**Bridge Design Pattern**

Just like [Adapter pattern](https://www.journaldev.com/1487/adapter-design-pattern-java), bridge design pattern is one of the **Structural design pattern**.

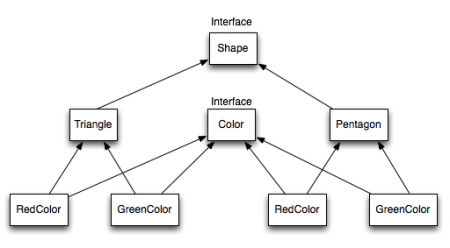
According to GoF bridge design pattern is:

Decouple an abstraction from its implementation so that the two can vary independently

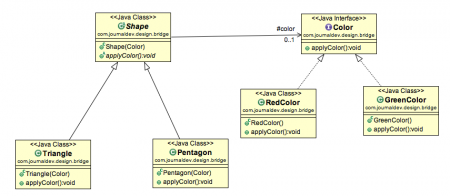
The implementation of bridge design pattern follows the notion to prefer [Composition](https://www.journaldev.com/1325/composition-in-java-example) over [inheritance](https://www.journaldev.com/644/inheritance-java-example).

**Bridge Design Pattern in Java Example**

If we look into bridge design pattern with example, it will be easy to understand. Lets say we have an interface hierarchy in both interfaces and implementations like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Bridge-Interface-Hierarchy.png)

Now we will use bridge design pattern to decouple the interfaces from implementation. UML diagram for the classes and interfaces after applying bridge pattern will look like below image.

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/bridge-design-pattern.png)

Notice the bridge between Shape and Color interfaces and use of composition in implementing the bridge pattern.

Here is the java code for Shape and Color interfaces.

Color.java

package com.journaldev.design.bridge;

public interface Color {

public void applyColor();

}

Shape.java

package com.journaldev.design.bridge;

public abstract class Shape {

//Composition - implementor

protected Color color;

//constructor with implementor as input argument

public Shape(Color c){

this.color=c;

}

abstract public void applyColor();

}

We have Triangle and Pentagon implementation classes as below.

Triangle.java

package com.journaldev.design.bridge;

public class Triangle extends Shape{

public Triangle(Color c) {

super(c);

}

@Override

public void applyColor() {

System.out.print("Triangle filled with color ");

color.applyColor();

}

}

Pentagon.java

package com.journaldev.design.bridge;

public class Pentagon extends Shape{

public Pentagon(Color c) {

super(c);

}

@Override

public void applyColor() {

System.out.print("Pentagon filled with color ");

color.applyColor();

}

}

Here are the implementation classes for RedColor and GreenColor.

RedColor.java

package com.journaldev.design.bridge;

public class RedColor implements Color{

public void applyColor(){

System.out.println("red.");

}

}

GreenColor.java

package com.journaldev.design.bridge;

public class GreenColor implements Color{

public void applyColor(){

System.out.println("green.");

}

}

Lets test our bridge pattern implementation with a test program.

BridgePatternTest.java

package com.journaldev.design.test;

import com.journaldev.design.bridge.GreenColor;

import com.journaldev.design.bridge.Pentagon;

import com.journaldev.design.bridge.RedColor;

import com.journaldev.design.bridge.Shape;

import com.journaldev.design.bridge.Triangle;

public class BridgePatternTest {

public static void main(String[] args) {

Shape tri = new Triangle(new RedColor());

tri.applyColor();

Shape pent = new Pentagon(new GreenColor());

pent.applyColor();

}

}

Output of above bridge pattern example program is:

Triangle filled with color red.

Pentagon filled with color green.

Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.