Objectives

- Garbage Collection
- Inheritance

Review

- How do you print formatted output in Java?
  - What are the various components? How do they work?
- Some code needs to return a private variable from a public method
  - Why could that be a problem?
  - How should we implement that method?
- How does Java pass parameters?
  - What are the consequences of that choice? (How does that affect how we call methods?)
- Assignment 4: How do you ensure that there is only one variable/object for a class? Example: only one random number generator for Birthday class

Review: Providing Private Data

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

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

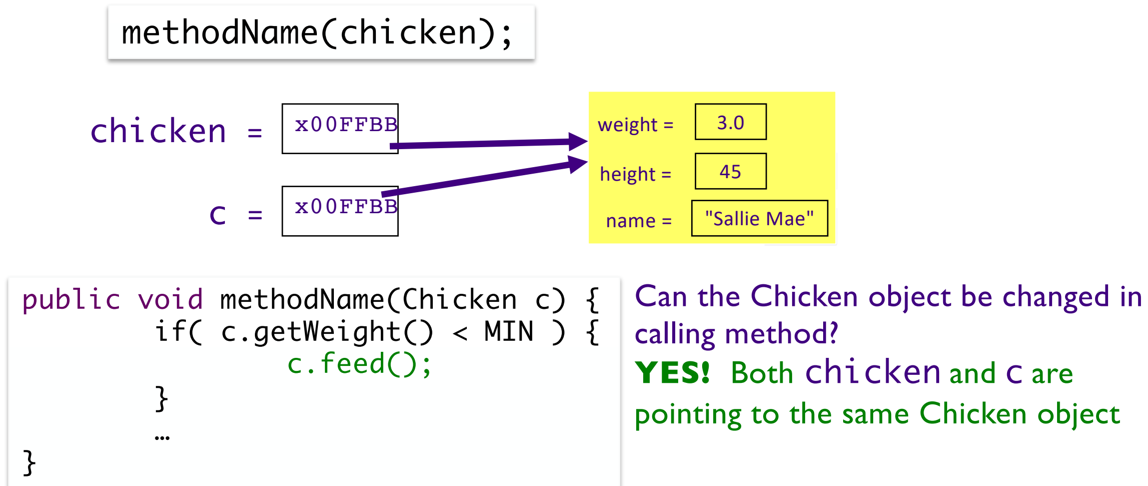
Method is available to all objects (inherited from `Object`)

- 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

Review: Method Parameters in Java

- Java always passes parameters into methods ***by value***
  - Meaning: the formal parameter becomes a copy of the argument/actual parameter's value
    - Method caller and callee have two independent variables with the same value
  - Consequence: Methods can**not** change the ***variables*** used as input parameters

Review: Pass by Value: Objects



Can the Chicken object be changed in calling method?

YES! Both chicken and c are pointing to the same Chicken object

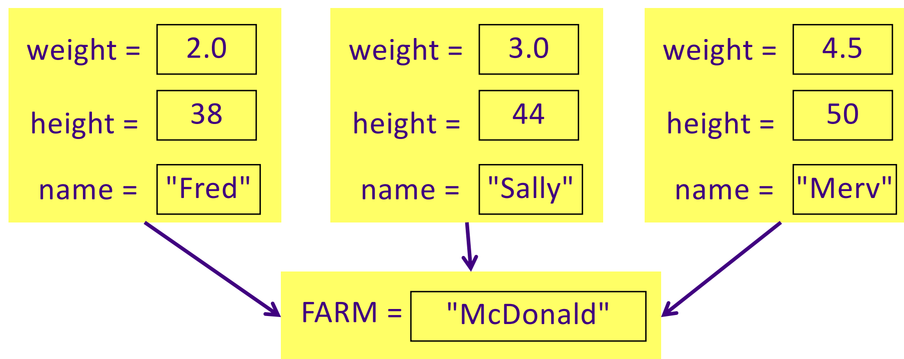
Review: Summary of Method Parameters

- Everything is passed ***by value*** in Java
  - Formal parameter copies the actual parameter
- An ***object variable*** (not an object) is passed into a method
  - Changing the *state* of an object in a method changes the state of object outside the method
  - Method does not see a copy of the original object

Review: Chicken static Field Example

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

A bunch of Chicken objects



One variable shared by all members of the class.

What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```

baby

Whoops! Lost "baby" chicken! -- No object variable references it
Memory leak!

Luckily Java has ***garbage collectors*** to clean up the memory leak

GARBAGE COLLECTION

Memory Management

- Early languages (e.g., C): free memory when you're done with it
- In C++ and some other OOP languages, classes have explicit *destructor* methods that run when an object is no longer in scope
- Java provides ***automatic garbage collection***
  - Reclaims memory allocated for objects that are no longer referenced

Garbage Collector (GC)

- Garbage collector is low-priority thread or runs when available memory gets tight
- Before GC can clean up an object, the object may have opened resources
  - Ex: generated temp files or open network connections that should be deleted/closed first
- GC calls object's `finalize()` method
  - Object's chance to clean up resources

`finalize()`

- Inherited from `java.lang.Object`
- Called before garbage collector sweeps away an object and reclaims its memory
- Should not be used for reclaiming resources
  - i.e., *close resources as soon as possible*
  - Why?
    - *When* method is called is not deterministic or consistent
    - Only know it will run sometime before garbage collection
- Clean up anything that cannot be atomically cleaned up by the garbage collector
  - Close file handles, network connections, database connections, etc.
- Note: no finalizer chaining
  - Must explicitly call parent object's `finalize` method

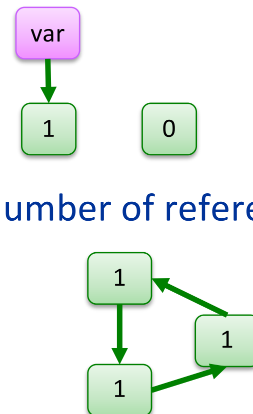
Alternatives to `finalize`

- Recall: unknown when `finalize` will execute—or *if* it will execute
  - Also *heavy performance cost*
- Solution: create your own terminating method
  - User of class terminates when done using object
- Examples: `Scanner's` or `Window's` `close` method
- May still want `finalize()` as a safety net if user didn't call the terminate method
  - Log a warning message so user knows error in code

Do you know what Python does?

Python Garbage Collection

- Python also does garbage collection
- Python does ***reference counting***
  - On each reference/dereference, update the number of references to the object
    - Can't handle reference cycles
- Python also does ***generational garbage collection*** to handle reference cycles
- Tradeoffs with Java's Garbage Collection
  - Synchronous (not asynchronous) process (i.e., slows down execution)
  - Cheaper memory costs than Java for keeping track of what can be garbage collected



Discussion: Benefits and limitations of garbage collection?

Garbage Collection

Benefits

- Programmer doesn't need to worry about memory management
- Cleans up unreferenced memory automatically, eventually
- Programmer can never release memory that is then accessed (a.k.a. seg faults)

Drawbacks

- Programmer doesn't worry about memory management
  - May not be as careful to avoid memory leaks
- Memory could be cleaned up sooner
- Requires resources (CPU, memory) to keep track of memory
- Slows program execution

Garbage Collection

Benefits

- Programmer doesn't need to worry about memory management
- Cleans up unreferenced memory automatically, eventually
- Programmer can never release memory that is then accessed (a.k.a. seg faults)

> - Programmer time is more valuable than computer resources.
> - Less buggy code is preferred to more efficient code in many domains

Drawbacks

- Programmer doesn't worry about memory management
  - May not be as careful to avoid memory leaks
- Memory could be cleaned up sooner
- Requires resources (CPU, memory) to keep track of memory
- Slows program execution

INHERITANCE

Review: Inheritance (from CSCI112)

- What are the benefits of inheritance?
- What are examples of inheritance?
- When should you use inheritance?

Inheritance

- Build new classes based on existing classes
  - Allows *code reuse*
- Start with a class (**parent** or **super class**)
- Create another class that extends or *specializes* the class
  - Called **the child**, **subclass,** or **derived class**
  - Use `extends` keyword to make a subclass

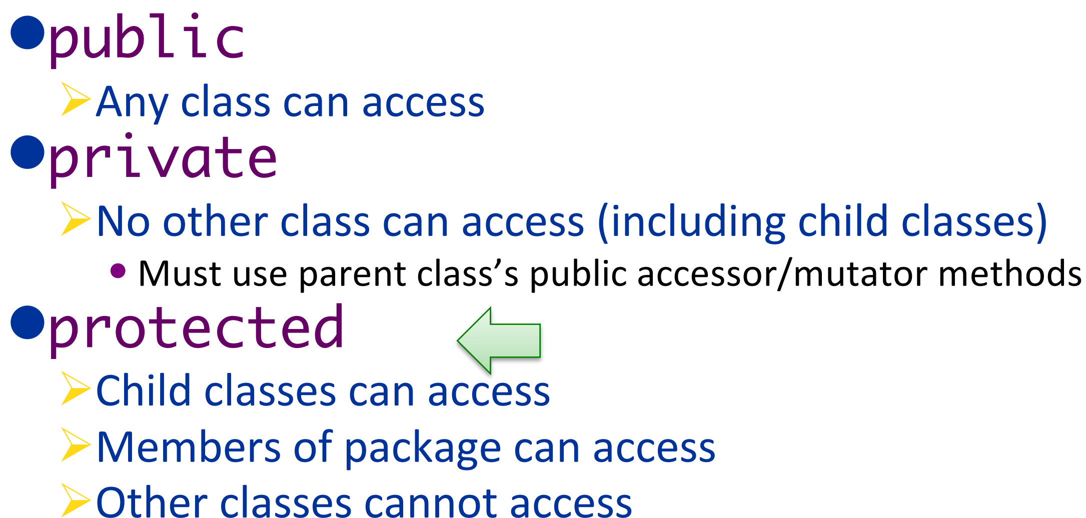
Child class

- Inherits all of parent class's methods and fields
  - Note on `private` fields: all are *inherited*, just can't *access*
- Constructors are ***not*** inherited
- Can ***override*** methods
  - Recall: overriding - methods have the same name and parameters, but implementation is different
- Can add methods or fields for *additional functionality*
- Use `super` object to call parent's method
  - Even if child class redefines parent class's method

`Rooster` class

- Could write class from scratch, but …
- A rooster *is a* chicken
  - But it adds something to (or *specializes*) what a chicken is/does
- Classic mark of inheritance: ***is a*** relationship
- Rooster is child class
- Chicken is parent class

Access Modifiers



Access Modes

Default (if none specified) → package

Accessible to	Member Visibility			
	public	protected	package	private
Defining class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

- Visibility for variables: who can access/change
- Visibility for methods: who can call

protected

- Accessible to subclasses and members of package
- Can't keep encapsulation "pure"
  - Don't want others to access fields directly
  - May break code if you change your implementation
- Assumption?
  - Someone extending your class with protected access knows what they are doing

Guidance on Access Modifiers

- If you're uncertain which to use (protected, package, or private), use the *most restrictive*
  - Changing to less restrictive later → easy
  - Changing to more restrictive → may break code that uses your classes

Changes to Chicken Class

- Added a new instance variable called is_female
- Added getter and setter for is_female
- Updated toString, equals methods accordingly

- 2 Chicken classes in examples
  - Chicken.java private instance variables
  - Chicken2.java protected instance variables

Rooster class

extends means that Rooster is a child of Chicken

```
public class Rooster extends Chicken {
   public Rooster( String name, int height, double weight ) {
```

Call to super constructor must be **first** statement in constructor

```
        super(name, height, weight, false);
   }

   // new functionality
   public void crow() { … }

   …
}
```


Rooster class

extends means that Rooster is a child of Chicken

```
public class Rooster extends Chicken {
   public Rooster( String name, int height, double weight ) {
        // all instance fields inherited
        // from super class
        this.name = name;
        this.height = height;
        this.weight = weight;
        this.is_female = false;
   }

   // new functionality
   public void crow() {… }
   …
```

By default, calls *default* super constructor with no parameters

(not one of the examples posted online)

Constructor Chaining

- Constructor ***automatically*** calls constructor of parent class if not done explicitly
  - `super();`
- What if parent class does not have a constructor with no parameters?
  - Compilation error
  - Forces child classes to call a constructor with parameters

Overriding and New Methods

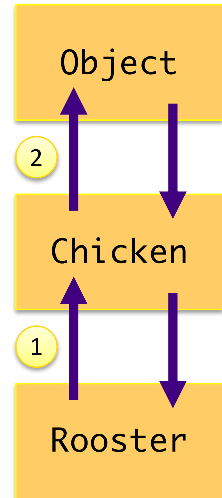
```
public class Rooster extends Chicken {
   …

   // overrides superclass; greater gains
   @Override
   public void feed() {
      weight += .5;
      height += 2;
   }

   // new functionality
   public void crow() {
      System.out.println("Cocka-Doodle-Doo!");
   }
}
```

Same method signature as parent class

Specializes the class

Inheritance Tree: Constructor Chaining

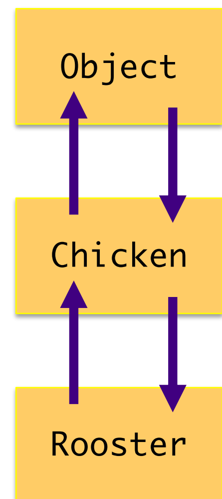
- java.lang.Object
  - Chicken
    - Rooster



- Call parent class's constructor first
  - Know you have fields of parent class before implementing constructor for your class

Inheritance Tree

- java.lang.Object
  - Chicken
    - Rooster



- No `finalize()` chaining
  - Should explicitly call `super.finalize()` inside of `finalize` method

Shadowing Parent Class Fields

- Child class has field with same name as parent class
  - You probably shouldn't be doing this
  - But could happen
    - Examples: more precision for a constant (or more weight gain for a rooster)

```
field       // this class's field
this.field  // this class's field
super.field // super class's field
```


Multiple Inheritance

- In Python, a class can inherit more than one parent class
  - Child class has the fields from both parent classes
- This is NOT possible in Java.
  - A class may extend (or inherit from) **only one** class

Assignment 5

- Start of a simple video game
  - `Game` class to run
  - `GamePiece` is parent class of other moving objects
- Some less-than-ideal design
  - Can’t fix until see other Java structures (Friday)
- Don’t need to understand all of the code (yet), just some of it
- Create a `Goblin` class and a `Treasure` class
  - Move `Goblin` and Treasure
- Due Tuesday at midnight