Objectives

Packages

More on Inheritance



Review: Packages

- Hierarchical structure of Java classes
 - Similar to Python's modules

└ Scanner

Directories of directories java lang bject String rully qualified name: java.lang.String net util

Oct 7, 2024

Importing Packages

- Can import one class at a time or all the classes within a package
- Examples:

>* form may increase compile time

• BUT, no effect on run-time performance

Oct 7, 2024

Why Packages?

- Organizes code
 - Groups related code into directory structure
- Reduces chance of a conflict between names of classes
 - Example: Java's library has two classes named Array:

java.lang.reflect.Array
java.sql.Array

Packaging Code

- Use package keyword to say that a class belongs to a package:
 - >package my.package.name;
 - First line in class file
 - Classes without a declared package (like what we've been doing) are in the *default* package
- Typically, use a unique prefix, similar to domain names
 - ≻ com.ibm
 - >edu.wlu.cs.graffiti
- Use package name to create directory hierarchy
 - Example: code in edu.wlu.cs.graffiti package would be in a graffiti directory inside a cs directory inside a wlu directory inside an edu directory

We will start organizing our code in packages soon...

Oct 7, 2024

INHERITANCE

Review: Inheritance (from CSCI112)

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

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

Oct 7, 2024

Child class

- Inherits all of parent class's methods and fields
 - > private fields are inherited but can't access
 - > Static methods are inherited but cannot be overridden
- 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

Oct 7, 2024

Inheriting Private Variables



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

Parent brivate	Child
-------------------	-------

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

Oct 7, 2024

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
 - \succ Changing to less restrictive later \rightarrow easy

Changes to Chicken Class

- Added a new instance variable called isFemale
- Added getter and setter for isFemale
- Updated toString, equals methods accordingly
- 2 Chicken classes in examples
 > PrivateChicken.java *private* instance variables
 > ProtectedChicken.java *protected* instance variables

Oct 7, 2024





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



Comparing Rooster Implementations

```
@Override Parent class's weight and height are protected
public void feed() {
    //overrides superclass; greater gains by rooster
    weight += WEIGHT_GAIN;
    height += HEIGHT_GAIN;
} Need to trust child classes
won't mess up fields
```

```
@Override Parent class's weight and height are private
public void feed() {
    // overrides superclass; greater gains by rooster
    this.setWeight(this.getWeight() + WEIGHT_GAIN);
    this.setHeight(this.getHeight() + HEIGHT_GAIN);
}
Code is bulkier
```

Oct 7, 2024

23

Shadowing Parent Class Fields

- Shadowing: Child class has field with same name as parent class
 - >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

Oct 7, 2024

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

POLYMORPHISM & DISPATCH

Oct 7, 2024

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
```

Oct 7, 2024

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

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

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 a Rooster, but to compiler, it's a Chicken so

chickens[1].crow(); will not compile

Oct 7, 2024

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.

 \rightarrow 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 n new Chicken(...);

Oct 7, 2024

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?

Oct 7, 2024

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



 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!

Oct 7, 2024

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 (known at compile time) 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) 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 {
                                                   Think on your own for 1 minute
    public Parent() {}
    public void method1() {
        System.out.println("Parent: method1");
                                                 public class DynamicDispatchExample {
    }
                                                     public static void main(String[] args) {
                                                         Parent p = new Parent();
    public void method2() {
                                                         Child c = new Child();
        System.out.println("Parent: method2");
        method1();
                                                         p.method1();
    }
                                                         System.out.println("");
}
                                                         c.method1();
class Child extends Parent {
                                                         System.out.println("");
    public Child() {}
                                                         p.method2();
    public void method1() {
                                                         System.out.println("");
        System.out.println("Child: method1");
    }
                                                         c.method2();
}
                                                         System.out.println("");
                                                     }
                                                 }
    Oct 7, 2024
```

What Will This Code O	utput? Parent: method1		
<pre>class Parent { public Parent() {}</pre>	Child: method1		
<pre>public void method1() { System.out.println("Parent: method1"); }</pre>	public class DynamicDispat public static void mai Parent: method2 Parent: method2		
<pre>public void method2() { System.out.println("Parent: method2"); mathod1();</pre>	Parent p = new Par Fulcher method2 Child c = new Chil Child: method1		
<pre>method1(); }</pre>	p.method1(); System.out.println("");		
<pre>class Child extends Parent { public Child() {}</pre>	c.method1(); System.out.println("");		
<pre>public void method1() { System.out.println("Child: method1");</pre>	p.method2(); System.out.println("");		
} }	<pre>c.method2(); System.out.println(""); }</pre>		
Oct 7, 2024 Sprenkle	}		

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

Oct 7, 2024

PREVENTING INHERITANCE

Oct 7, 2024

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 3

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)

Oct 7, 2024