Objectives

- Inheritance
  - Overriding methods
- Garbage collection
- Parameter passing in Java

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

More on `toString()`

- Automatically called when object is passed to print methods
- Default implementation: Class name followed by @ followed by unsigned hexidecimal representation of hashcode
  - 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*)
  - Use the == operator to check if the argument is a reference to this object
  - 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
  - Cast the argument to the correct type
  - 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?

@Override

- 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

- 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;
      }
      . . .
}
```

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;
}
```

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

GARBAGE COLLECTION

Memory Management

- In C++ and some other OOP languages, classes have explicit *destructor* methods that run when an object is no longer used
- Java does not support destructors because it provides ***automatic garbage collection***
  - Waits until there are no references to an object
  - Reclaims memory allocated for the object that is no longer referenced

Do you know what Python does?

Garbage Collector

- 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

Discussion: Benefits and limitations of garbage collection?

`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: `File`'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

PARAMETER PASSING

Method Parameters in Java

- Java always passes parameters into methods **by value**
  - Methods cannot change the variables used as input parameters
  - A subtle point, so we need to go through several examples

- Python is something that's not quite pass-by-value—it depends on if the object is mutable or immutable
  - *Pass-by-alias* is one term used

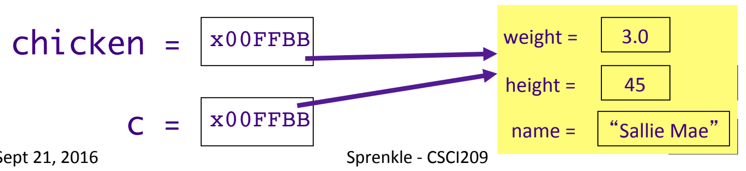
Pass by Value: Objects

- Primitive types are a little more obvious
  - Can't change original variable
- For objects, passing a copy of the parameter looks like

```
public void methodName(Chicken c)
```

Pass Chicken object to `methodName` when calling method

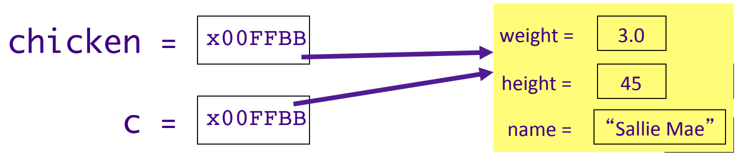
```
methodName(chicken);
```



Pass by Value: Objects

- What happens in this case?

```
methodName(chicken);
```



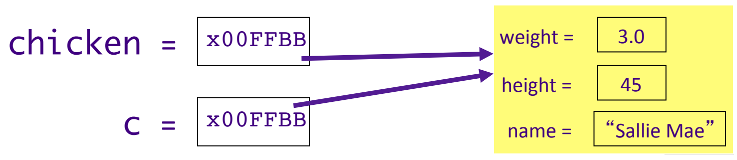
```
public void methodName(Chicken c) {
      if( c.getWeight() < MIN ) {
            c.feed();
      }
      …
}
```

Does chicken change in calling method?

Pass by Value: Objects

- What happens in this case?

```
methodName(chicken);
```



```
public void methodName(Chicken c) {
        if( c.getWeight() < MIN ) {
                c.feed();
        }
        …
}
```

Does chicken change in calling method?

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