

Objectives

- Parameter passing in Java
- Inheritance

Sept 21, 2011

Sprenkle - CSCI209

1

Review

- What does **static** mean?
- 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?

Sept 21, 2011

Sprenkle - CSCI209

2

Assignment 2 Review

```
private int oneVar;

public Assign2(int arg) {
    oneVar = arg;
}
```

- Is the above code correct?

Sept 21, 2011

Sprenkle - CSCI209

3

Analyzing java.lang.String

- **String toUpperCase()**
 - Converts all of the characters in *this* String to upper case
- **static String valueOf(boolean b)**
 - Returns the string representation of the **boolean** argument

Why can (should) the second method be **static**?

When should a method be declared **static**?

Sept 21, 2011

Sprenkle - CSCI209

4

PARAMETER PASSING

Sept 21, 2011

Sprenkle - CSCI209

5

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

Sept 21, 2011

Sprenkle - CSCI209

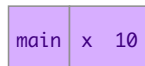
6

Method Parameters in Java

```
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):



Sept 21, 2011

Sprenkle - CSCI209

7

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;
}
```

Sept 21, 2011

Sprenkle - CSCI209

8

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

Sept 21, 2011

Sprenkle - CSCI209

9

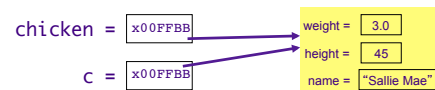
Pass by Value: Objects

- 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);
```



Sept 21, 2011

Sprenkle - CSCI209

10

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?

Sept 21, 2011

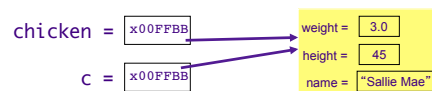
Sprenkle - CSCI209

11

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

Sept 21, 2011

Sprenkle - CSCI209

12

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());
...

// From Farm class
void feedChicken(Chicken c) {
    c.setWeight( c.getWeight() + .5);
}
```

Sept 21, 2011

Sprengle - CSCI209

13

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());
...

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

Sept 21, 2011

Sprengle - CSCI209

14

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());
...

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



Sept 21, 2011

Sprengle - CSCI209

15

What's the Difference?

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



Sept 21, 2011

Sprengle - CSCI209

16

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

Sept 21, 2011

Sprengle - CSCI209

17

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.

Sept 21, 2011

Sprengle - CSCI209

18

What is “bad” about this class?

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

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

Sept 21, 2011

Sprenkle - CSCI209

19

Fixing the Problem: Cloning

```
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

Sept 21, 2011

Sprenkle - CSCI209

20

Cloning

- Cloning is a more complicated topic than it seems from the example
- We may examine cloning in more detail later

Sept 21, 2011

Sprenkle - CSCI209

21

Review: Class Design/Organization

- Fields
 - Chosen first
 - Placed at the beginning or end of the class defn
 - 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 `static` ...

Sept 21, 2011

Sprenkle - CSCI209

22

INHERITANCE

Sept 21, 2011

Sprenkle - CSCI209

23

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?

Sept 21, 2011

Sprenkle - CSCI209

24

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*
- Use **super** object to call parent's method
 - Even if child class redefines parent class's method

Sept 21, 2011

Sprenkle - CSCI209

25

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`

Sept 21, 2011

Sprenkle - CSCI209

26

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

Sept 21, 2011

Sprenkle - CSCI209

27

Modify Chicken Class


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

Sept 21, 2011

Sprenkle - CSCI209

28

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

Sept 21, 2011

Sprenkle - CSCI209

29

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?

Sept 21, 2011

Sprenkle - CSCI209

30

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

Sept 21, 2011

Sprenkle - CSCI209

31

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

Sept 21, 2011

Sprenkle - CSCI209

32

Inheritance Rules: Access Modifiers

Access modifiers in child classes

- Can make access to child class **less** restrictive but not more restrictive

- **Why?**
- What would happen if a method in the parent class is **public** but the child class's method is **private**?

Sept 21, 2011

Sprenkle - CSCI209

33

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

Sept 21, 2011

Sprenkle - CSCI209

34

Look at Modified Chicken Class

Sept 21, 2011

Sprenkle - CSCI209

35

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;
        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, false);
    }

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

    ...
}
```

Sept 21, 2011

Sprenkle - CSCI209

37

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

Sept 21, 2011

Sprenkle - CSCI209

38

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-Do!");
    }
}
```

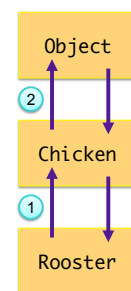
Sept 21, 2011

Sprenkle - CSCI209

41

Inheritance Tree

- `java.lang.Object`
 - `Chicken`
 - `Rooster`
- Call parent class's constructor first
 - Know you have fields of parent class before implementing constructor for your class



Sept 21, 2011

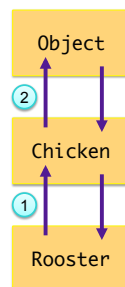
Sprenkle - CSCI209

40

Inheritance Tree

- `java.lang.Object`
 - `Chicken`
 - `Rooster`

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



Sept 21, 2011

Sprenkle - CSCI209

41

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
```

Sept 21, 2011

Sprenkle - CSCI209

42

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

Sept 21, 2011

Sprenkle - CSCI209

43

Assignment 4

- 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
- 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`

Sept 21, 2011

Sprenkle - CSCI209

44