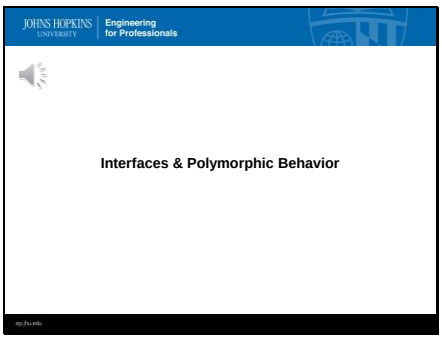
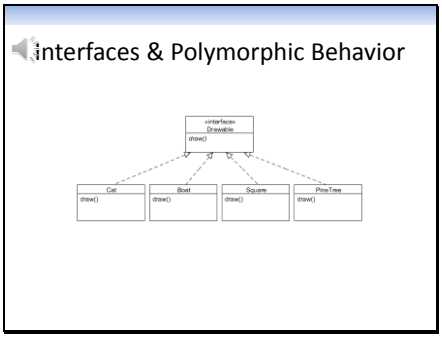
1



In this lecture you will learn how to use interfaces to implement polymorphic behavior.

2



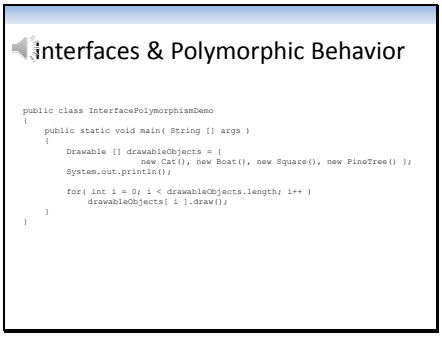
Let's assume we had a number of classes that implemented the Drawable interface…as indicated by this design diagram.

We can actually use the Drawable interface to invoke the draw() method in the four classes polymorphically.

We do this in the same way as we would if we were using inheritance and Drawable was the base class.

Let's take a look at how this can be done.

3



For simplicity, we'll assume that we've already written the code for the Cat, Boat, Square, and PineTree classes.

By now you know what that code would look like.

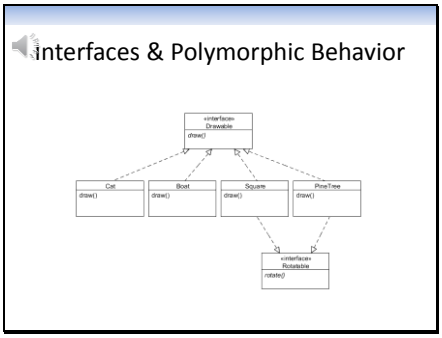
So…let's just write the code for a program that will invoke the draw method on those classes polymorphically.

Here's a program called InterfacePolymorphismDemo that does the job.

An array of Drawable objects is created and contains references to each of the 4 Drawable objects…a Cat, a Boat, a Square, and a PineTree.

Then, the program simply loops through each element in the array and invokes the draw() method for the object that is referenced.

4

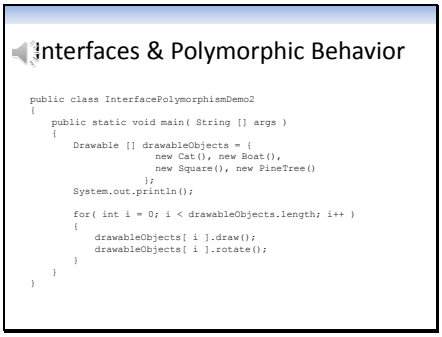


Let's take another example.

In this design, PineTree and Square classes implement both the Drawable and Rotatable interfaces.

Suppose we wanted to invoke both the draw() and rotate() methods polymorphically. How might that be done.

5



Let's look at this code.

This program is a slight variation on the last one. The only change is calling the rotate() method from within the for loop.

So…what do you think…will this work?

Go ahead and pause this lecture now, then look at the code and decide. You can resume the lecture when you're ready.

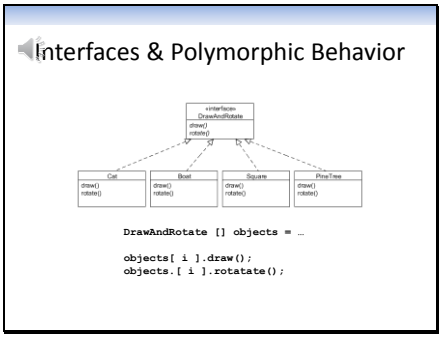
If you think that this won't work…you are correct.

The rotate() method is not part of the Drawable interface…so polymorphic behavior can't be achieved

by using a reference to Drawable.

Since this won't work…how might we change the design so that we can get polymorphic behavior for both the rotate() and draw() methods?

6

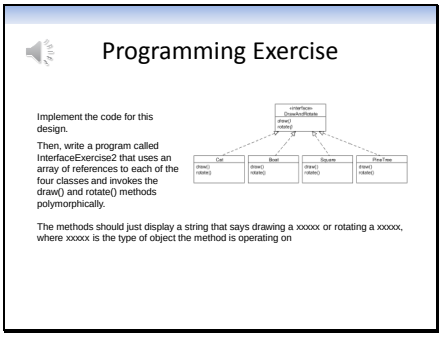


The trick here is that we must use an interface that declares both the draw() and rotate() methods.

In this diagram we have an interface called DrawAndRotate that declares both methods.

If we store references to each object type in an array of DrawAndRotate references, we can get polymorphic behavior, as indicated by this code.

7



At this point I'd like you to do another programming exercise.

Please pause this lecture now, and resume it once you have your program running.

8

Sample Solution

```
public interface DrawAndRotate
{
    void draw();
    void rotate();
}

public class Cat implements DrawAndRotate
{
    public void draw()
    {
        System.out.println( "Drawing a cat" );
    }

    public void rotate()
    {
        System.out.println( "Rotating a cat" );
    }
}
```

Here's a partial solution to the exercise.

The interface DrawAndRotate declares the draw() and rotate() methods, and these are implemented in the Cat class.

The Boat, Square, and PineTree classes would have implementations of draw() and rotate() as well.

9

Sample Solution

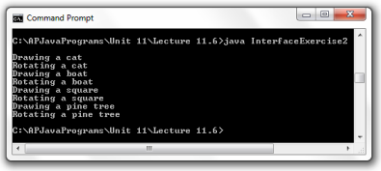
```
public class InterfaceExercise2
{
    public static void main( String [] args )
    {
        DrawAndRotate [] objects = {
                    new Cat(), new Boat(),
                    new Square(), new PineTree()
                        };
        System.out.println();

        for( int i = 0; i < drawableObjects.length; i++ )
        {
            objects[ i ].draw();
            objects[ i ].rotate();
        }
    }
}
```

And here's a program that uses polymorphic behavior to invoked the draw() and rotate() methods.

Let's take a look at the output.

10

Sample Solution



Here's the output of the program.

Your program's output doesn't have to look exactly like this, but it should produce similar results.

If it doesn't, please check your code against our sample solution and make any changes necessary to have your program produce the correct results.