An Exercise

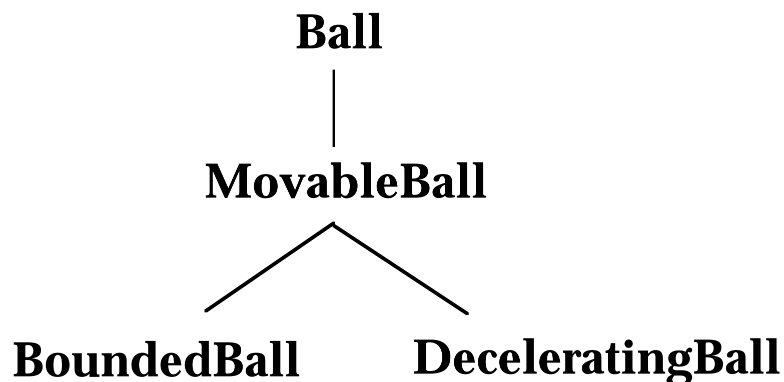
Suppose that we are writing a new program using objects from our `Ball` hierarchy. In this program, some `MovableBall`s must decelerate. Every time one of these decelerating balls moves, its speed decreases by 5%.

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;

   ...
}
```

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 );
   }
}
```



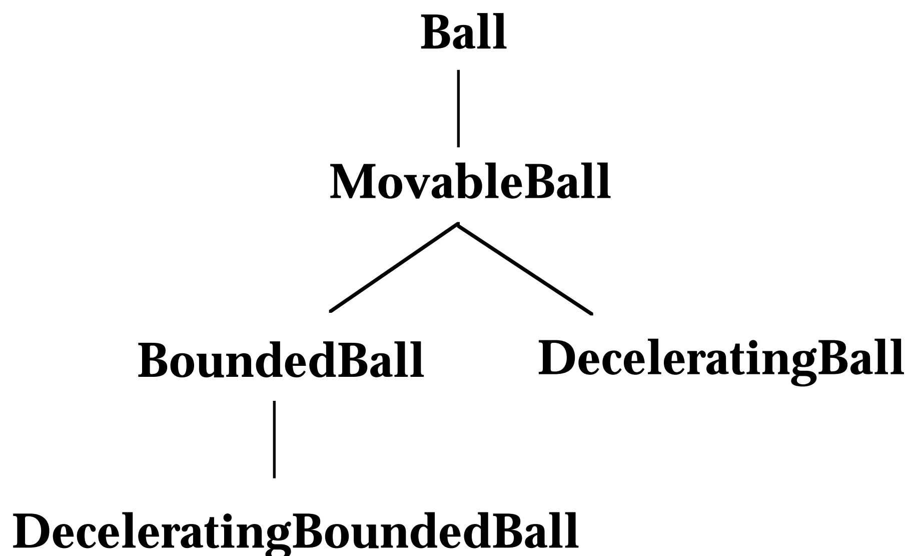
We create a `DeceleratingBall` just as we would a `MovableBall`:

```
DeceleratingBall b = new DeceleratingBall( 10, 15, 5 );
```

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 );
   }
}
```



We create a `DeceleratingBoundedBall` as we would a `BoundedBall`...

How Good is Our Solution?

What are the strengths of our approach?

- It is simple.
- It is easy to implement right now.

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`?

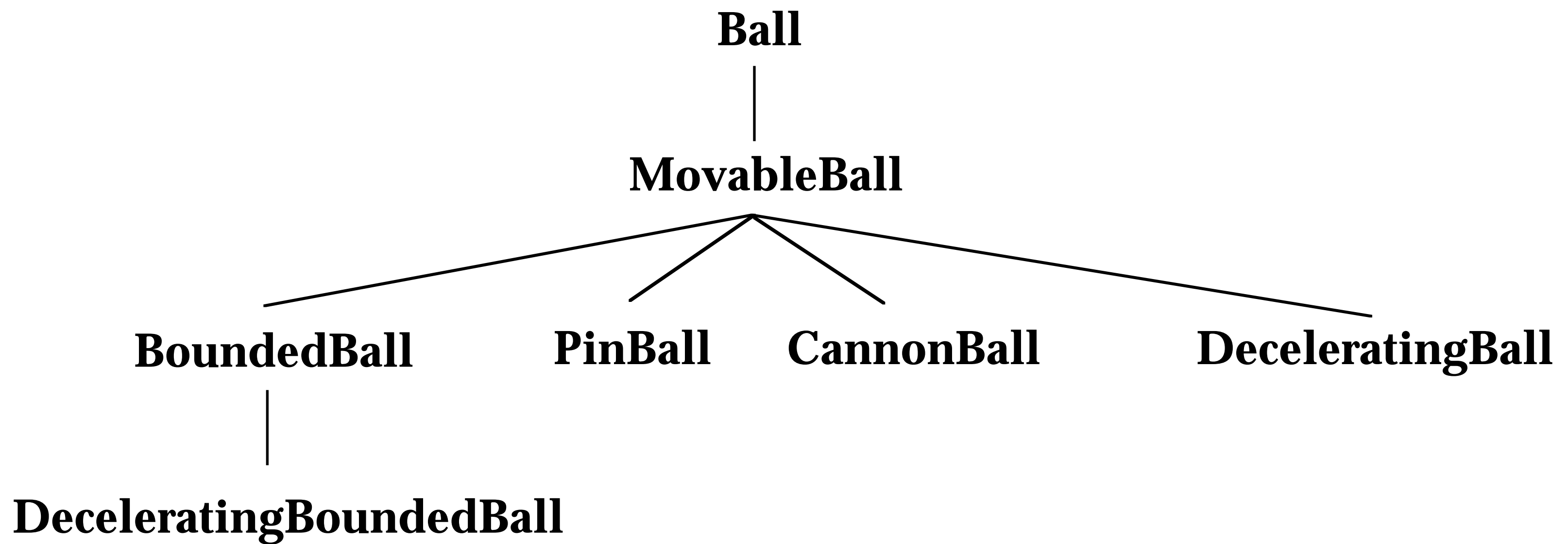
  More subclasses!

- 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...

The Full `Ball` Hierarchy



An Alternative Solution

BoundedBalls respond to the same set of messages as MovableBalls.
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 ENTIRELY TO THE IV

   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 );
   }
```

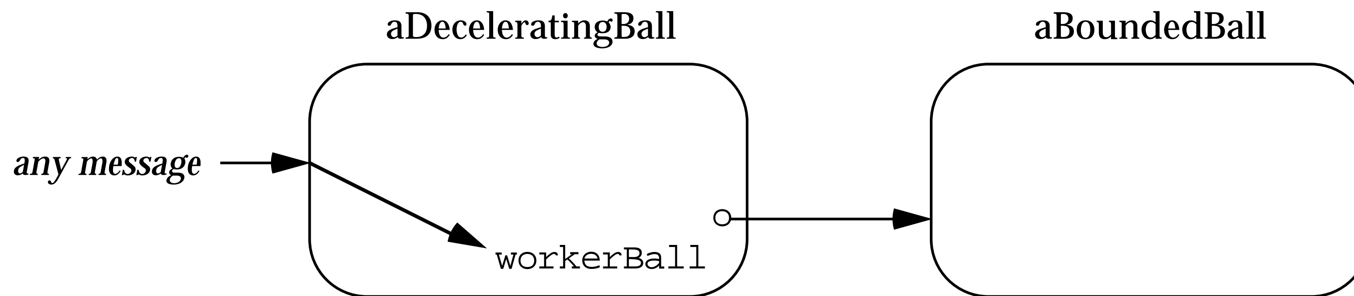
```
    public Color getColor()
    {
        return workerBall.getColor();
    }

    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 );
    }
}
```

Now, we create a `DeceleratingBall` by giving it a `MovableBall` to direct:

```
DeceleratingBall b =
    new DeceleratingBall( new MovableBall( 10, 15, 5 ) );
```

How the "Decorator" Works

```
DeceleratingBall b = new DeceleratingBall( new BoundedBall( x, y, r, this );
```



This approach takes advantage of substitutability to create an object that delegates all of its responsibility to another object... without the client ever knowing the difference!

How Good is Our New Solution?

What are the weaknesses of our new approach?

- It is more complex.
- Decelerating balls are a bit “bigger” and “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.

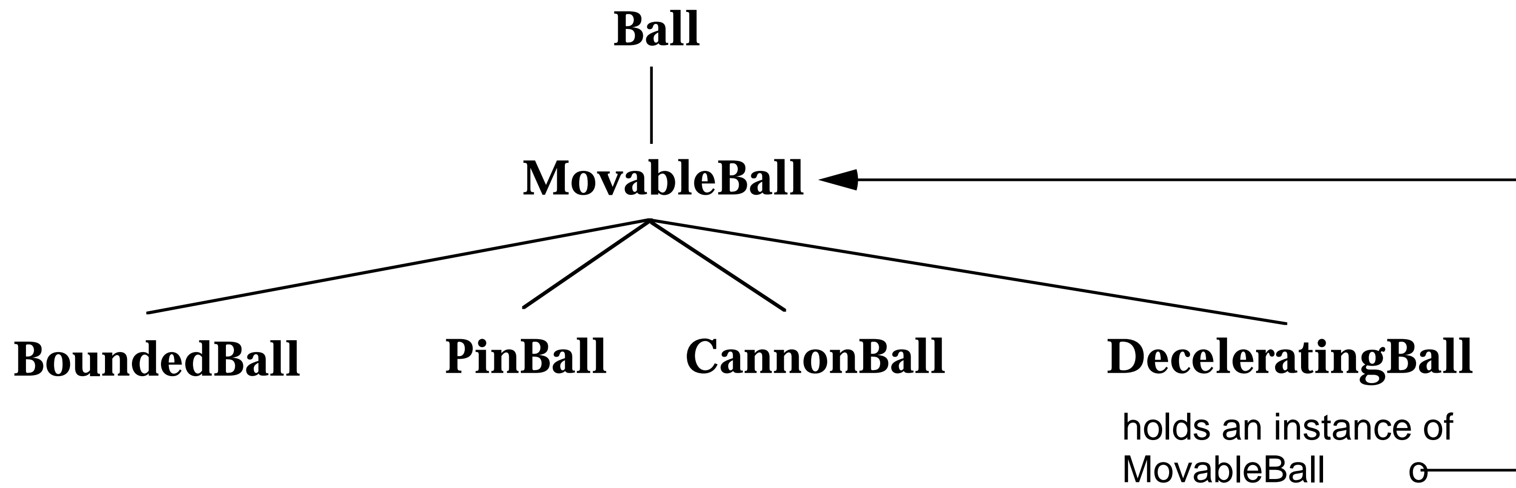
- We can add deceleration behavior to any `MovableBall` with this same class!

- 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. In any case, writing them once seems more palatable than writing multiple subclasses for deceleration throughout the hierarchy.

As is often the case, we sometimes choose a more complex solution when it offers *extensibility* and *flexibility* as benefits.

The New `Ball` Hierarchy



An Exercise

Add an `ExpandingBall` class to the `MovableBall` hierarchy.
An `ExpandingBall` becomes a little bit larger every time it moves.

Make `ExpandingBall` work like our new `DeceleratingBall` class.

```
public class ExpandingBall extends MovableBall
{
   private MovableBall workerBall;

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

   public void move ()
   {
      workerBall.move();
      workerBall.region().height =
         ( workerBall.region().height * 11 ) / 10;
      workerBall.region().width  =
         ( workerBall.region().width  * 11 ) / 10;
   }

   // *** 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(); }

   ... // all the rest of the messages in MovableBall, too
}
```

Using An `ExpandingBall`

Here's how we might use an `ExpandingBall` in the `MultiBallWorld`:

```
protected void initializeArrayOfBalls( Color ballColor )
{
   ballArray = new MovableBall [ BallArraySize ];
   for (int i = 0; i < BallArraySize; i++)
   {
      ballArray[i] =
         new ExpandingBall(
             new BoundedBall( 10, 15, 5, this ) );
      ballArray[i].setColor( ballColor );
      ballArray[i].setMotion( 3.0+i, 6.0-i );
   }
}
```

But here's a beautiful example of what using a decorator can do for you:

```
protected void initializeArrayOfBalls( Color ballColor )
{
   ballArray = new MovableBall [ BallArraySize ];
   for (int i = 0; i < BallArraySize; i++)
   {
      ballArray[i] =
         new ExpandingBall(
             new DeceleratingBall(
                 new BoundedBall( 10, 15, 5, this ) ) );
      ballArray[i].setColor( ballColor );
      ballArray[i].setMotion( 3.0+i, 6.0-i );
   }
}
```

Since a decorator is substitutable for instances of its base class, you can decorate a decorator!

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: combining **composition** and **inheritance**.

- 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. Why?

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 Problem Solved by 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

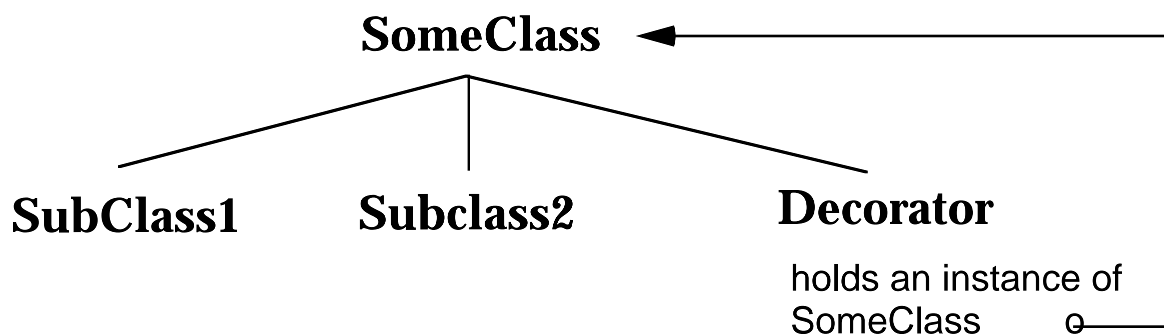
Create a decorator class.

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

   Implement the new behavior in the decorator. Delegate as much of the new behavior to the instance variable as possible.

   Send all other messages recursively to the encapsulated object.

2. Use *inheritance* to extend the decorator class from the contained class.



How Does the Decorator Pattern Work?

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

Inheritance creates substitutable classes. This allows decorated object to be used in all the same places as the encapsulated object!

The superclass acts as an interface for all of its subclasses.

An application won't know—or need to know—what sort of `MovableBall` it is using; the ball responds to all the same messages.

Composition uses substitution to reuse the code in the base class, but in a way that is controlled by the decorator.

This allows us to add the same behavior to **all** of the classes in the hierarchy!

In a way, the decorator pattern allows us to add new behavior to a single instance, rather than to the whole class.

Using a decorator makes sense any time there are two varieties of some object, but the variations are not directly related.

- balls that *bounce* of the walls and balls that *decelerate*

Polymorphism

Decorators are a great example of polymorphism.

Polymorphism is about how we can use different objects in the same place in our program, because they respond to the same set of messages.

We can use interfaces to specify polymorphic classes. `Tree`

We can use inheritance to define polymorphic classes. `MovableBall`

Decorators use polymorphism in two ways:

- Since the decorator class is a subclass, instances of the decorator can substitute for instances of the base class.
- Since the decorator class holds an instance of the base class, it can hold instances of anything substitutable for the base class.

As you study these examples, focus on:

- the use of inheritance to make polymorphic objects.
- the use of composition and inheritance together.

Try playing with the code to create new substitutable subclasses and new decorators. Polymorphism comes alive when you see it run!