CS 112 Introduction to Programming

(Spring 2012)

Lecture #33: Polymorphism and Interface

Zhong Shao

Department of Computer Science
Yale University
Office: 314 Watson

http://flint.cs.yale.edu/cs112

Acknowledgements: some slides used in this class are taken directly or adapted from those accompanying the two textbooks: Introduction to Programming in Java: An Interdisciplinary Approach by Robert Sedgewick and Kevin Wayne and Building Java Programs: A Back to Basics Approach by Stuart Reges and Marty Stepp
What is Polymorphism?

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.

  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.
Recap: Reference Variables

- Interaction with an object occurs through object reference variables

- An object reference variable holds the reference (address, the location) of an object

```java
ChessPiece bishop1 = new ChessPiece();
```

![Diagram of bishop1 reference to an object]
Recap: Object Reference Variable

- **Object reference variable assignment:**

```java
bishop2 = bishop1;
```

**Before**

```
bishop1
  └──► bishop
```

**After**

```
bishop1
  └──► bishop
```

```
bishop2
  └──► bishop
```

```
bishop2
  └──► bishop
```
Object References

- An object may be referenced by multiple variables; these are aliases of each other.

- An object may be referenced by no variable, e.g.,

  
  ```java
  Employee alice = new Employee("Alice");
  alice = null;
  ```

- When an object has no reference, it is called garbage in Java, and will be garbage collected
  - garbage collection avoids running out of memory because unused memory has not been reclaimed
  - garbage collection is done automatically, and we have no control
Polymorphism Reference

- Polymorphism is implemented by polymorphic references, which are variables that can refer to different types of objects.

- It is the type of the object being referenced, not the reference type, that determines which method is invoked.
  - Polymorphic references are therefore resolved at run-time, not during compilation; this is called dynamic binding.

- Careful use of polymorphic references can lead to elegant, robust software designs.
Polymorphic Reference through Inheritance

- A common way to implement polymorphic reference is through inheritance:
  - A variable of type $T$ can hold an object of class $T$ or descendent of $T$.
    
    ```java
    Employee emp = new Lawyer("Larry");
    emp = LegalSecretary("Lisa");
    ```

- You can call any methods defined in the Employee class on `emp`.
- When you invoke a method defined in Employee on an Lawyer object, the behavior is that of the object type (Lawyer), not the variable type (Employee).
Polymorphic Reference through Inheritance

Employee ed

Reference variable type

ed.vacationDays()
// 15

ed.vacationDays()
// 10

Object type: Lawyer

Object type: Secretary
Polymorphic Reference: Example

Employee ed = new Lawyer("Larry");
System.out.println ( ed.vacationDays() );
// OUTPUT: 15
System.out.println ( ed.vacationForm() );
// OUTPUT: pink

ed = new LegalSecretary("Lisa");
System.out.println ( ed.vacationDays() );
// OUTPUT: 10
System.out.println ( ed.vacationForm() );
// OUTPUT: yellow
Comment: Variable Type and Method

Through a given type of reference variable, we can invoke only the methods defined in that type.

```java
class Employee{
    public double pay()
    {
    }
}

class Lawyer extends Employee {
    public void sue()
    {
    }

Employee ed = new Lawyer(“Larry”);

Can we do the following statements:
    ed.pay();
    ed.sue();
```
Comment: Variable Type and Method

- We can “promote” an object back to its original type through an explicit narrowing cast:

```java
Employee ed = new Lawyer("Larry");
Lawyer larry = (Lawyer)ed;
larry.sue();
```

If the type of object referred to by ed is not Lawyer, program error.
Polymorphic Parameters

- Define a method that takes a reference to a base type and apply to all derived types.

- This is how `print` in `PrintStream` is defined:
  ```java
  void print(Object obj) {
    // all objects have the `toString()` method
    // convert to string and then output
  }
  ```
Polymorphic Parameters: Example

```java
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer();
        Secretary steve = new Secretary();
        printInfo(lisa);
        printInfo(steve);
    }

    public static void printInfo(Employee empl) {
        System.out.println("salary: "+ empl.pay());
        System.out.println("v.days: "+ empl.vacationDays());
        System.out.println("v.form: "+ empl.vacationForm());
    }
}

OUTPUT:
salary: 50000.0
v.days: 15
v.form: pink
salary: 50000.0
v.days: 10
v.form: yellow
```
Polymorphism and Arrays

- A common usage of polymorphism is to define an array of a base type, but different entries refer to different types of objects
  - To handle a heterogeneous population of objects (e.g., critters) with uniformity
Polymorphism and Arrays: Example

```java
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[4];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
    }

    public void payday() {
        for (int count = 0; count < staffList.length; count++) {
            System.out.printf("%-10s:", staffList[count].name());
            System.out.printf("$%.2f\n", staffList[count].pay());
            System.out.println("-----------------------------------");
        }
    }
}
```
Extending the Program: Hourly

- Include a new type of secretary who are paid by hours.
Add a new Type of Employee: Hourly

```java
public class Hourly extends Secretary {
    private double payRate;

    public Hourly(String name, double payRate)
    {
        super(name);
        this.payRate = payRate;
    }

    public double pay() {
        return hours() * payRate;
    }
}
```
Polymorphism and Arrays: Example

```java
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[5];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
        staffList[4] = new Hourly("Holly");
    }

    public void payday() {
        for (int count = 0; count < staffList.length; count++) {
            System.out.printf("%-10s:", staffList[count].name());
            System.out.printf("$%.2f\n", staffList[count].pay());
        }
    }
}
```
The pay-roll of a firm
Exercise: A Polymorphism Problem

- Suppose that the following four classes have been declared:

```java
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }
    public void method2() {
        System.out.println("foo 2");
    }
    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}

public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }
    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```
A Polymorphism Problem

What would be the output of the following client code?

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};

for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```
### Finding output with tables

<table>
<thead>
<tr>
<th>method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td></td>
<td>baz 1</td>
<td></td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td></td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td></td>
<td>baz</td>
<td></td>
</tr>
</tbody>
</table>
## Finding output with tables

<table>
<thead>
<tr>
<th>method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td></td>
<td>baz 1</td>
<td></td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td></td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td></td>
<td>baz</td>
<td></td>
</tr>
</tbody>
</table>
# Finding output with tables

<table>
<thead>
<tr>
<th>method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td>foo 1</td>
<td>baz 1</td>
<td>baz 1</td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td>foo 2</td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td>foo</td>
<td>baz</td>
<td>baz</td>
</tr>
</tbody>
</table>
Use Diagramming

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.
Polymorphism Answer

Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}

- Output:
  baz
  baz 1
  foo 2
  foo
  foo 1
  bar 2
  baz
  baz 1
  mumble 2
  foo
  foo 1
  foo 2
Exercise 2:

- The methods sometimes call other methods (tricky!).

```java
class Ham {
  public void a() {
    System.out.print("Ham a   ");
    b(); // whose b()? Ham's?
  }
  public void b() {
    System.out.print("Ham b   ");
  }
  public String toString() {
    return "Ham";
  }
}

public class Lamb extends Ham {
  public void b() {
    System.out.print("Lamb b   ");
  }
}
```
public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b ");
    }
}
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a ");
        super.a();
    }
    public String toString() {
        return "Yam";
    }
}

What would be the output of the following client code?

Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println(); // to end the line of output
    food[i].b();
    System.out.println(); // to end the line of output
}
Class Diagram

- Ham
  - a0
  - b0
  - toString()

- Lamb
  - a0
  - b0
  - toString()

- Yam
  - a0
  - b0
  - toString()

- Spam
  - a0
  - b0
  - toString()
<table>
<thead>
<tr>
<th>method</th>
<th>Ham</th>
<th>Lamb</th>
<th>Yam</th>
<th>Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Ham a</td>
<td></td>
<td>Yam a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b()</td>
<td></td>
<td>super.a()</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Ham b</td>
<td>Lamb b</td>
<td></td>
<td>Spam b</td>
</tr>
<tr>
<td>toString</td>
<td>Ham</td>
<td></td>
<td>Yam</td>
<td></td>
</tr>
<tr>
<td>method</td>
<td>Ham</td>
<td>Lamb</td>
<td>Yam</td>
<td>Spam</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
<td>-------</td>
</tr>
</tbody>
</table>
| a      | Ham a  
b() |     | Yam a super.a() |       |
<p>| b      | Ham b | Lamb b |       | Spam b |
| toString | Ham       |     | Yam       |       |</p>
<table>
<thead>
<tr>
<th>method</th>
<th>Ham</th>
<th>Lamb</th>
<th>Yam</th>
<th>Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Ham a</td>
<td>Ham a</td>
<td>Yam a</td>
<td>Yam a</td>
</tr>
<tr>
<td>b()</td>
<td>b()</td>
<td>b()</td>
<td>b()</td>
<td>b()</td>
</tr>
<tr>
<td>b</td>
<td>Ham b</td>
<td>Lamb b</td>
<td>Lamb b</td>
<td>Spam b</td>
</tr>
<tr>
<td>toString</td>
<td>Ham</td>
<td>Ham</td>
<td>Yam</td>
<td>Yam</td>
</tr>
</tbody>
</table>
Ham[ ] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
An *interface* provides an abstraction to write more reusable programs

Instead of writing a program for a single class of objects, we want to write a program to handle all classes with a given set of behaviors

- An interface is an abstraction for the common behaviors of these behaviors

Often interface represents abstract concepts
Interface

interface: A list of methods that classes can promise to implement.

- Analogous to non-programming idea of roles or certifications
  - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits, and do management consulting."
Inheritance and Interface

- **Inheritance** gives you an is-a relationship and code-sharing.
  - A Lawyer object can be treated as an Employee, and Lawyer inherits Employee's code.

- **Interfaces** give you an is-a relationship without code sharing.
Interface Syntax

- An *interface* is a collection of *constants* and *abstract methods*.
  - Abstract method: a method header without a method body; we declare an abstract method using the modifier `abstract`.
  - Since all methods in an interface are abstract, the `abstract` modifier is usually left off.
public interface Movable {
  public double getSpeed();
  public void setSpeed(double speed);
  public void setDirection(int direction);
  public int getDirection();
}

This interface describes the behaviors common to all movable things.
(Every Movable thing should have these methods.)

A semicolon follows each method header immediately

No method in an interface has a definition (body)
Implementing an interface

- A class can declare that it *implements* an interface.
  - This means the class contains an implementation for each of the abstract methods in that interface.
    (Otherwise, the class will fail to compile.)

- Implementing an interface, general syntax:
  
  ```java
  public class <name> implements <interface names> {
    ...
  }
  ```

- Example:
  ```java
  public class Bicycle implements Movable {
    ...
  }
  ```

  (What must be true about the Bicycle class for it to compile?)
Interface Implementation

- If we write a class that claims to be a `Movable` but doesn't implement all of the methods defined in the interface, it will not compile.

  Example:
  ```java
  public class Bicycle implements Movable {
  }
  ```

  The compiler error message:
  ```java
  Bicycle.java:1: Bicycle is not abstract and does not override abstract method getSpeed() in Movable
  ```
Example: Shape interface

- An interface for shapes:
  
  ```java
  public interface Shape {
    public double area();
    public double perimeter();
  }
  ```

- This interface describes the common features that all shapes should have in your design. (Every shape has an area and perimeter.)
Example: Circle class

// Represents circles.
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of this circle.
    public double area() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of this circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}

Example: Rectangle class

// Represents rectangles.
public class Rectangle implements Shape {
    private double width;
    private double height;

    // Constructs a new rectangle with the given dimensions.
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    // Returns the area of this rectangle.
    public double area() {
        return width * height;
    }

    // Returns the perimeter of this rectangle.
    public double perimeter() {
        return 2.0 * (width + height);
    }
}
Example: Triangle class

// Represents triangles.
public class Triangle implements Shape {
    private double a;
    private double b;
    private double c;

    // Constructs a new Triangle given side lengths.
    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // Returns this triangle's area using Heron's formula.
    public double area() {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    // Returns the perimeter of this triangle.
    public double perimeter() {
        return a + b + c;
    }
}
Diagrams of Interfaces

```
<<interface>>
Shape

area()
perimeter()

Circle
radius
Circle(radius)
area()
perimeter()

Rectangle
width, height
Rectangle(w,h)
area()
perimeter()

Triangle
a, b, c
Triangle(a, b, c)
area()
perimeter()
```
Polymorphic Reference through Interface

- A variable of interface type $T$ can hold an object of any class implementing $T$.
  
  ```java
  Movable mobj = new Bicycle();
  ```

  - You can call any methods defined in the `Movable` interface on `mobj`.

  - When you invoke a method through the interface variable, the behavior is that of the object type.
Interface Polymorphism: Example

```java
public static void printInfo(Shape s) {
    System.out.println("area : "+ s.area());
    System.out.println("perim: "+ s.perimeter());
    System.out.println();
}

- Any object that implements the interface may be passed as the parameter to the above method.
  Circle circ = new Circle(12.0);
  Triangle tri = new Triangle(5, 12, 13);
  printInfo(circ);
  printInfo(tri);
```
Interface Polymorphism: Example

- We can create an array of an interface type, and store any object implementing that interface as an element.

```java
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);

Shape[] shapes = {circ, tri, rect};
for (int i = 0; i < shapes.length; i++) {
    printInfo(shapes[i]);
}
```

- Each element of the array executes the appropriate behavior for its object when it is passed to the `printInfo` method.
Using Interface for General Programming

- When implementing a class or method (e.g., sorting), think about the essence of the properties/behaviors of the objects you require.

- Define those properties in an interface.

- Implement the class/method for the interface only.