Objectives

- Inheritance

1

Review

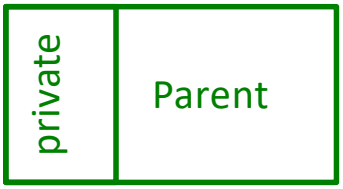
- When should you apply inheritance?
  - i.e., what is the relationship you should look for?
- What are Java's inheritance rules?
  - i.e., what is inherited? What is not inherited?
- How do you refer to the parent class in Java?

2

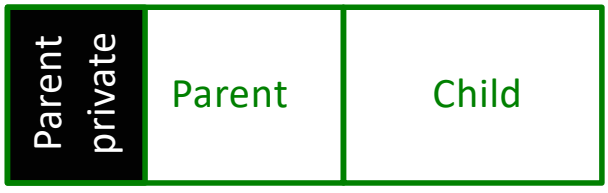
Child class

- 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

Inheriting Private Variables



Parent has private variables.
Objects of Parent class can access.



Child class inherits the private variables from Parent but cannot *directly* access them.
Call Parent methods that can!

`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

- `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

- Visibility for fields: 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 (or in same package) knows what they are doing

Guidance on Access Modifiers

- If you're uncertain which access modifier to use (public, protected, package/default, 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() {… }
   …
```

If no explicit call to super, calls *default* `super` constructor with no parameters

(not one of the examples posted online)

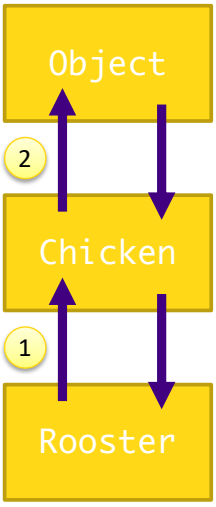
Constructor Chaining

- 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

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



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

Shadowing Parent Class Fields

- Shadowing: Child class has field with same name as parent class
  - You probably shouldn't shadow a field
- Example: more precision for a constant (e.g., 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

17

POLYMORPHISM & DISPATCH

18

Polymorphism

- ***Polymorphism*** is an object's ability 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

Somewhere Else…

- These objects were instantiated at some point in time …

```
Rooster foghorn = new Rooster(…);
Hen momma = new Hen(…);
Chicken baby = new Chicken(…);
```

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

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;
```

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`?

Polymorphism

- Which method do we call when we call `chicken[1].feed()`?
  Rooster's or Chicken's?
- In Java: 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

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 method corresponding to the actual class of each object at run time
  - This is the power of polymorphism and dynamic dispatch!

Dynamic Dispatch vs. Static Dispatch

- Dynamic dispatch is not necessarily a property of statically typed object-oriented programming languages 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

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");
    }
}
```

Think on your own for 1 minute

```
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("");
    }
}
```


27

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 DynamicDispat
    public static void mai
        Parent p = new Par
        Child c = new Chil

        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
```


28

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`?

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

PREVENTING INHERITANCE

Preventing Inheritance: `final` Class

- If you have a class and you do **not** want child/derived classes, you can define the class as `final`

```
public final class Rooster extends Chicken {
        . . .
}
```

- Examples of `final` class: `java.lang.System` and `java.lang.String`

Preventing Overriding: `final` Method

- If you don't want child classes to override a method, you can make that method `final`

```
class Chicken {
        . . .
        public final String getName() { . . . }
        . . .
}
```

Why would we want to make a method `final`?
What are possible benefits to us, the compiler, …?

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

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 (yet), just some of it
- Create a Goblin class and a Treasure class
  - Move Goblin and Treasure
- Due *next* Wednesday before class
  - Can start on Parts 0-2 now (harder parts than part 3)