Topic 32 - Polymorphism
You’re on a team developing a drawing program

- **Shape class**
  - Ellipse class
    - Circle class
  - Rectangle class
- **Arrow class**
- Want to be able to print a description of any drawable object: shape & arrow alike
public class Shape {
    public double area() {
        return 0.0;
    }

    public void describeArea() {
        System.out.println("The area of this shape is "+area()+");
    }
}
Rectangle.java

```java
public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() {
        return width * height;
    }
}
```
public class Ellipse extends Shape {
    private double semiMajorAxis;
    private double semiMinorAxis;

    public Ellipse(double semiMajorAxis, double semiMinorAxis) {
        this.semiMajorAxis = semiMajorAxis;
        this.semiMinorAxis = semiMinorAxis;
    }

    public double area() {
        return Math.PI * semiMajorAxis * semiMinorAxis;
    }
}
public class Circle extends Ellipse {
    public Circle(double radius) {
        super(radius, radius);
    }
}

Circle.java
public class ShapeCreator {
    public static void main(String[] args) { 
        Shape[] shapes = {new Rectangle(3, 4), new Circle(3), 
            new Ellipse(3, 4)};

        for (int i = 0; i < shapes.length; i++) {
            shapes[i].describeArea();
        }
    }
}

describeArea() is a method implemented in the Shape class, but it calls the area() method implemented in the derived class (Rectangle or Ellipse classes depending on the type of Shape).
But here’s a problem:

What if someone else comes in and implements a Triangle class that extends Shape, but forgets to implement the area() method inside the Triangle class?
Triangle.java

public class Triangle extends Shape {
    private double base;
    private double height;

    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }
}

What would be the output of the following code?

Triangle triangle = new Triangle(3, 4);
System.out.println(triangle.area());

A. Syntax error
B. Runtime error
C. 0
D. 6
E. 12
Shape.java

public interface Shape {
    public double area();
}

An interface is nothing but a list of methods, but it acts like another “type.”

If a class is defined as “implements Shape”, then it is forced to implement its own area() method.

Every class that “implements Shape” is now a data type that contains a public void area() method.
public class Ellipse implements Shape {
    private double semiMajorAxis;
    private double semiMinorAxis;

    public Ellipse(double semiMajorAxis, int semiMinorAxis) {
        this.semiMajorAxis = semiMajorAxis;
        this.semiMinorAxis = semiMinorAxis;
    }
}

Code won’t compile unless you implement the area() method from the Shape interface!
public class Ellipse implements Shape {
    private double semiMajorAxis;
    private double semiMinorAxis;

    public Ellipse(double semiMajorAxis, int semiMinorAxis) {
        this.semiMajorAxis = semiMajorAxis;
        this.semiMinorAxis = semiMinorAxis;
    }

    public double area() {
        return Math.PI * semiMajorAxis * semiMinorAxis;
    }
}
A class can “implement” many interfaces

```java
public interface Describable {
    public void describe();
}
```
public class Ellipse implements Shape, Describable {
    private double semiMajorAxis;
    private double semiMinorAxis;

    public Ellipse(double semiMajorAxis, int semiMinorAxis) {
        this.semiMajorAxis = semiMajorAxis;
        this.semiMinorAxis = semiMinorAxis;
    }

    public double area() {
        return Math.PI * semiMajorAxis * semiMinorAxis;
    }

    public void describe() {
        System.out.print("This is an ellipse with semimajor axis ");
        System.out.print(semiMajorAxis);
        System.out.print("", semiMinorAxis);
        System.out.print("", semiminor axis ");
        System.out.print(semiMinorAxis);
        System.out.print("", and area ");
        System.out.print(area());
        System.out.println(".");
    }
}

public class Arrow implements Describable {
    public void describe() {
        System.out.println("This is a line with a V at the end.");
    }
}
public class ShapeCreator {  
    public static void main(String[] args) {  
        Describable[] stuff = {new Ellipse(3, 4), new Arrow()};  
        for (int i = 0; i < stuff.length; i++) {  
            stuff[i].describe();  
        }  
    }  
}  

Output:  
This is an ellipse with semimajor axis 4.0, semiminor axis 3.0, and area 37.69911184307752.  
This is a line with a V at the end.
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.
Coding with polymorphism

- A variable of type $T$ can refer to an object of any subclass of $T$.

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

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

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

- When a method is called on ed, it behaves as a Lawyer.
Polymorphism and parameters

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

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

**OUTPUT:**

```
salary: 50000.0
v.days: 15
v.form: pink
```
Polymorphism and arrays

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

```java
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
v.days: 15
salary: 50000.0
v.days: 10
salary: 60000.0
v.days: 10
salary: 55000.0
v.days: 10
```
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");
    }
}

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();
}
Diagramming the classes

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

```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();
}
```

- **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!).

```java
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";
    }
}
```
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].toString());
    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
Polymorphism at work

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

```java
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

<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 b()</td>
<td>Ham a b()</td>
<td>Yam a b()</td>
<td>Yam a 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>
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
Overriding Object's equals Method

- The Object class contains this method:
  ```java
  public boolean equals(Object obj)
  ```

- many classes override this method
- many students mistakenly overload the method
- many headaches when placing objects in data structures
Overriding Object's equals Method

• overriding equals correctly follows a pattern

• So, it isn't that hard, if you follow the pattern

• Override equals for a Standard Playing Card