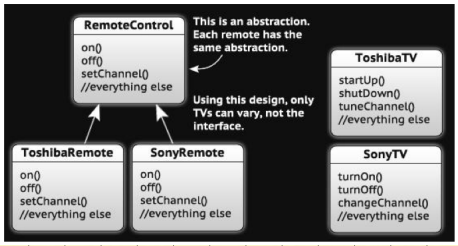
Bridge Pattern

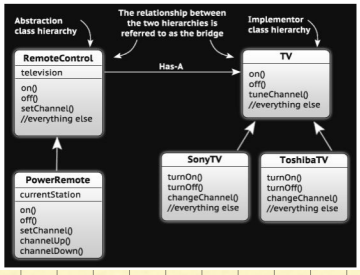
The Scenario

- You're writing code for a new remote control for TVs
- You're a good OO designer, so you'll make good use of abstraction
  - You'll have a RemoteControl base class and multiple implementations, one for each model of TV



The Dilemma

- The problem is that only the implementations can vary
- You need to make it possible for the abstraction to vary
- Do you remember "favor composition over inheritance"?
  - How can we employ that?



The Result

- Now you have two separate class hierarchies: one for the abstraction, and one for the implementation
  - The "has-a" relationship between the two is the "bridge"
- Benefits
  - Decouples an implementation so it is not permanently bound to an unchanging interface
  - Abstraction and implementation can be extended independently
  - Changes to the concrete abstraction classes don't affect the client
- Uses
  - Useful in graphics and windowing systems that need to run over multiple platforms
  - Useful anytime you need to vary an interface and an implementation in different ways
- Disadvantages
  - Increases complexity

Flyweight Pattern

The Scenario

- You want to add trees to a landscape design application
  - Trees are pretty dumb; they have x-y locations, they can draw themselves dynamically (depending on an age attribute)
  - A user might want to add lots and lots of trees
- A user of your application is complaining that, when he creates large groves of trees, the app gets sluggish
  - Yikes! There are THOUSANDS of tree objects!
- When you have tons of objects that are basically the same
  - Create a single instance of the object
  - A single client object that manages the state of all of those objects

Example

```
public class Test extends JFrame{
private static final Color colors[] = { Color.red, Color.blue,
                                        Color.yellow, Color.orange,
                                        Color.black,  Color.white };
   public Test() {
      // …
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            Graphics g = panel.getGraphics();
            for(int i=0; i < NUMBER_OF_LINES; ++i) {
               g.setColor(getRandomColor());
               g.drawLine(getRandomX(), getRandomY(),
                          getRandomX(), getRandomY());
            }
         }
      });
   }
   private Color getRandomColor() {
      return colors[(int)(Math.random()*colors.length)];
   }
   // helper methods
}
```

Example with Classes

```
public class Test extends JFrame{
   public Test() {
      //...
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            Graphics g = panel.getGraphics();
            for(int i=0; i < NUMBER_OF_LINES; ++i) {
               Color color = getRandomColor();
               Line line = new Line(color,
                                    getRandomX(), getRandomY(),
                                    getRandomX(), getRandomY());
               line.draw(g);
            }
         }
      });
   }
   ...
}

public class Line {
   private Color color = Color.black;
   private int x, y, x2, y2;
   public Line(Color color, int x, int y, int x2, int y2) {
      this.color = color;  this.x = x;   this.y = y;  this.x2 = x2; this.y2 = y2;
   }
   public void draw(Graphics g) {
      g.setColor(color);
      g.drawLine(x, y, x2, y2);
   }
}
```

Yikes!

- We made 10,000 Line objects. That's insane
- Instead, let's group lines by some fundamental characteristic, say color
  - Now we only need to create six lines; then we can reuse the implementation…

Flyweight Lines

```
public class Test extends JFrame {
   //...
   public Test() {
      //...
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent event) {
            Graphics g = panel.getGraphics();
            for(int i=0; i < NUMBER_OF_LINES; ++i) {
               Line line = LineFactory.getLine(getRandomColor());
               line.draw(g, getRandomX(), getRandomY(),
                           getRandomX(), getRandomY());
            }
         }
      });
   }
   //...
}
```

```
public class LineFactory {
   private static final HashMap linesByColor = new HashMap();
   public static Line getLine(Color color) {
      Line line = (Line)linesByColor.get(color);
      if(line == null) {
         line = new Line(color);
         linesByColor.put(color, line);
         System.out.println("Creating " + color + " line");
      }
      return line;
   }
}

public class Line {
   private Color color;
   public Line(Color color) {
      this.color = color;
   }
   public void draw(Graphics g, int x, int y, int x2, int y2) {
      g.setColor(color);
      g.drawLine(x, y, x2, y2);
   }
}
```