Polymorphism

Contents

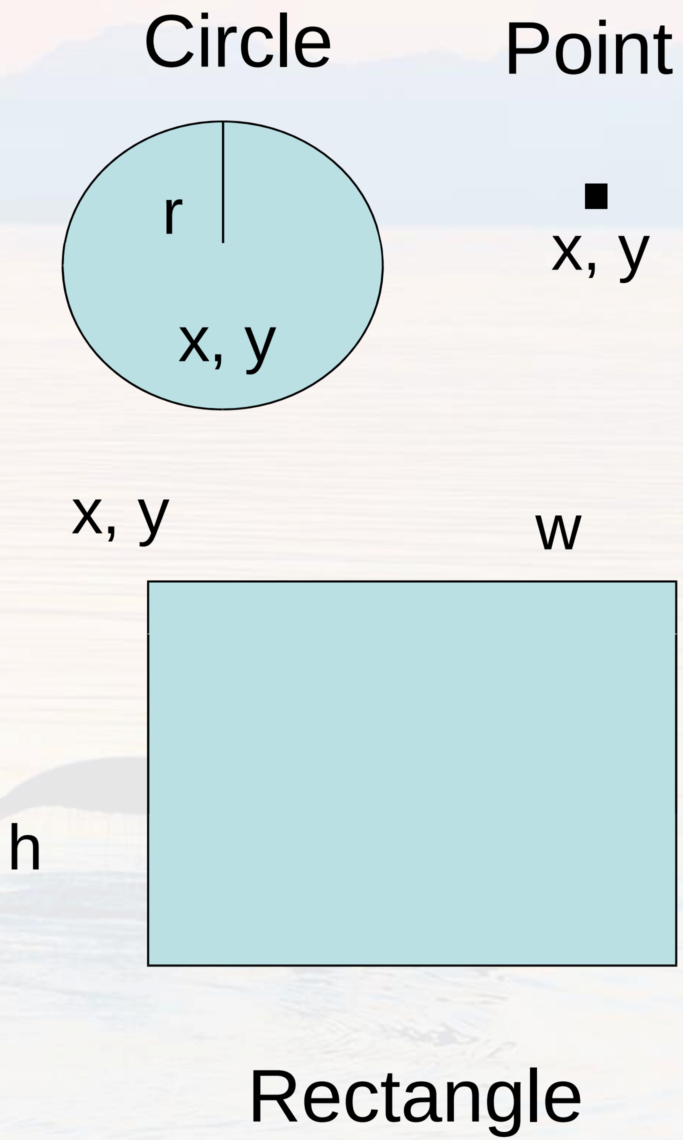
- Introduction
- Examples
- The Mechanics of Polymorphism
  - Static and Dynamic Binding
  - Abstract Classes
- Private and Static Methods

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: Drawing Shapes

- Write a program to maintain a list of shapes created by the user, and print the shapes when needed.
- The shapes needed in the application are:
  - points
  - lines
  - rectangles
  - circles
  - etc...



In Conventional Programs

- When you use the C language you should:
  - Define the *struct* data type to store parameters of the shape
    - One field is for the type of the shape: point, circle, etc.
  - Write the functions to draw each shape (separate for each shape).
  - Check the type of the shape first to select the right function to draw.

```
typedef struct shape {
    int typeS; // point = 0, circle = 1,
                   // ,line = 2, rectangle = 3
    int x, y // parameters of the shape
. . .
} ;
shape varShape;

. . .
if (varShape.typeS == 1) then
            DrawCircle(varShape);
else if (varShape.typeS == 3) then
            DrawRectangle(varShape);
else if (varShape.typeS == 0) then
            DrawPoint(varShape);
else if(varShape.typeS == 2) then
            DrawLine(varShape);
```

Using Polymorphism

- You need only to write:
  - varShape.Draw()

- How to do this?

Example 1

```
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 PolymorphismDemo1 {
    public static void main(String[] args) {
        Student s =
            new Student("Xiaoli","s115333");
        Person p = s;
        System.out.println(s.introduction());
        System.out.println(p.introduction());
    }
}
```

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

Comments on the Previous Slide

- Consider two simple classes:
  - Person
  - Student (this one is a subclass of Person)
- Why do they print the same output?
  - —System.out.println(s.introduction());
  - —System.out.println(p.introduction());
- Because the same message (*introduction()*) is sent to the same object, in this case Student.
- Why is the object the same (Student)?

Recall: Primitive Assignment

- The act of assignment takes a copy of a value and stores it in a variable.
- For primitive types:

```
num2 = num1;
```

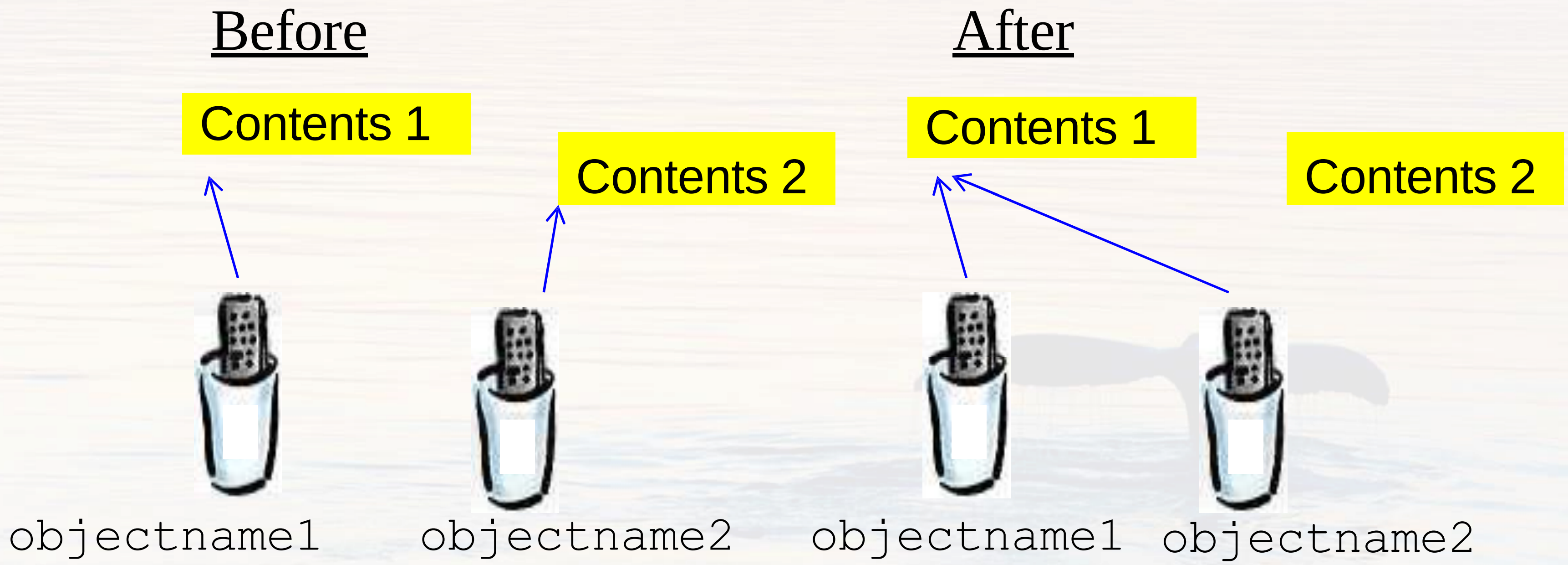
<u>Before</u> <u>After</u>

num1 num2 num1 num2

Recall: Reference Assignment

- For object references, the reference (address, the location) is copied:

```
objectname2 = objectname1;
```



Example 2

```
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 PolymorphismDemo2 {
    public static void main(String[] args) {
        m(new Student("Xiaoli", "s115333"));
        m(new Person( “Xiaowang"));
    }
    public static void m(Person x) {
        System.out.println(x.introduction());
    }
}
```

Comments on the Previous Slide

- Method *m* takes a parameter of the *Person* type. An object of a subtype can be used wherever its supertype value is required.
  - This feature is known as *polymorphism*.

- When the method *m(Person x)* is executed, the argument x's *introduction* method is invoked. *x* may be an instance of *Student* or *Person*. Classes *Student* and *Person* have their own implementation of the *introduction* method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.
  - This capability is known as *dynamic binding*.

Comments on Examples 1 and 2

- Example 1: The Java compiler cannot decide at compilation time which method must be called when the program is running:
  - Introduction() of the Person class or of the Student class
    - —System.out.println(s.introduction());
    - —System.out.println(p.introduction());
- Example 2: The same situation
    - —System.out.println(x.introduction());
- A decision is made when the program is running.

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(" Xiaoli "),
          new Student( “Xiaowang",  “s116000"),
          new Person(" Xiaozhang")};
        // print information about each person
        for (int i = 0; i < people.length; i++) {
              System.out.println(
                   people[i].introduction()); }
        }
 }
```

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("Xiaoli")* object.
  - The value of *people[1]* is a reference to the *Student("Xiaowang" , "s116000")* object.
  - The value of *people[2]* is a reference to the *Person("Xiaozhang")* 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("Xiaoli","s115333");
        Person p = s;
        System.out.println(s.getInfo());
        System.out.println(p.getInfo());
   }
```

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):
  - —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

Static and Dynamic Binding

- Non-polymorphic methods (static methods) are "bound"
  - at compile time
  - called *early binding* or static binding.
- Polymorphic methods are "bound"
  - at run time
  - called *late binding* or dynamic binding (also called dynamic dispatch).
- Alternate views of polymorphism:
  - One objects sends a message to another object without caring about the type of the receiving object.
  - The receiving object responds to a message appropriately for its type.
- Java methods are polymorphic by default
  - *static* or *final* (*private* methods are implicitly *final*) are bound at compile time.

Note: Polymorphic Methods

- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in a superclass is redefined in a subclass, the method defined in the superclass is hidden.

Note: Polymorphic Methods

```
class Parent {
    public static void myStaticMethod()  {
        System.out.println("A");
    }
    public void myInstanceMethod()   {
        System.out.println("B");
    }
}
public class Child extends Parent {
   public static void myStaticMethod()  {
        System.out.println("C");
    }
    public void myInstanceMethod()   {
        System.out.println("D");
    }
```

```
public static void main(String[] args)  {
       Parent o1 = new Parent();
       Parent   o2   =   new   Child();
       Child  o3 = new Child();

       Parent.myStaticMethod();     //
       Child.myStaticMethod();      //
       o1.myStaticMethod();         //
       o1.myInstanceMethod();  //
       o2.myStaticMethod();       //
       o2.myInstanceMethod();  //
       o3.myStaticMethod();       //
       o3.myInstanceMethod();  //
       myStaticMethod();          //
       myInstanceMethod();//
   }
}
```

Comments on the Previous Slide

- Notice that *o2.myStaticMethod* invokes *Parent.myStaticMethod()*. If this method were truly overridden, we should have invoked *Child.myStaticMethod*, but we didn't. Rather, when you invoke a static method, even if you invoke it on an instance, as we did here, you really invoke the method associated with the "compile-time type" of the variable. In this case, the compile-time type of *o2* is *Parent*. Therefore, we invoke *Parent.mStaticMethod()*.

- However, when we execute the line *o2.myInstanceMethod()*, we really invoke the method *Child.myInstanceMethod()*.

  That's because, unlike static methods, instance methods CAN be overridden. In such a case, we invoke the method associated with the run-time type of the object. Even though the compile-time type of *o2* is *Parent*, the run-time type (the type of the object *o2* references) is *Child*. Therefore, we invoke *Child.myInstanceMethod* rather than *Parent.myInstanceMethod()*.

Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two issues.
  - The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
- A method may be implemented in several subclasses.
- The Java Virtual Machine dynamically binds the definition of the method at runtime.

Dynamic Binding in Java

- We can conceptually think of the dynamic binding mechanism as follows: Suppose an object *o* is an instance of classes $C_1, C_2, ..., C_{n-1}$, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n.
- That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the *Object* class.
- If *o* invokes a method *p*, the JVM searches the implementation for the method *p* in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

Since o is an instance of C_1, o is also an instance of C_2, C_3, . . . , C_{n-1}, and C_n

The `instanceof` Operator

- Often you will get into a situation in which you need to rediscover the exact type of the object so you can access the extended methods of that type (see Example 2, slide 11):

```
Person p = new Student("Xiaoli","s115333");
System.out.println(p.getID()); // Compile-time error:
          // There is no the getID method in the Person class.
```

- Use the `instanceof` operator to test whether an object is an instance of a class:

```
Person p = new Student("Xiaoli","s115333");
if (p instanceof Student) {
  System.out.println( "Student ID:"   + ((Student)p).getID());
}
```

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

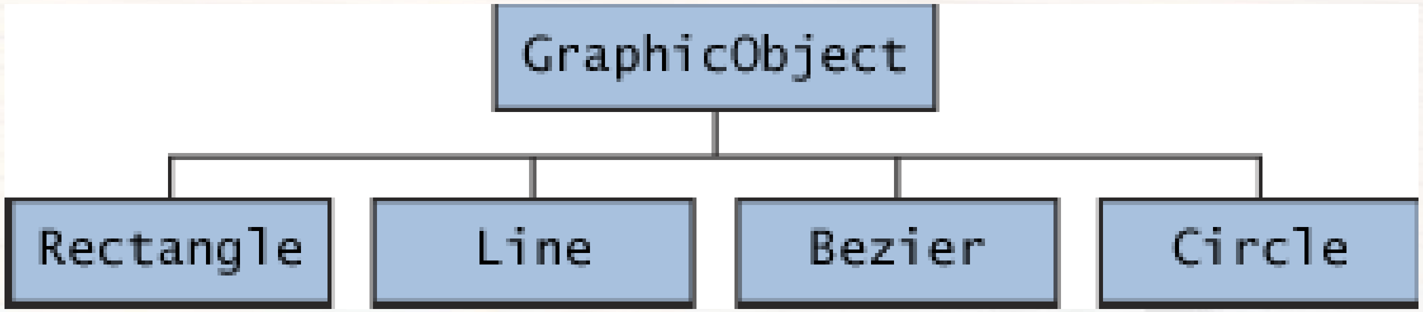
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

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() {
        ...
    }
}
```

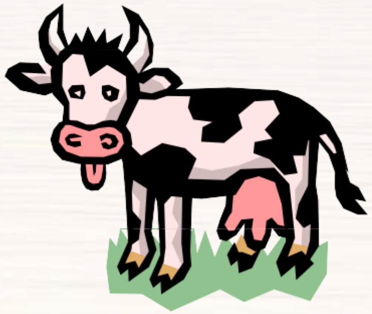
Animals: Different Ways of Talking

Animal

Dog

Cow

Cat



Talk

Talk

Talk

Dog: Woof

Cow: Moo

Cat: Meow

Solution: Different Ways of Talking

```java
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");
    }
}
```

```java
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();
   }
}
```

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()");
    }
}
```

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());
  }
}
```

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

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.