## Objectives

- Object Oriented Programming
  - Constructors
  - Initializing object state
- Overloading constructors, methods
- Inheritance
  - Overriding methods



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 some of the syntax rules?
- What is the process for creating a class?
- What is the Java equivalent of None?
- What is the Java equivalent of self?
- What does a variable to an object contain?



2

# Assignment 4 Review

```
private int oneVar;

public Assign4(int par) {
   oneVar = par;
}
```

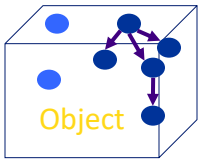
- Is the above code correct?





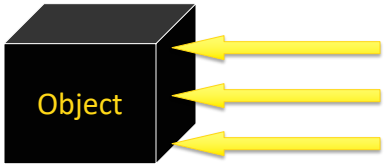
# Review

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

## Review: Access Modifiers

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





## Review: `Chicken.java`

```
public class Chicken {

   // --------- INSTANCE VARIABLES ---------------
   private String name;
   private int height; // in cm

   // --------- CONSTRUCTORS ---------------
   public Chicken(String name, int h,
                                double weight) {
      this.name = name;
      this.height = h;
      this.weight = weight;
   }
   …
```

Constructor name same as class's name

**Type** and name for each parameter

Params 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)

# Review: Chicken.java

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

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

**Type** the method returns

Chicken object's instance variables

Note that you don't *have* to use `this` when variables are unambiguous





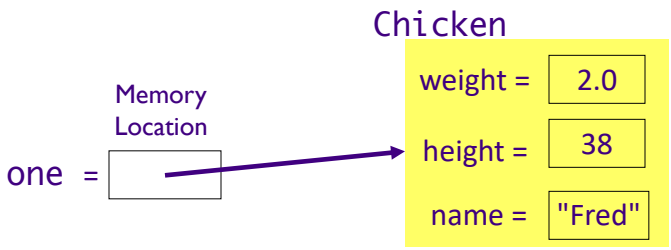
# Review: Class Development Process

1. Determine state
   - Declare state at top of class
2. Write constructor
   - Test
3. Repeat
   - Write method or constructor
   - Test

## Review: Object References

- Variable of type Object: value is memory location

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

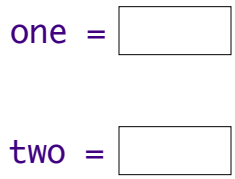







## Review: Object References

- Variable of type Object: value is memory location




If I haven't called the constructor, only *declared* the variables, e.g.,

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

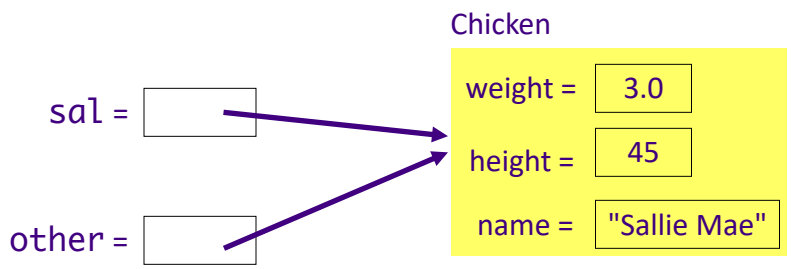
Both one and two are equal to `null`

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

## Review: Multiple Object Variables

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

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



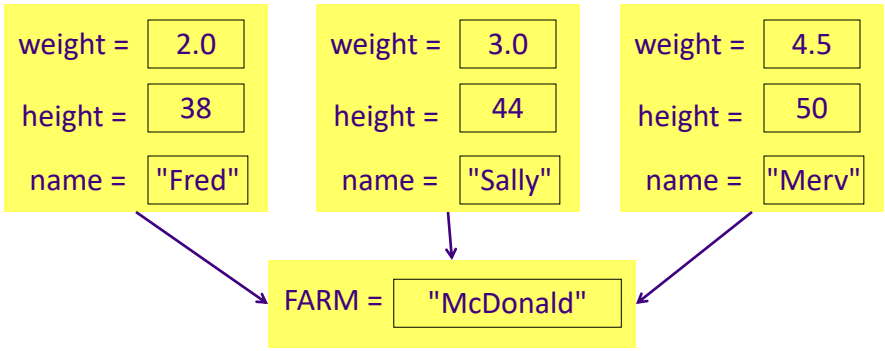





## Chicken static field example

```
static String FARM = "McDonald";
```

A bunch of Chicken objects




One variable shared by all members of the class.

# 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`
  - Local variables are not assigned defaults
- **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

- Or in a static method call

```
class Employee {
      private static int nextID = 0;
      private int id = assignID();
      . . .
      private static int assignID() {
            int assignedID = nextID;
            nextID++;
            return assignedID;
      }
}
```

## Explicit Field Initialization

- Explicit field initialization happens before any constructor runs
- A constructor can change an instance field that was set explicitly
- If the constructor does not set the field explicitly, explicit field initialization is used

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

Change explicit field initialization

17

17

## final keyword

- An instance field can be `final`
- `final` instance fields **must** be set in the constructor or in the field declaration
  - Cannot be changed ***after object is constructed***

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

Sep 7, 2020 Sprenkle - CSCI209 18

18

## More on Constructors

- A class can have **more than one** constructor
  - Whoa! Let that sink in for a bit
- A constructor can have zero, one, or multiple parameters
- A constructor has no return value
- A constructor is always called with the new operator





## 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
    - 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();
```

- Is a compiler error

## Constructors Calling Constructors

- Can call a constructor from inside another constructor
- The **first** statement of constructor must be

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

to call another constructor of the same class

  - `this` refers to the object being constructed

Why would you want to call another constructor?





## Constructors Calling Constructors

- Why would you 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

# BASICS OF JAVA INHERITANCE





## Parent Class: `Object`

- Every new class you create *automatically* inherits from the `Object` class
  - See Java API
- Useful `Object` methods 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__`
  - `void finalize()`
    - Called when object is destroyed
    - Clean up resources

Method signature

# 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.java 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)`

- 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?





# Aside

- 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
  - Better to compare fields directly

## @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 you do not correctly override the method, then the compiler will give you a warning
- The point: use @Override so you don't make silly—yet costly—mistakes





## Encapsulation Revisited

- Encapsulation/Black-box programming
- Objects should hide their data and only allow other objects to access this data through **accessor** and **mutator** methods
- Common programmer mistake:
  - Creating an accessor method that returns a reference to a mutable (changeable) object

## What is "bad" about this class?

```
public class Farm {
      . . .
      private Chicken headRooster;

      public Chicken getHeadRooster() {
            return headRooster;
      }
      . . .
}
```



33

## What is "bad" about this class?

```
public class Farm {
      . . .
      private Chicken headRooster;

      public Chicken getHeadRooster() {
            return headRooster;
      }
      . . .
}
```

Problem: Giving others access to Farm's headRooster
Others can then feed your rooster or change his name!!
(Silly example; understand consequences)



34

## Fixing the Problem: Cloning

```
public class Farm {
      . . .
      private Chicken headRooster;

      public Chicken getHeadRooster() {
            return (Chicken) headRooster.clone();
      }
      . . .
}
```

Method is available to all objects (inherited from `Object`)

- In previous example, could modify returned object's state
- Another `Chicken` object, with the same data as `headRooster`, is created and returned to the user
- If the user modifies (e.g., feeds) that object, `headRooster` is not affected





## Cloning

- Cloning is a more complicated topic than it seems from the example
  - Out of scope for this class

## What is "bad" about this class?

```
public class Farm {
      . . .
      private Chicken headRooster;

      public Chicken getHeadRooster() {
            return headRooster;
      }
      . . .
}
```

Problem: Giving others access to Farm's headRooster
Others can then feed your rooster or change his name!!
(Silly example; understand consequences)

**But, then, why is it okay to return the name, height, or weight of a chicken?**
Similar to Python, primitive types and Strings are immutable.
Since those attributes have data types (String, int, double, respectively) that are immutable, others can't change those attributes.





## Review: Class Design/Organization

- Fields
  - Chosen first
  - Placed at the beginning or end of class definition
  - Have an access modifier, data type, variable name, and some optional other modifiers
  - Use `this` keyword to access the object
- Constructors
- Methods
  - Need to declare the return type
  - Have an access modifier

# Looking Ahead

- Assignment 5 – due Friday before class
  - Building on the Birthday class
    - Overloading constructor
    - Overriding methods
  - Creating an application, practicing
    - Control structures
    - Using your own class
    - Using classes from the Java API
  - Good capstone for the course so far
    - Brings together a lot of concepts of the last ~2 weeks
- Textbook: Continuing "Defining Classes in Java"
  - Up to but not including "Abstract Classes and Methods"