Adapter

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
This pattern involves a single class, which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader, which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Question: ***Which creational pattern is used in the following code?***
In this question, we have an audio player device that can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.
We have a *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default. We have another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files.
We want to make *AudioPlayer* to play other formats as well. To attain this, we have created an class *Pattern* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.
AudioPlayer uses the class *Pattern* passing it the desired audio type without knowing the actual class, which can play the desired format. *PatternDemo*, our demo class will use *AudioPlayer* class to play various formats.

1. Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

```
public interface MediaPlayer {
  public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
  public void playVlc(String fileName);
  public void playMp4(String fileName);
}
```

2. Create concrete classes implementing the AdvancedMediaPlayer interface.

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
  @Override
  public void playVlc(String fileName) {
    System.out.println("Playing vlc file. Name: "+ fileName);
  }

  @Override
  public void playMp4(String fileName) {  //do nothing
  }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

  @Override
  public void playVlc(String fileName) {
    //do nothing
  }

  @Override
  public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: "+ fileName);
  }
}
```

3. Create pattern class implementing the MediaPlayer interface.

Pattern.java

```
public class Pattern implements MediaPlayer {

  AdvancedMediaPlayer advancedMusicPlayer;

  public Pattern(String audioType){

    if(audioType.equalsIgnoreCase("vlc") ){
      advancedMusicPlayer = new VlcPlayer();

    }else if (audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer = new Mp4Player();
    }
  }

  @Override
  public void play(String audioType, String fileName) {

    if(audioType.equalsIgnoreCase("vlc")){
      advancedMusicPlayer.playVlc(fileName);
    }
    else if(audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer.playMp4(fileName);
    }
  }
}
```

4. Create concrete class implementing the MediaPlayer interface.

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
  Pattern pattern;

  @Override
  public void play(String audioType, String fileName) {

    //inbuilt support to play mp3 music files
    if(audioType.equalsIgnoreCase("mp3")){
      System.out.println("Playing mp3 file. Name: " + fileName);
    }

    //pattern is providing support to play other file formats
    else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
      pattern = new Pattern(audioType);
      pattern.play(audioType, fileName);
    }

    else{
      System.out.println("Invalid media. " + audioType + " format not supported");
    }
  }
}
```

5. Use the AudioPlayer to play different types of audio formats.

PatternDemo.java

```
public class PatternDemo {
  public static void main(String[] args) {
    AudioPlayer audioPlayer = new AudioPlayer();

    audioPlayer.play("mp3", "beyond the horizon.mp3");
    audioPlayer.play("mp4", "alone.mp4");
    audioPlayer.play("vlc", "far far away.vlc");
    audioPlayer.play("avi", "mind me.avi");
  }
}
```

6. Verify the output.

(Output)

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

Reference: https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

Bridge

Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.
This pattern involves an interface, which acts as a bridge that makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

Question: ***Which creational pattern is used in the following code?***
In this question we use a pattern for drawing a circle with different colors using same abstract class method but different pattern implementer classes.
We have a *DrawAPI* interface which is acting as a pattern implementer and concrete classes *RedCircle*, *GreenCircle* implementing the *DrawAPI* interface. *Shape* is an abstract class and will use object of *DrawAPI*. *PatternDemo*, our demo class will use *Shape* class to draw different colored circle.

1. Create pattern implementer interface.

DrawAPI.java

```
public interface DrawAPI {
  public void drawCircle(int radius, int x, int y);
}
```

2. Create concrete pattern implementer classes implementing the DrawAPI interface.

RedCircle.java

```
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 + "]");
  }
}
```

GreenCircle.java

```
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 + "]");
  }
}
```

3. Create an abstract class Shape using the DrawAPI interface.

Shape.java

```
public abstract class Shape {
  protected DrawAPI drawAPI;

  protected Shape(DrawAPI drawAPI){
    this.drawAPI = drawAPI;
  }
  public abstract void draw();
}
```

4. Create concrete class implementing the Shape interface.

Circle.java

```
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);
  }
}
```

5. Use the Shape and DrawAPI classes to draw different colored circles.

PatternDemo.java

```
public class PatternDemo {
  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();
  }
}
```

6. Verify the output.

(Output)

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[ color: green, radius: 10, x: 100, 100]
```

Reference: https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm

Composite

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.
This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

Question: ***Which creational pattern is used in the following code?***
This question is about the employees' hierarchy of an organization. We have a class *Employee*, which acts as pattern actor class. *PatternDemo*, our demo class will use *Employee* class to add department level hierarchy and print all employees.

1. Create Employee class having list of Employee objects.

Employee.java

```
import java.util.ArrayList;
import java.util.List;

public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;

   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }

   public String toString(){
      return ("Employee :[ Name : " + name + ", dept : " + dept + ", salary :" + salary+" ]");
   }
}
```

2. Use the Employee class to create and print employee hierarchy.

PatternDemo.java

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

    Employee CEO = new Employee("John","CEO", 30000);

    Employee headSales = new Employee("Robert","Head Sales", 20000);

    Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

    Employee clerk1 = new Employee("Laura","Marketing", 10000);
    Employee clerk2 = new Employee("Bob","Marketing", 10000);

    Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
    Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

    CEO.add(headSales);
    CEO.add(headMarketing);

    headSales.add(salesExecutive1);
    headSales.add(salesExecutive2);

    headMarketing.add(clerk1);
    headMarketing.add(clerk2);

    System.out.println(CEO);

    for (Employee headEmployee : CEO.getSubordinates()) {
      System.out.println(headEmployee);

      for (Employee employee : headEmployee.getSubordinates()) {
        System.out.println(employee);
      }
    }
  }
}
```

3. Verify the output.

(Output)

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

Reference: https://www.tutorialspoint.com/design_pattern/composite_pattern.htm