CMSC 202H

Polymorphism

Topics

- Binding (early and late)
- Upcasting and downcasting
- Extensibility
- The final modifier with
  - methods
  - classes

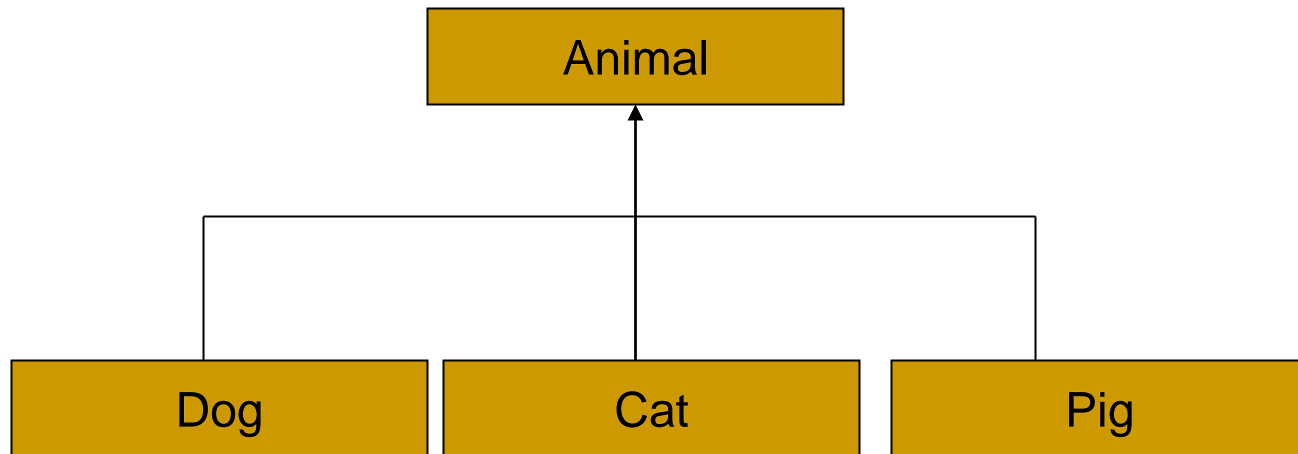
Introduction to Polymorphism

- Object-oriented programming mechanisms
  - Encapsulation - data and methods together
  - Inheritance - extending a class for specialization
  - Polymorphism

- Polymorphism
  - The ability to associate many meanings with one method name.
  - Accomplished through a mechanism known as `late binding` or `dynamic binding`.

Copyright © 2008 Pearson Addison-Wesley.
All rights reserved

Animal Hierarchy



Animals That Speak

```
public class Animal
{
   public void speak( int x )
   { System.out.println(" Animal " + x );}
}
public class Dog extends Animal
{
   public void speak (int x )
   { System.out.println( "Dog " + x ); }
}
public class Cat extends Animal
{
   public void speak (int x )
   { System.out.println( "Cat " + x ); }
}
public class Pig extends Animal
{
   public void speak (int x )
   { System.out.println( "Pig " + x ); }
}
```

The ZooDemo Class

In the ZooDemo, we ask each Animal to say hello to the audience.

```
public class ZooDemo
{
   // Overloaded type-specific sayHello method
   // for each kind of Animal

   public static void sayHello( Dog d, int i )
        { d.speak( i ); }

   public static void sayHello( Cat c, int i )
        { c.speak( i ); }

   public static void sayHello( Pig p, int i)
        { p.speak( i ); }
```

(continued)

The ZooDemo Class

```
    public static void main( String[ ] args )
    {
        Dog dusty = new Dog( );
        Cat fluffy = new Cat( );
        Pig sam = new Pig( );

        sayHello( dusty, 7 );
        sayHello( fluffy, 17 );
        sayHello( sam, 27 );
    }
} // end Zoo Demo

//------- output -----
Dog 7
Cat 17
Pig 27
```

Problems with ZooDemo?

- The ZooDemo class contains a type-specific version of sayHello for each type of Animal.
- What if we add more types of Animals?
- Wouldn’t it be nice to write just one sayHello method that works for all animals?

New ZooDemo

```
public class ZooDemo
{
   // One sayHello method whose parameter
   // is the base class works for all Animals

   public static void sayHello( Animal a, int x )
        { a.speak( x ); }

   public static void main( String[ ] args )
   {
        Dog dusty = new Dog( );
        Cat fluffy = new Cat( );
        Pig sam = new Pig( );

        sayHello( dusty, 7 );
        sayHello( fluffy, 17 );
        sayHello( sam, 27 );
   }
}
```

How Does New ZooDemo work?

- Associating the appropriate method definition with the method invocation is known as binding.
- Early binding occurs when the method definition is associated with its invocation when code is compiled.
  - With early binding, the method invoked is determined by the reference variable type.
- How can the compiler know which Animal’s speak method to call in sayHello? It can’t!

Late Binding

- The solution is to use late (dynamic) binding.
- Late binding
  - The appropriate method definition is associated with its invocation *at run-time*.
  - The method invoked is determined by the *type of object* to which the variable refers, **NOT** by the type of the reference variable.
- Java uses late binding for all methods except
  - `final`,
  - private (which are implicitly final), and
  - static methods.

An Object Knows the Definitions of Its Methods

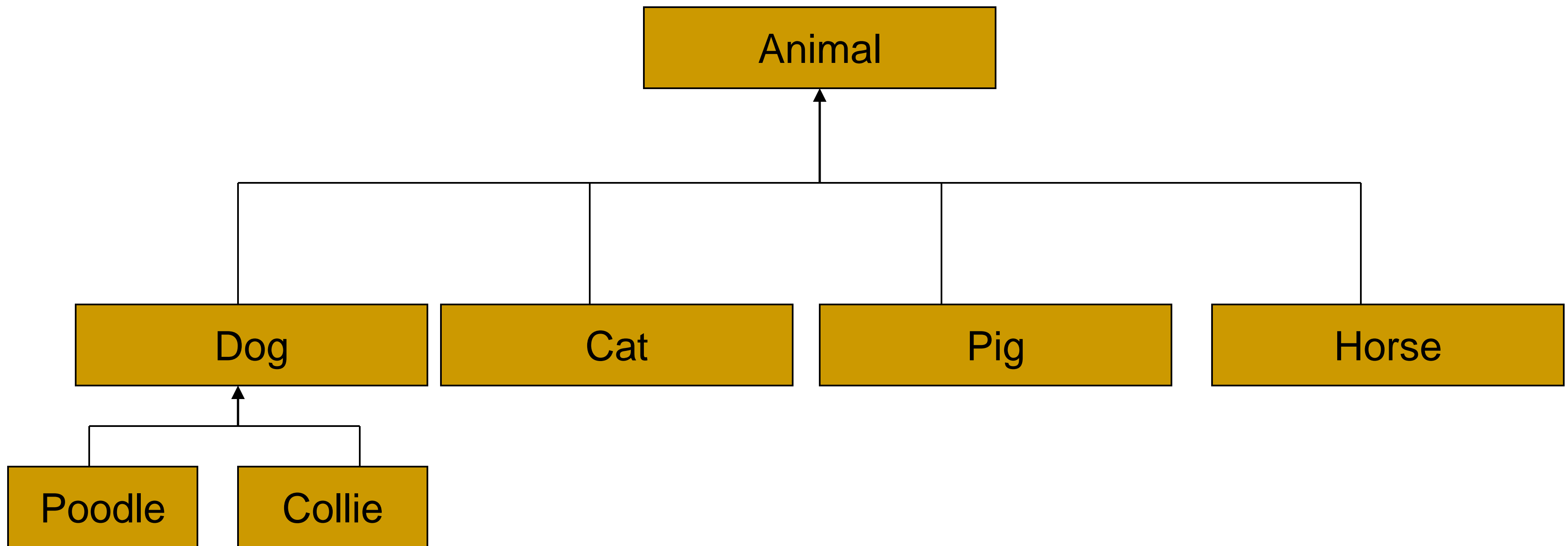
- The type of a class variable determines which method names can be used with the variable.
  - However, the object named by the variable determines which definition with the same method name is used.
- A special case of this rule:
  - The type of a class parameter determines which method names can be used with the parameter.
  - The argument determines which definition of the method name is used.

Copyright © 2008 Pearson Addison-Wesley.
All rights reserved

Using Polymorphism

- How do we take advantage of Polymorphism?
  - Write code to talk to base class objects (e.g. use base class references as method parameters).
  - Late binding will ensure that the appropriate method definition is used, even if a reference to a derived class is passed to the method.

More Animals



Extensibility

- Suppose more Animals were added to the hierarchy as shown in the previous diagram.
- All of these new classes work correctly with the old, unchanged sayHello method of the ZooDemo because sayHello's parameter is a base class reference.
- In a well designed OOP program, most of your methods will follow the model of sayHello and communicate with a base class reference and let late binding and polymorphism determine which sayHello method to call.
- Such a program is called ***extensible*** because you can add new functionality by deriving new classes from the base class without changing existing code.

The `final` Modifier

- A *method* marked `final` indicates that it cannot be overridden with a new definition in a derived class.
  - If `final`, the compiler can use early binding with the method.

```
public final void someMethod() { . . . }
```

- A *class* marked `final` indicates that it cannot be used as a base class from which to derive any other classes.

Copyright © 2008 Pearson Addison-Wesley.
All rights reserved

Late Binding with `toString`

- If an appropriate `toString` method is defined for a class, then an object of that class can be output using

  ```
  System.out.println( );
  ```

  As in this code snippet

  ```
  Animal max = new Animal( );
  System.out.println(max);
  ```

- This works because of late binding.

Copyright © 2008 Pearson Addison-Wesley.
All rights reserved

Late Binding with `toString`

- One definition of the method `println` takes a single argument of type `Object`:

```
public void println(Object theObject)
{
  System.out.println(theObject.toString());
}
```

  - In turn, It invokes the version of `println` that takes a `String` argument.

- Note that the `println` method was defined before the Animal class existed.

- Because of late binding, the `toString` method from the Animal class is used, not the `toString` from the `Object` class.

Copyright © 2008 Pearson Addison-Wesley.
All rights reserved

Upcasting and Downcasting

- **Upcasting** occurs when an object of a derived class is assigned to a variable of a base class (or any ancestor class).

```
Animal animalVariable;              // base class
Dog dogVariable = new Dog( );       // derived class
animalVariable = dogVariable;       // upcasting

animalVariable.speak(42);           // "Dog 42" is printed
```

Or we could do something equivalent, such as

```
Animal animal = new Dog( );
```

- Because of late binding, `speak` uses the definition of `speak` given in the `Dog` class.

Copyright © 2008 Pearson Addison-Wesley.
All rights reserved

Upcasting and Downcasting

- **Downcasting** occurs when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class).
  - Downcasting must be done very carefully.
  - In many cases it doesn't make sense, or is illegal:

```
dogVariable =                       //will produce
     (Dog)animalVariable;           //run-time error
dogVariable = animalVariable        //will produce
                                    //compiler error
```

  - There are times when downcasting is necessary; e.g., inside the `equals` method for a class:

```
Dog otherDog = (Dog)otherDog;          //downcasting
```


Copyright © 2008 Pearson Addison-Wesley.
All rights reserved