Design for Change
Patterns & Principles

Dr. Javaria Imtiaz,
Mr. Basharat Hussain,
Mr. Majid Hussain

Content

- Observer
- Adaptor
- Strategy

- *"Gang of Four" design patterns* describing how to solve recurring design challenges in order to design flexible and reusable object-oriented software, i.e. objects which are easier to implement, change, test, and reuse

Strategy Pattern

Intent:

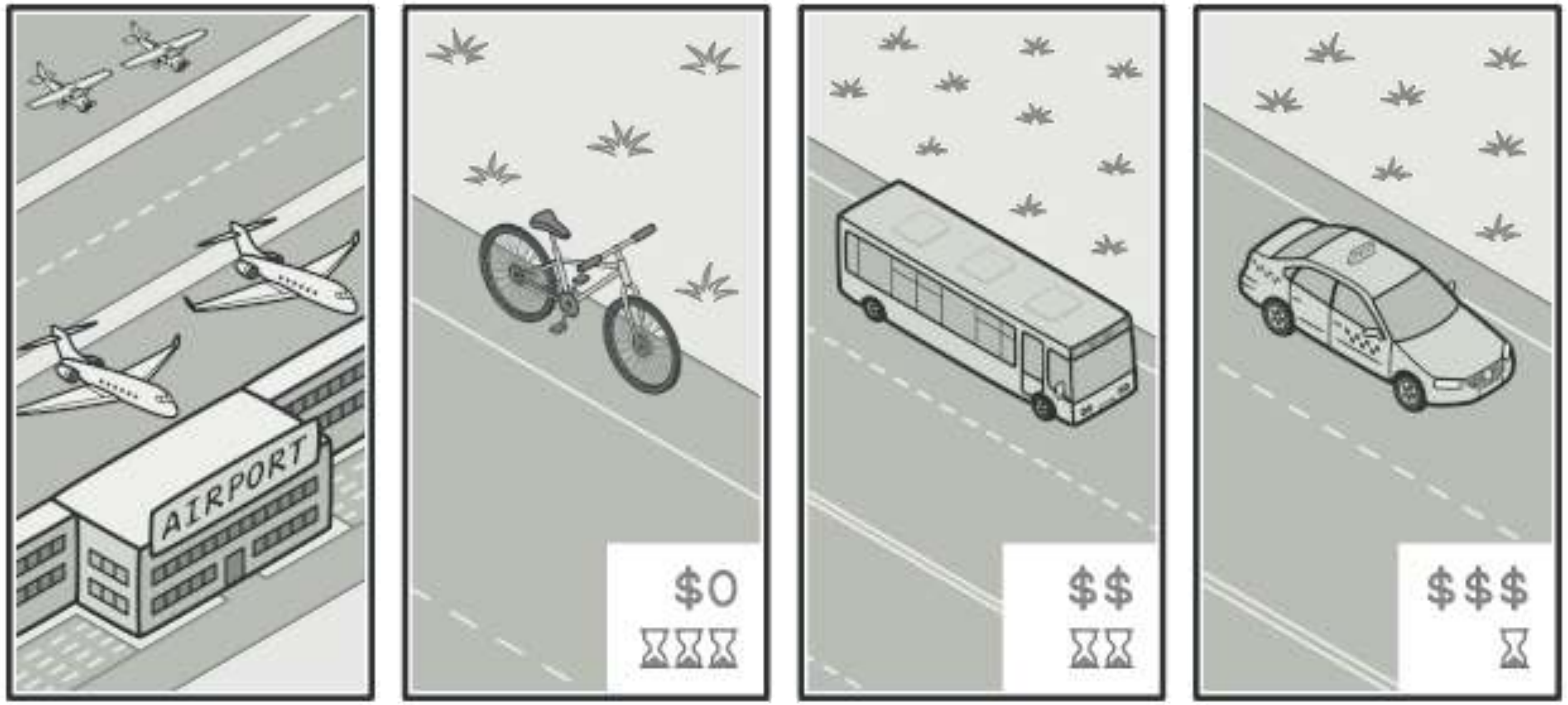
- **Strategy** is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Problem

1. One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.

1. One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.

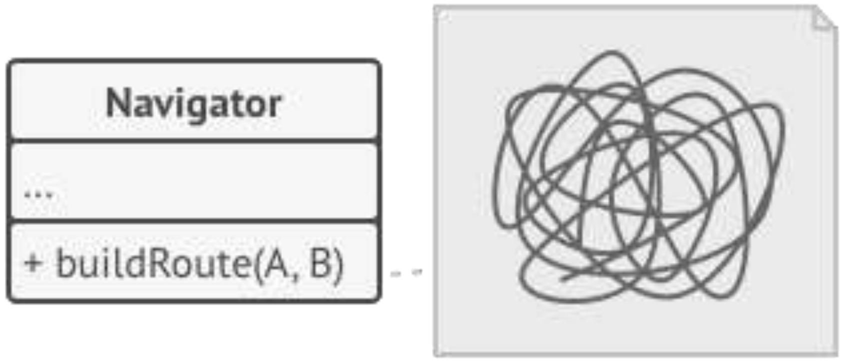
1. The first version of the app could only build the routes over roads. People who traveled by car were bursting with joy. But apparently, not everybody likes to drive on their vacation. So with the next update, you added an option to build walking routes. Right after that, you added another option to let people use public transport in their routes.



Various strategies for getting to the airport.

Cont..

- However, that was only the beginning. Later you planned to add route building for cyclists. And even later, another option for building routes through all of a city's tourist attractions.



The code of the navigator became bloated.

Cont..

1. From a business perspective the app was a success, the technical part caused many headaches.

1. Each time you added a new routing algorithm, the main class of the navigator doubled in size. At some point, the beast became too hard to maintain.

1. Any change to one of the algorithms, whether it was a simple bug fix or a slight adjustment of the street score, affected the whole class, increasing the chance of creating an error in already-working code.

Solution

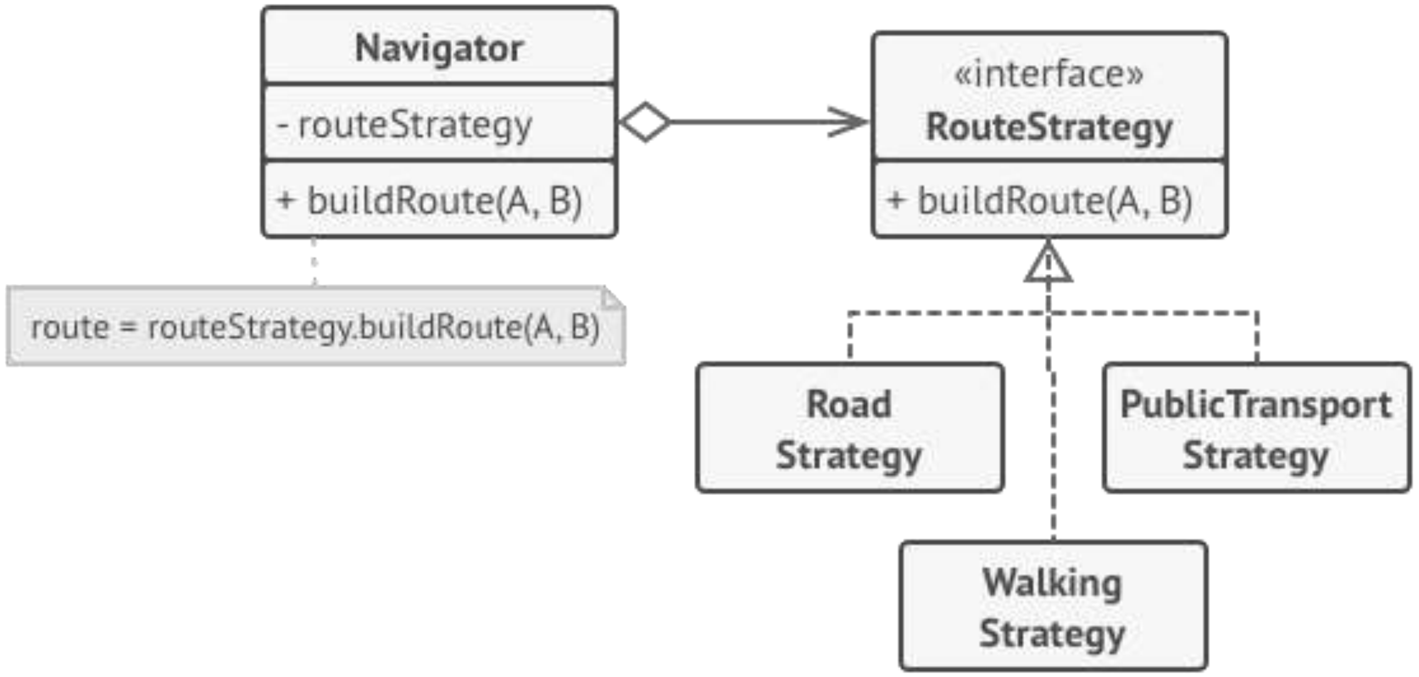
1. The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.

1. The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

1. The context isn't responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context. In fact, the context doesn't know much about strategies. It works with all strategies through the same generic interface, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

Cont..

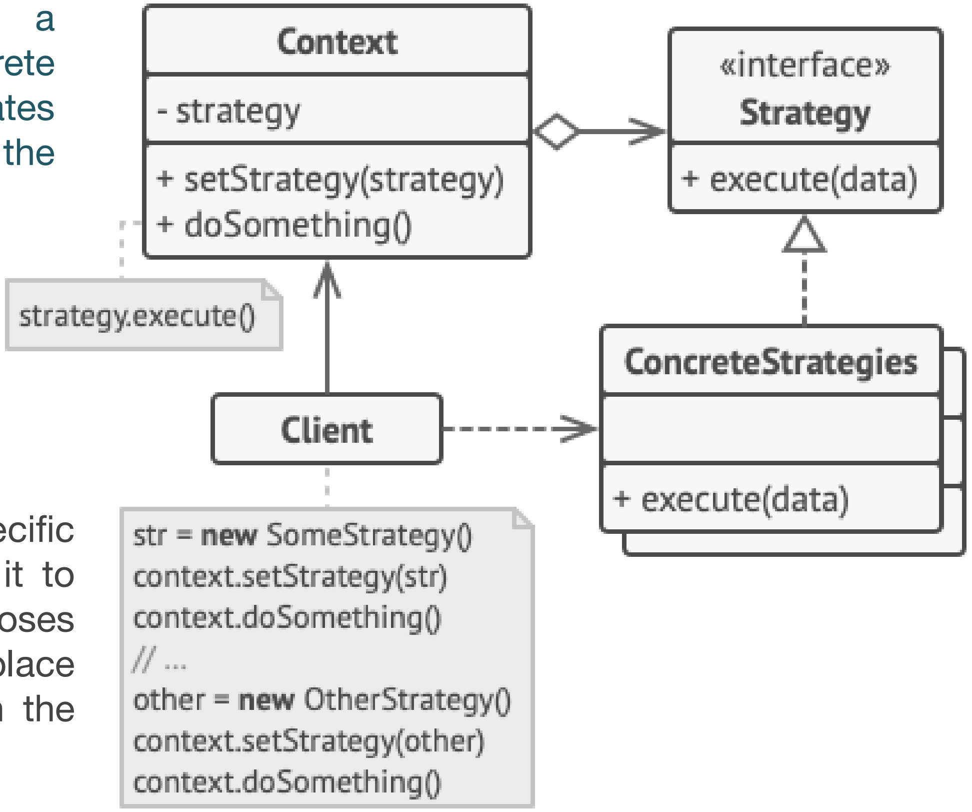
- This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones without changing the code of the context or other strategies.



Route planning strategies.

UML for Strategy pattern

The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface



```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

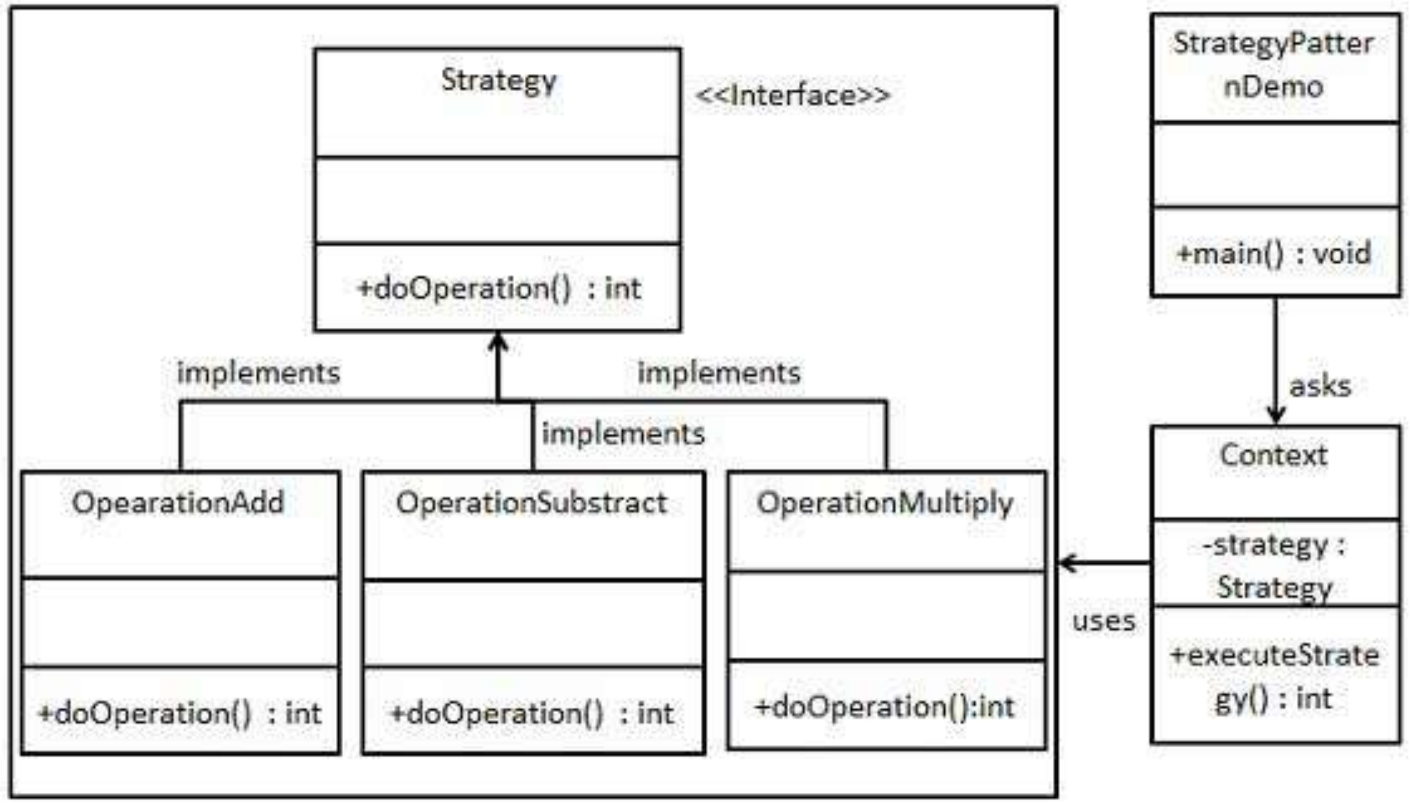
The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

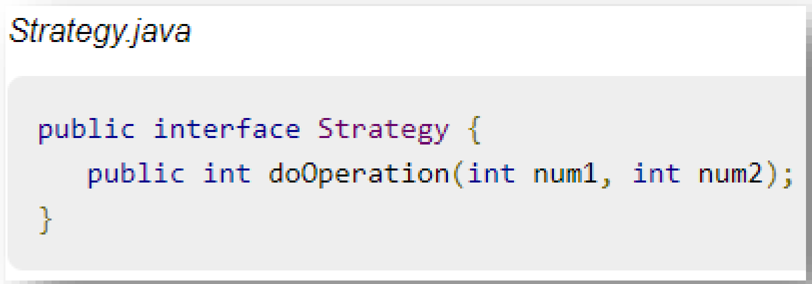
Concrete Strategies implement different variations of an algorithm the context uses.

The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

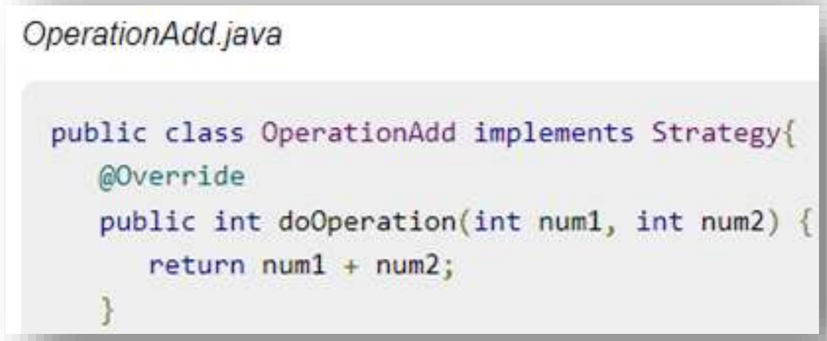
Implementation In JAVA

Example

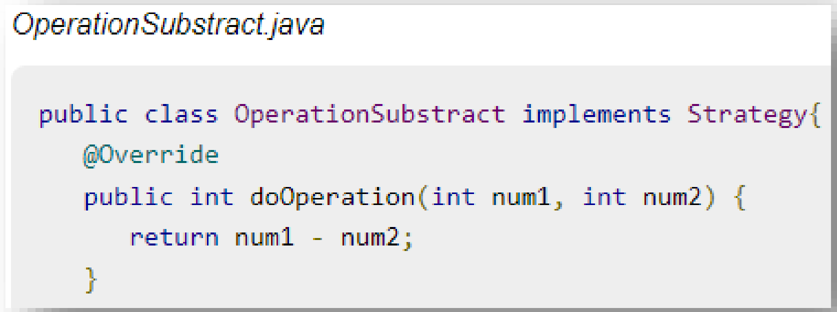

Strategy
<<Interface>>
+doOperation() : int
implements
implements
implements
OpearationAdd
+doOperation() : int
OperationSubstract
+doOperation() : int
OperationMultiply
+doOperation():int
StrategyPatternDemo
+main() : void
asks
Context
-strategy : Strategy
+executeStrategy() : int
uses


Strategy.java

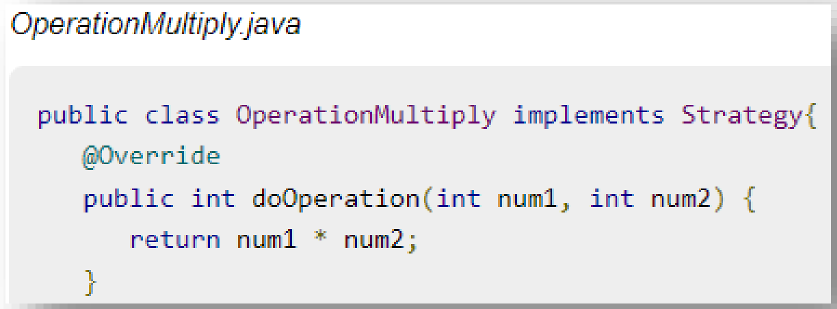
```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```


OperationAdd.java

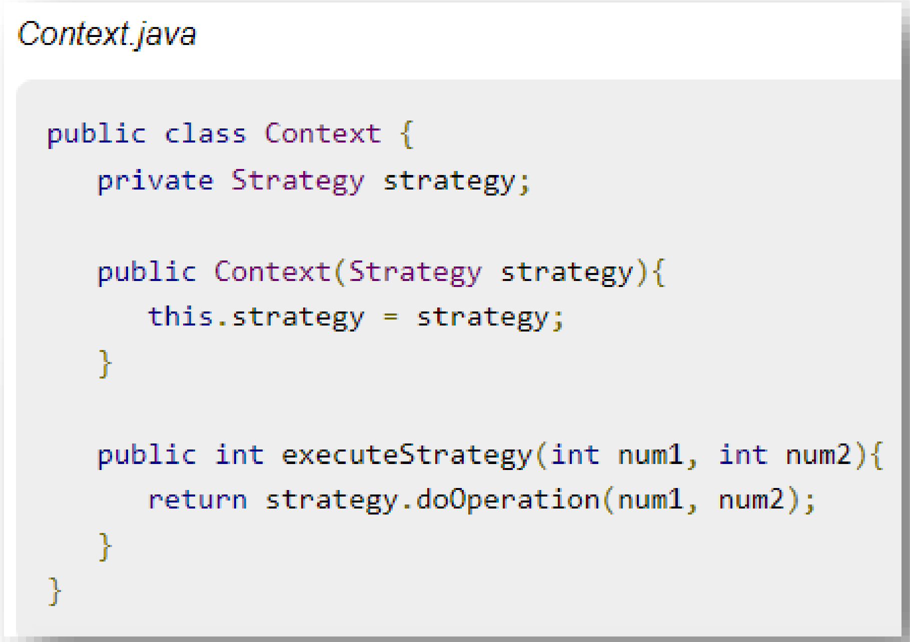
```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
```


OperationSubstract.java

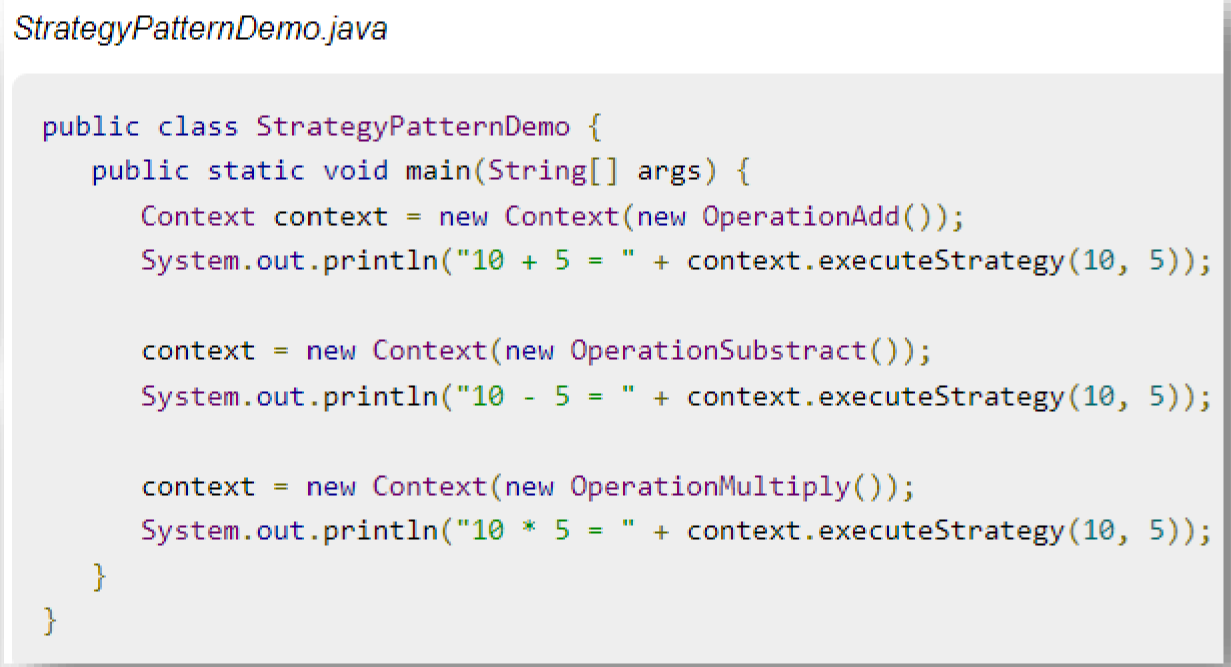
```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
```


OperationMultiply.java

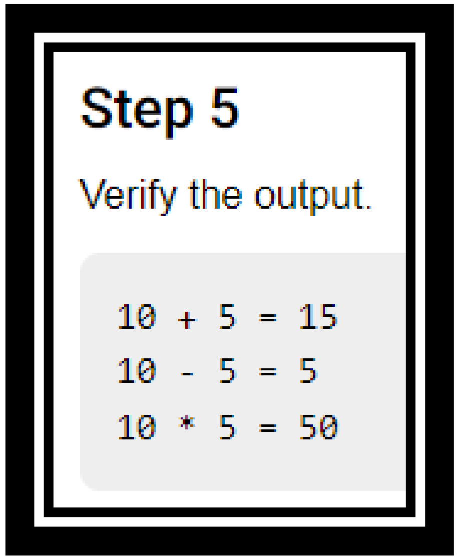
```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
```


Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```


StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```


Step 5

Verify the output.

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

Difference between Factory and Strategy

- The Factory Pattern.
  - Create concrete instances only. Different arguments may result in different objects. It depends on the logic etc.

- The Strategy Pattern.
  - Encapsulate the algorithm (steps) to perform an action. So you can change the strategy and use another algorithm.

Observer Pattern

- The Observer Design Pattern maintains one-to-many dependency between Subject (Observable) and its dependents (Observer) in such a way that whenever state of Subject changes, its dependents get notified.

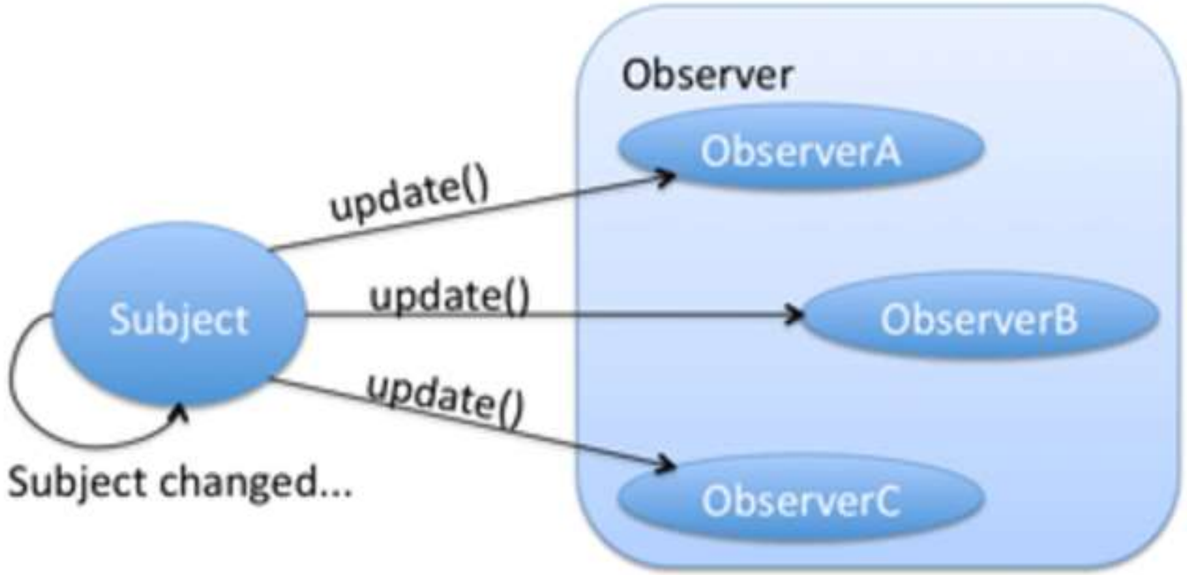
- The Observer Design Pattern is a design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

- The Observer Design Pattern is used when we like to have notification upon changes in the objects state.

- The Observer Design Pattern is one of twenty-three well known Gang of Four design patterns that defines an one-to-many dependency objects so that when one object changes state, all of its dependents get notified and updated automatically.

Real world example of observer pattern

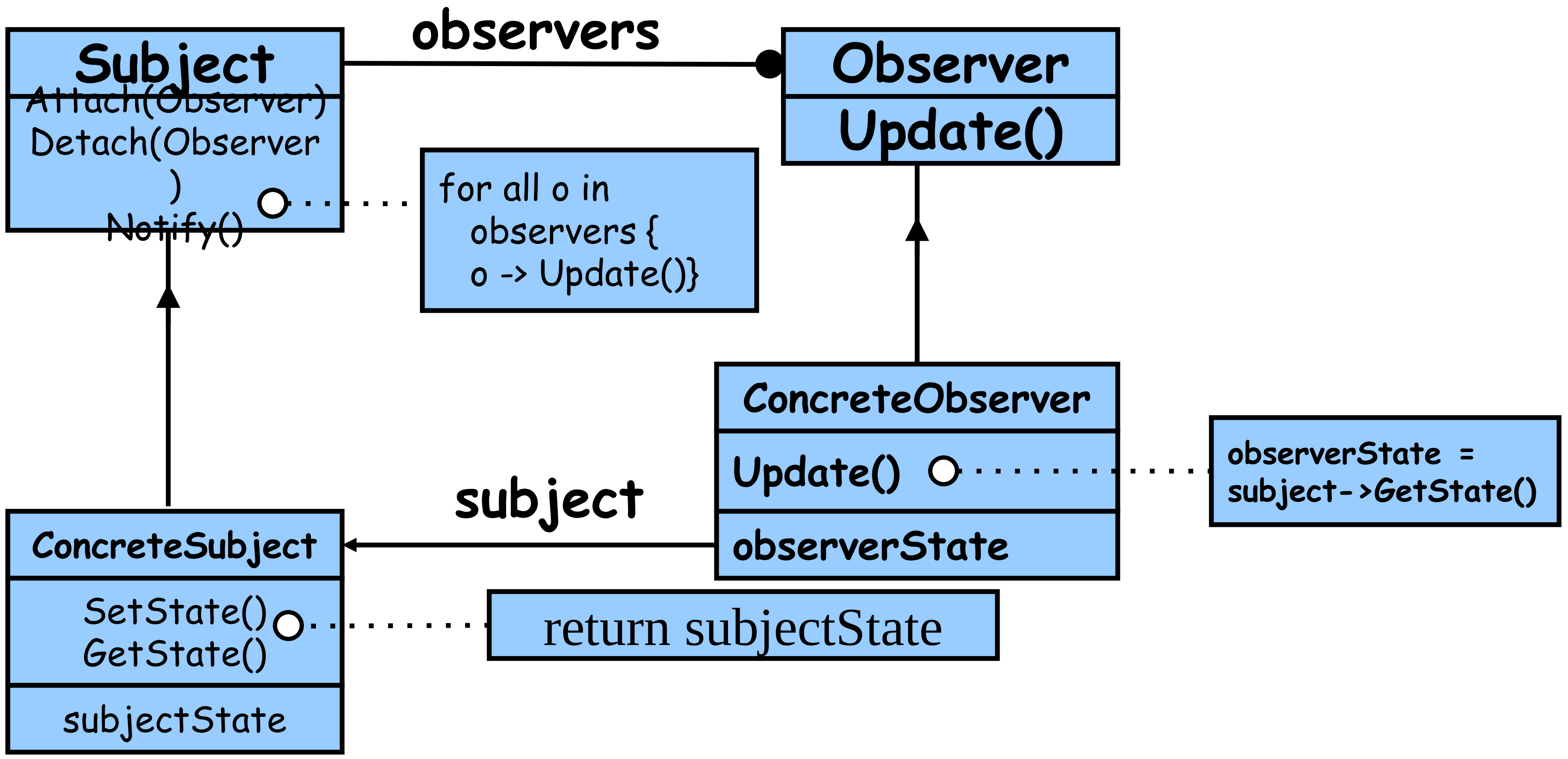
- A real world example of observer pattern can be any social media platform such as Facebook or twitter. When a person updates his status – all his followers gets the notification.
- A follower can follow or unfollow another person at any point of time. Once unfollowed, person will not get the notifications from subject in future.

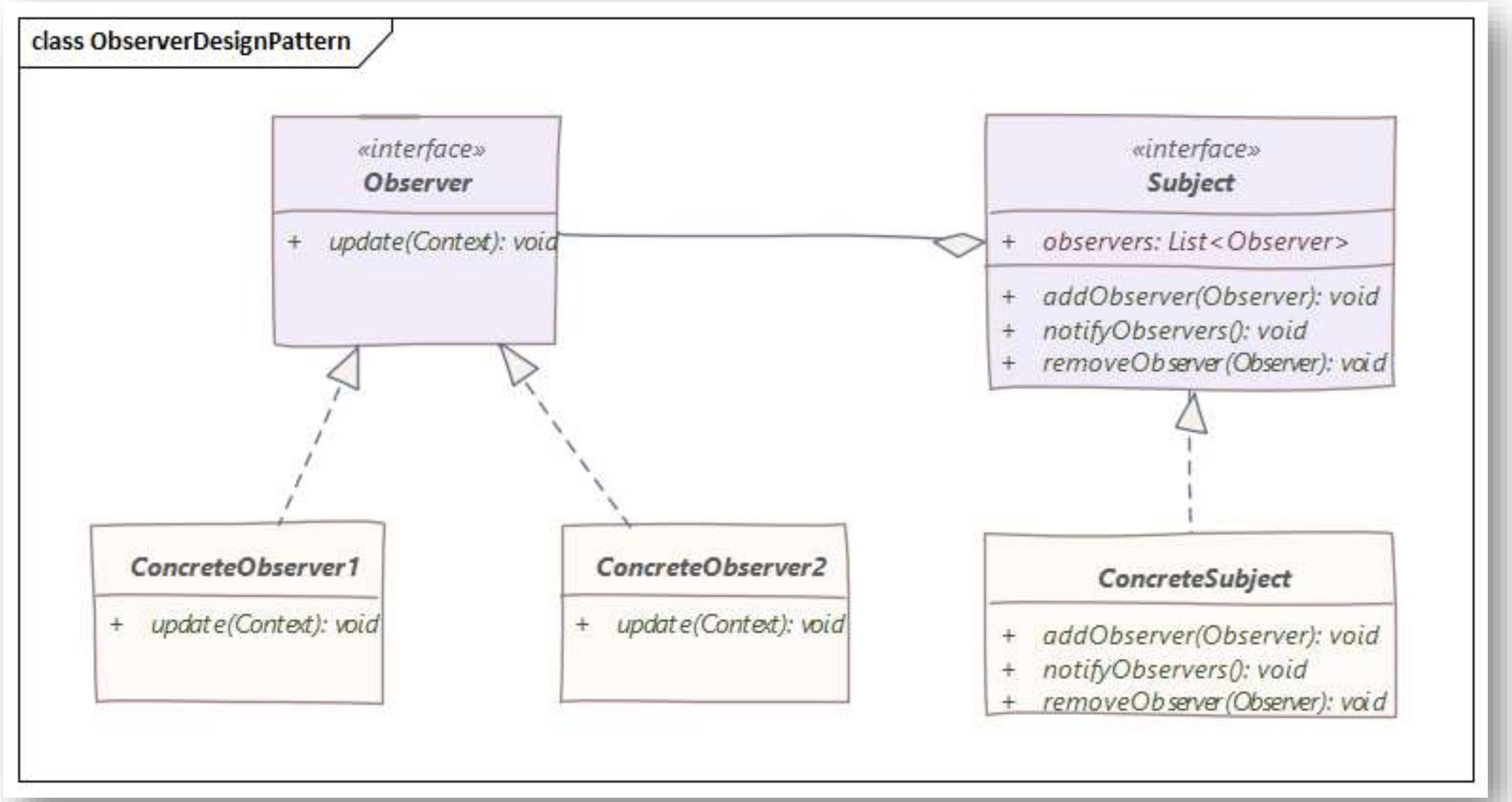


Subject & Observer

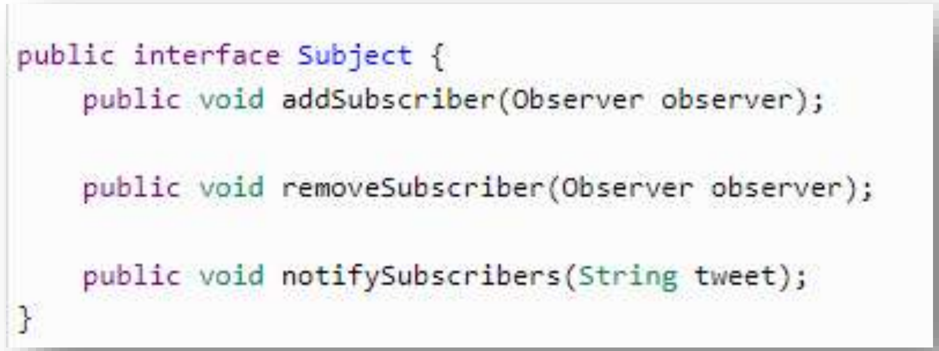
- **Subject**
  - the object which will frequently change its state and upon which other objects depend
- **Observer**
  - the object which depends on a subject and updates according to its subject's state.

Observer Pattern - UML

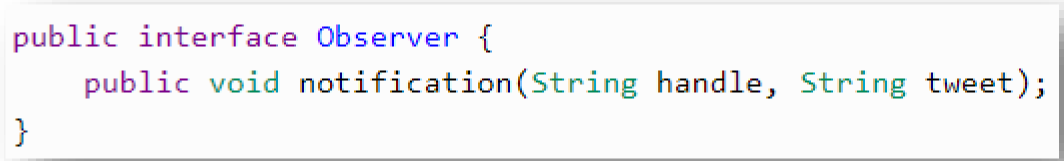



class ObserverDesignPattern
«interface»
Observer
+ update(Context): void
«interface»
Subject
+ observers: List<Observer>
+ addObserver(Observer): void
+ notifyObservers(): void
+ removeObserver(Observer): void
ConcreteObserver1
+ update(Context): void
ConcreteObserver2
+ update(Context): void
ConcreteSubject
+ addObserver(Observer): void
+ notifyObservers(): void
+ removeObserver(Observer): void

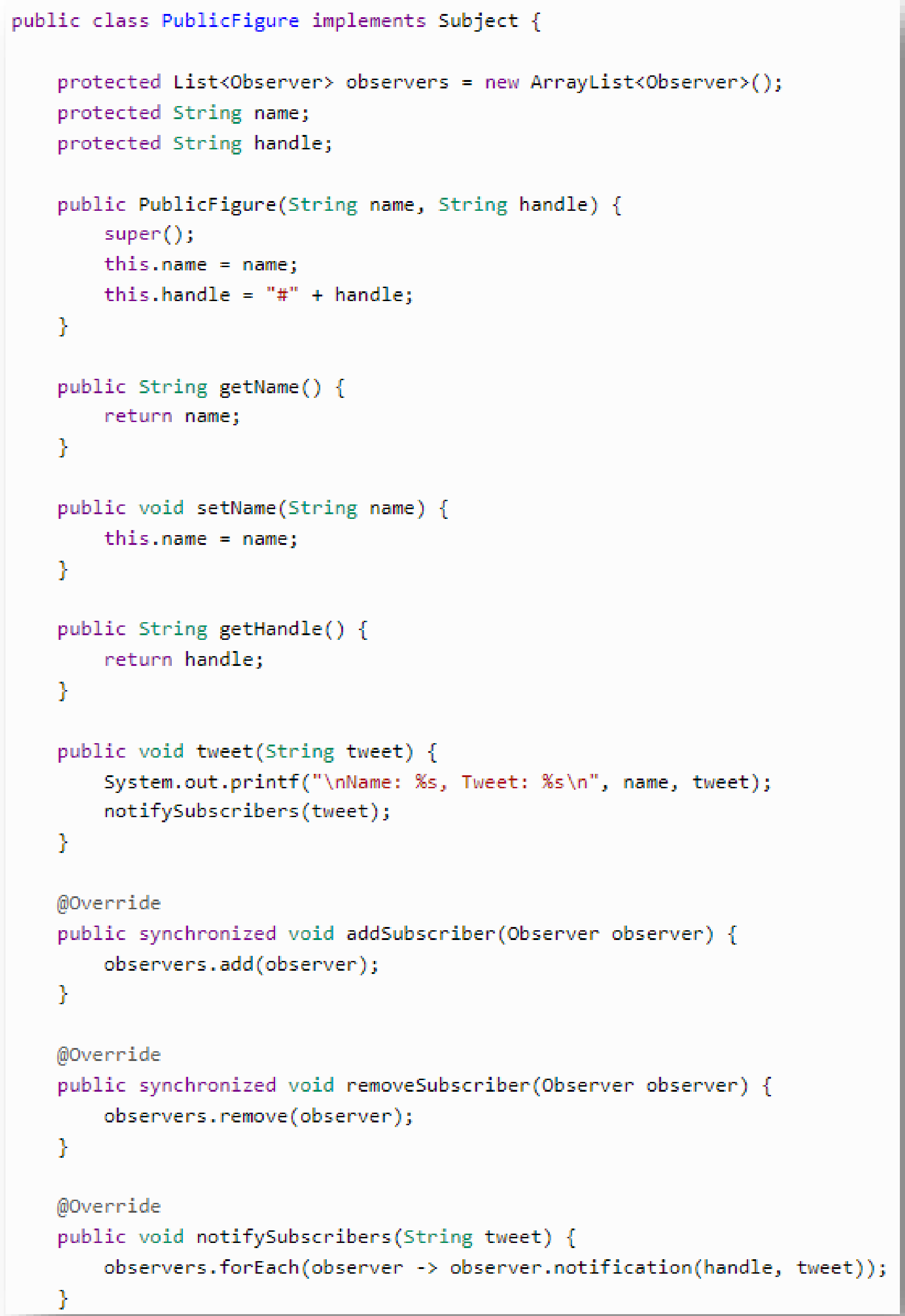
Suppose there are some public figures like politicians or celebrities for which there are some followers. Whenever these public figures do any tweet, there registered followers get the notification on that.



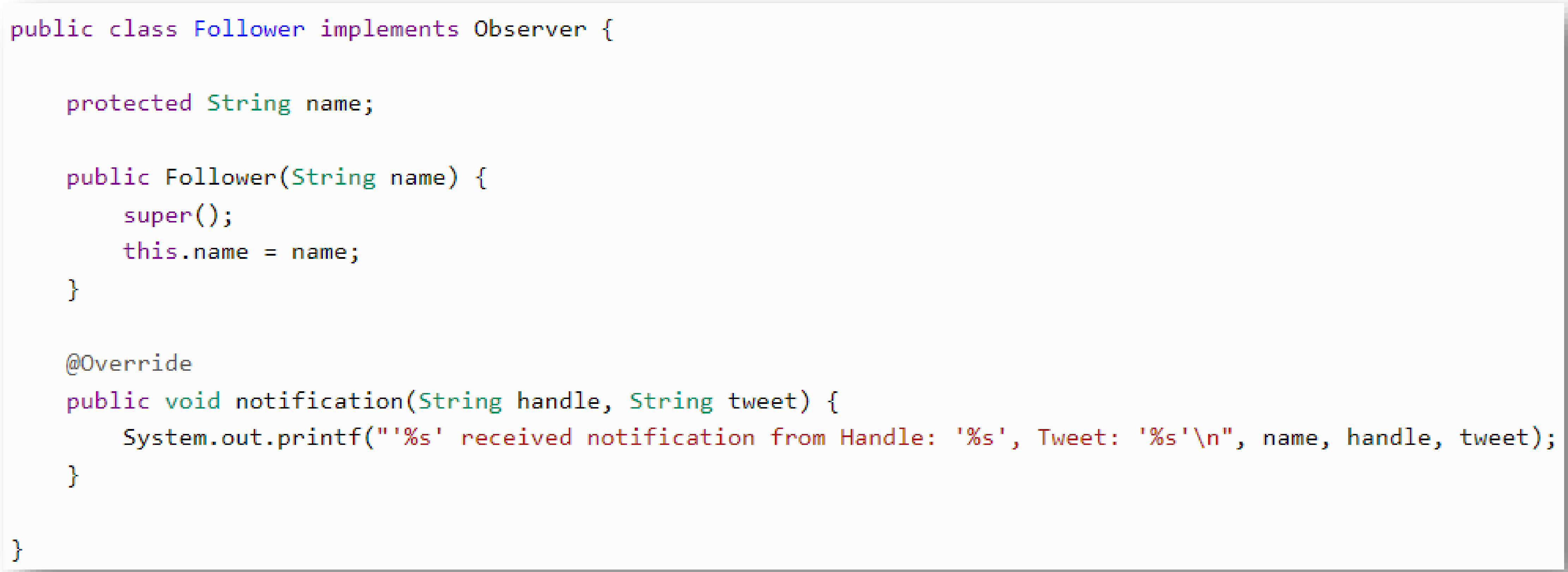
```java
public interface Subject {
    public void addSubscriber(Observer observer);

    public void removeSubscriber(Observer observer);

    public void notifySubscribers(String tweet);
}
```



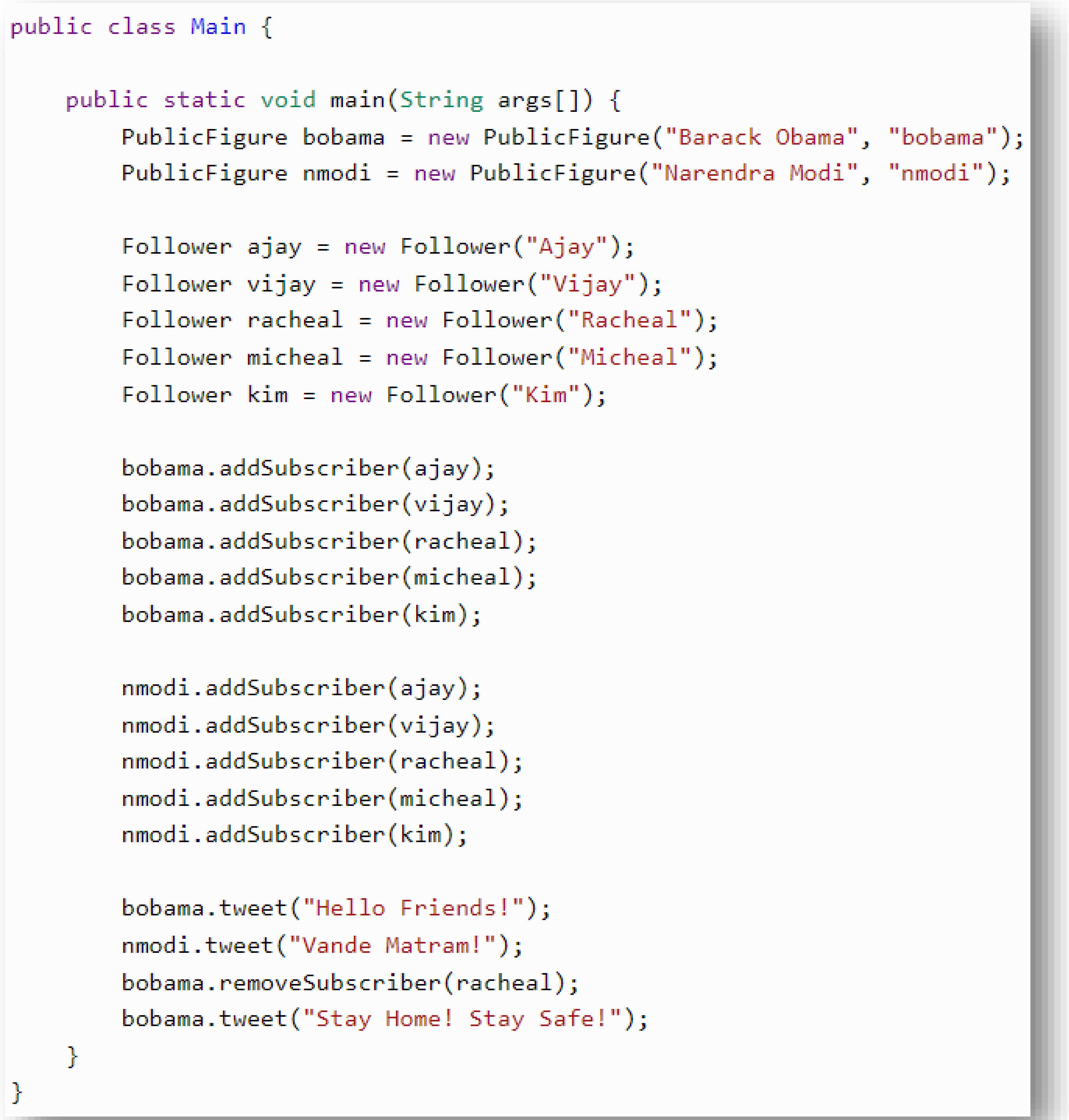
```java
public interface Observer {
    public void notification(String handle, String tweet);
}
```



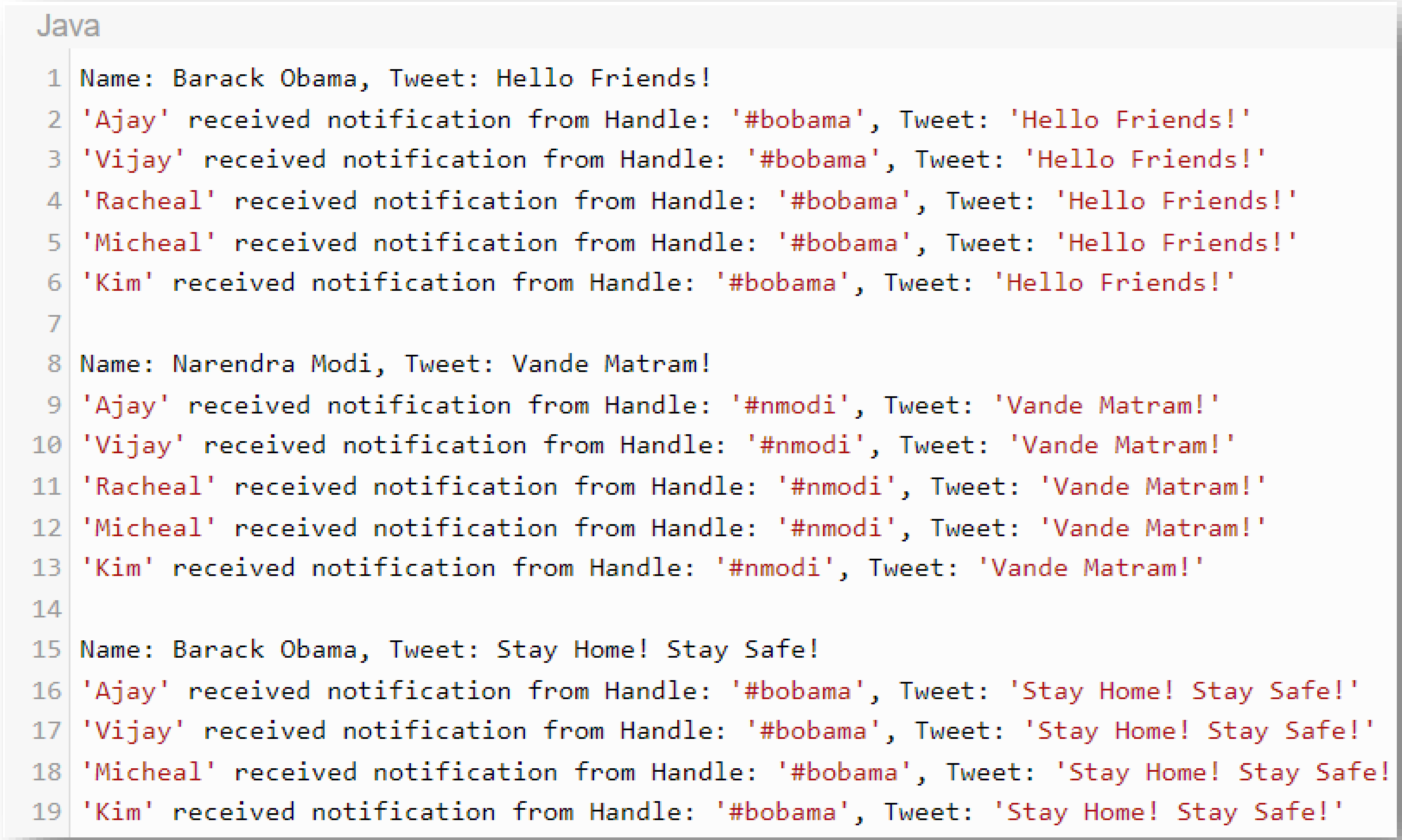
```java
public class PublicFigure implements Subject {

    protected List<Observer> observers = new ArrayList<Observer>();
    protected String name;
    protected String handle;

    public PublicFigure(String name, String handle) {
        super();
        this.name = name;
        this.handle = "#" + handle;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getHandle() {
        return handle;
    }

    public void tweet(String tweet) {
        System.out.printf("\nName: %s, Tweet: %s\n", name, tweet);
        notifySubscribers(tweet);
    }

    @Override
    public synchronized void addSubscriber(Observer observer) {
        observers.add(observer);
    }

    @Override
    public synchronized void removeSubscriber(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifySubscribers(String tweet) {
        observers.forEach(observer -> observer.notification(handle, tweet));
    }
```



```
public class Follower implements Observer {

    protected String name;

    public Follower(String name) {
        super();
        this.name = name;
    }

    @Override
    public void notification(String handle, String tweet) {
        System.out.printf("'%s' received notification from Handle: '%s', Tweet: '%s'\n", name, handle, tweet);
    }

}
```



```java
public class Main {

    public static void main(String args[]) {
        PublicFigure bobama = new PublicFigure("Barack Obama", "bobama");
        PublicFigure nmodi = new PublicFigure("Narendra Modi", "nmodi");

        Follower ajay = new Follower("Ajay");
        Follower vijay = new Follower("Vijay");
        Follower racheal = new Follower("Racheal");
        Follower micheal = new Follower("Micheal");
        Follower kim = new Follower("Kim");

        bobama.addSubscriber(ajay);
        bobama.addSubscriber(vijay);
        bobama.addSubscriber(racheal);
        bobama.addSubscriber(micheal);
        bobama.addSubscriber(kim);

        nmodi.addSubscriber(ajay);
        nmodi.addSubscriber(vijay);
        nmodi.addSubscriber(racheal);
        nmodi.addSubscriber(micheal);
        nmodi.addSubscriber(kim);

        bobama.tweet("Hello Friends!");
        nmodi.tweet("Vande Matram!");
        bobama.removeSubscriber(racheal);
        bobama.tweet("Stay Home! Stay Safe!");
    }
}
```


Java

```
Name: Barack Obama, Tweet: Hello Friends!
'Ajay' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Vijay' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Racheal' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Micheal' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'
'Kim' received notification from Handle: '#bobama', Tweet: 'Hello Friends!'

Name: Narendra Modi, Tweet: Vande Matram!
'Ajay' received notification from Handle: '#nmodi', Tweet: 'Vande Matram!'
'Vijay' received notification from Handle: '#nmodi', Tweet: 'Vande Matram!'
'Racheal' received notification from Handle: '#nmodi', Tweet: 'Vande Matram!'
'Micheal' received notification from Handle: '#nmodi', Tweet: 'Vande Matram!'
'Kim' received notification from Handle: '#nmodi', Tweet: 'Vande Matram!'

Name: Barack Obama, Tweet: Stay Home! Stay Safe!
'Ajay' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay Safe!'
'Vijay' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay Safe!'
'Micheal' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay Safe!
'Kim' received notification from Handle: '#bobama', Tweet: 'Stay Home! Stay Safe!'
```

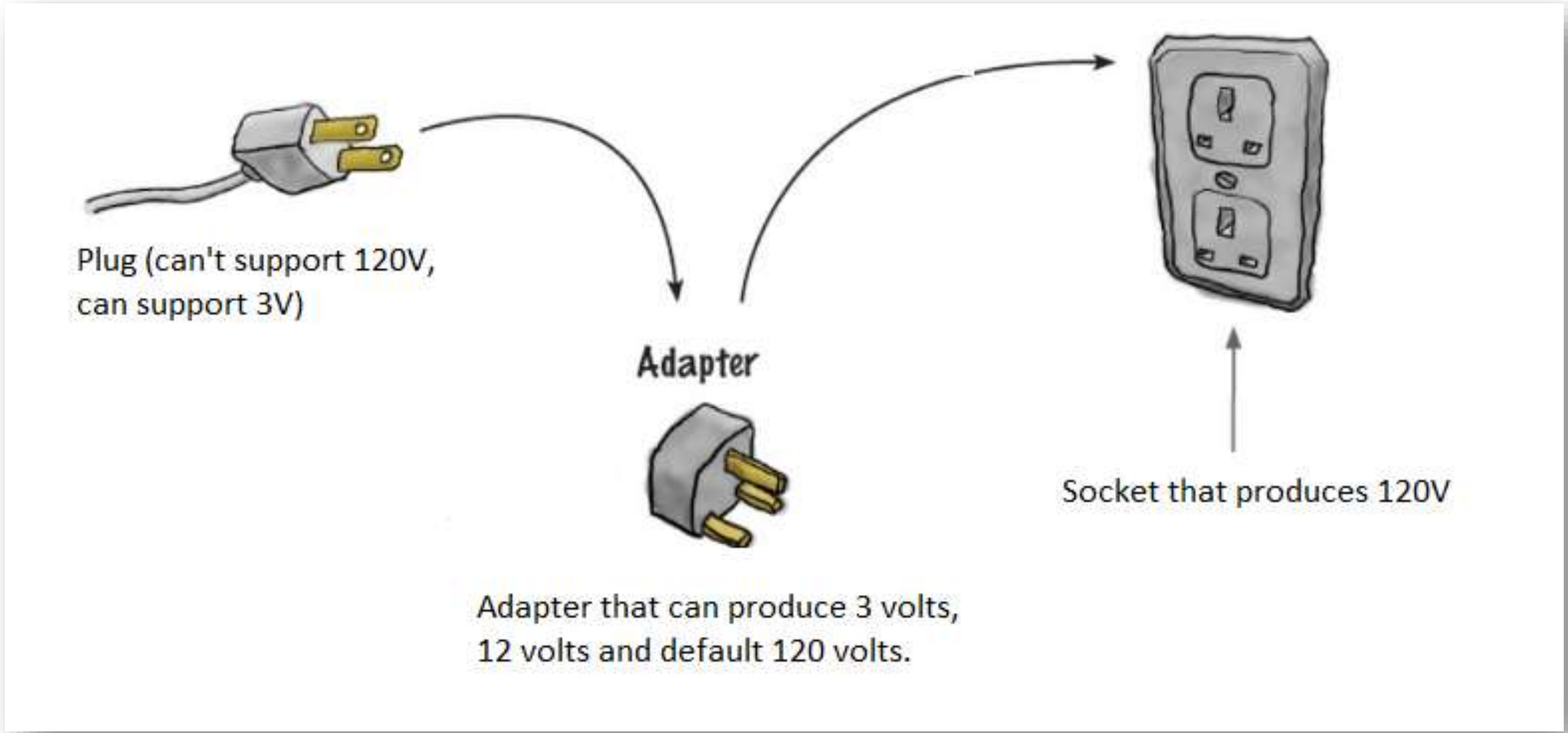
Adapter

- 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.
- It can convert the interface of a class, to make it compatible with a client who is expecting a different interface, without changing the source code of the class.
- The Adapter Pattern is also known as Wrapper.

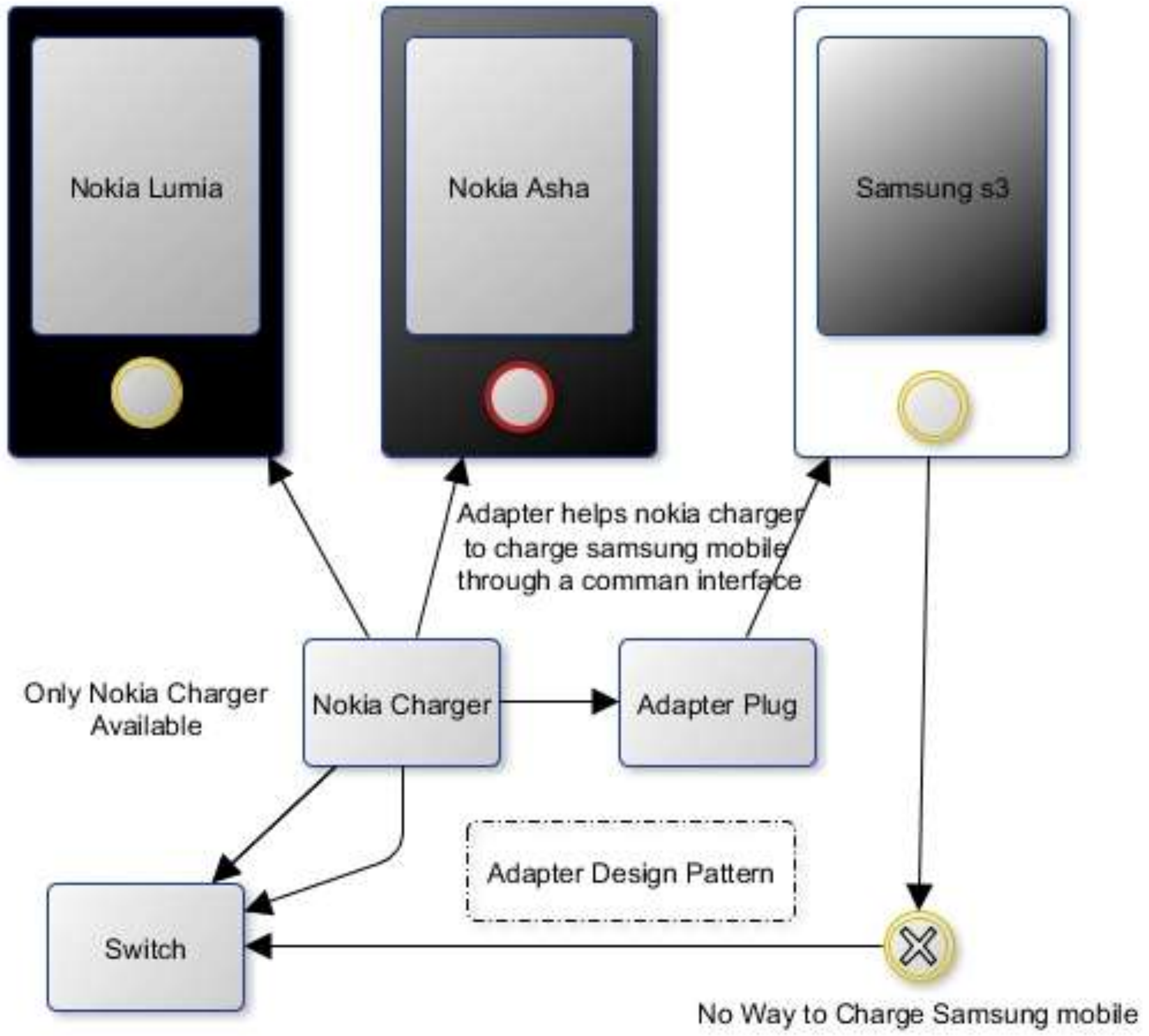
Examples

- Like we have a card reader adapter which acts as a link between our computer and memory card.
- Another example will be a charging adapter that acts as a link between a device and the charging port.

Examples


Plug (can't support 120V,
can support 3V)
Adapter
Adapter that can produce 3 volts,
12 volts and default 120 volts.
Socket that produces 120V

Examples


Nokia Lumia
Nokia Asha
Samsung s3
Adapter helps nokia charger to charge samsung mobile through a comman interface
Only Nokia Charger Available
Nokia Charger
Adapter Plug
Adapter Design Pattern
Switch
No Way to Charge Samsung mobile

Examples

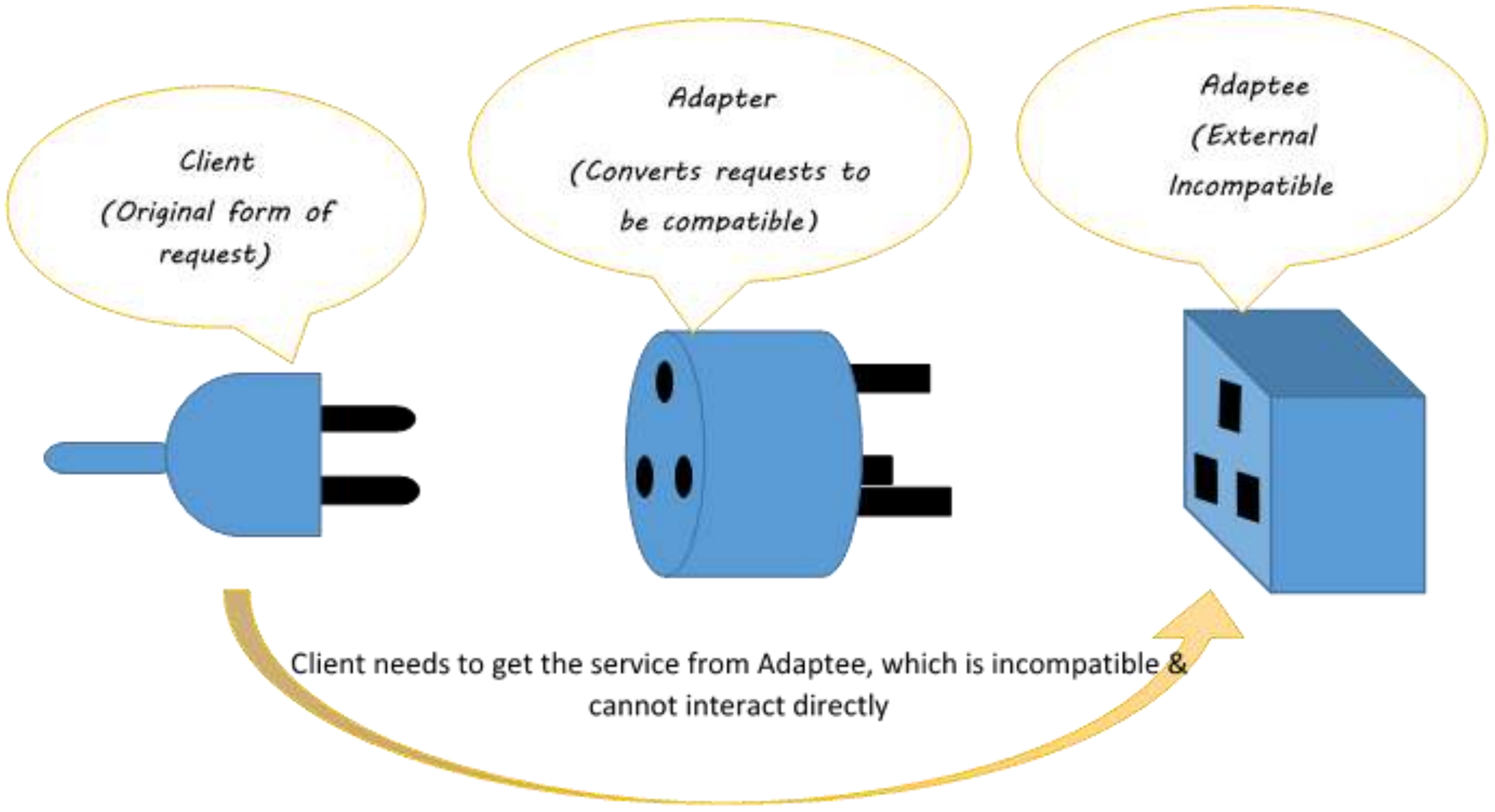
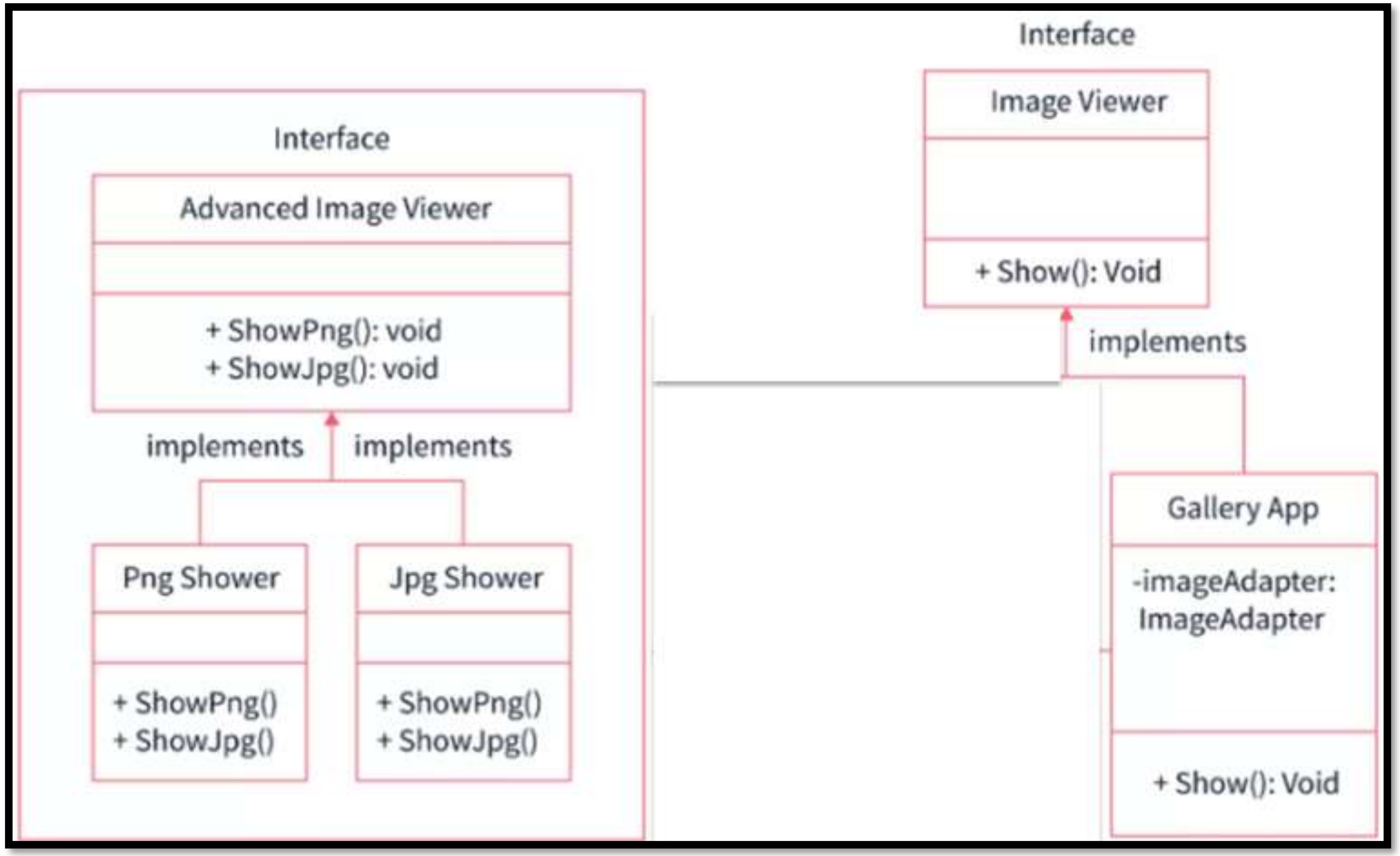


Figure 1-Adapter Pattern Concept

Problem

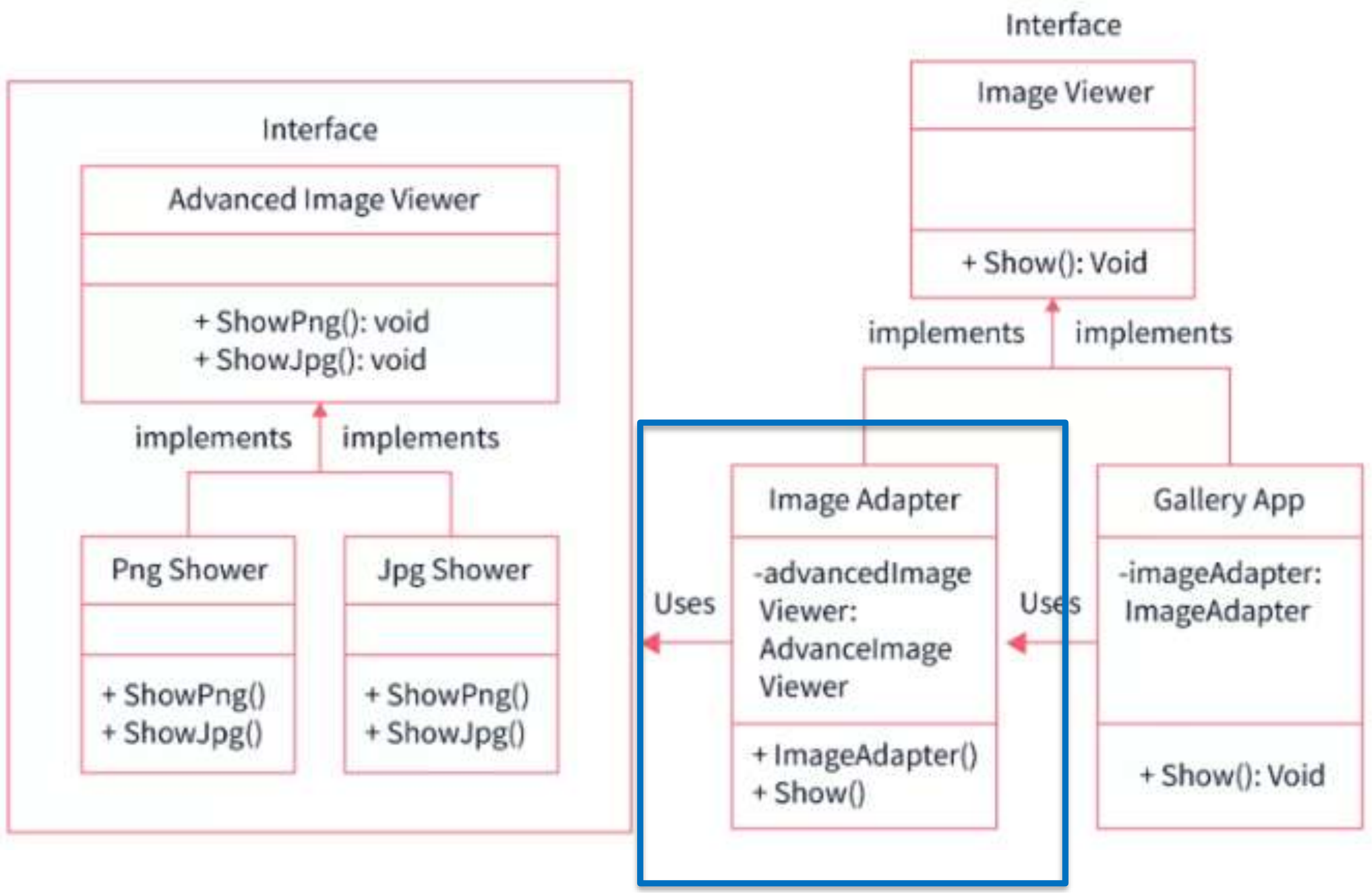
- Let us say we have an image showing application that can show only jpeg images, but we want to use an advanced image application that is capable of showing (.png) and (.jpg) files too. So we can use Adapter Design Pattern here to act as a link between these two different interfaces.

AdvancedImageViewer and concrete classes implementing this interface.These classes can show png and jpg files by default.



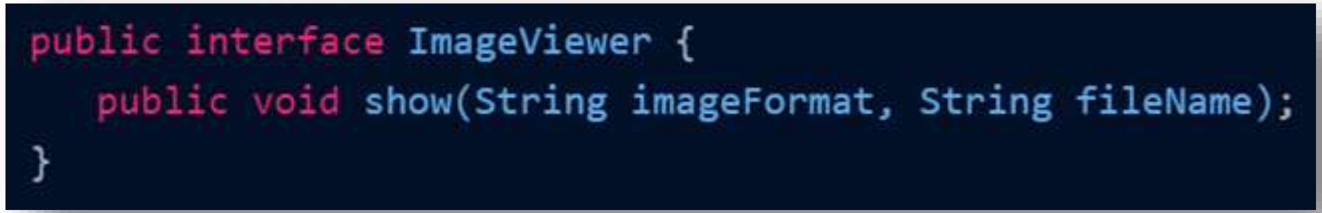
We have an ImageViewer interface and a concrete class named GalleryApp which is implementing the ImageViewer interface. GalleryApp can show jpeg files by default.

- Now we want our GalleryApp class to be able to show png and jpg files without changing the source code of the ImageViewer interface.

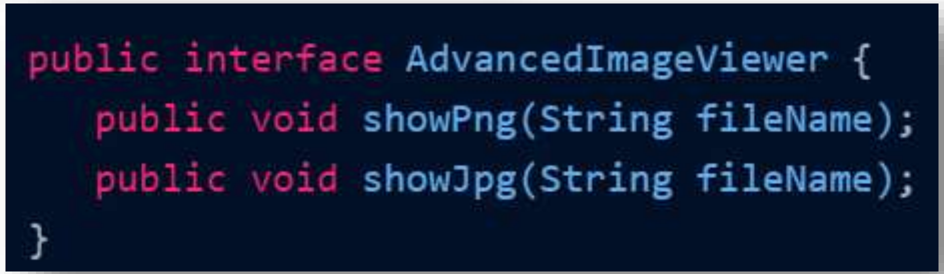


- So we can use an Adapter design pattern to achieve this. We will create a ImageAdapter which implements ImageViewer and uses a AdvancedImageViewer object to show the desired image format.

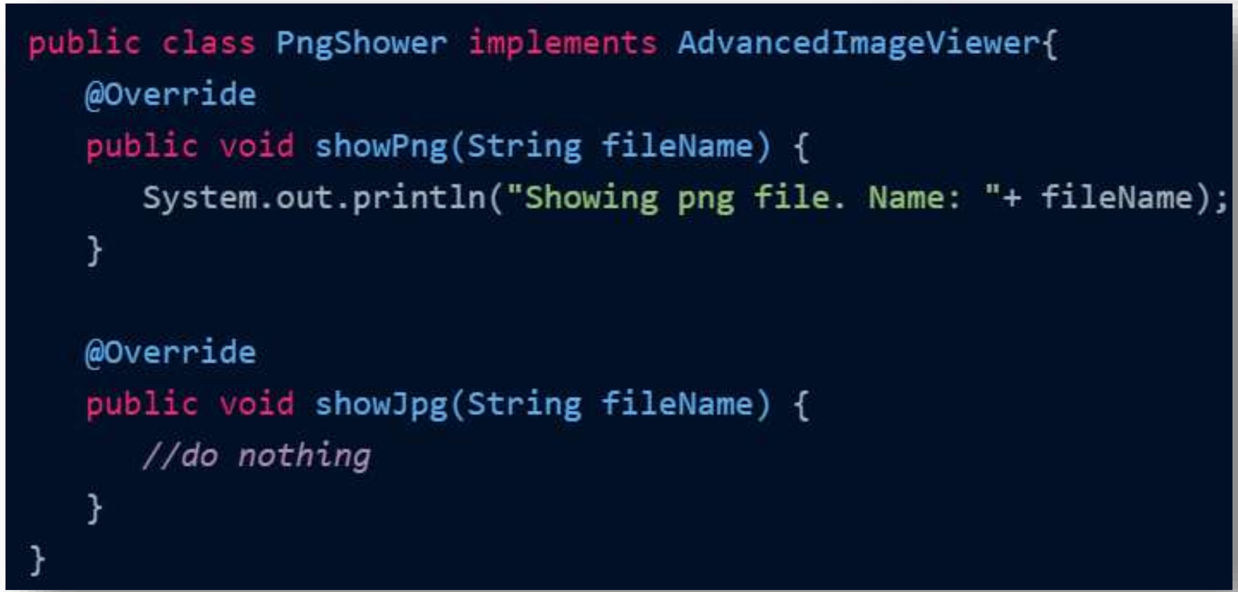
- Our Gallery application will use the ImageAdapter and simply pass the desired image format without worrying about which class can show what image format.



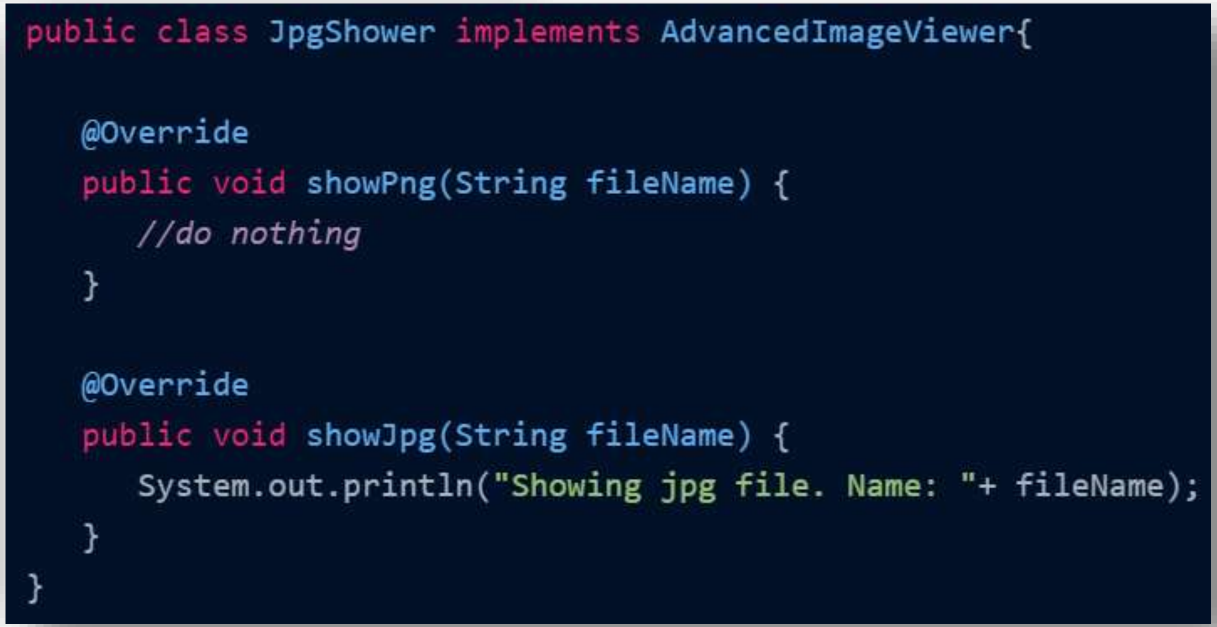
```java
public interface ImageViewer {
   public void show(String imageFormat, String fileName);
}
```



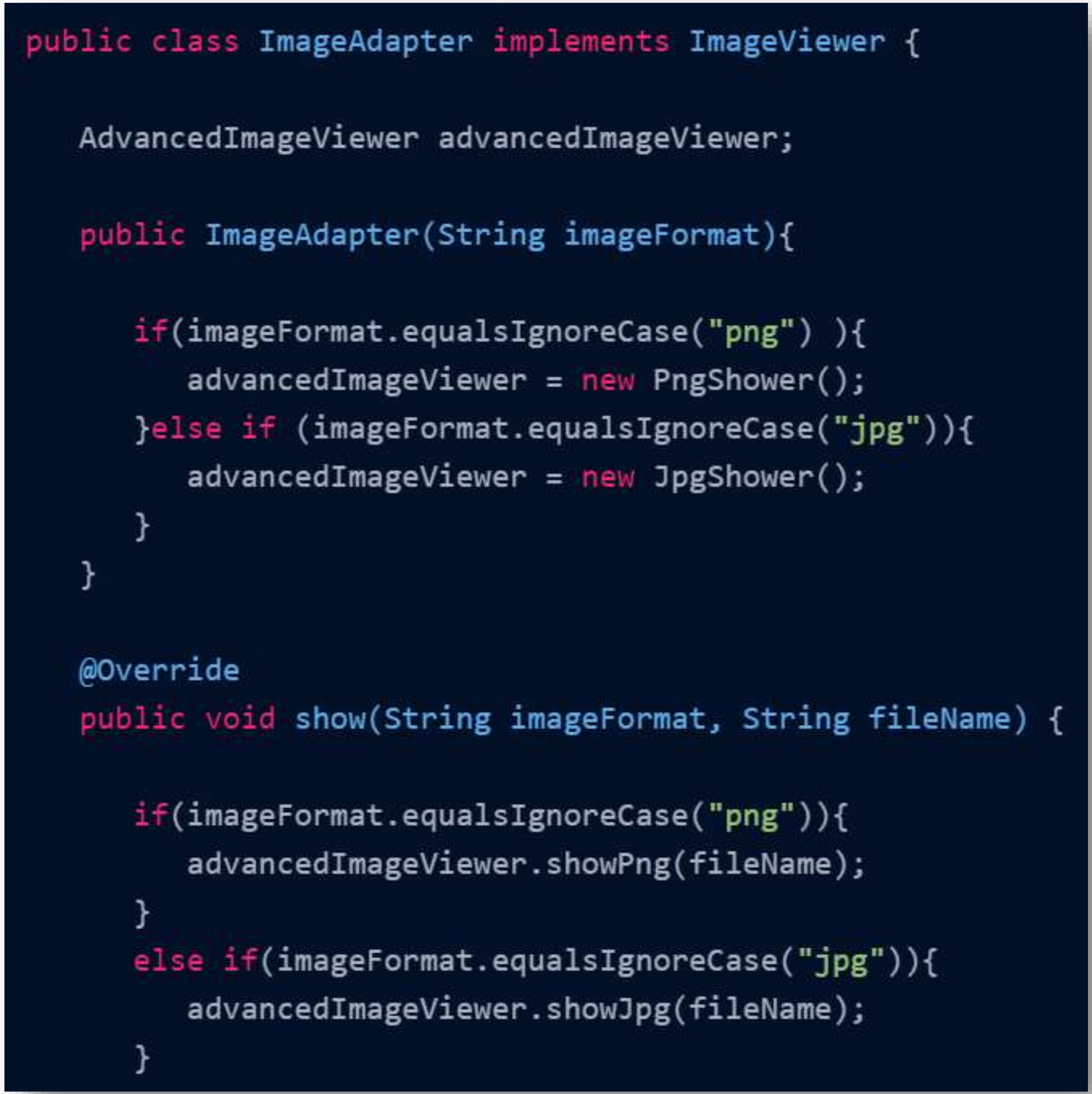
```java
public interface AdvancedImageViewer {
   public void showPng(String fileName);
   public void showJpg(String fileName);
}
```



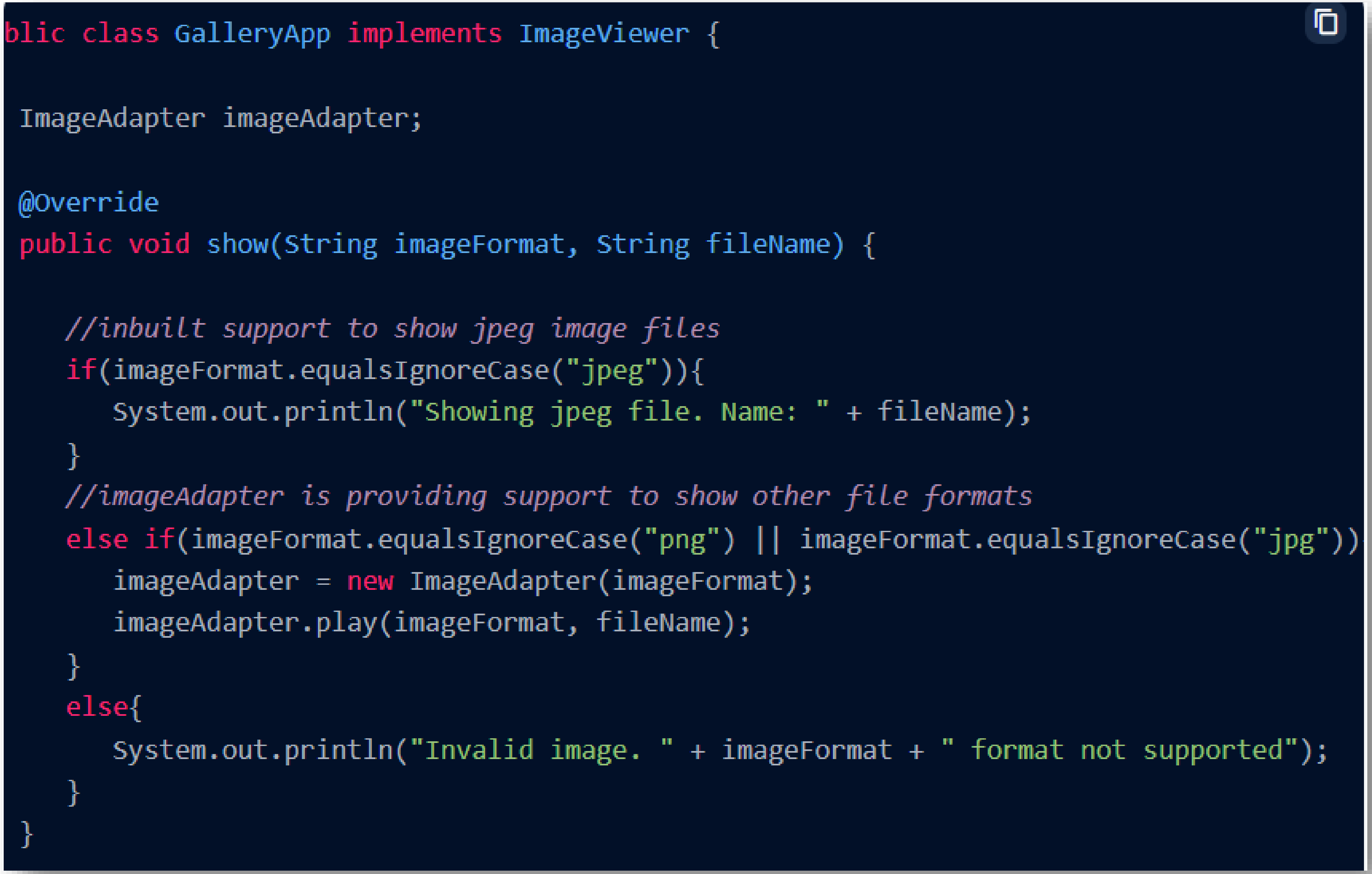
```java
public class PngShower implements AdvancedImageViewer{
   @Override
   public void showPng(String fileName) {
      System.out.println("Showing png file. Name: "+ fileName);
   }

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



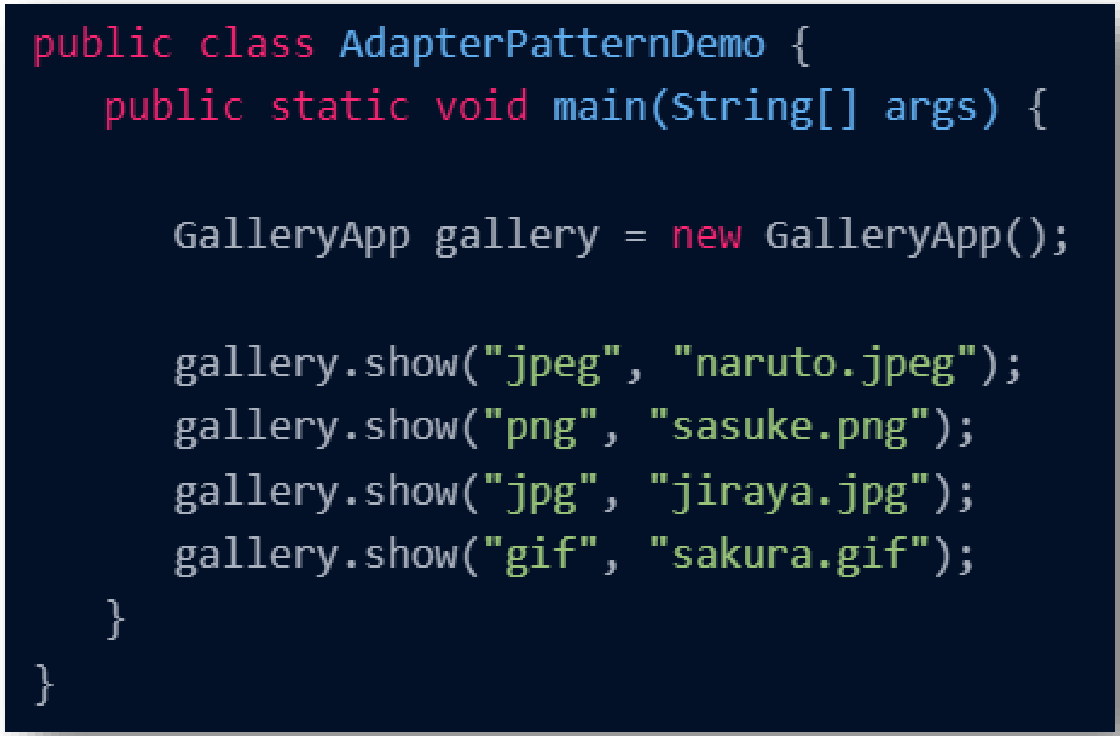
```java
public class JpgShower implements AdvancedImageViewer{

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

   @Override
   public void showJpg(String fileName) {
      System.out.println("Showing jpg file. Name: "+ fileName);
   }
}
```



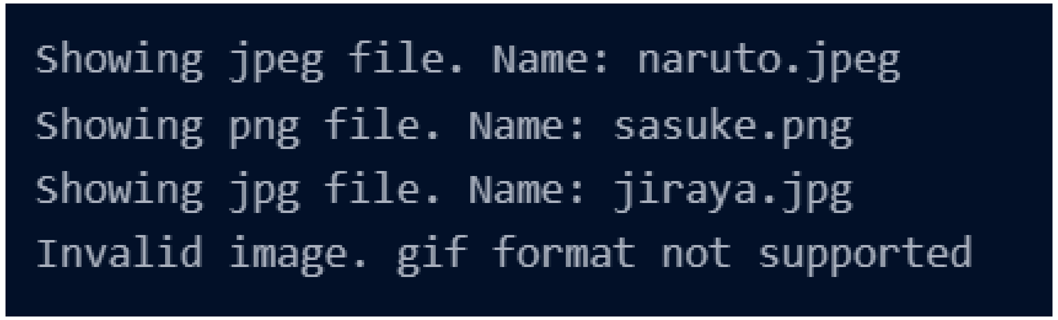
```
public class ImageAdapter implements ImageViewer {

   AdvancedImageViewer advancedImageViewer;

   public ImageAdapter(String imageFormat){

      if(imageFormat.equalsIgnoreCase("png") ){
         advancedImageViewer = new PngShower();
      }else if (imageFormat.equalsIgnoreCase("jpg")){
         advancedImageViewer = new JpgShower();
      }
   }

   @Override
   public void show(String imageFormat, String fileName) {

      if(imageFormat.equalsIgnoreCase("png")){
         advancedImageViewer.showPng(fileName);
      }
      else if(imageFormat.equalsIgnoreCase("jpg")){
         advancedImageViewer.showJpg(fileName);
      }
```



```
blic class GalleryApp implements ImageViewer {

 ImageAdapter imageAdapter;

 @Override
 public void show(String imageFormat, String fileName) {

    //inbuilt support to show jpeg image files
    if(imageFormat.equalsIgnoreCase("jpeg")){
       System.out.println("Showing jpeg file. Name: " + fileName);
    }
    //imageAdapter is providing support to show other file formats
    else if(imageFormat.equalsIgnoreCase("png") || imageFormat.equalsIgnoreCase("jpg"))
       imageAdapter = new ImageAdapter(imageFormat);
       imageAdapter.play(imageFormat, fileName);
    }
    else{
       System.out.println("Invalid image. " + imageFormat + " format not supported");
    }
 }
```



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

      GalleryApp gallery = new GalleryApp();

      gallery.show("jpeg", "naruto.jpeg");
      gallery.show("png", "sasuke.png");
      gallery.show("jpg", "jiraya.jpg");
      gallery.show("gif", "sakura.gif");
   }
}
```



```
Showing jpeg file. Name: naruto.jpeg
Showing png file. Name: sasuke.png
Showing jpg file. Name: jiraya.jpg
Invalid image. gif format not supported
```

Benefits

- **Separation of Concern:** We can separate the interface or data conversion code from the main business logic part of the code.
- **Independence of Code:** We can implement and use the various adapters without breaking the existing client or main code.