Java Programming 1

Polymorphism

Part 2

Contents

- Examples
- The Mechanics of Polymorphism
  - Abstract Classes (Chapter 5)
- Private and Static Methods
- Order of Constructor Calls
- Designing Classes
- Summary of Polymorphism

- Reference
  - Bruce Eckel, Thinking in Java, Chapter 7 http://www.faqs.org/docs/think_java/TIJ309.htm

Introduction

- Polymorphism is one of the most important concepts of object-oriented programming.
- In general, it means the occurrence of something in multiple forms.
- In programming, polymorphism is the ability for same code to be used with several different types of objects and behave differently depending on the actual type of object used.

Example 3

```
class Person {
  private String name;
 public Person(String name) {
   this.name = name;
 }
 public String introduction() {
    return "My name is " + name + ".";
  }
}
class Student extends Person {
  private String id;
  public Student(String name, String id){
    super(name);
    this.id = id;
  }
 public String getID() { return id; }
 public String introduction() {
    return "I am a student. " +
    super.introduction() + " My ID is "+ id + ".";
  }
}
```

```
public class PolymorphismDemo3 {
    public static void main(String[] args) {
        Person[] people = {
          new Person(" Suzuki "),
          new Student(“Tanaka", “s116000"),
          new Person(" Murakawa")};
 // print information about each person
        for (int i = 0; i < people.length; i++) {
            System.out.println(
              people[i].introduction()); }
         }
}
```

- Output of this program:
  - My name is Suzuki .
  - I am a student. My name is Tanaka. My ID is s116000.
  - My name is Murakawa.

Comments on the Previous Slide

- Example 3:
  - An array of three (people) objects is created.
  - The value of *people[0]* is a reference to the *Person(“Suzuki”)* object.
  - The value of *people[1]* is a reference to the *Student(“Tanaka”, “s116000”)* object.
  - The value of *people[2]* is a reference to the *Person(“Murakawa”)* object.

Example 4

```
class Person {
  private String name;
 public Person(String name) {
   this.name = name;
 }
 public String introduction() {
    return "My name is " + name +".";
  }
 public String getInfo() {
    return introduction();
  }
}
class Student extends Person {
  private String id;
  public Student(String name, String id){
    super(name);
    this.id = id;
  }
public String getID() { return id; }
 public String introduction() {
    return "I am a student. "+
     super.introduction()+" My ID is " +
        getID()+".";
  }
}
public class PolymorphismDemo4 {
  public static void main(String[] args) {
       Student s =
          new Student("Saito","s115333");
        Person p = s;
        System.out.println(s.getInfo());
        System.out.println(p.getInfo());
   }
}
```

- Output of this program:
  - I am a student. My name is Saito. My ID is s115333.
  - I am a student. My name is Saito. My ID is s115333.

Comments on the Previous Slide

- The difference between Example 1 and this example:
  - The *Person* class has a *public String getInfo()* method.
- Why do they print the same output? (The reason is the same as for Example 1 from the previous lecture):
  - System.out.println(s.getInfo());
  - System.out.println(p.getInfo());
- The following statement is wrong. (The *getID* method is not in the set of the *Person* class methods; the compiler produces an error message):
  - System.out.println(p.getID()); // Error

Abstract Classes

- An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:
  - abstract void moveTo(double deltaX, double deltaY);
- If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject {
   // declare fields
   // declare non-abstract methods
   abstract void draw();
}
```

Abstract Classes

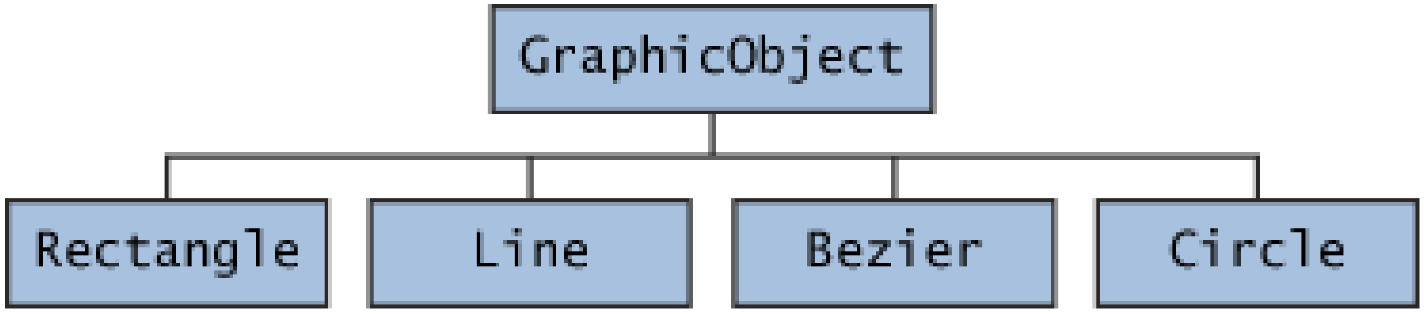
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

An Abstract Class Example

- In an object-oriented drawing application, you can draw circles, rectangles, lines, and many other graphic objects.
- These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common.
- Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw.
- All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass.

An Abstract Class Example

- You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, GraphicObject, as shown in the following figure.



An Abstract Class Example

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

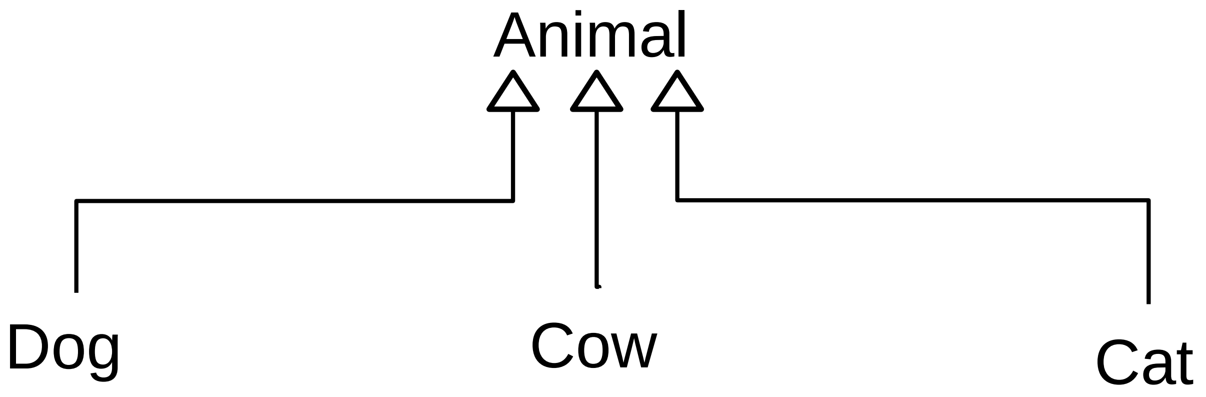
- GraphicObject is an abstract class. It has member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method.
- GraphicObject declares abstract methods (draw or resize), that need to be implemented by all subclasses but must be implemented in different ways.

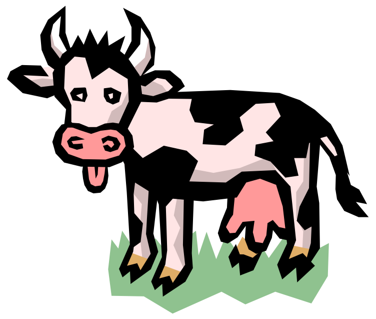
An Abstract Class Example

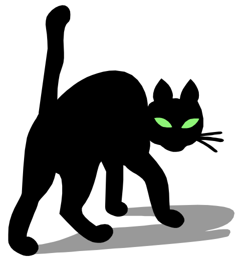
- Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize. methods

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

Animals: Different Ways of Talking







Talk

Talk

Talk

Dog: Woof

Cow: Moo

Cat: Meow

Solution: Different Ways of Talking

```
abstract class Animal   {
   private String name;
   public Animal(String name) {
              this.name=name; }
   public String getName() { return name; }
   public abstract void talk();
}
class Dog extends Animal {
   public Dog(String name) { super(name); }
   public void talk() {
       System.out.println(getName()+" Woof");
   }
 }
class Cat extends Animal {
   public Cat(String name) { super(name); }
   public void talk() {
      System.out.println(getName()+" Meow");
   }
 }
```

```
class Cow extends Animal {
 public Cow(String name) { super(name);
}
   public void talk() {
   System.out.println(getName()+" Moo");
   }
 }
public class AnimalReference {
  public static void main(String[] args){
       Animal ref = new Cow("Edna");
       Dog aDog = new Dog("Humi");
        ref.talk();
        ref = aDog;
        ref.talk();
        ref = new Cat("Aya");
        ref.talk();
   }
}
```

Output:

```
Edna  Moo
Humi  Woof
Aya  Meow
```

Comments on the Previous Slide

- The *Animal* class is abstract:
  - There is no implementation for the *talk* method.
- *AnimalArray* class:

  Animal[] ref = new Animal[3];
  - Declaration of the *ref* array to store three objects of *Animal* type or its subclass.

```
public class AnimalArray {
   public static void main(String[] args) {
// assign space for an array
         Animal[] ref = new Animal[3];
         Random rand = new Random();
// create specific objects and put them in array
         ref[0] = new Cow("Edna");
         ref[1] = new Dog("Humi");
         ref[2] = new Cat("Aya");
         ref[rand.nextInt(3)] = new Cat("Kitty");
// Compiler does not know where Kitty  is
          for (int i=0;i<3;++i) {
               ref[i].talk(); }
     }
}
```

Note: Private Methods

- A *private* method is automatically final, and is also hidden from the derived class.
- *f( )* in the *Derived* class is a brand new method; it's not even overloaded, since the base-class version of *f( )* isn't visible in *Derived.*

```
public class PrivateOverride {
  private void f() {
    System.out.println("private f()");
  }
  public static void main(String[] args) {
    PrivateOverride po = new Derived();
    po.f();
  }
}
class Derived extends PrivateOverride {
  public void f() {
    System.out.println("public f()");
  }
}
```

Output :

```
private f()
```

Note: Static Methods

```
class Mother {
      public static String staticGet()
           {  return "Mother staticGet()"; }
      public String dynamicGet()
            { return "Mother dynamicGet()";  }
}
class Child extends Mother {
      public static String staticGet()
         { return "Child staticGet()"; }
       public String dynamicGet()
          { return "Child dynamicGet()"; }
}
public class StaticPolymorphism {
  public static void main(String[] args) {
       Mother child = new Child();
       System.out.println(child.staticGet());
       System.out.println(child.dynamicGet());
  }
}
```

Output:

```
Mother staticGet()
Child dynamicGet()
```

- If a method is *static*, it does not behave polymorphically.
- *Static* methods are associated with the class and not the individual objects.

Order of Constructor Calls

```
class Meal {
  Meal() { System.out.println("Meal()"); }
}
class Bread {
  Bread() { System.out.println("Bread()"); }
}
class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}
class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch {
  PortableLunch() {
       System.out.println("PortableLunch()");}
}
```

```
public class Sandwich extends
                              PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
}
```

Output:

```
      Meal()
      Lunch()
      PortableLunch()
      Bread()
      Cheese()
      Lettuce()
      Sandwich()
```

Comments on the Previous Example

- This example creates a complex class out of other classes, and each class has a constructor that announces itself.
- The important class is *Sandwich*, which reflects three levels of inheritance (four, if the implicit inheritance from *Object* is counted) and three member objects.
- The order of constructor calls for a complex object:
  - The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructed first, followed by the next-derived class, etc., until the most-derived class is reached.
  - Member initializers are called in the order of declaration.
  - The body of the derived-class constructor is called.

Comments on the Previous Example

- The order of the constructor calls is important
  - When you inherit, you know all about the base class and can access any **public** and **protected** members of the base class.
  - This means that you must be able to assume that all the members of the base class are valid when you're in the derived class.
  - Inside the constructor, you must be able to assume that all members that you use have been built.
  - The only way to guarantee this is for the base-class constructor to be called first.

Designing Classes: HAS-A vs. IS-A

- Inheritance
  - We say an object X is-a Y, if everywhere you can use an object of type Y, you can use instead of object of type X. In other words, X is a proper subtype of Y.   So, you know that X and Y conform to the same set of method type signatures, however, their implementation may be different.
- Composition
  - We say an object X has-a Y, if Y is a part-of X. So, you typically think of X containing an instance of Y, not X inheriting from Y.
- Compare: Slide 4, Lecture 6

Designing Classes: HAS-A vs. IS-A

- Once you learn about polymorphism, you may think that everything should be inherited, because polymorphism is such a clever tool. This can burden your designs; in fact, if you choose *inheritance first* when you're using an existing class to make a new class, things can become complicated.
- A better approach is to choose *composition first*, especially when it's not clear which one you should use. Composition does not force a design into an inheritance hierarchy. But composition is also more flexible since it's possible to dynamically choose a type (and thus behavior) when using composition, whereas inheritance requires an exact type to be known at compile time.

Designing Classes: HAS-A vs. IS-A

```
abstract class Actor {
  public abstract void act();
}
class HappyActor extends Actor {
  public void act() {
    System.out.println("HappyActor");
  }
}
class SadActor extends Actor {
  public void act() {
    System.out.println("SadActor");
  }
}
class Stage {
  private Actor actor = new HappyActor();
  public void change()
      {  actor = new SadActor(); }
  public void performPlay() { actor.act(); }
}
```

```
public class Transmogrify {
   public static void main(String[] args) {
    Stage stage = new Stage();
    stage.performPlay();
    stage.change();
    stage.performPlay();
   }
 }
```

Output:

```
 HappyActor
 SadActor
```

Comments on the Previous Slide

- A *Stage* object contains a reference to an *Actor*, which is initialized to a *HappyActor* object. This means *performPlay( )* produces a particular behavior. But since a reference can be rebound to a different object at run time, a reference for a *SadActor* object can be substituted in *actor*, and then the behavior produced by *performPlay( )* changes. Thus you gain dynamic flexibility at run time.
- In contrast, you can’t decide to inherit differently at run time; that must be completely determined at compile time.

Designing Classes: HAS-A vs. IS-A

- A general guideline is:
  - Use inheritance to express differences in behavior, and
  - Use fields to express variations in state.
- In the preceding example, both are used; two different classes are inherited to express the difference in the *act( )* method, and *Stage* uses composition to allow its state to be changed. In this case, that change in state happens to produce a change in behavior.

Classes are in Different Files

```
// Person.java
public class Person {
  private String name;
 public Person(String name) {
   this.name = name;
 }
 public String introduction() {
    return "My name is " + name + ".";
  }
}
// Student.java
public class Student extends Person {
  private String id;
  public Student(String name, String id){
    super(name);
    this.id = id;
  }
 public String introduction() {
    return "I am a student. " +
    super.introduction() + " My ID is "+ id + ".";
  }
}
```

```
// PolymorphismDemo3.java
public class PolymorphismDemo3 {
    public static void main(String[] args) {
        Person[] people = {
          new Person("Suzuki"),
          new Student("Tanaka", "s116000"),
          new Person("Murakawa")};
 // print information about each person
        for (int i = 0; i < people.length; i++) {
            System.out.println(
              people[i].introduction()); }
         }
}
```

- Compiling:
  - javac *.java
- Running
  - java PolymorphismDemo3

Comments on the Previous Slide

- Every public class that you write must be in a separate .java file where the first part of the file name is identical to the class name.
  - Person.java
  - Student.java
  - PolymorphismDemo3.java
- Files should be in the same directory.
- Run
  - java PolymorphismDemo3

  where PolymorphismDemo3 is the class which contains the *public static void main(String[] args)* method.

Summary of Polymorphism

- Polymorphism means “multiple forms”.
  - In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the dynamically bound methods.
- Polymorphism is a feature that cannot be viewed in isolation (like a **switch** statement can, for example), but instead works only in concert, as part of a “big picture” of class relationships.
- To use polymorphism (and thus object-oriented techniques) effectively in your programs, you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other.
  - The results are faster program development, better code organization, extensible programs, and easier code maintenance.