Objectives

- Java fundamentals: Standard Streams
- Creating our own classes
  - Documentation: Javadocs
  - Object variables
  - Object initialization
  - Overloading
  - Overriding

1

Review

- What is black-box programming?
  - What are the benefits of black-box programming?
  - How does Java help enforce black-box programming?
- What is the structure of a Java class?
  - What does it contain?
  - What are the syntax rules?
  - What are our conventions for ordering the class?
- What is the Java equivalent of `self`?
- What is our process for developing a class?

2

Access Modifier Example

- If a method is *private* to a class, other classes cannot call that method.

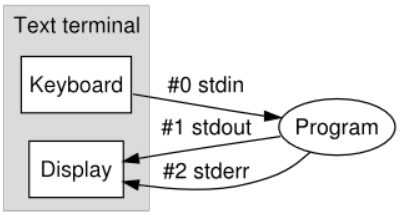
```
public static void main(String[] args) {
    Chicken myChicken = new Chicken("Fred", 10, 2);
    myChicken.feed();

    // this will result in a compiler error:
    myChicken.privateMethod();
}
```

NotAChickenClass.java

STANDARD ERROR

Standard Streams



- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`

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

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

- Recall earlier this semester

```
$ java Assign1 > debugged.out
```

  - Redirected `stdout` to `debugged.out`
  - `stderr` would still go to terminal

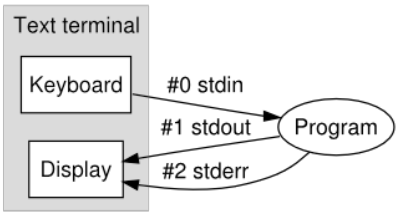
- To redirect `stderr` to same file as well

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

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

`StandardStreamsExample.java`

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)

JAVADOCS

JavaDocs for Methods

From `String` class

```
/**
 * Returns the string representation of the boolean argument.
 *
 * @param b - a boolean
 * @return if the argument is true, a string equal to "true" is
 *         returned; otherwise, a string equal to "false" is
 *         returned.
 */
public static boolean valueOf(boolean b) {
```

- Use format similar to class comments
- Use `@param` tag(s) to describe what method takes as parameter(s)
- Use `@return` tag to describe what method returns

JavaDocs for Methods: Chicken Example

```
/**
 * Sets the name of the chicken
 *
 * @param n the name of the chicken
 */
public void setName(String n) {
```

setName

`public void setName(String n)`

Sets the name of the chicken

Parameters:

n - the name of the chicken

Generated on Web Page:

https://cs.wlu.edu/~sprenkles/cs209/javadocs/07-oo/Chicken.html

JavaDocs for Methods

```
/**
 * Returns the string representation of the boolean argument.
 *
 * @param b - a boolean
 * @return if the argument is true, a string equal to "true" is
 *         returned; otherwise, a string equal to "false" is
 *         returned.
 */
public static boolean valueOf(boolean b) {
```

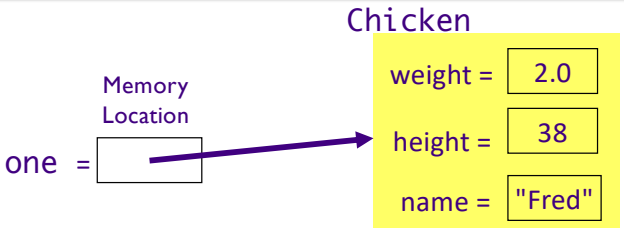
- Expectation in CSCI209: **All** methods will have JavaDoc comments
  - Exception: `main` method – sometimes covered by the *class*'s JavaDoc but sometimes needs more explanation

OBJECT REFERENCES

Variables: Object References

- Variable of type Object (not a primitive type): value is memory location

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



1. Constructor creates the object in memory
2. The variable stores the object's location in memory

Object References

- Variable of type Object: value is memory location

```
Chicken one;
Chicken two;
```

one =

two =

Variables are declared (only).
There are no memory locations to reference, so both one and two are equal to `null`

This is the case for *objects.* Primitive types are *not* `null`.

Null Object Variables

- An object variable can be explicitly set to `null`
  - Means that the object variable does not currently refer to any object

- Can test if an object variable is set to `null`:

```
Chicken chick = null;
    … … …
if (chick == null) {
        . . .
}
```

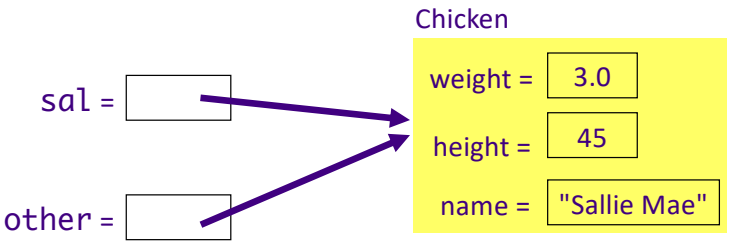

Recall This Error Message

From Kroger <noreply@kroger.com>
Subject **Your null Comments Have Been Received**
To Sara Sprenkle

Multiple Object Variables

- More than one object variable can refer to the same object

```
Chicken sal = new Chicken("Sallie Mae");
Chicken other = sal;
```



Constructor Fun Facts

- A constructor can have zero, one, or multiple parameters
- A constructor has no return value
- A constructor is always called with the new operator

- A class can have **more than one** constructor
  - Whoa! Let that sink in for a bit

Overloading

- Allowing > 1 constructor (or any method) with the same name is called ***overloading***
  - Constraint: Each of the methods that have the same name or constructor **must** have **different parameters**
    - "**different**" → *Number* and/or *type*
- *Compiler* handles ***overload resolution***
  - Process of matching a method call to the correct method by matching the parameters
- Can't overload functions in Python

Why isn't overloading possible in Python?

`overload.py`

Constructor Overloading

- Allowing > 1 constructor (or any method) with the same name is called ***overloading***
  - Constraint: Each of the methods that have the same name **must** have **different parameters** so that compiler can distinguish between them
    - "**different**" → *Number* and/or *type*
- Compiler handles ***overload resolution***
  - Process of matching a method call to the correct method by matching the parameters
- No function overloading in Python

Why isn't overloading possible in Python?

`overload.py`

Default Constructor

- ***Default constructor*: constructor with no parameters**
- If class has *no constructors,* ***compiler*** provides a default constructor (automatically)
  - Sets all instance fields to their default values
- If a class has at least one constructor and no default constructor, default constructor is NOT provided

Default Constructor

- Chicken class has one constructor:

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

➡ No default constructor

```
Chicken chicken = new Chicken();
```

  - Above code is a compiler error

Constructors Calling Constructors

- Can call a constructor from another constructor
- To call another constructor of the same class, the **first** statement of constructor must be

```
this( . . . );
```

  - `this` refers to the object being constructed

Why would you want to call another constructor?

Constructors Calling Constructors

- Why would a constructor call another constructor?
  - Reduce code size, reduce duplicate code
- Ex: if Chicken's name is not provided, use default name

```
Chicken( int height, double weight ) {
    this( "Bubba", height, weight );
}
```

- Another example:

```
Chicken( int height, double weight ) {
    this();
    this.height = height;
    this.weight = weight;
}
```

Not in example code online

Summary: Overloading

- Overloading is when you define multiple constructors or multiple methods with the same name
- Constraint: Each of the methods that have the same name or the constructor **must** have **different parameters**
  - "**different**" → *Number* and/or *type*
- Compiler distinguishes between the methods/constructor

MORE ON OBJECT INITIALIZATION

Default Object State Initialization

- If instance field is not explicitly set in constructor, automatically set to default value
  - Numbers set to zero
  - Booleans set to `false`
  - Object variables set to `null`

(Aside: recall that local variables are **not** assigned defaults)

- But, **do not** rely on defaults
  - Code is harder to understand

Clean Code Recommendation:
Set all instance fields in the constructor(s)

Explicit Field Initialization

- If more than one constructor needs an instance field set to same value, the field can be set explicitly in the field declaration

```
class Chicken {
        private String name = "";
        . . .
}
```

Set value here for all constructors

Explicit Field Initialization

- Explicit field initialization happens before any constructor runs
- A constructor can change an instance field that was set explicitly

```
class Chicken {
        private String name = "";

        public Chicken( String name, … ) {
                this.name = name;
                …
        }
…
```

Explicit field initialization

Change explicit field initialization

`final` keyword

- Meaning when modifier a field: ***field cannot be changed after object is constructed***
- `final` instance fields **must** be set in the constructor or in the field declaration

```
private final String dbName = "invoices";
private final String id;
…
public MyObject( String id ) {
        this.id = id;
}
```

33

BASICS OF JAVA INHERITANCE

Parent Class: `Object`

- Every class you create *automatically* inherits from the `Object` class
  - See Java API
- Examples of class hierarchies (from Java API):

Class String

java.lang.Object
    java.lang.String

Class JFrame

java.lang.Object
    java.awt.Component
        java.awt.Container
            java.awt.Window
                java.awt.Frame
                    javax.swing.JFrame

Overriding Methods

- You can ***override*** methods from parent classes
- Useful `Object` methods to override to customize your class
  - `String toString()`
    - Returns a string representation of the object
    - Like Python's `__str__`
  - `boolean equals(Object o)`
    - Return `true` iff this object and `o` are equivalent
    - Like Python's `__eq__`

Note method signatures

@Override

```
@Override
public boolean equals(Object obj) {
```

- Annotation
- Tells compiler "This method overrides a method in a parent class. It should have the same signature as that method in the parent class."
- If your method signature does not match the overridden method, then the compiler will give you a error
- The point: use @Override so you don't make silly—yet costly—mistakes

String toString()

- Automatically called when object is passed to `print` methods
- Default implementation: Class name followed by @ followed by unsigned hexidecimal representation of hashcode
  - Hashcode is typically the internal address of the object
  - Example: `Chicken@163b91`
- General contract:
  - "A concise but informative representation that is easy for a person to read"
- Your responsibility: ***Document the format***

Chicken's toString

- What would be a good string representation of a Chicken object?
  - Look at output before and after `toString` method implemented

boolean equals(Object o)

Note method signature

- Procedure (Source: *Effective Java*)
  1. Use the == operator to check if the argument is a reference to this object
  2. Use the `instanceof` operator to check if the argument has the correct type
     - If a variable is a null reference, then `instanceof` will be false
  3. Cast the argument to the correct type
  4. For each "significant" field in the class, check if that field of the argument matches the corresponding field of this object
     - For doubles, use Double.compare and for floats use Float.compare

How should we determine that two Chickens are equivalent?

Checking an Object's Type

- Use the `instanceof` operator to see if an object implements an interface or is an object of the given type
  - e.g., to determine if an object is a `String`

```
if (obj instanceof String) {
        // runs if obj is an object variable of type String
}
else {
        // runs if obj is not an object variable of type String
}
```

What Not to Do

- It is not recommended that you turn the objects into Strings (using `toString`) and then comparing
- While the outcome may be correct, String operations are expensive
- String representation may not represent all of the object
- Better to compare fields directly

Summary: Inheritance So Far

- Every class inherits from `Object` class
- Can override methods of parent class(es)
- Useful `Object` methods to override:
  - `String toString()`
  - `boolean equals(Object o)`

Looking Ahead

- Assignment 3 – updated! Reload page
  - Due *Wednesday* before class
  - Broke into more parts