Polymorphism

Introduction

- Polymorphism is the third essential feature of an object-oriented programming language, after data abstraction and inheritance
- Polymorphism allows improved code organization and readability as well as the creation of extensible programs
- Learn about polymorphism (also called dynamic binding or late binding or run-time binding)

Polymorphism

- Polymorphism is the capability of a method to do different things based on the object that it is acting upon

- In other words, polymorphism allows you define one interface and have multiple implementations
  - Upcasting (Inheritance)
  - Method override (Inheritance)

Upcasting revisited

- Taking an object reference and treating it as a reference to its base type is called *upcasting*
- You can see a problem arise
- Upcasting from Wind to Instrument may "narrow" that interface

```
1  package polymorphism.music;
2
3  public enum Note {
4      MIDDLE_C, C_SHARP, B_FLAT; // Etc.
5  } ///:~
```

```
1  package polymorphism.music;
2  import static net.mindview.util.Print.*;
3
4  class Instrument {
5    public void play(Note n) {
6      print("Instrument.play()");
7    }
8  }
```

```
1  package polymorphism.music;
2
3  // Wind objects are instruments
4  // because they have the same interface:
5  public class Wind extends Instrument {
6    // Redefine interface method:
7    public void play(Note n) {
8      System.out.println("Wind.play() " + n);
9    }
10 } ///:~
```

```
1  package polymorphism.music;
2
3  public class Music {
4    public static void tune(Instrument i) {
5      // ...
6      i.play(Note.MIDDLE_C);
7    }
8    public static void main(String[] args) {
9      Wind flute = new Wind();
10     tune(flute); // Upcasting
11   }
12 }
```

Forgetting the object type

```
1  package polymorphism.music;
2  import static net.mindview.util.Print.*;
3
4  class Stringed extends Instrument {
5    public void play(Note n) {
6      print("Stringed.play() " + n);
7    }
8  }
9
10 class Brass extends Instrument {
11   public void play(Note n) {
12     print("Brass.play() " + n);
13   }
14 }
15
16 public class Music2 {
17   public static void tune(Wind i) {
18     i.play(Note.MIDDLE_C);
19   }
20   public static void tune(Stringed i) {
21     i.play(Note.MIDDLE_C);
22   }
23   public static void tune(Brass i) {
24     i.play(Note.MIDDLE_C);
25   }
26   public static void main(String[] args) {
27     Wind flute = new Wind();
28     Stringed violin = new Stringed();
29     Brass frenchHorn = new Brass();
30     tune(flute); // No upcasting
31     tune(violin);
32     tune(frenchHorn);
33   }
34 }
```

- You must write type-specific methods for each new Instrument class you add
- Write a single method that takes the base class as its argument, and not any of the specific derived classes
- That's exactly what polymorphism allows you to do

The twist

- **How can the compiler possibly know that this Instrument reference points to a Wind in this case and not a Brass or Stringed?**

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

- **The compiler can't**

- **To get a deeper understanding of the issue, it's helpful to examine the subject of *binding***

Method-call binding

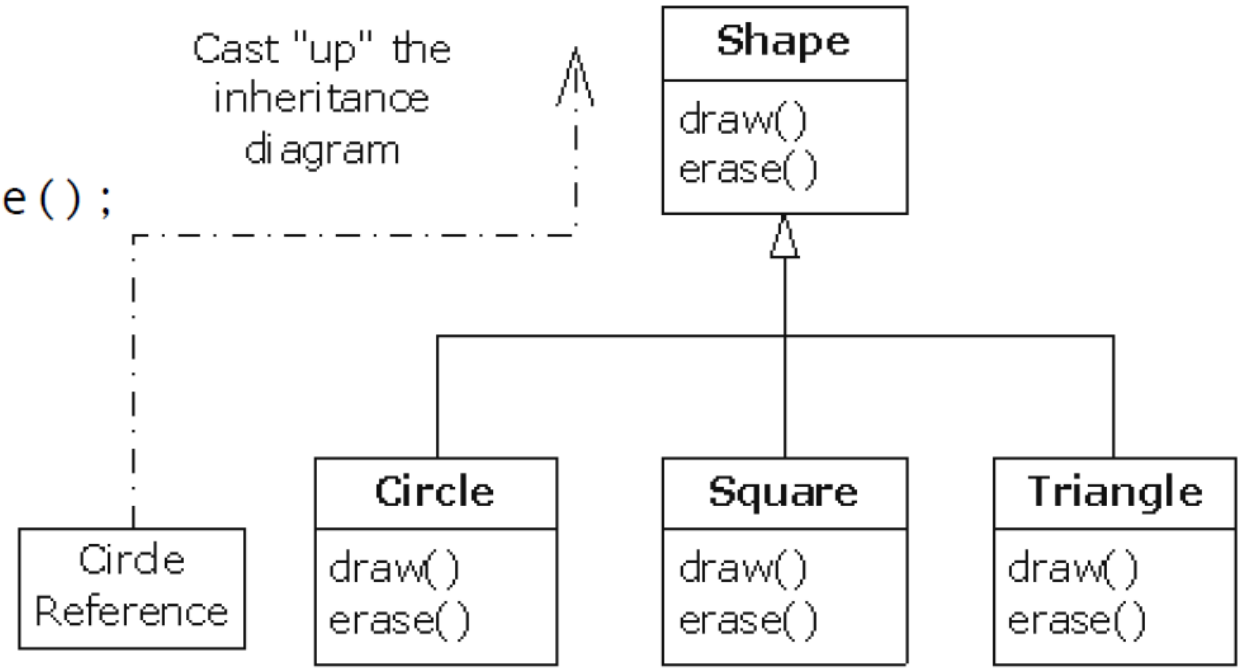
- **Connecting a method call to a method body is called *binding***
- **When binding is performed before the program is run, it's called *early binding***
- **The binding occurs at run time, based on the type of object**
  - ***Late binding* is also called *dynamic binding* or *runtime binding***
- **All method binding in Java uses late binding unless the method is *static* or *final* (*private* methods are implicitly *final*)**
- **A *final* method "turns off" dynamic binding**
  - **It tells the compiler that dynamic binding isn't necessary**

Producing the right behavior

- All method binding in Java happens polymorphically via late binding
  - You can write your code to talk to the base class and know that all the derived-class cases will work correctly using the same code
- The classic example in OOP is the “shape” example

```
Shape s = new Circle();

s.draw();
```



Producing the right behavior (Cont.)

```
1  package polymorphism.shape;
2
3  public class Shape {
4    public void draw() {}
5    public void erase() {}
6  } ///:~
```

```
1  package polymorphism.shape;
2  import static net.mindview.util.Print.*;
3
4  public class Circle extends Shape {
5    public void draw() { print("Circle.draw()"); }
6    public void erase() { print("Circle.erase()"); }
7  } ///:~
```

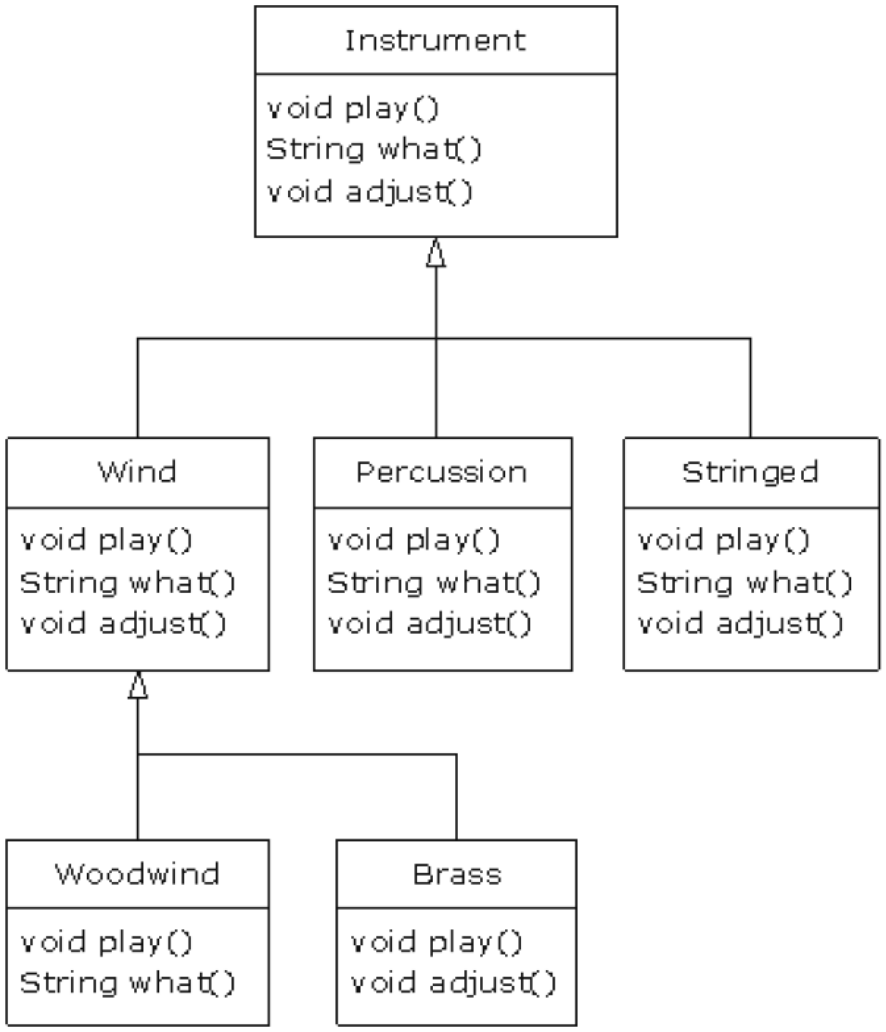
```
1  package polymorphism.shape;
2  import static net.mindview.util.Print.*;
3
4  public class Square extends Shape {
5    public void draw() { print("Square.draw()"); }
6    public void erase() { print("Square.erase()"); }
7  } ///:~
```

```
1  package polymorphism.shape;
2  import static net.mindview.util.Print.*;
3
4  public class Triangle extends Shape {
5    public void draw() { print("Triangle.draw()"); }
6    public void erase() { print("Triangle.erase()"); }
7  } ///:~
```

```
1  package polymorphism.shape;
2  import java.util.*;
3
4  public class RandomShapeGenerator {
5    private Random rand = new Random(47);
6    public Shape next() {
7      switch(rand.nextInt(3)) {
8        default:
9        case 0: return new Circle();
10       case 1: return new Square();
11       case 2: return new Triangle();
12     }
13   }
14 } ///:~
```

```
1  import polymorphism.shape.*;
2
3  public class Shapes {
4    private static RandomShapeGenerator gen =
5      new RandomShapeGenerator();
6    public static void main(String[] args) {
7      Shape[] s = new Shape[9];
8      // Fill up the array with shapes:
9      for(int i = 0; i < s.length; i++)
10       s[i] = gen.next();
11     // Make polymorphic method calls:
12     for(Shape shp : s)
13       shp.draw();
14   }
15 }
```

Extensibility



- Return to the musical instrument example
- Because of polymorphism, you can add as many new types as you want to the system without changing the *tune( )* method
- Extensible
  - Add new functionality by inheriting new data types from the common base class
  - The methods that manipulate the base-class interface will not be changed at all to accommodate the new classes

Extensibility (Cont.)

```
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
  void play(Note n) { print("Instrument.play() " + n); }
  String what() { return "Instrument"; }
  void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
  void play(Note n) { print("Wind.play() " + n); }
  String what() { return "Wind"; }
  void adjust() { print("Adjusting Wind"); }
}

class Percussion extends Instrument {
  void play(Note n) { print("Percussion.play() " + n); }
  String what() { return "Percussion"; }
  void adjust() { print("Adjusting Percussion"); }
}

class Stringed extends Instrument {
  void play(Note n) { print("Stringed.play() " + n); }
  String what() { return "Stringed"; }
  void adjust() { print("Adjusting Stringed"); }
}

class Brass extends Wind {
  void play(Note n) { print("Brass.play() " + n); }
  void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
  void play(Note n) { print("Woodwind.play() " + n); }
  String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
```

Pitfall: “overriding” private methods

- **The result of this is that only non-private methods may be overridden**
  - **You should watch out for the appearance of overriding *private* methods, which generates no compiler warnings**
  - **To be clear, you should use a different name from a *private* base-class method in your derived class**

```
1  package polymorphism;
2  import static net.mindview.util.Print.*;
3
4  public class PrivateOverride {
5    private void f() { print("private f()"); }
6    public static void main(String[] args) {
7      PrivateOverride po = new Derived();
8      po.f();
9    }
10 }
11
12 class Derived extends PrivateOverride {
13   public void f() { print("public f()"); }
14 }
```

Pitfall: fields and static methods

- **Only ordinary method calls can be polymorphic**

```
1  class Super {
2    public int field = 0;
3    public int getField() { return field; }
4  }
5
6  class Sub extends Super {
7    public int field = 1;
8    public int getField() { return field; }
9    public int getSuperField() { return super.field; }
10 }
11
12 public class FieldAccess {
13   public static void main(String[] args) {
14     Super sup = new Sub(); // Upcast
15     System.out.println("sup.field = " + sup.field +
16       ", sup.getField() = " + sup.getField());
17     Sub sub = new Sub();
18     System.out.println("sub.field = " +
19       sub.field + ", sub.getField() = " +
20       sub.getField() +
21       ", sub.getSuperField() = " +
22       sub.getSuperField());
23   }
24 }
```

Pitfall: fields and static methods (Cont.)

- If a method is ***static***, it doesn't behave polymorphically

```
1  class StaticSuper {
2    public static String staticGet() {
3      return "Base staticGet()";
4    }
5    public String dynamicGet() {
6      return "Base dynamicGet()";
7    }
8  }
9
10 class StaticSub extends StaticSuper {
11   public static String staticGet() {
12     return "Derived staticGet()";
13   }
14   public String dynamicGet() {
15     return "Derived dynamicGet()";
16   }
17 }
18
19 public class StaticPolymorphism {
20   public static void main(String[] args) {
21     StaticSuper sup = new StaticSub(); // Upcast
22     System.out.println(sup.staticGet());
23     System.out.println(sup.dynamicGet());
24   }
25 }
```

Constructors and polymorphism

- Constructors are different from other kinds of methods
- This is also true when polymorphism is involved
- Constructors are not polymorphic
  - They're actually ***static*** methods, but the ***static*** declaration is implicit
- It's important to understand the way constructors work in complex hierarchies and with polymorphism

Order of constructor calls

- A *constructor* for the base class is always called during the construction process
- Only the base-class *constructor* has the proper knowledge and access to initialize its own elements
  - It's essential that all *constructors* get called
  - Otherwise the entire object wouldn't be constructed
- The compiler will silently call the *default constructor* if you don't explicitly call a base-class *constructor* in the derived-class *constructor* body
  - If there is no default constructor, the compiler will complain

Order of constructor calls

```
1  package polymorphism;
2  import static net.mindview.util.Print.*;
3
4  class Meal {
5    Meal() { print("Meal()"); }
6  }
7
8  class Bread {
9    Bread() { print("Bread()"); }
10 }
11
12 class Cheese {
13   Cheese() { print("Cheese()"); }
14 }
15
16 class Lettuce {
17   Lettuce() { print("Lettuce()"); }
18 }
19
20 class Lunch extends Meal {
21   Lunch() { print("Lunch()"); }
22 }
23
24 class PortableLunch extends Lunch {
25   PortableLunch() { print("PortableLunch()");}
26 }
27
28 public class Sandwich extends PortableLunch {
29   private Bread b = new Bread();
30   private Cheese c = new Cheese();
31   private Lettuce l = new Lettuce();
32   public Sandwich() { print("Sandwich()"); }
33   public static void main(String[] args) {
34     new Sandwich();
35   }
36 }
```

- The order of constructor calls for a complex object is as follows:
- The base-class constructor is called
- Member initializers are called in the order of declaration
- The body of the derived-class constructor is called

Behavior of polymorphic methods inside constructors

- What happens if you're inside a constructor and you call a dynamically-bound method of the object being constructed?
- If you call a dynamically-bound method inside a constructor, the overridden definition for that method is used

```
1  import static net.mindview.util.Print.*;
2
3  class Glyph {
4    void draw() { print("Glyph.draw()"); }
5    Glyph() {
6      print("Glyph() before draw()");
7      draw();
8      print("Glyph() after draw()");
9    }
10 }
11
12 class RoundGlyph extends Glyph {
13   private int radius = 1;
14   RoundGlyph(int r) {
15     radius = r;
16     print("RoundGlyph.RoundGlyph(), radius = " + radius);
17   }
18   void draw() {
19     print("RoundGlyph.draw(), radius = " + radius);
20   }
21 }
22
23 public class PolyConstructors {
24   public static void main(String[] args) {
25     new RoundGlyph(5);
26   }
27 }
```

Behavior of polymorphic methods inside constructors (Cont.)

- The actual process of initialization is:
- The storage allocated for the object is initialized to binary zero before anything else happens
- The base-class constructors are called
- Member initializers are called in the order of declaration
- The body of the derived-class constructor is called

Behavior of polymorphic methods inside constructors (Cont.)

- **A good guideline for constructors**
  - **"Do as little as possible to set the object into a good state, and if you can possibly avoid it, don't call any other methods in this class."**
- **The only safe methods to call inside a constructor are those that are *final* in the base class**
  - **This also applies to *private* methods, which are automatically *final***
  - **These cannot be overridden and thus cannot produce this kind of surprise**

Covariant return types

- Java SE5 adds covariant return types
- An overridden method in a derived class can return a type derived from the type returned by the base-class method

```
1  class Grain {
2    public String toString() { return "Grain"; }
3  }
4
5  class Wheat extends Grain {
6    public String toString() { return "Wheat"; }
7  }
8
9  class Mill {
10   Grain process() { return new Grain(); }
11 }
12
13 class WheatMill extends Mill {
14   Wheat process() { return new Wheat(); }
15 }
16
17 public class CovariantReturn {
18   public static void main(String[] args) {
19     Mill m = new Mill();
20     Grain g = m.process();
21     System.out.println(g);
22     m = new WheatMill();
23     g = m.process();
24     System.out.println(g);
25   }
26 }
```

Designing with inheritance

- Once you learn about polymorphism, it can seem that everything ought to be inherited, because polymorphism is such a clever tool
- If you choose inheritance first when you're using an existing class to make a new class, things can become needlessly complicated
- A better approach is to choose composition first
  - Composition does not force a design into an inheritance hierarchy
  - Composition is also more flexible since it's possible to dynamically choose a type
  - Inheritance requires an exact type to be known at compile time

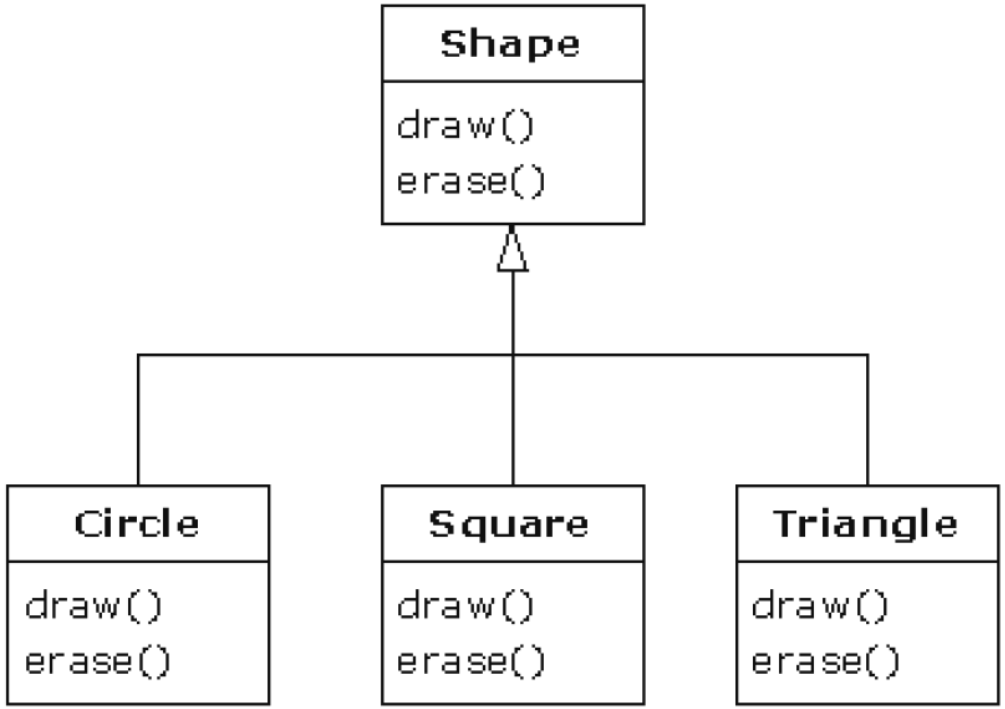
Designing with inheritance

```
1  import static net.mindview.util.Print.*;
2
3  class Actor {
4    public void act() {}
5  }
6
7  class HappyActor extends Actor {
8    public void act() { print("HappyActor"); }
9  }
10
11 class SadActor extends Actor {
12   public void act() { print("SadActor"); }
13 }
14
15 class Stage {
16   private Actor actor = new HappyActor();
17   public void change() { actor = new SadActor(); }
18   public void performPlay() { actor.act(); }
19 }
20
21 public class Transmogrify {
22   public static void main(String[] args) {
23     Stage stage = new Stage();
24     stage.performPlay();
25     stage.change();
26     stage.performPlay();
27   }
28 }
```

- **You gain dynamic flexibility at run time**
  - **This is also called the State Pattern**
  - **A general guideline is "Use inheritance to express differences in behavior, and fields to express variations in state."**
  - **In this case, that change in state happens to produce a change in behavior**

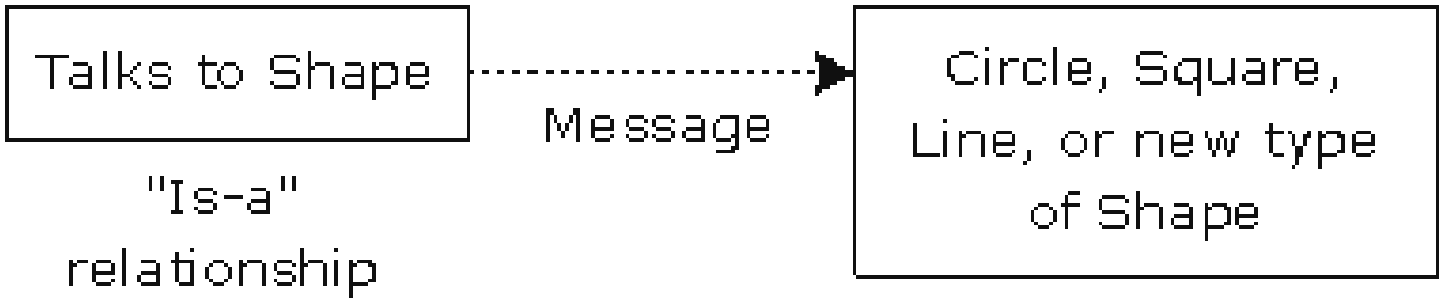
Substitution vs. extension

- The cleanest way to create an inheritance hierarchy is to take the "pure" approach
  - Only methods that have been established in the base class are overridden in the derived class
  - Inheritance guarantees that any derived class will have the interface of the base class and nothing less



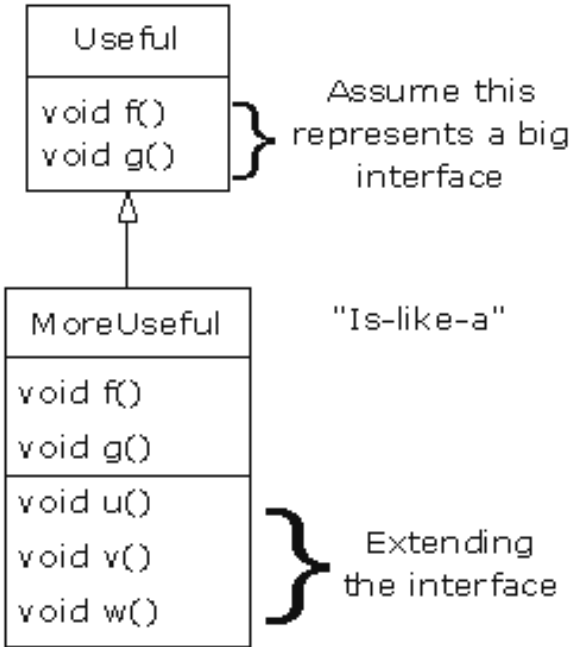
Substitution vs. extension (Cont.)

- **This can be thought of as *pure substitution***
  - **Derived class objects can be perfectly substituted for the base class, and you never need to know any extra information about the subclasses when you're using them**
  - **The base class can receive any message you can send to the derived class**
  - **Everything is handled through polymorphism**



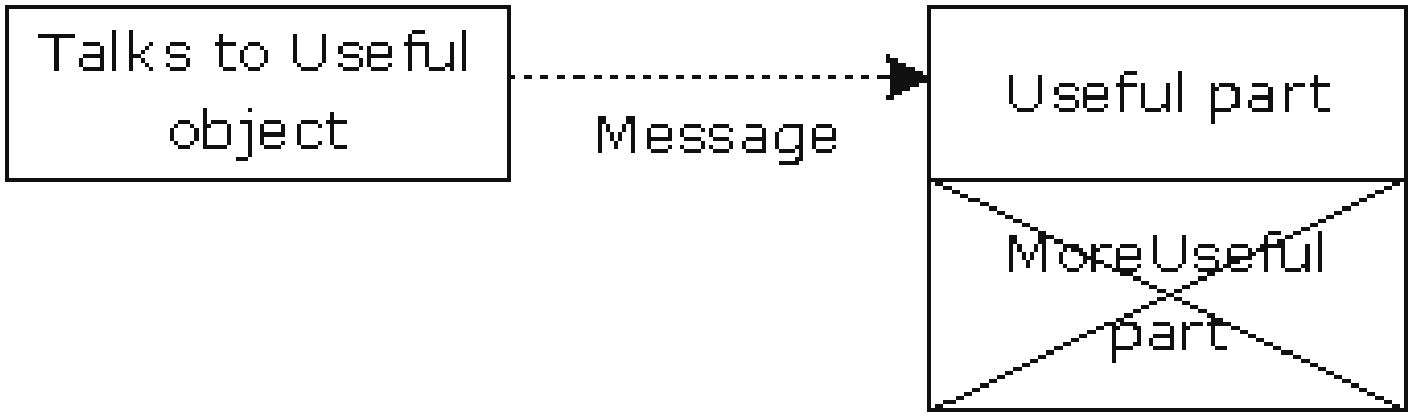
Substitution vs. extension (Cont.)

- It seems like a pure *is-a* relationship is the only sensible way to do things
- Extending the interface is the perfect solution to a particular problem
  - This can be termed an "*is-like-a*" relationship
  - It has the same fundamental interface
  - It has other features that require additional methods to implement



Substitution vs. extension (Cont.)

- ***Extension*** **has a drawback**
  - **The extended part of the interface in the derived class is not available from the base class**
  - **Once you upcast, you can't call the new methods**



Downcasting and runtime type information

- Lose the specific type information via an *upcast*
- An upcast is always safe
  - Because the base class cannot have a bigger interface than the derived class
  - Every message you send through the base class interface is guaranteed to be accepted
- With a downcast, you don't really know that a *shape* (for example) is actually a *circle*
  - To solve this problem, there must be some way to guarantee that a downcast is correct
- In Java, *every* cast is checked
  - At run time the cast is checked
  - If it isn't, you get a ClassCastException
  - *runtime type identification* (RTTI)

Downcasting and runtime type information (Cont.)

```
1  class Useful {
2    public void f() {}
3    public void g() {}
4  }
5
6  class MoreUseful extends Useful {
7    public void f() {}
8    public void g() {}
9    public void u() {}
10   public void v() {}
11   public void w() {}
12 }
13
14 public class RTTI {
15   public static void main(String[] args) {
16     Useful[] x = {
17       new Useful(),
18       new MoreUseful()
19     };
20     x[0].f();
21     x[1].g();
22     // Compile time: method not found in Useful:
23     //! x[1].u();
24     ((MoreUseful)x[1]).u(); // Downcast/RTTI
25     ((MoreUseful)x[0]).u(); // Exception thrown
26   }
27 } ///:~
```

Thank you

zhenling@seu.edu.cn