Topic 32 - Polymorphism
Clicker 1

What is output by the following code?
Critter c1 = new Hippo(7);
System.out.print(c1.toString());

A. 7
B. ?
C. null
D. No output due to a syntax error
E. No output due to a runtime error
Polymorphism

- **polymorphism**: Ability for the same method to be called 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 type \( T \) and any descendants of \( T \).

```java
Critter c1 = new Hippo(7);
```

- You can call any methods from the Critter class on \( c1 \).

- When a method is called on \( c1 \), it behaves as a Hippo.
  ```java
  System.out.println(c1.getColor()); // GRAY
  System.out.println(c1.toString()); // 7
  ```
Polymorphism and parameters

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

```java
public class CriiterMain {
    public static void main(String[] args) {
        Hippo henry = new Hippo(7);
        Bird angry = new Bird();
        printInfo(henry);
        printInfo(angry);
    }

    public static void printInfo(Critter crit) {
        System.out.println(" eat?: " + crit.eat());
        System.out.println(" fight: " + crit.fight("?"));
        System.out.println(" move: " + crit.getMove());
        System.out.println();
    }
}
```

OUTPUT???
Arrays of superclass types can store any subtype as elements.

```java
public class CritterMain2 {
    public static void main(String[] args) {
        Critter[] crits = { new Bird(),
                           new Vulture(),
                           new Hippo(7),
                           new Ant(true) };

        for (Critter crit : crits) {
            System.out.println(" color: "+ crit.getColor());
            System.out.println(" move: "+ crit.getMove());
            System.out.println();
        }
    }
}
```

Output:
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 String toString() {
        return "baz";
    }
}

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

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>method2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toString</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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[] foos = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < foos.length; i++) {
    System.out.println(foos[i]);
    foos[i].method1();
    foos[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?

```java
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
- toString0

Lamb
- a0
- b0
- toString0

Yam
- a0
- b0
- toString0

Spam
- a0
- b0
- toString0
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</td>
<td>Ham a</td>
<td>Yam a</td>
<td>Yam a</td>
</tr>
<tr>
<td></td>
<td>b()</td>
<td>b()</td>
<td>Ham a</td>
<td>Ham a</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) {
    return this == 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
- Override equals for a Snake Critter
  - Demo array of Critter objects