Polymorphism (part 2)

November 29, 2006

Quick Review of Last Lecture

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:

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

- 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

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

Polymorphism in Nature
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

Polymorphism via Inheritance

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();
```

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 runtime -- this is called dynamic binding or late binding
- Late binding provides flexibility in program design

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 a 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

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

```java
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
We can do this...

```java
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

But if we add more classes to the Class hierarchy:

```java
public abstract class Animal {
    abstract void makeSound();
    public void move() {
        System.out.println("walk");
    }
}

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");
    }
}
```

Define a new method called `move()`, it is not abstract and will be inherited by all children of Animal.
public class Cow extends Animal {
    public abstract void makeSound();
    public void move() {
        System.out.println("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 abstract class Animal {
    abstract void makeSound();
    public void move() {
        System.out.println("Walk");
    }
}

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

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

public class Duck extends Animal {
    public void makeSound() {
        System.out.println("Quack-Quack");
    }
    public void dive() {
        System.out.println("Diving");
    }
    public void move() {
        System.out.println("Swim");
    }
}

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].move();
    }
}

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

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

Chapter 9
Section 9.1 & 9.2
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)

Firm Class Hierarchy

Employee Class Hierarchy

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)
Polymorphism via Interfaces

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

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

•

•

© 2004 Pearson Addison-Wesley. All rights reserved

The version of `speak` via the current reference can be used to point to any object of any class that implements the `Speaker` interface.

The following line invokes one version and the second invokes another:

```java
Speaker guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.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:

```java
Speaker guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

The Animals example with interfaces implements

implements

implements

In this case `Animal` is an interface.

```java
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

© 2004 Pearson Addison-Wesley. All rights reserved

© 2004 Pearson Addison-Wesley. All rights reserved

© 2004 Pearson Addison-Wesley. All rights reserved

© 2004 Pearson Addison-Wesley. All rights reserved
```java
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
fly

```java
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 use a cast
    }
}
```

Result:
Ducks can dive.

```
public interface Animal
{
    public void makeSound();
    public void move();
}
```

```java
public class Cow implements Animal
{
    public void move() {
        System.out.println("walk");
        System.out.println("Moo-Moo");
    }
}
```

```java
public class Dog implements Animal
{
    public void makeSound() {
        System.out.println("walk");
    }
}
```

```java
public class Duck implements Animal
{
    public void move();
    public void makeSound() {
        System.out.println("fly");
        System.out.println("Quack-Quack");
    }
}
```

Only Ducks can dive.

THE END