

Objectives

- Inheritance
 - Polymorphism
 - Dynamic dispatch

Sep 11, 2020

Sprenkle - CSCI209

1

1

Review

- We would like to return a private variable from a public method
 - Why could that be a problem?
 - How should we implement that method?
- How does Java handle memory management?
 - What are the benefits and limitations of that approach?
- How does Java pass parameters?
 - What are the consequences of that choice? (How does that affect how we call methods?)

Sep 11, 2020

Sprenkle - CSCI209

2

2

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

Sep 9, 2020

Sprenkle - CSCI209

3

3

Review: Garbage Collection

Benefits

- Programmer doesn't need to worry about memory management
- Cleans up unused 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

Sep 11, 2020

Sprenkle - CSCI209

4

4

Review: Garbage Collection

Benefits

- ➔ Programmer doesn't need to worry about memory management
- Cleans up unused 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.

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

09

5

5

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 **cannot** change the **variables** used as input parameters

6

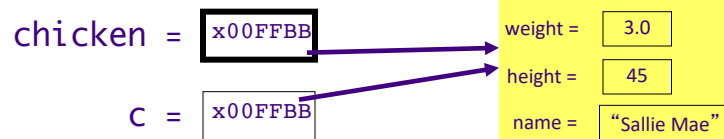
Review: Pass by Value - Objects

- Primitive types are a little more obvious
 - Can't change passed-in 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);
```



Sep 11, 2020

Sprenkle - CSCI209

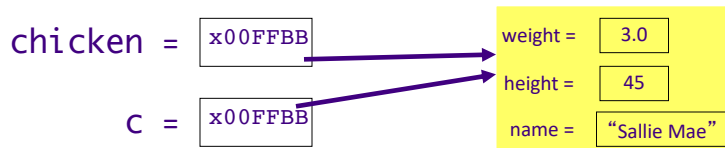
7

7

Review: Pass by Value: Objects

- What happens in this case?

```
methodName(chicken);
```



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

Can the Chicken object be changed in calling method?

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

Sep 11, 2020

Sprenkle - CSCI209

8

8

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

Sep 11, 2020

Sprenkle - CSCI209

9

9

INHERITANCE

Sep 11, 2020

Sprenkle - CSCI209

10

10

Review: Inheritance (from CSCI112)

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

Sep 11, 2020

Sprenkle - CSCI209

11

11

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

Sep 11, 2020

Sprenkle - CSCI209

12

12

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

Sep 11, 2020

Sprenkle - CSCI209

13

13

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


Sep 11, 2020

Sprenkle - CSCI209

14

14

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

Sep 11, 2020

Sprenkle - CSCI209

15

15

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

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

Sep 11, 2020

Sprenkle - CSCI209

16

16

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

Sep 11, 2020

Sprenkle - CSCI209

17

17

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

Sep 11, 2020

Sprenkle - CSCI209

18

18

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

Sep 11, 2020

Sprenkle - CSCI209

19

19

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;
    }
}
```

By default calls *default super* constructor with no parameters

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

(not one of the examples posted online)

20

Rooster class

```
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() { ... }

    ...
}
```

Sep 11, 2020

Sprenkle - CSCI209

21

21

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

Sep 11, 2020

Sprenkle - CSCI209

22

22

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!");
    }
}
```

Same method signature
as parent class

Specializes the class

Sep 11, 2020

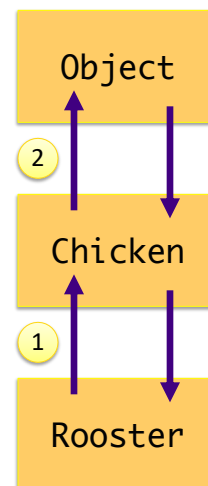
Sprenkle - CSCI209

23

23

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



Sep 11, 2020

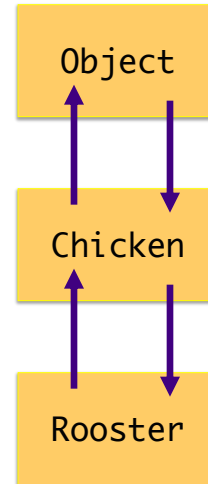
Sprenkle - CSCI209

24

24

Inheritance Tree

- `java.lang.Object`
 - `Chicken`
 - `Rooster`
- No `finalize()` chaining
 - Should call `super.finalize()` inside of `finalize` method



Sep 11, 2020

Sprenkle - CSCI209

25

25

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
  
```

Sep 11, 2020

Sprenkle - CSCI209

26

26

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

Sep 11, 2020

Sprenkle - CSCI209

27

27

POLYMORPHISM & DISPATCH

Sep 11, 2020

Sprenkle - CSCI209

28

28

Polymorphism

- **Polymorphism** is the ability for an object to vary behavior based on its type
- You can use a child class object whenever the program expects an object of the parent class
- Object variables are **polymorphic**
- A `Chicken` object variable can refer to an object of class `Chicken`, `Rooster`, `Hen`, or any class that *inherits from* `Chicken`

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

We can guess the actual types
But compiler can't

Sep 11, 2020

Sprenkle - CSCI209

29

29

Compiler's Behavior

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma; // a Hen
chickens[1] = foghorn; // a Rooster
chickens[2] = baby; // a Chicken
```

- We know `chickens[1]` is probably a `Rooster`, but to *compiler*, it's a `Chicken` so ~~`chickens[1].crow();`~~ will not compile

Sep 11, 2020

Sprenkle - CSCI209

30

30

Compiler's Behavior

- When we refer to a `Rooster` object through a `Rooster` object variable, compiler sees it as a `Rooster` object
- If we refer to a `Rooster` object through a `Chicken` object variable, compiler sees it as a `Chicken` object.

→ Object variable determines how compiler sees object.

- We cannot assign a parent class object to a child class object variable
 - Ex: `Rooster` is a `Chicken`, but a `Chicken` is not necessarily a `Rooster`

~~`Rooster r = chicken;`~~

Sep 11, 2020

Sprenkle - CSCI209

31

31

Polymorphism

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

```
chickens[1].feed();
```

Compiles because `Chicken` has a `feed` method.

But, which `feed` method is called –
`Chicken`'s or `Rooster`'s?

Sep 11, 2020

Sprenkle - CSCI209

32

32

Polymorphism

- Which method do we call when we call `chicken[1].feed()`
Rooster's or Chicken's?
- In Java (and Python): Rooster's!
 - Object is a Rooster
 - JVM figures out object's class at runtime and runs the appropriate method
- **Dynamic dispatch**
 - At runtime, the object's class is determined
 - Appropriate method for that class is dispatched

Sep 11, 2020

Sprenkle - CSCI209

33

33

Feed the Chickens!

Think on your own for 1 minute

Recall:

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

```
for( Chicken c: chickens ) {
    c.feed();
}
```

How to read this code?

What happens in execution?

- **Dynamic dispatch** calls the appropriate method in each case, corresponding to the actual class of each object
 - This is the power of polymorphism and dynamic dispatch!

Sep 11, 2020

Sprenkle - CSCI209

34

34

Dynamic Dispatch vs. Static Dispatch

- Dynamic dispatch is not necessarily a property of object-oriented programming in general
- Some OOP languages use **static dispatch**
 - Type of the object variable that the method is called on determines which version of method gets run
- The primary difference is **when decision on which method to call is made...**
 - Static dispatch (C#) decides at compile time
 - Dynamic dispatch (Java, Python) decides at run time
- Dynamic dispatch is slower
 - In mid to late 90s, active research on how to decrease time

Sep 11, 2020

Sprenkle - CSCI209

35

35

What Will This Code Output?

```

class Parent {
    public Parent() {}

    public void method1() {
        System.out.println("Parent: method1");
    }

    public void method2() {
        System.out.println("Parent: method2");
        method1();
    }
}

class Child extends Parent {
    public Child() {}

    public void method1() {
        System.out.println("Child: method1");
    }
}

public class DynamicDispatchExample {
    public static void main(String[] args) {
        Parent p = new Parent();
        Child c = new Child();

        p.method1();
        System.out.println("");

        c.method1();
        System.out.println("");

        p.method2();
        System.out.println("");

        c.method2();
        System.out.println("");
    }
}

```

Think on your own for 1 minute

See handout

Sep 11, 2020

36

What Will This Code Output?

```

class Parent {
    public Parent() {}

    public void method1() {
        System.out.println("Parent: method1");
    }

    public void method2() {
        System.out.println("Parent: method2");
    }
}

class Child extends Parent {
    public Child() {}

    public void method1() {
        System.out.println("Child: method1");
    }

    public void method2() {
        System.out.println("Child: method2");
    }
}

public class DynamicDispatch {
    public static void main(String[] args) {
        Parent p = new Parent();
        Child c = new Child();

        p.method1();
        System.out.println("");

        c.method1();
        System.out.println("");

        p.method2();
        System.out.println("");

        c.method2();
        System.out.println("");
    }
}

```

Parent: method1
 Child: method1
 Parent: method2
 Parent: method1
 Parent: method2
 Child: method1

See handout
 Sep 11, 2020

37

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**?

Sep 11, 2020

Sprenkle - CSCI209

38

38

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

Sep 11, 2020

Sprenkle - CSCI209

39

39

Summary of Inheritance

- Remove repetitive code by modeling the “**is-a**” hierarchy
 - Move “common denominator” code up the inheritance chain
- Don't use inheritance unless *all* inherited methods make sense
- Use polymorphism

Sep 11, 2020

Sprenkle - CSCI209

40

40

Assignment 6

- 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 (Monday)
- 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 Wednesday

Sep 11, 2020

Sprenkle - CSCI209

41