Bridge

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y + "]");
   }
}
public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " + y + "]");
   }
}
public abstract class Shape {
   protected DrawAPI drawAPI;
   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}



public class Circle extends Shape {
 private int x, y, radius;
```

```
  public Circle(int x, int y, int radius, DrawAPI drawAPI) {
    super(drawAPI);
    this.x = x;
    this.y = y;
    this.radius = radius;
  }
  public void draw() {
    drawAPI.drawCircle(radius,x,y);
  }
}
public class BridgePatternDemo {
  public static void main(String[] args) {
    Shape redCircle = new Circle(100,100, 10, new RedCircle());
    Shape greenCircle = new Circle(100,100, 10, new GreenCircle());
    redCircle.draw();
    greenCircle.draw();
  }
}
```

Composite

/*source https://sourcemaking.com/design_patterns/composite/java/2*/

```
public interface Component {
        public void traverse();

        public int getValue();
}
```

==========

```
import java.util.ArrayList;

abstract class Composite implements Component {
        private ArrayList<Component>  parts = new ArrayList<Component>();
        private int        total    = 0;
        private int        value;
        public Composite( int val )    { value = val; }
        public void add( Component c ) {
                total++;
                parts.add(c);
        }
        public void traverse() {
          System.out.print( value + "  " );
          for (int i=0; i < total; i++){
             (parts.get(i)).traverse();
          }
```

```
        }
          public int getValue()          {return value;}
}

========================

class Primitive implements Component {
          private int value;
          public Primitive( int val ) { value = val; }
          public void traverse()     { System.out.print( value + "  " ); }
          public int getValue()          {return value;}
        }

====================================

class Column extends Composite {
          public Column( int val ) { super( val ); }
          public void traverse() {
            System.out.print( "Col" );
            super.traverse();
        } }

======================

class Row extends Composite {
          public Row( int val ) { super( val ); }
          public void traverse() {
```

```
            System.out.print( "Row" );
            super.traverse();
        }
}
=====================
public class CompositeDemo {
          public static void main( String[] args ) {
            Composite first  = new Row( 1 );
            Composite second = new Column( 2 );
            Composite third  = new Column( 3 );
            Composite fourth = new Column( 4 );
            Composite fifth  = new Column( 5 );
            first.add( second );
            first.add( third  );
            third.add( fourth );
            third.add( fifth  );
            first.add(  new Primitive( 6 ) );
            second.add( new Primitive( 7 ) );
            third.add(  new Primitive( 8 ) );
            fourth.add( new Primitive( 9 ) );
            fifth.add(  new Primitive(10 ) );
            first.traverse();
        } }
```

Decorator

```
public interface Shape {
  void draw();
}

============================

public class Rectangle implements Shape {

  @Override
  public void draw() {
    System.out.println("Shape: Rectangle");
  }
}

==================================

public class Circle implements Shape {

  @Override
  public void draw() {
    System.out.println("Shape: Circle");
  }
}

========================

public abstract class ShapeDecorator implements Shape {
  protected Shape decoratedShape;
```

```
  public ShapeDecorator(Shape decoratedShape){
    this.decoratedShape = decoratedShape;
  }

  public void draw(){
    decoratedShape.draw();
  }
}
==============================
public class RedShapeDecorator extends ShapeDecorator {

  public RedShapeDecorator(Shape decoratedShape) {
    super(decoratedShape);
  }

  @Override
  public void draw() {
    decoratedShape.draw();
    setRedBorder(decoratedShape);
  }

  private void setRedBorder(Shape decoratedShape){
    System.out.println("Border Color: Red");
  }
}
================================
public class DecoratorPatternDemo {
```

```
  public static void main(String[] args) {

    Shape circle = new Circle();

    Shape redCircle = new RedShapeDecorator(new Circle());

    Shape redRectangle = new RedShapeDecorator(new Rectangle());
    System.out.println("Circle with normal border");
    circle.draw();

    System.out.println("\nCircle of red border");
    redCircle.draw();

    System.out.println("\nRectangle of red border");
    redRectangle.draw();
  }
}
```

Proxy

```
public interface Image {
	public void showImage();
	public String getFilePath();
}
```

==========

```
public class HighResolutionImage implements Image {
	private String imageFilePath;
	public HighResolutionImage(String imageFilePath) {
		this.imageFilePath = imageFilePath;
		loadImage(imageFilePath);
	}
	private void loadImage(String imageFilePath) {
		// load Image from disk into memory
		// this is heavy and costly operation
	}
	public String getFilePath(){
		return imageFilePath;
	}
	@Override
	public void showImage() {
		// Actual Image rendering logic
	}

}
```

==========

```
public class ImageProxy implements Image {
	/**
	 * Private Proxy data
	 */
	private String imageFilePath;
	/**
	 * Reference to RealSubject
	 */
	private Image proxifiedImage;
	public String getFilePath(){
		return imageFilePath;
	}
	public ImageProxy(String imageFilePath) {
		this.imageFilePath= imageFilePath;
	}

	@Override
	public void showImage() {
		// create the Image Object only when the image is required to be shown
		proxifiedImage = new HighResolutionImage(imageFilePath);
		// now call showImage on realSubject
		proxifiedImage.showImage();

	}

}
```

==========

```
public class ImageViewer {

	public static void main(String[] args) {

	// assuming that the user selects a folder that has 3 images

	//create the 3 images

	Image highResolutionImage1 = new ImageProxy("sample/veryHighResPhoto1.jpeg");

	Image highResolutionImage2 = new ImageProxy("sample/veryHighResPhoto2.jpeg");

	Image highResolutionImage3 = new ImageProxy("sample/veryHighResPhoto3.jpeg");

	System.out.println(highResolutionImage2.getFilePath());

	// at this point, no image is loaded to memory

	// assume that the user clicks on Image one item in a list

	// this would cause the program to call showImage() for that image only

	// note that in this case only image one was loaded into memory

	highResolutionImage1.showImage();

	// consider using the high resolution image object directly

	Image highResolutionImageNoProxy1 = new
HighResolutionImage("sample/veryHighResPhoto1.jpeg");

	Image highResolutionImageNoProxy2 = new
HighResolutionImage("sample/veryHighResPhoto2.jpeg");

	Image highResolutionImageBoProxy3 = new
HighResolutionImage("sample/veryHighResPhoto3.jpeg");

	// assume that the user selects image two item from images list

	highResolutionImageNoProxy2.showImage();

	// note that in this case all images have been loaded into memory

	// and not all have been actually displayed

	// this is a waste of memory resources


	}

}
```