Add figures to open slots below.

Finish with an exercise adding ShrinkingBall as a Decorator.

Look at benefits of polymorphism by Decorator inheritance:
nested decorators!

Add a final slide (couple of minutes) on the polymorhism ideas at work in this pattern and examples.

For each stopping point, show working code. SEE substitution!

Questions got to the idea of how clients create instances of the decorator.

Modify place where I talk about benefits of inheritance + composition. Inheritance is for interface and *gives* substitutability of the new objects for the old. Composition *uses* substitutability as a means for adding behavior to many different classes at once!

An Exercise

Suppose that we are writing a new program using objects from our `Ball` hierarchy. In this program, some `MovableBall`s must decelerate. Every time one of these decelerating balls moves, its speed decreases by 5%.

Add a `DeceleratingBall` class to the `Ball` hierarchy for this purpose.

```
public class Ball
{
   private Rectangle location;
   private Color color;

   ...
}

public class MovableBall extends Ball
{
   private double dx;
   private double dy;

   ...
}

public class BoundedBall extends MovableBall
{
   private int maxHeight;
   private int maxWidth;

   ...
}
```

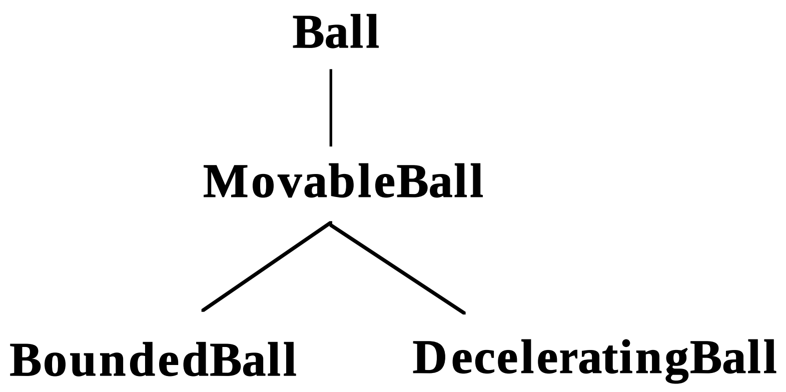
```
import java.awt.*;

public  class  Ball
{
   private Rectangle location;
   private Color     color;

   public Ball (int x, int y, int r)
   {
      location = new Rectangle(x-r, y-r, 2*r, 2*r);
      color = Color.blue;
   }

   public void  setColor (Color newColor) { color = newColor; }
   public Color getColor()                { return color; }

   public int       radius () { return location.width / 2; }
   public int       x ()      { return location.x + radius(); }
   public int       y ()      { return location.y + radius(); }
   public Rectangle region () { return location; }

   public void paint (Graphics g)
   {
      g.setColor (color);
      g.fillOval (location.x, location.y, location.width, location.height);
   }
}
```

```
import java.awt.*;

public  class  MovableBall  extends  Ball
{
   private double dx;
   private double dy;

   public MovableBall (int x, int y, int r)
   {
      super(x, y, r);
      dx = 0;
      dy = 0;
   }

   public void setMotion (double ndx, double ndy)
   {
      dx = ndx;
      dy = ndy;
   }

   public double xMotion () { return dx; }
   public double yMotion () { return dy; }
   public void moveTo (int x, int y)
   {
      region().setLocation(x, y);
   }

   public void move ()
   {
      region().translate ( (int) dx, (int) dy );
   }
}
```

```
import java.awt.*;

public  class  BoundedBall  extends  MovableBall
{
   private Frame myWorld;

   public BoundedBall (int x, int y, int r, Frame aWorld)
   {
      super(x, y, r);
      myWorld = aWorld;
   }

   public BoundedBall()
   {
      super( 0, 0, 0 );
      myWorld = null;
   }

   protected Frame world()
   {
      return myWorld;
   }

   public void move ()
   {
      super.move();

      int maximumHeight = myWorld.getSize().height;
      int maximumWidth  = myWorld.getSize().width;

      if ( (x() < 0) || (x() > maximumWidth) )
         setMotion (-xMotion(), yMotion());

      if ( (y() < 0) || (y() > maximumHeight) )
         setMotion (xMotion(), -yMotion());
   }
}
```

A Possible Solution

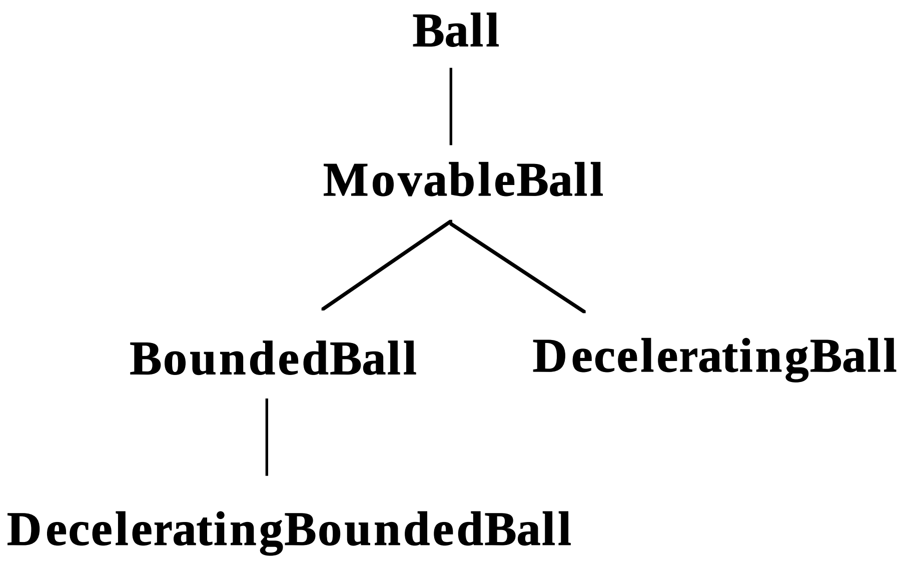
```
import java.awt.*;

public class DeceleratingBall extends MovableBall
{
   public DeceleratingBall( int x, int y, int r )
   {
      super(x, y, r);
   }

   public void move()
   {
      super.move();
      setMotion( xMotion() * 0.95, yMotion() * 0.95 );
   }
}
```



A Wrinkle in Our Solution

It turns out that we sometimes need to use `BoundedBall`s that also decelerate. Can you fix the problem?

```
import java.awt.*;

public class DeceleratingBoundedBall extends BoundedBall
{
   public DeceleratingBoundedBall( int x, int y, int r,
                                   Frame f )
   {
      super(x, y, r, f);
   }

   public void move()
   {
      super.move();
      setMotion( xMotion() * 0.95, yMotion() * 0.95 );
   }
}
```



How Good is Our Solution?

What are the strengths of our approach?

- It is simple.

What are the weaknesses of our approach?

- It repeats codes. The `move()` methods in `DeceleratingBall` and `DeceleratingBoundedBall` are identical!

You may be asking yourself, “So what? It works.”

- What happens if we need to change the deceleration factor, say, from 95% to 80%?

  We must remember to make the change in two different classes.

- What happens if we need to add deceleration behavior to other classes that inherit from `MovableBall`?

  More subclasses!

- What happens if we need to add more behavior to our decelerating ball classes?

  Even more subclasses!

Solutions that make future extensions to the system unbearable are probably not very good solutions at all...

Insert slide of “full” ball hierarchy from Persuasion...

An Alternative Solution

All `BoundedBall`s respond to the same basic set of messages, inherited from `Ball` and `MovableBall`. So they are *substitutable* for one another. Can we use this to our advantage?

```
import java.awt.*;

public class DeceleratingBall extends MovableBall
{
   private MovableBall workerBall;

   public DeceleratingBall( MovableBall aBall )
   {
      super();
      workerBall = aBall;
   }

   public void move()
   {
      workerBall.move();
      workerBall.setMotion( workerBall.xMotion() * 0.95,
                            workerBall.yMotion() * 0.95 );
   }

   // *** MESSAGES DELEGATED TO THE INSTANCE VARIABLE

   public int radius()       { return workerBall.radius(); }

   public int x()            { return workerBall.x(); }
   public int y()            { return workerBall.y(); }

   public Rectangle region() { return workerBall.region(); }

   public void setColor (Color newColor)
   {
      workerBall.setColor( newColor );
   }
   public Color getColor()
   {
      return workerBall.getColor();
   }

   public void paint( Graphics g )
   {
      workerBall.paint( g );
   }

   public void setMotion (double ndx, double ndy)
   {
      workerBall.setMotion( ndx, ndy );
   }

   public double xMotion ()
   {
      return workerBall.xMotion();
   }

   public double yMotion ()
   {
      return workerBall.yMotion();
   }

   public void moveTo (int x, int y)
   {
      workerBall.moveTo( x, y );
   }
}
```

How Good is Our New Solution?

What are the weaknesses of our new approach?

- It is more complex.
- It is a bit slower at run time.

What are the strengths of our new approach?

- It “says it once and only once”. The `move()` method specific to deceleration behavior occurs in one class. The deceleration factor lives in exactly one class.

- We can add deceleration behavior to any `MovableBall` with a single class.

- We can add deceleration behavior to any future subclass of `MovableBall`—with no new code!!

- The tedious task of writing the delegation methods can be done automatically within many OO programming tools.

As is usually the case, we favor a somewhat more complex solution when it offers *extensibility* and *flexibility* as benefits.

Insert slide of the new “full” ball hierarchy from Persuasion...

Do You Recognize a Pattern?

We added flexibility and extensibility to our system using the same ideas we used in our previous two class sessions:

- substitution
- delegation
- recursion

The new twist in this solution is that `DeceleratingBall` uses substitution on a class in its own class hierarchy!

This new twist is so common that it has its own name: **decorator**.

The Problem

> We would like to add a behavior to a set of classes that share a common interface.

A Tempting Solution that Fails

> Use inheritance to create a new class of objects that has the behavior. Use instances of this class when you need the behavior, and use instances of the superclass otherwise.
>
> This solution is impractical.
>
> What if we would like to add the behavior to many different classes in the hierarchy? We will need to create multiple subclasses and replicate the behavior in each.
>
> What if we would like to add *more* behavior to the extended object? We have to make (many!) more subclasses!

The Decorator Pattern

Why Does This Problem Matter?

It occurs in many domains and in many applications:

We want to add features to individual balls in our ball games or to individual card piles in our card games.

We want to add features to individual streams in the Java library, such as buffering the input we read from a stream.

We want to add windowing features to individual objects in a word processor or drawing program.

GoF figure bottom of page 175

The Decorator Pattern

The Solution

Create a *decorator* class. Encapsulate an instance of the base class as an instance variable of the decorator.

Implement the new behavior in the decorator.

Delegate all other messages to the encapsulated object.

...

Use inheritance to extend the decorator class from the contained class. This allows decorated object to be used in all the same places as the encapsulated object!

This is a second example of *using inheritance and composition together*:

By using inheritance, we preserve substitutability.
By using composition, we improve extensibility and flexibility.

Examples of the Decorator Pattern

GoF figures middle page 176

A figure based on Budd top page 169