Polymorphism (part 2)

November 29, 2006

ComS 207: Programming I (in Java)
Iowa State University, FALL 2006
Instructor: Alexander Stoytchev

© 2004 Pearson Addison-Wesley. All rights reserved

Quick Review of Last Lecture

© 2004 Pearson Addison-Wesley. All rights reserved

Abstract Classes

- An ***abstract class*** is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
   // contents
}
```

© 2004 Pearson Addison-Wesley. All rights reserved

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so

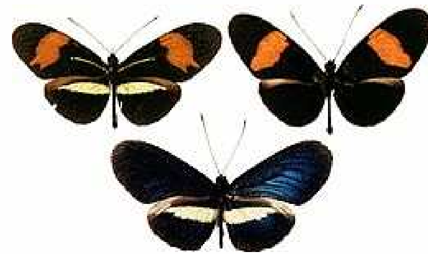
© 2004 Pearson Addison-Wesley. All rights reserved

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as `final` or `static`
- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too generic to instantiate

© 2004 Pearson Addison-Wesley. All rights reserved

Polymorphism in Nature



[http://www.blackwellpublishing.com/ridley/images/h_erato.jpg]

© 2004 Pearson Addison-Wesley. All rights reserved

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

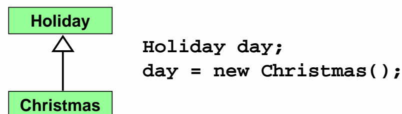
© 2004 Pearson Addison-Wesley. All rights reserved

Polymorphism via Inheritance

© 2004 Pearson Addison-Wesley. All rights reserved

References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a class called `Christmas`, then a `Holiday` reference could be used to point to a `Christmas` object




```
Holiday day;
day = new Christmas();
```

© 2004 Pearson Addison-Wesley. All rights reserved

Binding

- Consider the following method invocation:

```
obj.doIt();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes
- If this binding occurred at compile time, then that line of code would call the same method every time
- However, Java defers method binding until run time -- this is called *dynamic binding* or *late binding*
- Late binding provides flexibility in program design

© 2004 Pearson Addison-Wesley. All rights reserved

References and Inheritance

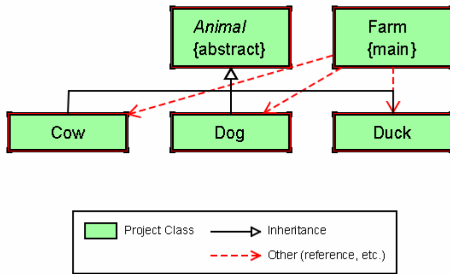
- Assigning a child object to a parent reference is considered to be a widening conversion, and can be performed by simple assignment
- Assigning an parent object to a child reference can be done also, but it is considered a narrowing conversion and must be done with a cast
- The widening conversion is the most useful

© 2004 Pearson Addison-Wesley. All rights reserved

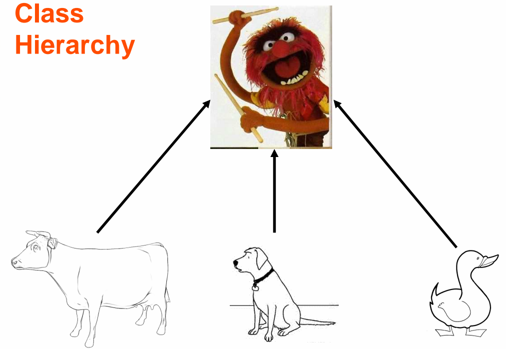
Example: Animals class hierarchy

- Animal.java
- Cow.java
- Duck.java
- Dog.java
- Farm.java

You can use jGrasp to draw diagram like this one



Class Hierarchy



```
public abstract class Animal
{
  abstract void makeSound();
}
```

```
public class Cow extends Animal
{
   public void makeSound()
   {
      System.out.println("Moo-Moo");
   }
}
```

```
public class Dog extends Animal
{
   public void makeSound()
   {
      System.out.println("Wuf-Wuf");
   }
}
```

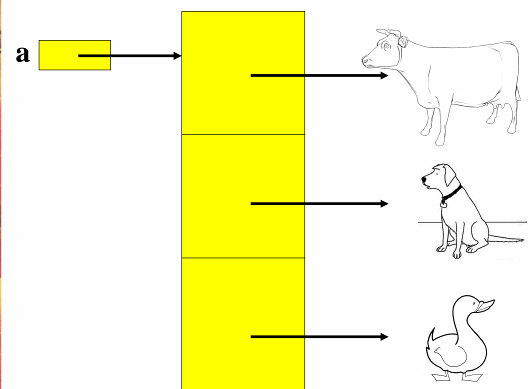
```
public class Duck extends Animal
{
   public void makeSound()
   {
      System.out.println("Quack-Quack");
   }
}
```

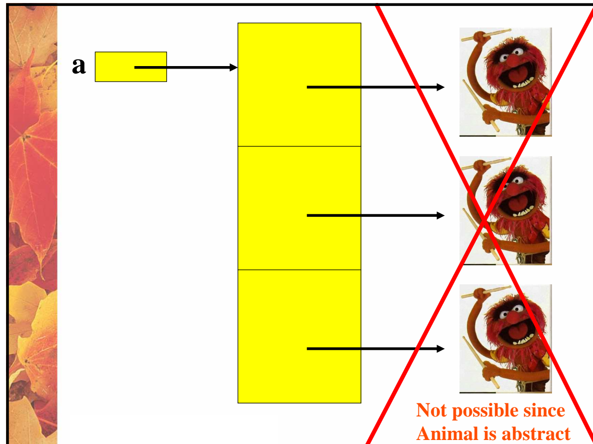
```
public class Farm
{
      public static void main(String[] args)
      {
            Cow c=new Cow();
            Dog d=new Dog();
            Duck k= new Duck();

            c.makeSound();
            d.makeSound();
            k.makeSound();
      }
}
```

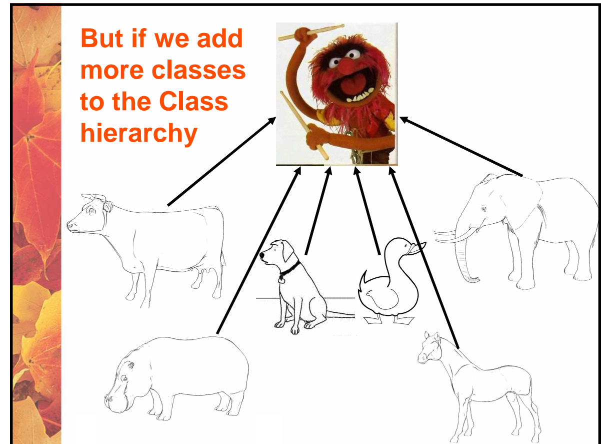
Result:
Moo-Moo
Wuf-Wuf
Quack-Quack

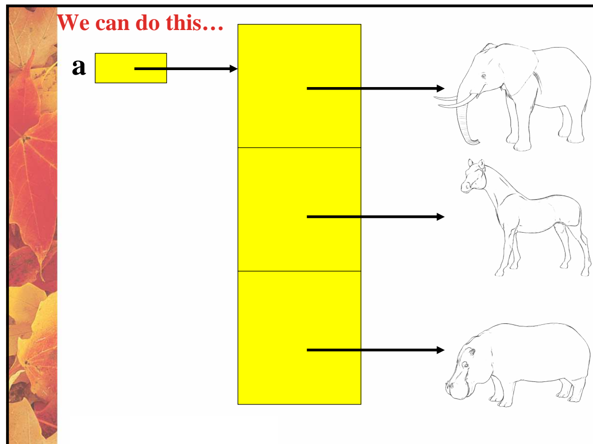
```
public class Farm2
{
      public static void main(String[] args)
      {
            Animal[] a = new Animal[3];

            a[0] = new Cow();
            a[1] = new Dog();
            a[2] = new Duck();

            for(int i=0; i< a.length; i++)
                  a[i].makeSound();
      }
}
```

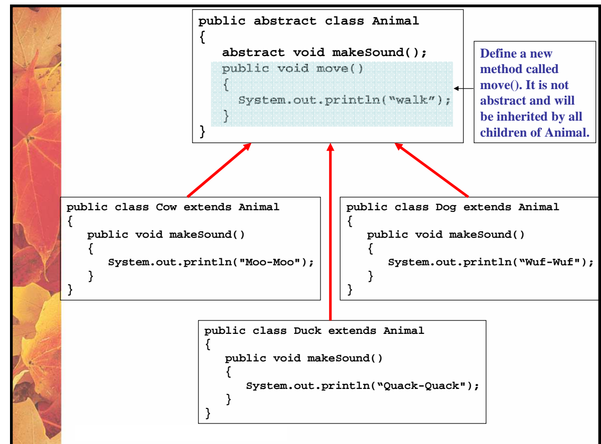
Result:
Moo-Moo
Wuf-Wuf
Quack-Quack





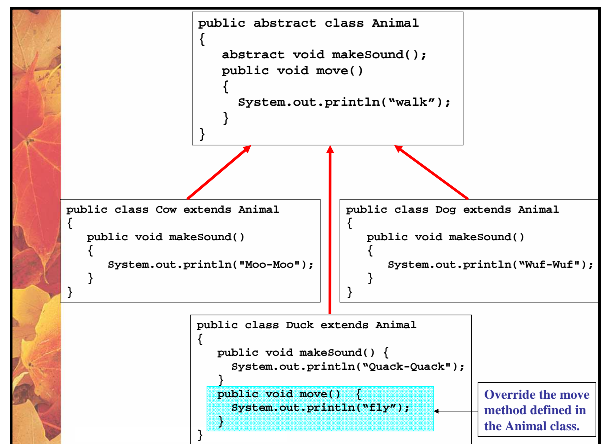






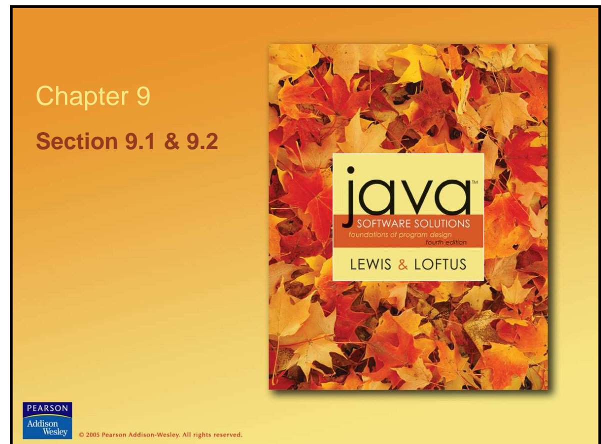

```
public class Farm2b
{
      public static void main(String[] args)
      {
            Animal[] a = new Animal[3];

            a[0] = new Cow();
            a[1] = new Dog();
            a[2] = new Duck();

            for(int i=0; i< a.length; i++)
                  a[i].move();
      }
}
```

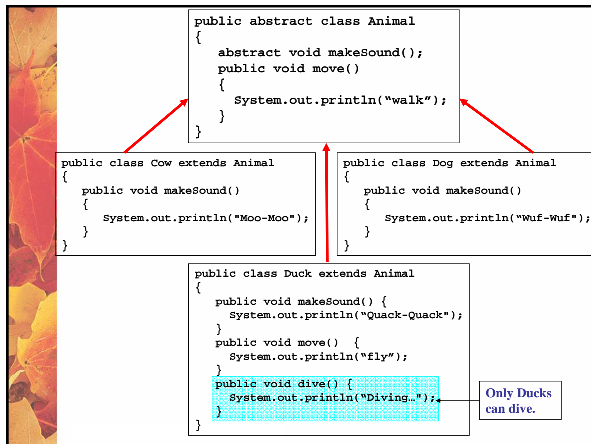
Result:
walk
walk
walk



```
public class Farm2c
{
      public static void main(String[] args)
      {
            Animal[] a = new Animal[3];

            a[0] = new Cow();
            a[1] = new Dog();
            a[2] = new Duck();

            for(int i=0; i< a.length; i++)
                  a[i].move();
      }
}
```

Result:
Walk
Walk
Fly






```
public class Farm2d
{
      public static void main(String[] args)
      {
            Animal[] a = new Animal[3];

            a[0] = new Cow();
            a[1] = new Dog();
            a[2] = new Duck();

            for(int i=0; i< a.length; i++)
                  a[i].dive();
      }
}
```

Compile Error, since dive() is defined only for Duck objects and not for all objects derived from Animal.

```
public class Farm2d
{
      public static void main(String[] args)
      {
            Animal[] a = new Animal[3];

            a[0] = new Cow();
            a[1] = new Dog();
            a[2] = new Duck();

            ((Duck)a[2]).dive();
      }
}
```

This works OK, but requires a cast from a reference to Animal to a reference to Duck.

```
public class Farm2d
{
      public static void main(String[] args)
      {
            Animal[] a = new Animal[3];

            a[0] = new Cow();
            a[1] = new Dog();
            a[2] = new Duck();

            ((Duck)a[2]).dive();
      }
}
```

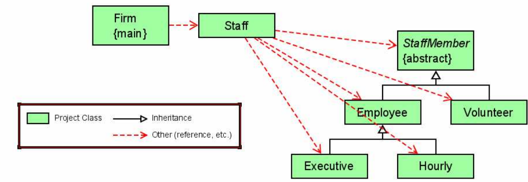
Result:
Diving…

Polymorphism via Inheritance

- Now let's look at an example that pays a set of diverse employees using a polymorphic method
- See `Firm.java` (page 486)
- See `Staff.java` (page 487)
- See `StaffMember.java` (page 489)
- See `Volunteer.java` (page 491)
- See `Employee.java` (page 492)
- See `Executive.java` (page 493)
- See `Hourly.java` (page 494)

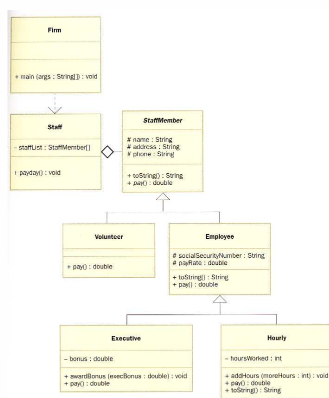
© 2004 Pearson Addison-Wesley. All rights reserved

Firm Class Hierarchy



© 2004 Pearson Addison-Wesley. All rights reserved

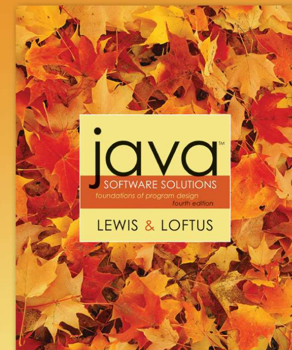
Employee Class Hierarchy



© 2004 Pearson Addison-Wesley. All rights reserved

Chapter 9

Section 9.3



PEARSON Addison Wesley

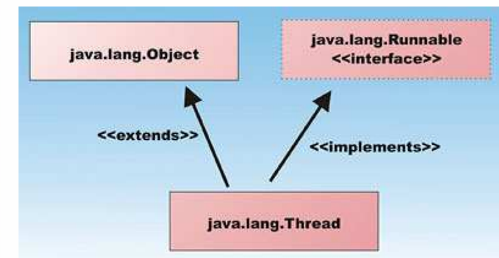
© 2005 Pearson Addison-Wesley. All rights reserved.

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

© 2004 Pearson Addison-Wesley. All rights reserved

This example shows how multiple inheritance can be faked in java



© 2004 Pearson Addison-Wesley. All rights reserved

[http://www.vsj.co.uk/pix/articleimages/may05/javathread3.jpg]

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

© 2004 Pearson Addison-Wesley. All rights reserved

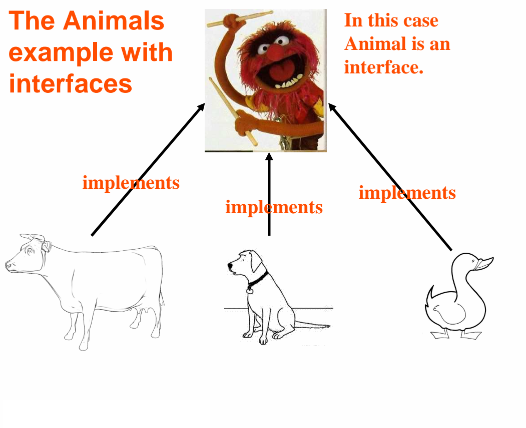
Polymorphism via Interfaces

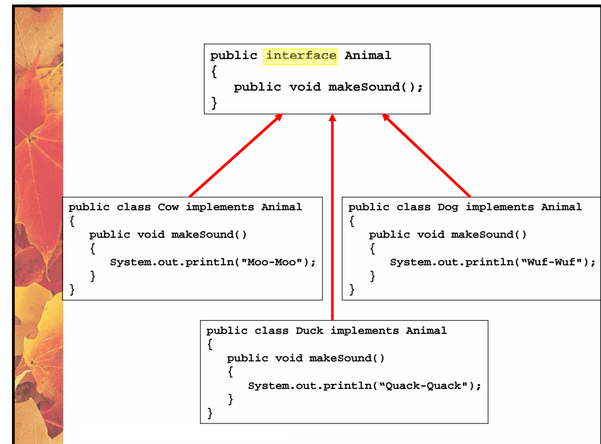
- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```

© 2004 Pearson Addison-Wesley. All rights reserved

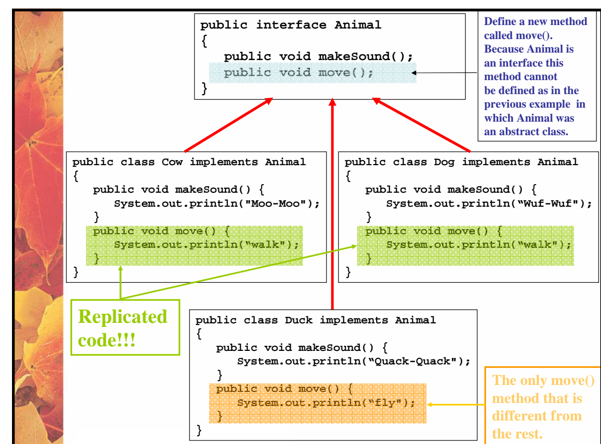
The Animals example with interfaces





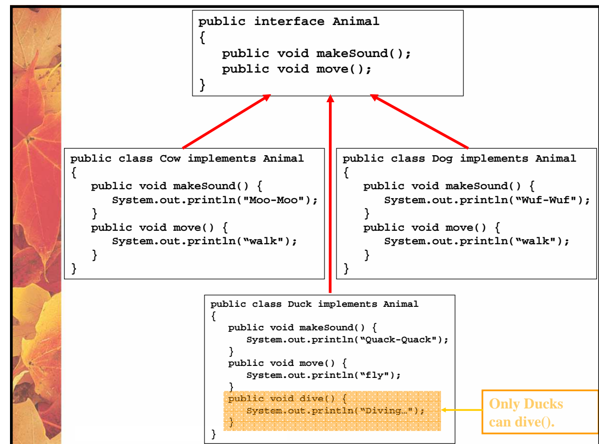

```
public class iFarm
{
      public static void main(String[] args)
      {
            Animal domestic;
            domestic = new Cow();
            domestic.makeSound();

            domestic = new Dog();
            domestic.makeSound();

            domestic = new Duck();
            domestic.makeSound();
      }
}
```

Result:
Moo-Moo
Wuf-Wuf
Quack-Quack



```
public class iFarm2
{
      public static void main(String[] args)
      {
            Animal domestic;
            domestic = new Cow();
            domestic.move();

            domestic = new Dog();
            domestic.move();

            domestic = new Duck();
            domestic.move();
      }
}
```

Result:
walk
walk
fly




```
public class iFarm3
{
      public static void main(String[] args)
      {
            Animal domestic;
            domestic = new Cow();
            //domestic.dive(); // error

            domestic = new Dog();
            //domestic.dive(); // error

            domestic = new Duck();
            // domestic.dive(); // error

            ((Duck)domestic).dive(); // OK, but uses a cast
      }
}
```

Result:
Ducks can dive.

THE END

© 2004 Pearson Addison-Wesley. All rights reserved