // declare variables that represent object's state
    private int instVar;

    // ----- CONSTRUCTORS -----
    public ClassName() {
        // initialize data structures
    }

    // ----- METHODS -----
    public int getInfo() {
        return instVar;
    }
}
```

Example: Chicken class

- State

- Name, weight, height

Discussion: what data types for instance variables?

- Behavior

- Accessor methods

- getWeight, getHeight, getName

- Convention: “get” for “getter” methods

- Mutator methods

- feed: adds weight, height to the chicken

- setName

- Convention: “set” for “setter” methods



Instance Variables: Chicken.java

```
public class Chicken {  
  
    // ----- INSTANCE VARIABLES -----  
    private String name;  
    private int height; // in cm  
    private double weight; // in lbs  
}
```

Instance variables are *declared*, with access modifier
(**private**, in this case)

Constructor: Chicken.java

```
public class Chicken {  
  
    // ----- INSTANCE VARIABLES -----  
    private String name;  
    private int height; // in cm  
    private double weight; // in lbs  
  
    // ----- CONSTRUCTORS -----  
    public Chicken(String name, int h, double weight) {  
        this.name = name;  
        height = h;  
        this.weight = weight;  
    }  
    ...  
}
```

Observations? Thoughts? Questions?

Constructor: Chicken.java

```
public class Chicken {
```

```
// ----- INSTANCE VARIABLES -----
```

```
private String name;
```

```
private int height; // in cm
```

Constructor name same as class's name

Type and name for each parameter

```
// ----- CONSTRUCTORS -----
```

```
public Chicken(String name, int h, double weight) {
```

```
    this.name = name;
```

```
    height = h;
```

```
    this.weight = weight;
```

```
    ...
```

Don't have to use **this** when variables are unambiguous

Parameters don't need to be same names as instance var names

this: Special name for the constructed object, like **self** in Python (differentiate from parameters)

Constructors

- **Constructor:** a special method that constructs and initializes an object
 - Does not return anything
- A constructor has the same name as its class
- Like `__init__` in Python
- After construction, can call methods on object

Example: Chicken class

- State
 - Name, weight, height
- Behavior
 - Accessor methods
 - `getWeight`, `getHeight`, `getName`
 - Convention: “get” for “getter” methods
 - Mutator methods
 - `feed`: adds weight, height to this Chicken
 - `setName`



Discussion: What are the methods' **input** (parameters) and **output** (what is returned and its data type)?

Methods: Chicken.java

```
private String name;
private int height; // in cm
private double weight; // in lbs

...

// ----- Getter Methods -----
public String getName() {
    return this.name;
}

// ----- Mutator Methods -----
public void feed() {
    weight += .3;
    height += 1;
}

...
```

Methods: Chicken.java

```
... Type the method returns
// ----- Getter Methods -----
public String getName() {
    return this.name;
}

// ----- Mutator Methods -----
public void feed() {
    weight += .3;
    height += 1;
}
...
}
```

Chicken object's instance variables

Recall: you don't *have* to use **this** when variables are unambiguous

Constructing objects

- Given the **Chicken** constructor

`Chicken( String name, int height, double weight )`

create a chicken with the following characteristics

➤ Name is “Fred”, weight is 2.0, height is 38

```
Chicken chicken = new Chicken("Fred", 38, 2.0);
```

Note: Static vs Instance Methods

- `main` is a **static** method
 - Latest version of Java: doesn't *have* to be **static**
- The methods we've been defining so far are ***not*** static
 - They do not have the **static** keyword as a modifier
 - They are therefore *instance* methods

More on this later...

TODO

- Assignment 2 – due *next* Wednesday before class
 - OO programming
 - Recommendation
 - Parts 1 and 2 before Friday's class
 - Part 3 before Monday's class
 - Part 4 before Wednesday's class (most can be completed by Monday)
- Textbook – Read “Defining Classes in Java” up to but not including Inheritance