CS18000: Problem Solving and Object-Oriented Programming

Polymorphism
Video 1
Polymorphism and Abstract Classes
Polymorphism

Abstract Classes
Polymorphism
Dynamic Binding
Polymorphism

• “Many forms”
• Animals can take on many forms ... yet exhibit similar behaviors.
• Java allows a superclass variable to contain a reference to a subclass object
• The compiler chooses the subclass implementation of any overridden methods
• account.withdraw(amount); could be savings acct, checking acct, money market acct, ....
Polymorphism

Account account = new Student (...);

account.withdraw(50.00);

interest = account.getInterest();
Code Reuse

• Suppose you’re modeling animals
  – Dog
  – Cat
  – Fish
  – Horse
  – ...

• Lots of redundancy, so you create a superclass
  – Animal
  – All other subclasses extend Animal
  – Q: But what does “new Animal()” mean?
  – A: Nothing--don’t want to create a “generic” animal
Abstract Classes

• The Java solution for Animal: an abstract class
• Declaring a class abstract means that it cannot be instantiated
• Some methods may be unimplemented (just like an interface)
• But an abstract class may also include some implemented methods for default behavior
public abstract class Animal {
    abstract void speak();
    
    public static void main(String[] args) {
        Animal[] animals = new Animal[2];
        
        animals[0] = new Cat();
        animals[1] = new Dog();
        
        for (int i = 0; i < animals.length; i++)
            animals[i].speak();
    }
}
public class Cat extends Animal {
    void speak() {
        System.out.printf("Meow\n");
    }
}

public class Dog extends Animal {
    void speak() {
        System.out.printf("Bark\n");
    }
}
Video 2
Dynamic Binding and Abstract Methods
Dynamic Binding

- Methods are selected at runtime based on the class of the object referenced, not the class of the variable that holds the object reference.

- Example
  
  ```java
  Animal[] animals = new Animal[100];
  animals[i].speak();
  ```

- If `animals[i]` is a Dog, calls the `speak()` method in Dog, even though the variable is of type Animal.
Why polymorphism?

• Allows generic treatment of objects
• An array of Animals
  – Some are Dogs
  – Some are Cats
  – Some are new animal classes defined after the superclass code is written
• Programmer must be disciplined: the overridden methods should implement “consistent” or “expected” behavior
• Example: In Java, all GUI widgets are a subclass of Component; allows uniform treatment by GUI code
Reminder: Subclass Object

Contains its fields as well as all the fields defined in its superclasses...

Dog object

- name
- ...
- name
- ...

Fields defined in Object

Fields defined in Animal

Fields defined in Dog
public class Dog extends Animal {
    private String name;

    public Dog(String name) {
        super(name);
        this.name = super.getName() + " Barker";
    }

    public String getName() {
        return name;
    }

    void speak() {
        System.out.printf("Bark\n");
    }
}

Revised: Dog
public abstract class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    abstract void speak();

    public static void main(String[] args) {
        Animal[] animals = new Animal[2];
        animals[0] = new Cat("Garfield");
        animals[1] = new Dog("Snoopy");

        for (int i = 0; i < animals.length; i++)
            animals[i].speak();

        Dog d = new Dog("Marmaduke");
        System.out.println(d.getName());
        Animal a = d;
        System.out.println(a.getName());
    }
}
Abstract Methods

• Methods may be declared abstract
  – Provide only the header (no body)
  – Class must then be declared abstract
• Methods in an interface are implicitly declared abstract
• When subclassing an abstract class
  – Generally provide method bodies for abstract methods
  – If abstract methods remain, then subclass is still abstract and must be declared so
Example: Abstract Methods

```java
abstract public class AbstractParent {
    abstract void doOne();
    abstract void doTwo();
}
abstract class AbstractChild extends AbstractParent {
    void doOne() {
        System.out.println("in AbstractChild");
    }
}
class ConcreteGrandChild extends AbstractChild {
    void doTwo() {
        System.out.println("in ConcreteGrandChild");
    }
}
public static void main(String[] args) {
    ConcreteGrandChild cc = new ConcreteGrandChild();
    cc.doOne();
    cc.doTwo();
}
```