Polymorphism

Hsuan-Tien Lin

Department of CSIE, NTU

OOP Class, April 8, 2013

Polymorphism: The Motto

One Thing, Many Shapes.....

One Variable, Many Values

```
1  char a;
2  switch(a){
3  case 1:
4    return 1 * 1;
5  case 2:
6    return 2 * 2;
7  case 3:
8    return 3 * 3;
9  ...
10 case 127:
11   return 127 * 127;
12 }
```

- better ways?
- mechanism? raw memory interpretation

One Class, Many Instances

```
1 Student s;
2 if (s equals student1)
3   show(score1);
4 else if (s equals student2)
5   show(score2);
6 ...
7 else if (s equals student100)
8   show(score100);
```

- better ways?
- mechanism? data abstraction

One Method Name, Many Parameter Lists

```
1  //no overloading
2  System.out.printString("abc");
3  System.out.printInt(3);
4  System.out.printDouble(4.0);
5  //overloading
6  System.out.print("abc");
7  System.out.print(3);
8  System.out.print(4.0);
```

- mechanism? signature by name + parameter types

A Twist in Method Overloading

```
1  //overloading
2  System.out.print("abc");
3  System.out.print(3);
4  System.out.print(4.0);
5  //a twist (of course, not exactly workable
6  String("abc").print();
7  Integer(3).print();
8  Double(4.0).print();
```

- mechanism? just write print() for every class (we'll see)

Polymorphic Behavior

- all cases above: some polymorphic behavior
- **BUT** almost no one calls them real “polymorphism”

Why Not?

Polymorphic Behavior on Known Stuff

- polymorphic behavior of **known** primitive-type variables
- polymorphic behavior of **known** classes (extended types)
- polymorphic behavior of **known** parameter types

Unknown Stuff: Future Extensions

Backward Compatibility

inheritance hierarchy

Forward Advance

newly added variables/methods

Polymorphism of Instance Content

one **advanced** content, many **compatible** ways to access

Polymorphism of Instance References

one **compatible** reference, many **advanced** contents to point to

Reference Upcast versus Reference Downcast

```
1  CSIEProfessor c = new CSIEProfessor();
2  Professor p = c;
3  //Professor p = new CSIEProfessor();
```

Upcast

simple (backward compatibility)

```
1  CSIEProfessor c = new CSIEProfessor();
2  Professor p = c;
3  CSIEProfessor c2 = (CSIEProfessor)p;
```

Downcast

need to check whether content fits (forward advance)

need RTTI (run-time type information/identification) to make downcast work **(where did we see it?)**

Content/Reference Polymorphism: Summary

backward compatibility handled

forward advance: only via downcast (RTTI)

simpler mechanism for forward advance?

Our Needs in Forward Advance

- new instance variables for advanced state
- new instance methods to manipulate new variables
- give new meanings to existing instance variables
- give new meanings to existing instance methods
- write an “updated but compatible” version of existing method

Method Overriding (Virtual Function)

calls the updated version through an upper-level reference

Method Invocation Polymorphism

one method (via upper-level reference),
many possible extended behaviors

Object.equals

```
1  class Coordinate extends Object{
2    double x, y;
3
4    bool equals(Object o){
5      if (o instanceof Coordinate){
6        Coordinate c = (Coordinate)o;
7        return (c.x == this.x && c.y == this.y);
8      }
9      return false;
10   }
11 }
```

Object.toString

```
1        Coordinate c = new Coordinate();
2        System.out.println(c);
```

Twist Revisited

```
1  System.out.print("abc");
2  System.out.print(3);
3  System.out.print(4.0);
4  //a twist (of course, not exactly workable
5  String("abc").print();
6  Integer(3).print();
7  Double(4.0).print();
```

System.out.print(Object) can have polymorphic behavior by internally calling the updated Object.print() (actually, Object.toString()) without overloading

V-Table: A Possible Mechanism of Method Overriding

again, not the only mechanism

RTTI revisited

- all we need is a link to the class area (stores name, vtable, etc.)
- where is the link? `java.lang.Object`
- how to access? `Object.getClass()`
- each area is an instance of `java.lang.Class`

Summary on Polymorphism

- one thing, many shapes
- important in strongly-typed platforms with inheritance
- view from content: one advanced content with many compatible access
- view from reference: one compatible reference can point to many advanced contents
- view from method: one compatible method “contract”, many different method “realization”

Abstract Class (1/3)

```
1  public class Professor(){
2    public void teach(){
3      System.out.println("not␣sure␣of␣what␣to␣teach!");
4    }
5  }
6  class CSIEProfessor extends Professor{
7    private void teach_oop(){ /* lalala */ }
8    public void teach(){ teach_oop(); }
9  }
10 class EEProfessor extends Professor{
11   private void teach_elec(){ /* lululu */ }
12   public void teach(){ teach_elec(); }
13 }
14 //in other places
15 Professor p = new Professor();
16 p.teach(); //not sure of what to teach!
```

- `teach` is a place-holder in Professor, expected to be overridden
- allows constructing a professor without any teaching ability! —absurd in some sense

Abstract Class (2/3)

```
1  public abstract class Professor(){
2    public abstract void teach();
3  }
4  class CSIEProfessor extends Professor{
5    private void teach_oop(){ /* lalala */ }
6    public void teach(){ teach_oop(); }
7  }
8  class EEProfessor extends Professor{
9    private void teach_elec(){ /* lululu */ }
10   public void teach(){ teach_elec(); }
11 }
12 //in other places
13 Professor p = new Professor(); //hahaha!!
14 Professor p = new CSIEProfessor(); //okay
```

- `teach` is a place-holder in Professor, expected to be overridden
- but cannot construct a pure Professor instance anymore!

Key Point: Abstract Class

a **contract** for future extensions

Final Words

- `static final` variable: accessed through class, and assigned once (in declaration or static constructor)
- `final` instance variable: accessed through instance, and assigned once (in declaration or every instance constructor)
- `final` instance method: cannot be overriden ($\approx$ assigned once)
- `static final` method: cannot be hidden by inheritance ($\approx$ assigned once)
- `final` class: cannot be inherited (and hence all methods final)