Objectives

- Parameter passing in Java
- Inheritance

Review

- What does `static` mean?
- When should a method be declared `static`?
- What does Java provide to prevent memory leaks?
- What method should we implement to allow pretty printing of objects?
  - To determine if two objects are equivalent?

PARAMETER PASSING

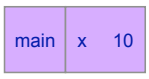
Method Parameters in Java

- Java always passes parameters into methods **by value**
  - Methods cannot change the variables used as input parameters
  - This is 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

Method Parameters

```
public static void main(String[] args) {
   int x = 10;
   int squared = square(x);
   System.out.println("The square of " + x + " is " +
                      squared);
}

public static int square(int num) {
   return num*=num;
}
```

Draw the stack as it changes (similar to Python):



What's the output?

```
public static void main(String[] args) {
     int x = 27;
     System.out.println(x);
     doubleValue(x);
     System.out.println(x);
}
. . .

void doubleValue(int p) {
     p = p * 2;
}
```

What's the output?

```
public static void main(String[] args) {
      int x = 27;
      System.out.println(x);
      doubleValue(x);
      System.out.println(x);
}
. . .

void doubleValue(int p) {
      p = p * 2;
}
```

27
27

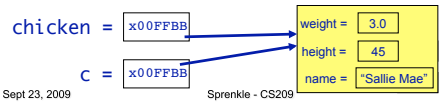
Pass by Value: Objects

- 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`

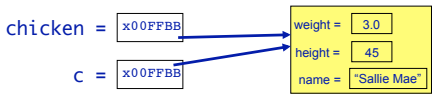
```
methodName(chicken);
```



Pass by Value: Objects

- What happens in this case?

```
methodName(chicken);
```



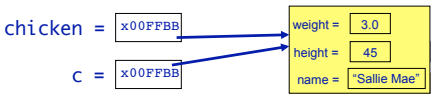

```
public void methodName(Chicken c) {
      if( c.getWeight() < MIN ) {
            c.feed();
      }
      …
}
```

Does `chicken` change in calling method?

Pass by Value: Objects

- What does this mean?

```
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

What's the output?

```
Farm farm = new Farm("OldMac");
Chicken sal = new Chicken("Sallie Mae", 50, 10);
System.out.println(sal.getWeight());
farm.feedChicken(sal);
System.out.println(sal.getWeight());
. . .

void feedChicken(Chicken c) {
      c.setWeight( c.getWeight() + .5);
}
```


What's the output?

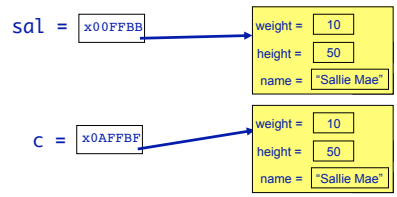
```
Farm farm = new Farm("OldMac");
Chicken sal = new Chicken("Sallie Mae", 50, 10);
System.out.println(sal.getWeight());
farm.feedChicken(sal);
System.out.println(sal.getWeight());
. . .

void feedChicken(Chicken c) {
      c = new Chicken(c.getName(), c.getWeight(),
            c.getHeight() );
      c.setWeight( c.getWeight() + .5);
}
```

What's the Difference?

```
Farm farm = new Farm("OldMac");
Chicken sal = new Chicken("Sallie Mae", 50, 10);
System.out.println(sal.getWeight());
farm.feedChicken(sal);
System.out.println(sal.getWeight());
. . .

void feedChicken(Chicken c) {
      c = new Chicken(c.getName(), c.getWeight(),
            c.getHeight() );
      c.setWeight( c.getWeight() + .5);
}
```

What's the Difference?

```
void feedChicken(Chicken c) {
      c = new Chicken(c.getName(), c.getWeight(),
            c.getHeight() );
      c.setWeight( c.getWeight() + .5);
}
```



Summary of Method Parameters

- Everything is passed **by value** in Java
- 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: Class Design/Organization

- Fields
  - Chosen first
  - Placed at the beginning or end of the class
  - Has 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
  - May be `public static …`

Encapsulation Revisited

- Objects should hide their data and only allow other objects to access this data through a **public interface**
- Common programmer mistake:
  - Creating an accessor method that returns a reference to a mutable (changeable) object.

What is "bad" about this class?

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

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

Fixing the Problem: Cloning

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

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

Method is available to all objects but must be defined

- 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
- We may examine cloning in more detail later

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

Examples?

Child class

- Inherits all of parent class's methods and fields
  - Unless they're `static`
  - Note on `private` fields: all are *inherited*, just can't *access*
- Can also ***override*** methods
  - Use the same name, but the implementation is different
- Adds methods or fields for *additional functionality*
- If child class redefines a parent class's method, can still call parent class's method using the `super` object

Inheritance Rules

- Class (`static`) fields and methods are ***not*** inherited
- Constructors are ***not*** inherited
  - For example: we will have to define `Rooster( String name, int height, double weight )` even though similar constructor in Chicken

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
- The is a relationship
  - Classic mark of inheritance
- Rooster is child class
- Chicken is parent class

Modify Chicken Class

- Want instance variables to be accessible by child class
  - Can't be `private`
- Add new boolean instance variable `is_female`

Access Modifiers

- `public`
  - Any class can access
- `private`
  - No other class can access (including child classes)
    - Must use parent class's public accessor/mutator methods
- `protected`
  - Child classes can access
  - Members of package can access
  - Other classes cannot access

Access Modes

Default (if none specified)

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

Which access modifier should we use for the `Chicken` instance variables?

`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

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 the code that uses your classes

Inheritance Rules: Access Modifiers

Access modifiers in child classes
- Can make access to child class ***less*** restrictive but not more restrictive

- **Why?**
- Consider what would happen if a method in the parent class is `public` but the child class is `private`

Inheritance Rules: Access Modifiers

Access modifiers in child classes
- Can make access to child class ***less*** restrictive but not more restrictive

- If a `public` method could be overridden as a `protected` or `private` method, child objects would not be able to respond to the same method calls as parent objects
- When a method is declared `public` in the parent, the method remains `public` for all that class's child classes
- Remembering the rule: compiler error to override a method with a more restricted access modifier

Look at Modified `Chicken` Class

`Rooster` class

```
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;
          is_female = false;
     }

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

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

`Rooster` class

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

Call to `super` constructor must be **first** line in constructor

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

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

     …
}
```


Constructor Chaining

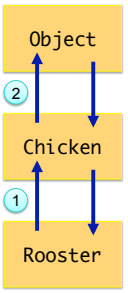
- Constructor automatically calls constructor of parent class if not done explicitly
  - `super();`
- What if parent class does not have a constructor with no parameters?
  - Compilation error
  - Forces subclasses to call a constructor with parameters

Overriding Methods in Parent Class

```
public class Rooster extends Chicken {
      …

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

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

}
```

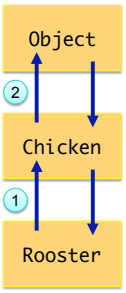

Inheritance Tree

- java.lang.Object
  - Chicken
    - Rooster
- Call constructor of parent class first
  - Know you have the fields of parent class before you implement constructor for your class



Inheritance Tree

- java.lang.Object
  - Chicken
    - Rooster
- No finalize() chaining
  - Should 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
    - Possibly: more precision for a constant

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


Multiple Inheritance

- In Python, it is possible for a class to inherit (or extend) 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 4

- Start of a simple video game
  - Game class to run
  - GamePiece is parent class of other moving objects
- Some poor design
  - Can't fix until see other Java structures
- Don't need to understand all of the code, just some of it
- Create a Goblin class and a Treasure class
  - Move Goblin and Treasure

Copy /home/courses/cs209/handouts/assign4