Polymorphism

November 26, 2007

Quick Review of Last Lecture

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```java
float tryMe(int x)
{
    return x + .375;
}
```

```java
float tryMe(int x, float y)
{
    return x*y;
}
```

Method Overriding

- A child class can override the definition of an inherited method in favor of its own

- The new method must have the same signature as the parent’s method, but can have a different body

- The type of the object executing the method determines which version of the method is invoked

Final Exam

- Time:
  - Thursday Dec 13 @ 4:30-6:30 p.m.

- Location:
  - Curtiss Hall, room 127 (classroom)
Overriding

• A method in the parent class can be invoked explicitly using the `super` reference

• If a method is declared with the `final` modifier, it cannot be overridden

• The concept of overriding can be applied to data and is called *shadowing variables*

• Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding

• Overloading deals with multiple methods with the same name in the same class, but with different signatures

• Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

Overloading vs. Overriding

• Overloading lets you define a similar operation in different ways for different parameters

• Overriding lets you define a similar operation in different ways for different object types

Class Hierarchies

• A child class of one parent can be the parent of another child, forming a *class hierarchy*

Class Hierarchies

• Two children of the same parent are called *siblings*

• Common features should be put as high in the hierarchy as is reasonable

• An inherited member is passed continually down the line

• Therefore, a child class inherits from all its ancestor classes

• There is no single class hierarchy that is appropriate for all situations

Object – the mother of all objects in Java

```java
boolean equals (Object obj) {
    // Returns true if this object is an alias of the specified object.
}

String toString () {
    // Returns a string representation of this object.
}

Object clone () {
    // Creates and returns a copy of this object.
}
```
The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- As we’ve seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters
- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

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
  }
  ```

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

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
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)

This example shows how multiple inheritance can be faked in Java

Visibility Cartoon

Visibility Revisited

- It’s important to understand one subtle issue related to inheritance and visibility
- All variables and methods of a parent class, even private members, are inherited by its children
- As we’ve mentioned, private members cannot be referenced by name in the child class
- However, private members inherited by child classes exist and can be referenced indirectly

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent’s methods
- The super reference can be used to refer to the parent class, even if no object of the parent exists
Chapter 9
Section 9.1 & 9.2

Polymorphism

- Polymorphism is an object-oriented concept that allows us to create versatile software designs
- Chapter 9 focuses on:
  - defining polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references
  - using polymorphism to implement sorting and searching algorithms
  - additional GUI components

Polymorphism in Nature

Binding

- Consider the following method invocation:
  ```java
  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

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

- Suppose we create the following reference variable:
  ```java
  Occupation job;
  ```
- Java allows this reference to point to an `Occupation` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs
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

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

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

Polymorphism via Inheritance

• It is the type of the object being referenced, not the reference type, that determines which method is invoked
• Suppose the Holiday class has a method called celebrate, and the Christmas class overrides it
• Now consider the following invocation:
  ```java
day.celebrate();
```
• If day refers to a Holiday object, it invokes the Holiday version of celebrate; if it refers to a Christmas object, it invokes the Christmas version

Example: Animals class hierarchy

• Animal.java
• Cow.java
• Duck.java
• Dog.java
• Farm.java

You can use jGrasp to draw diagram like this one
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("Woof-Woof");
    }
}

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

Polymorphism via Inheritance

- Consider the following class hierarchy:

```
StaffMember

Volunteer

Employee

Executive

Hourly
```
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)