Object Oriented Paradigms

College Requirements -Compulsive Courses

CSCR2105

Polymorphism and Interfaces

Lecture 3.2

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.

Coding with polymorphism

- A variable of type *T* can hold an object of any subclass of *T*.

```
Employee ed = new Lawyer();
```

  - You can call any methods from the `Employee` class on `ed`.

- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary()); // 50000.0
System.out.println(ed.getVacationForm());//pink
```

Polymorphism and parameters

- You can pass any subtype of a parameter's type.

```
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.getSalary());
  System.out.println("v.days:"+empl.getVacationDays());
  System.out.println("v.form:"+empl.getForm());
  System.out.println();
    }
}
```

OUTPUT:

```
salary: 50000.0                salary: 40000.0
v.days: 15                     v.days: 10
v.form: pink                   v.form: yellow
```

Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public class EmployeeMain2 {
   public static void main(String[] args) {
      Employee[] e = {  new Lawyer(),
                        new Secretary(),
                        new Marketer(),
                        new LegalSecretary()
                      };
     for (int i = 0; i < e.length; i++) {
        System.out.println("salary:"+e[i].getSalary());
        System.out.println("v.days:"+e[i].getVacationDays());
          System.out.println();
       }
     }
}
```

Output:

```
salary: 50000.0                salary: 50000.0
v.days: 15                      v.days: 10


salary: 60000.0             salary: 55000.0
v.days: 10                  v.days: 10
```

Polymorphism problems

- 4-5 classes with inheritance relationships are shown.

- A client program calls methods on objects of each class.

- You must read the code and determine the client's output.

- We always put such a question on our final exams!

A polymorphism problem

Suppose that the following four classes have been declared:

```
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");
    }
}
```

A polymorphism problem

```
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");
      }
  }
```

What would be the output of the following client code?

```
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

method	Foo	Bar	Baz	Mumble
method1	foo 1	*foo 1*	baz 1	*baz 1*
method2	foo 2	bar 2	*foo 2*	mumble 2
toString	foo	*foo*	baz	*baz*

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
```

Another problem

- The order of the classes is jumbled up.
- The methods sometimes call other methods (tricky!).

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

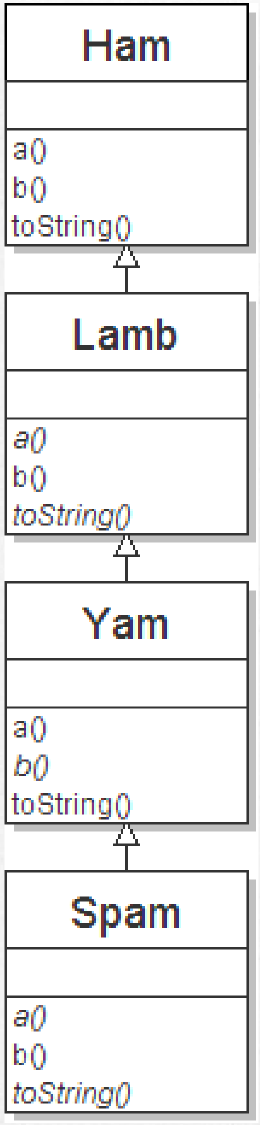
Another problem 2

```
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();
     food[i].b();
```

Class diagram



Ham

a()
b()
toString()

Lamb

a()
b()
toString()

Yam

a()
b()
toString()

Spam

a()
b()
toString()

Polymorphism at work

- Lamb inherits Ham's a. a calls b. But Lamb overrides b...

```
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    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   ");
    }
}
```

- Lamb's output from a:

```
Ham a   Lamb b
```

The table

method	Ham	Lamb	Yam	Spam
a	Ham a **b()**	*Ham a* ***b()***	Yam a Ham a **b()**	*Yam a* *Ham a* ***b()***
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	*Ham*	Yam	*Yam*

The answer

```
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();
}
```

- Output:

```
Ham
Ham a   Lamb b
Lamb b

Ham
Ham a   Ham b
Ham b

Yam
Yam a   Ham a   Spam b
Spam b

Yam
Yam a   Ham a   Lamb b
Lamb b
```

Casting references

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();
int hours = ed.getHours();   // ok; it's in
Employee
ed.sue();                    // compiler error
```

  - The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue` .

- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();                  // ok
((Lawyer) ed).sue();         // shorter version
```

More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");      // ok
((LegalSecretary) eric).fileLegalBriefs();//
exception

// (Secretary object doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");    // error
```

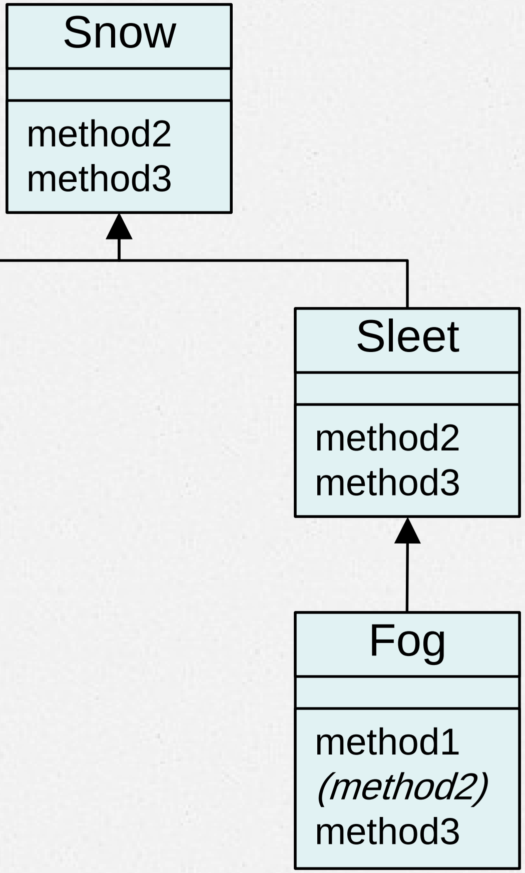
- Casting doesn't actually change the object's behavior.

  It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()    // pink
(Lawyer's)
```

Technique 1: diagram

- Diagram the classes from top (superclass) to bottom.



Technique 2: table

method	Snow	Rain	Sleet	Fog
method1		Rain 1		Fog 1
method2	Snow 2	Rain 2	Sleet 2 Snow 2 **method3()**	*Sleet 2* *Snow 2* ***method3()***
method3	Snow 3	*Snow 3*	Sleet 3	Fog 3

Italic - inherited behavior
Bold - dynamic method call

Interfaces

Lecture 4.2

Relatedness of types

Write a set of `Circle`, `Rectangle`, and `Triangle` classes.

- Certain operations that are common to all shapes.

  perimeter - distance around the outside of the shape

  area - amount of 2D space occupied by the shape

- Every shape has them but computes them differently.

Shape area, perimeter

- Rectangle (as defined by width w and height h):

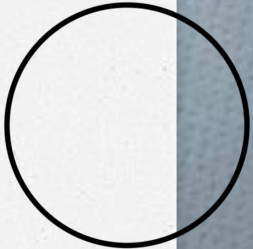
  area $= w\,h$

  perimeter $= 2w + 2h$

- Circle (as defined by radius r):

  area $= \pi\, r^2$

  perimeter $= 2\,\pi\, r$

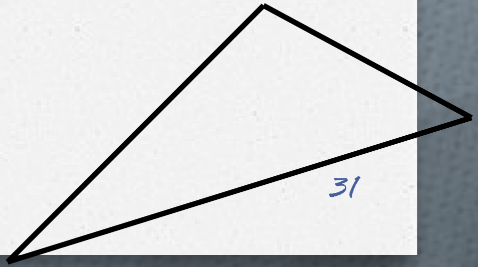


- Triangle (as defined by side lengths a, b, and c)

  area $= \sqrt{(s\,(s - a)\,(s - b)\,(s - c))}$

  where $s = ½\,(a + b + c)$

  perimeter $= a + b + c$



Common behavior

- Write shape classes with methods `perimeter` and `area`.

- We'd like to be able to write client code that treats different kinds of shape objects in the same way, such as:
  - Write a method that prints any shape's area and perimeter.
  - Create an array of shapes that could hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other shape we've written.
  - Make a `DrawingPanel` display many shapes on screen.

Interfaces

- **interface:** A list of methods that a class can implement.
  - 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.
    - A `Rectangle` object can be treated as a `Shape`.
      - "I'm certified as a Shape. That means I know how to compute my area and perimeter."

Declaring an interface

```
public interface name {
     public returntype Methodname(type name, ..., type name);
     public returntype Methodname(type name, ..., type name);
     ...
}
```

Example:

```
public interface car{
     public double speed();
     public void setDirection(int direction);
}
```

- **abstract method:** A header without an implementation.
  - The actual body is not specified, to allow/force different classes to implement the behavior in its own way.

Shape interface

```
public interface Shape {
    public double area();
    public double perimeter();
}
```

- This interface describes the features common to all shapes.
  (Every shape has an area and perimeter.)

Implementing an interface

```
public class name implements interface {
    ...
}
```

- Example:

  ```
  public class Bicycle implements car{
      ...
  }
  ```

- A class can declare that it *implements* an interface.
  - This means the class must contain each of the abstract methods in that interface. (Otherwise, it will not compile.)

Interface requirements

- If a class claims to be a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.
  - Example:

    ```
    public class Banana implements Shape {
        ...
    }
    ```

  - The compiler error message:

    ```
    Banana.java:1: Banana is not abstract and
    does not override abstract method area() in
    Shape
    public class Banana implements Shape {
                ^
    ```

Complete 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;
    }
}
```

Complete 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);
    }
}
```

Complete 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
    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;
    }
}
```

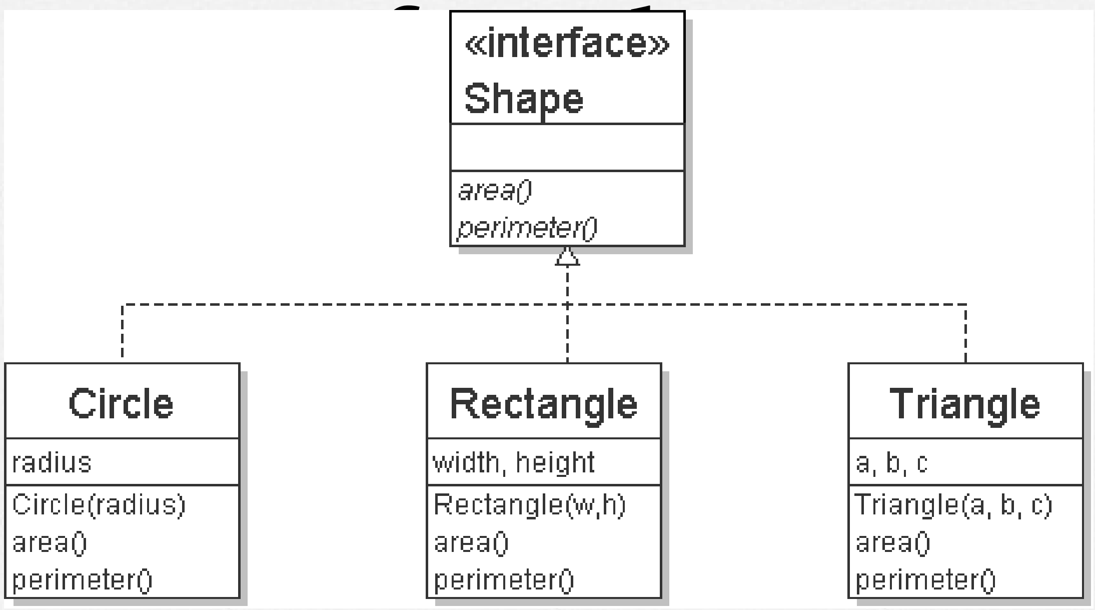
Interfaces + polymorphism

- Interfaces don't benefit the class so much as the *client.*
  - Interface's is-a relationship lets the client use polymorphism.

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

  - Any object that implements the interface may be passed.

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

Shape[] shapes = {tri, circ, rect};
```



- Arrow goes up from class to interface(s) it implements.
  - There is a supertype-subtype relationship here; e.g., all Circles are Shapes, but not all Shapes are Circles.
  - This kind of picture is also called a *UML class diagram*.

Waiting for your questions and comments

Lecture 4

Object-Oriented Programming: Exception handling