JAVA DESIGN PATTERNS

By Ayala Berkovich

JAVA DESIGN PATTERNS

WHAT IS DESIGN PATTERNS?

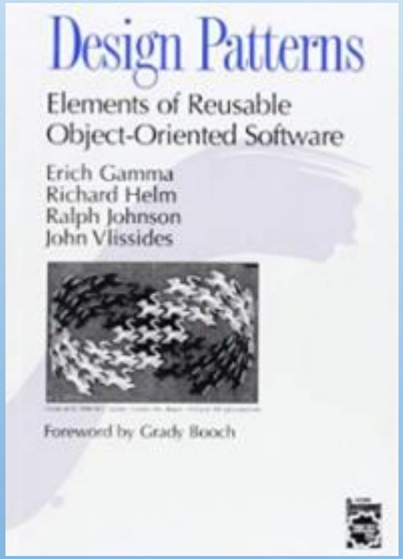
It is solutions already written by some of the advanced and experienced developers while facing and solving similar designing problems.
Patterns are not complete code, but it can use as a template which can be applied to a problem!
Named solution – for simple programmers language.

By Ayala Berkovich

JAVA DESIGN PATTERNS

The idea was introduced by the architect **Christopher Alexander** and has been adapted for various other disciplines, particularly software engineering

Computer scientists :
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
write the most popular design patterns book:



Design Patterns: Elements of Reusable Object-Oriented Software

Book consist 23 design patterns

By Ayala Berkovich

JAVA DESIGN PATTERNS

HOW TO SELECT AND USE ONE?

First, you need to identify the kind of design problem you are facing.

CATEGORIZATION OF PATTERNS

Design patterns can be categorized in the following categories:

- **Creational patterns**
- **Structural patterns**
- **Behavior patterns**

By Ayala Berkovich

JAVA DESIGN PATTERNS

CREATIONAL PATTERNS

Creational design patterns are used to design the instantiation process of **objects(creation of the objects**). The creational pattern uses the inheritance to vary the object creation.

STRUCTURAL PATTERNS

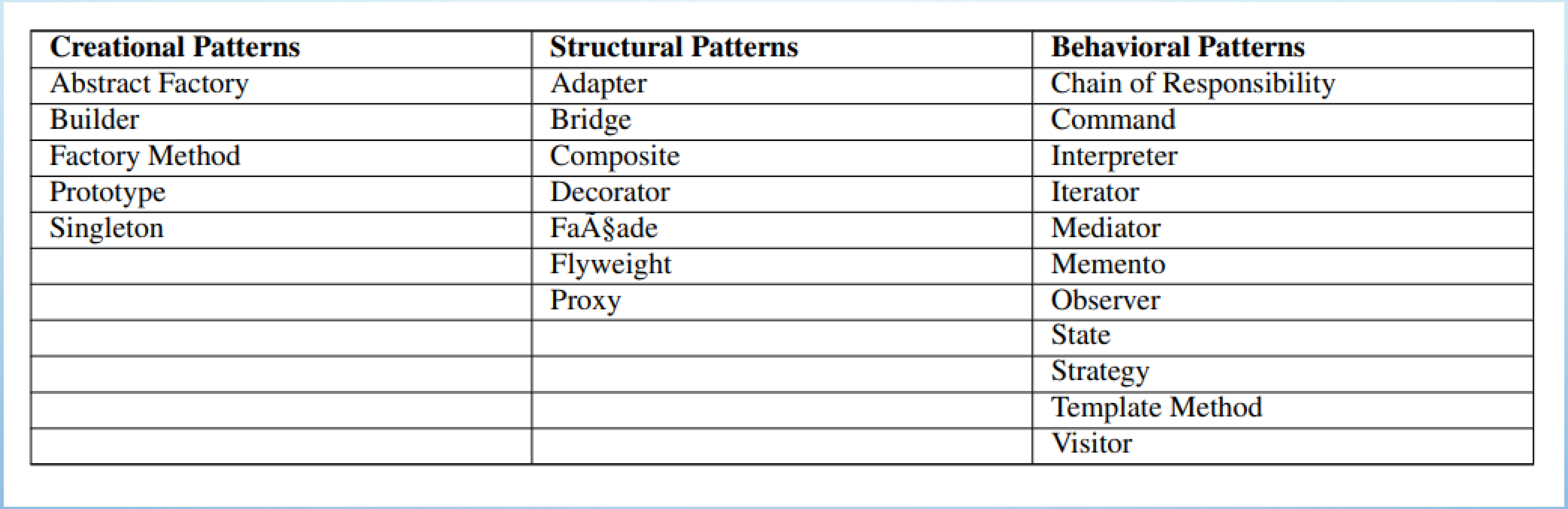
Structural patterns are concerned with how classes and objects are **composed to form larger structures.**

BEHAVIOR PATTERNS

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

By Ayala Berkovich

JAVA DESIGN PATTERNS

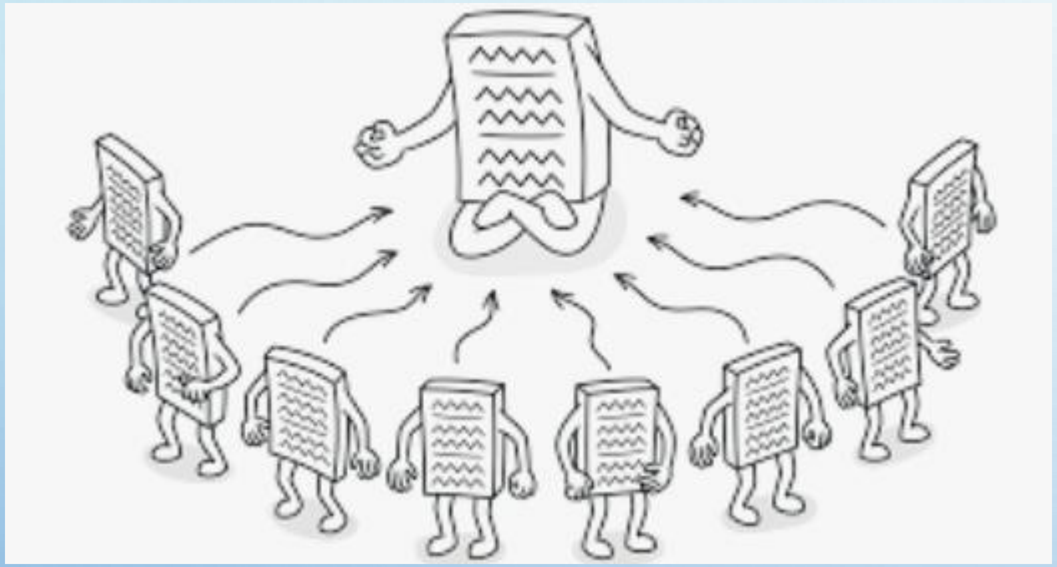


Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	FaÃ§ade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

By Ayala Berkovich

JAVA DESIGN PATTERNS

SINGLETON DESIGN PATTERN – CREATIONAL PATTERN



Sometimes it's important for some classes to have exactly one instance…

WHERE WE CAN USE SINGLETON?

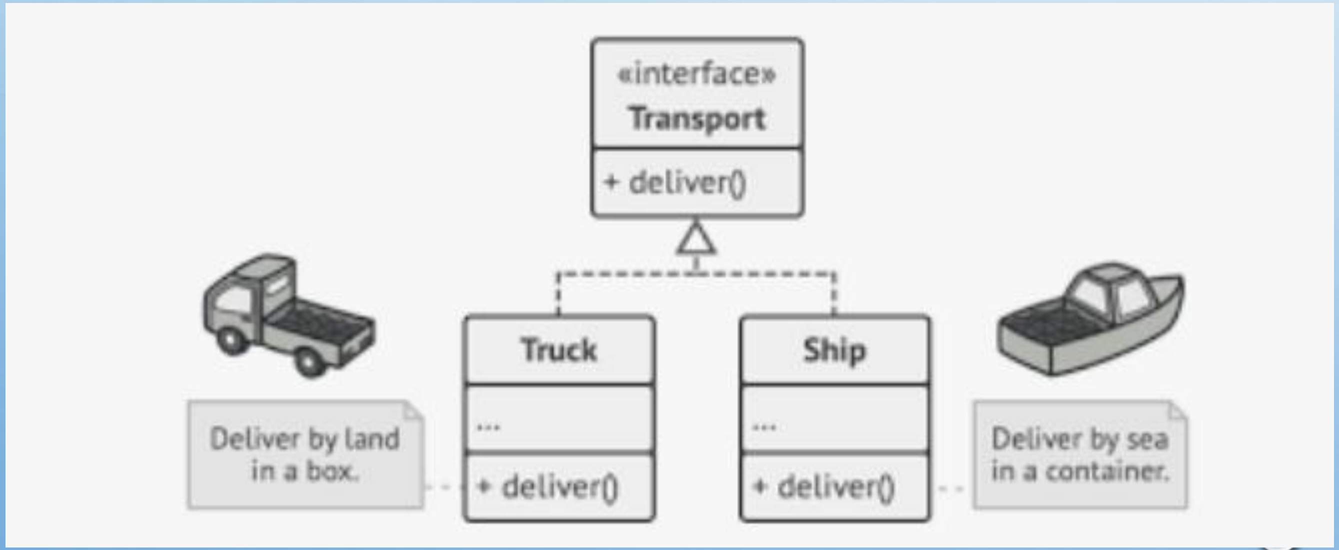
```
public class Singleton {
   private static Singleton sc = null;
   private Singleton(){}
public static Singleton getInstance()
{
        if(sc==null){
           sc = new Singleton();
         }
        return sc;
 }
}

public class Main{
   public static void main (String[] ergs){
      Singleton instance = Singleton.getInstance();
}
}
```

By Ayala Berkovich

JAVA DESIGN PATTERNS

FACTORY DESIGN PATTERN – CREATIONAL PATTERN (CREATION OF INTERFACE FOR OBJECTS CREATION)

The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method.
The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors.



By Ayala Berkovich

JAVA DESIGN PATTERNS

WRITE CODE WHERE WILL BE INPUTTED BY USER COUNTRY NAME AND PROGRAM WILL BACK COUNTRY COIN?

JAVA DESIGN PATTERNS

By Ayala Berkovich

SOLUTION:

1.CREATE INTERFACE:

```
Interface Country_Coin{
   public void getCoin();
}
```

2. CREATE CLASSES:

```
Class USA implements Country_Coin{
public void getCoin(){
System.out.println("Cent");
}}
Class ISRAEL implements Country_Coin{
 public void getCoin(){
System.out.println("Shekel");
}
}
…….
```

3.CREATE FACTORY CLASS:

```
Class Factory{
   public static Country_Coin getCountryCoin(String
country){
If(country.equals("USA"))
   return new USA();
If(country.equals("ISRAEL"))
   return new ISRAEL();
…..
}
```

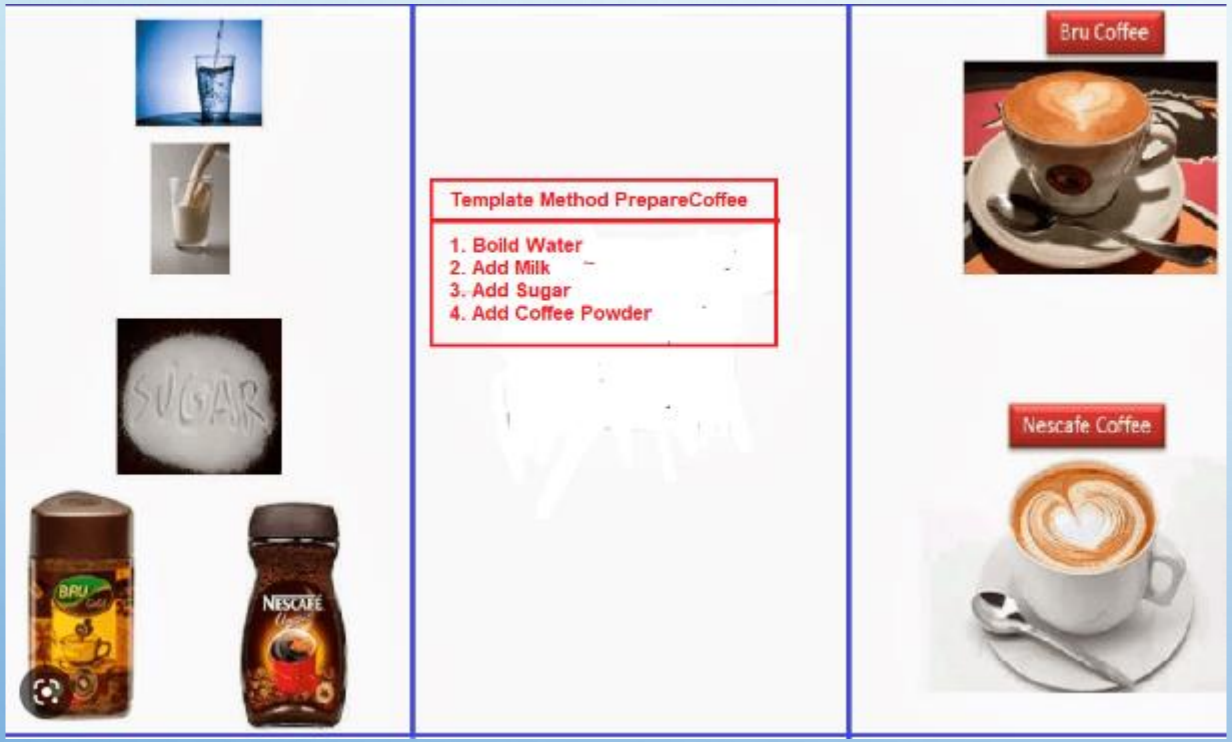
4. MAIN CLASS:

```
main(String[] args)
{
Country_Coin coin = Factory.getCountryCoin("USA");
Coin.getCoin();
}
```

By Ayala Berkovich

JAVA DESIGN PATTERNS

TEMPLATE DESIGN PATTERN – BEHAVIORAL PATTERN

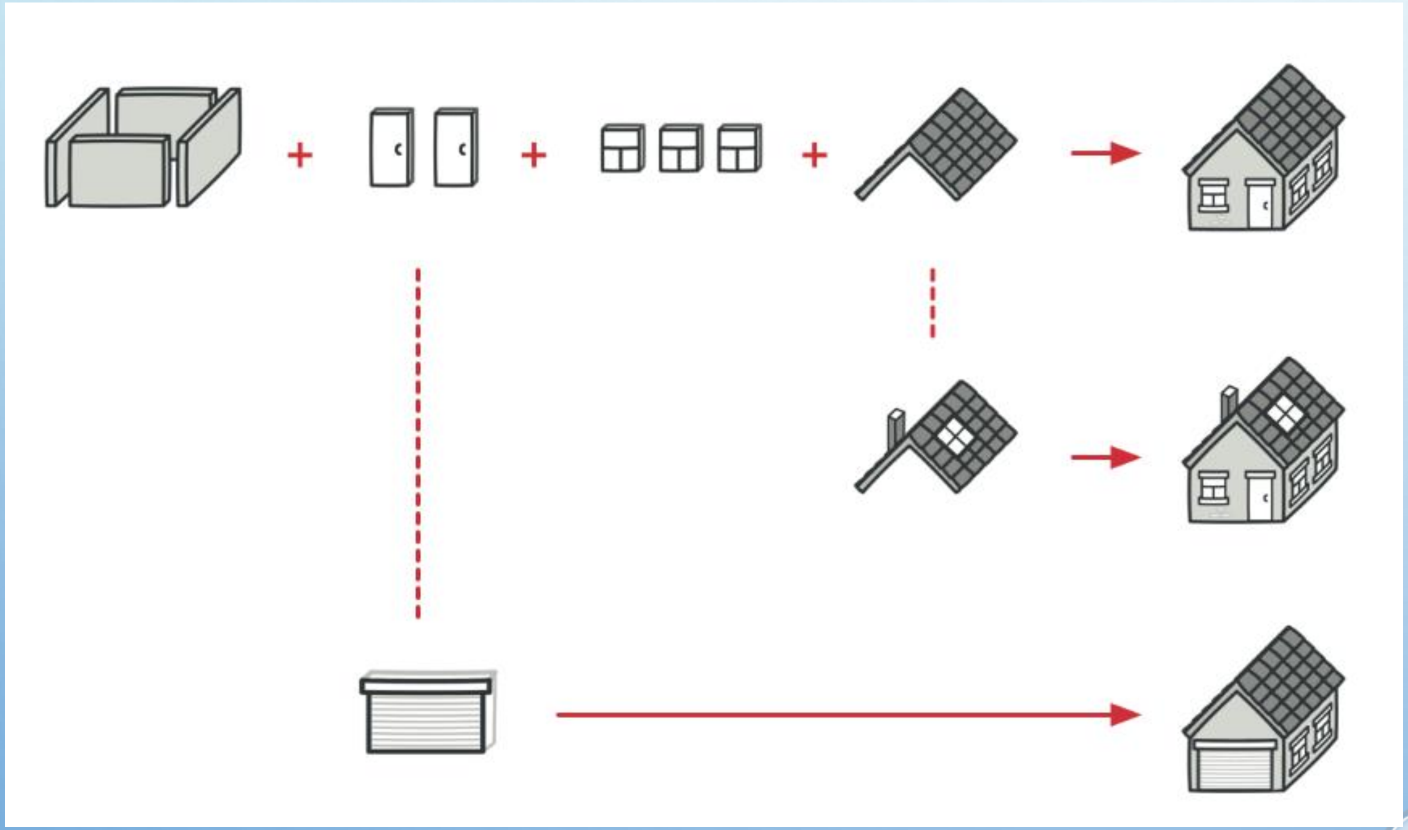


The **Template Pattern** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses to redefine certain steps of an algorithm without changing the algorithm's structure.

By Ayala Berkovich

TEMPLATE DESIGN PATTERN

Another Real-World Analogy:



By Ayala Berkovich

TEMPLATE DESIGN PATTERN

WRITE CODE FOR FASTFOOD SHOP THAT PREPARE:

HOTDOG, HUMBERGR, HOT SOUP, FRENCH TOST?

By Ayala Berkovich

TEMPLATE DESIGN PATTERN

SOLUTION:

1. CREATE ABSTRACT CLASS:

```
abstract Class FastFood {
final void putVegerables(){
  System.out.println(“put vegetables”);
}
final void roastBread(){
System.out.println()(“roast bread”);
}
….

abstract void prepare(){}
}
```

2. CREATE CLASSES:

```
Class HotDog extends FastFood {
void prepare(){
  roastBeaf();
  putSosage();
 putBegetable();
}
}
Class Hamburger extents FastFood{
 void prepare(){
  roastBread();
  putMeat();
putVegetables();
}
}
….
}
```

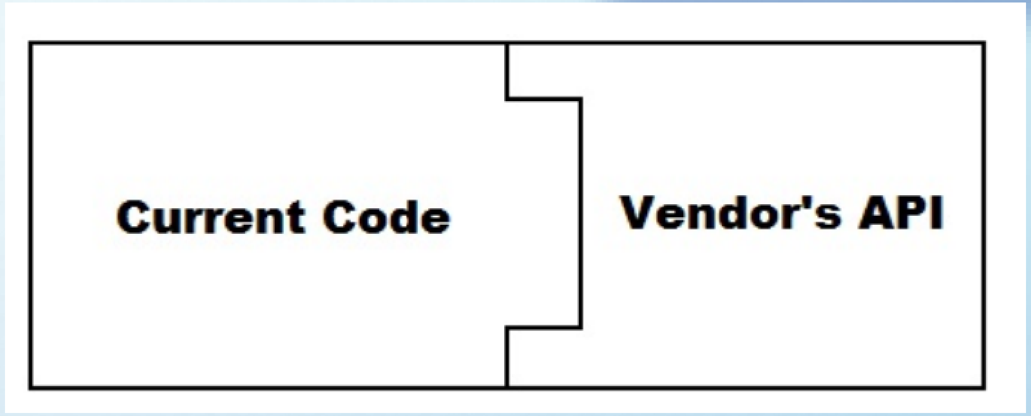
JAVA DESIGN PATTERNS

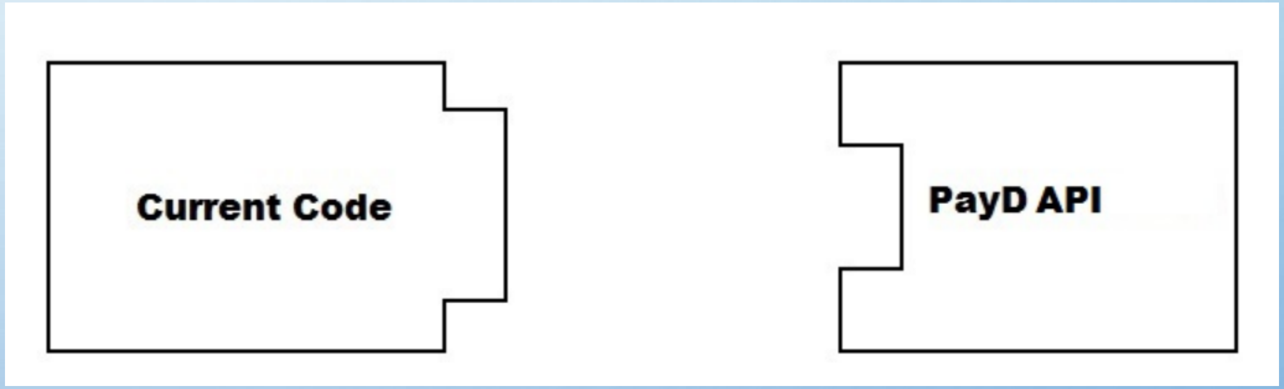
By Ayala Berkovich

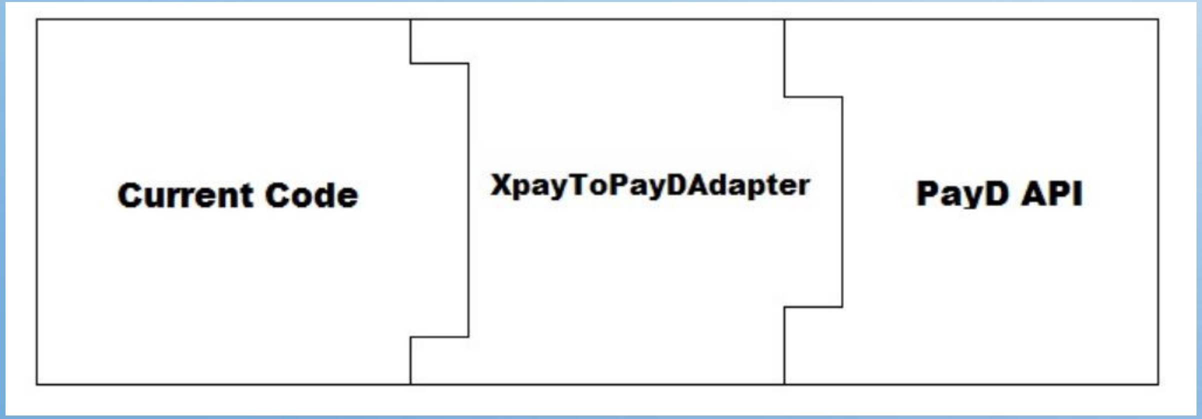
ADAPTER DESIGN PATTERN – STRUCTURAL PATTERN

A software developer has worked on an e-commerce website. The website allows users to shop and pay online. The site is integrated with a 3rd party payment gateway, through which users can pay their bills using their **credit card**.

Everything was going well, until his manager called him for a change in the project. The manager told him that they are planning to change the payment gateway vendor(Like **PayPal**), and he has to implement that in the code.



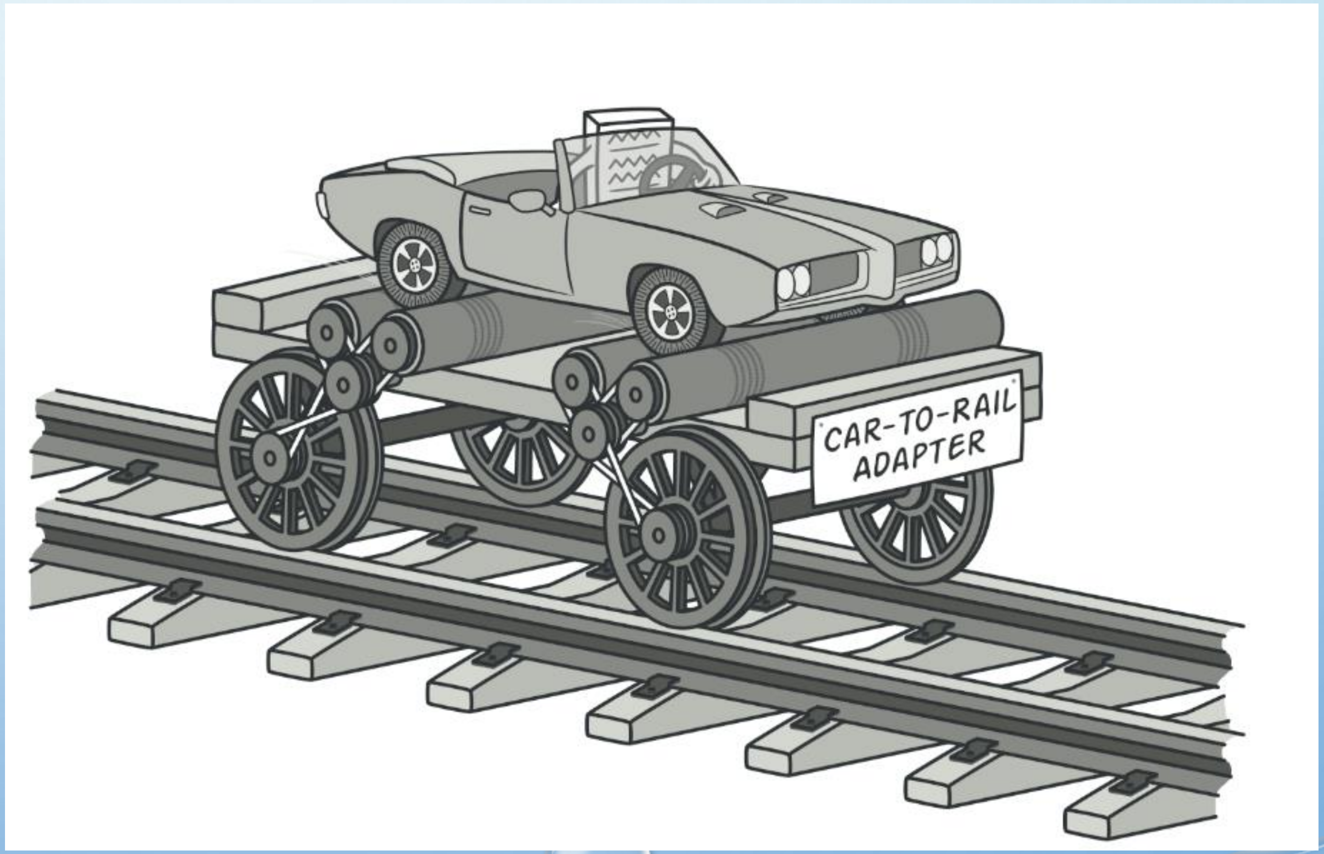




By Ayala Berkovich

ADAPTER DESIGN PATTERN

So it is simple…
Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.



By Ayala Berkovich

ADAPTER DESIGN PATTERNS

Example:

1.Define interface Bird:

```
interface Bird
{


    public void fly();
    public void makeSound();
}
```

2.Define Class Sparrow:

```
class Sparrow implements Bird
{

    public void fly()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}
```

3.Define interface ToyDuck:

```
interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}
```

4.Define Class PlasticToyDuck:

```
class PlasticToyDuck implements ToyDuck
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}
```

By Ayala Berkovich

ADAPTER DESIGN PATTERNS

5.Define Class Adapter:

```java
class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}
```

6.Define Main Class :

```java
class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

Output

```
Sparrow...
Flying
Chirp Chirp
ToyDuck...
Squeak
BirdAdapter...
Chirp Chirp
```

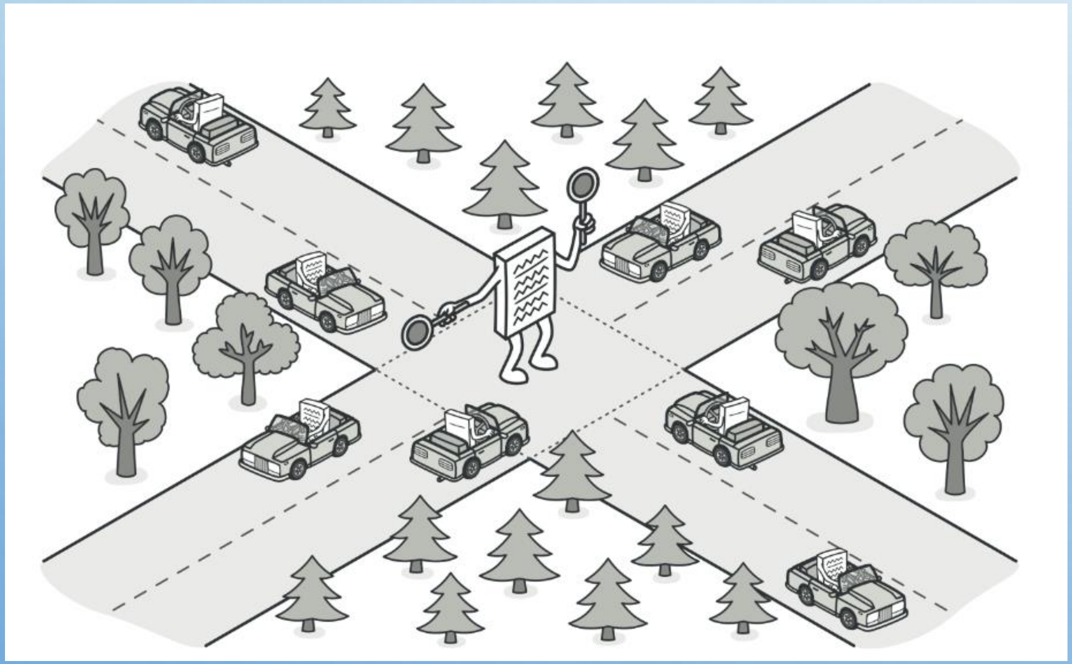
JAVA DESIGN PATTERNS

By Ayala Berkovich

MEDIATOR DESIGN PATTERN – BEHAVIORAL PATTERN

is one of the important and widely used behavioral design pattern

Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer.

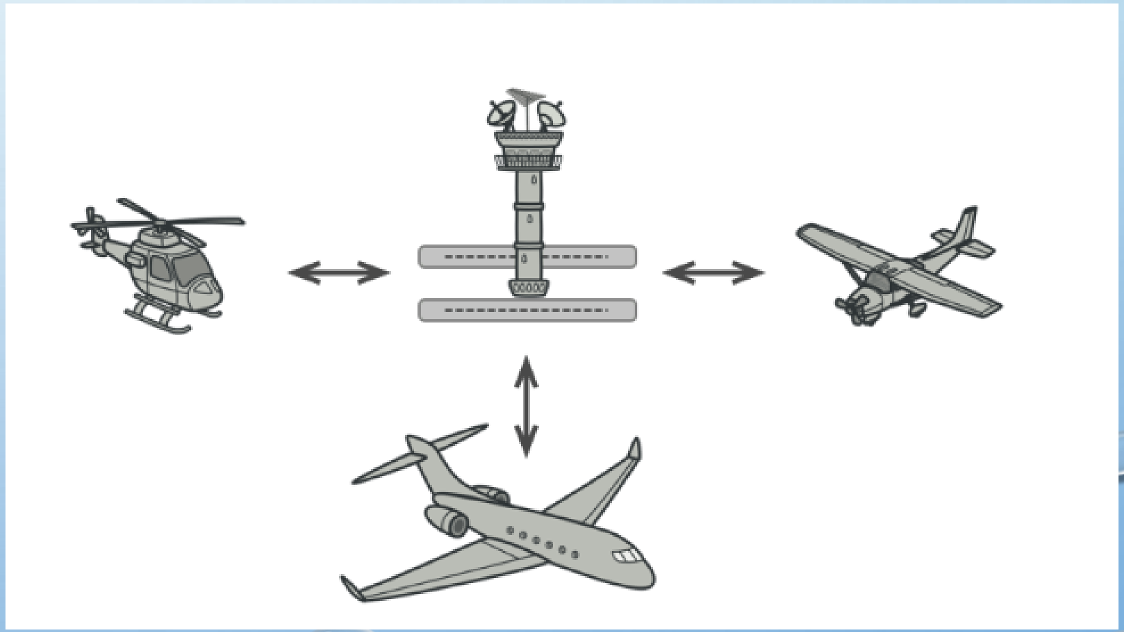


Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects.

By Ayala Berkovich

MEDIATOR DESIGN PATTERN

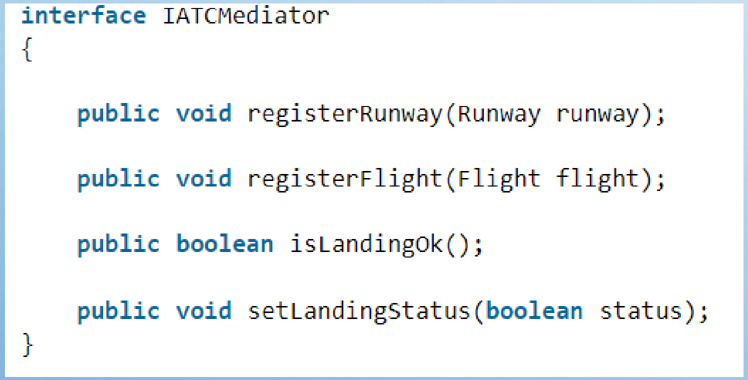
Air traffic controller is a great example of **mediator** pattern where the airport control room works as a mediator for communication between different flights.



MEDIATOR DESIGN PATTERN

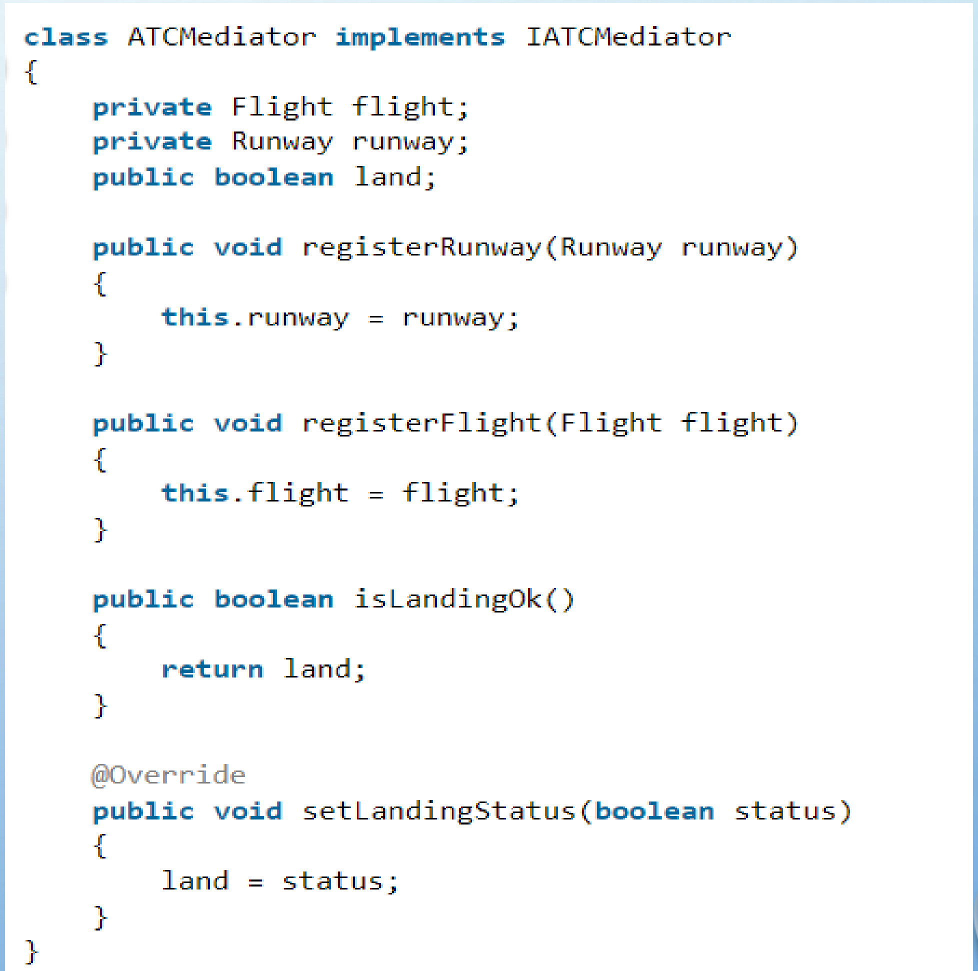
By Ayala Berkovich

1.Define Interface for Air traffic controller :



```
interface IATCMediator
{

    public void registerRunway(Runway runway);

    public void registerFlight(Flight flight);

    public boolean isLandingOk();

    public void setLandingStatus(boolean status);
}
```

2. Define Class for Air traffic controller :



```
class ATCMediator implements IATCMediator
{
    private Flight flight;
    private Runway runway;
    public boolean land;

    public void registerRunway(Runway runway)
    {
        this.runway = runway;
    }

    public void registerFlight(Flight flight)
    {
        this.flight = flight;
    }

    public boolean isLandingOk()
    {
        return land;
    }

    @Override
    public void setLandingStatus(boolean status)
    {
        land = status;
    }
}
```

MEDIATOR DESIGN PATTERN

By Ayala Berkovich

3. Define Class and interface for Flight:

```java
interface Command
{
    void land();
}
```

```java
class Flight implements Command
{
    private IATCMediator atcMediator;

    public Flight(IATCMediator atcMediator)
    {
        this.atcMediator = atcMediator;
    }

    public void land()
    {
        if (atcMediator.isLandingOk())
        {
            System.out.println("Successfully Landed.");
            atcMediator.setLandingStatus(true);
        }
        else
            System.out.println("Waiting for landing.");
    }

    public void getReady()
    {
        System.out.println("Ready for landing.");
    }

}
```

4. Define Class and interface for Runway:

```java
class Runway implements Command
{
    private IATCMediator atcMediator;

    public Runway(IATCMediator atcMediator)
    {
        this.atcMediator = atcMediator;
        atcMediator.setLandingStatus(true);
    }

    @Override
    public void land()
    {
        System.out.println("Landing permission granted.");
        atcMediator.setLandingStatus(true);
    }

}
```

5. Define Main Class:

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

        IATCMediator atcMediator = new ATCMediator();
        Flight sparrow101 = new Flight(atcMediator);
        Runway mainRunway = new Runway(atcMediator);
        atcMediator.registerFlight(sparrow101);
        atcMediator.registerRunway(mainRunway);
        sparrow101.getReady();
        mainRunway.land();
        sparrow101.land();

    }
}
```

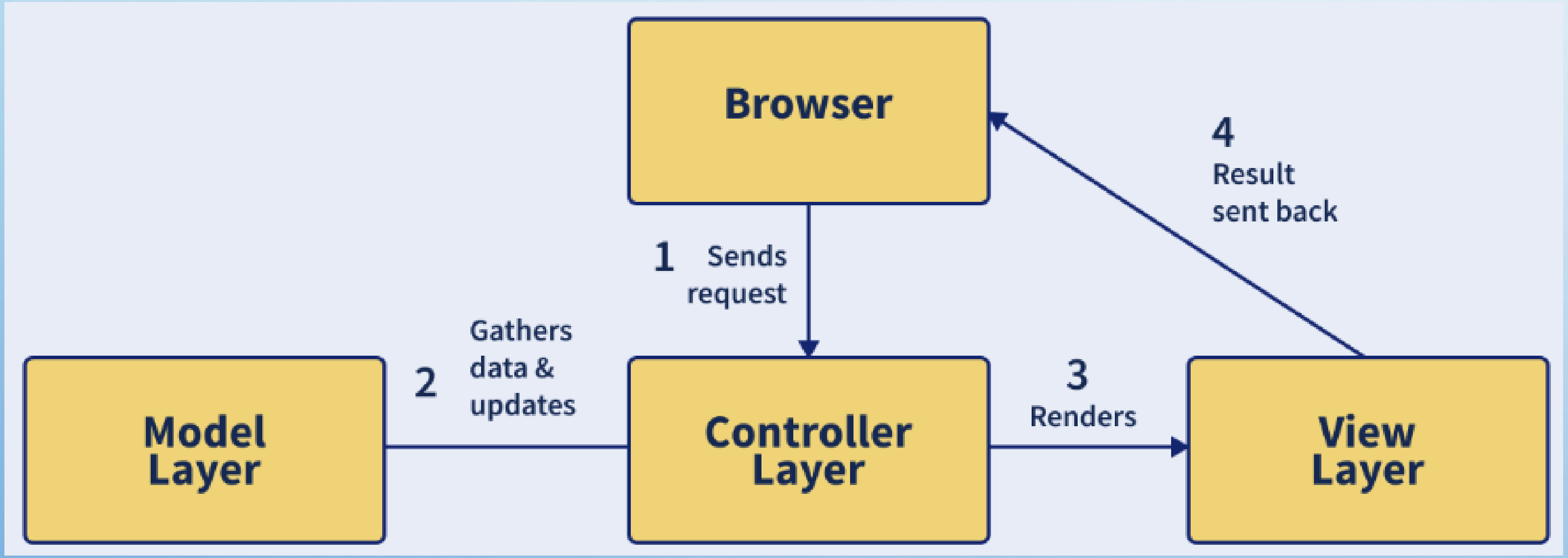
Output

```
Ready for landing.
Landing permission granted.
Successfully Landed.
```

By Ayala Berkovich

JAVA DESIGN PATTERNS

MVC(MODEL-VIEW-CONTROLLER) DESIGN PATTERN - this pattern is used for separating the application's concerns



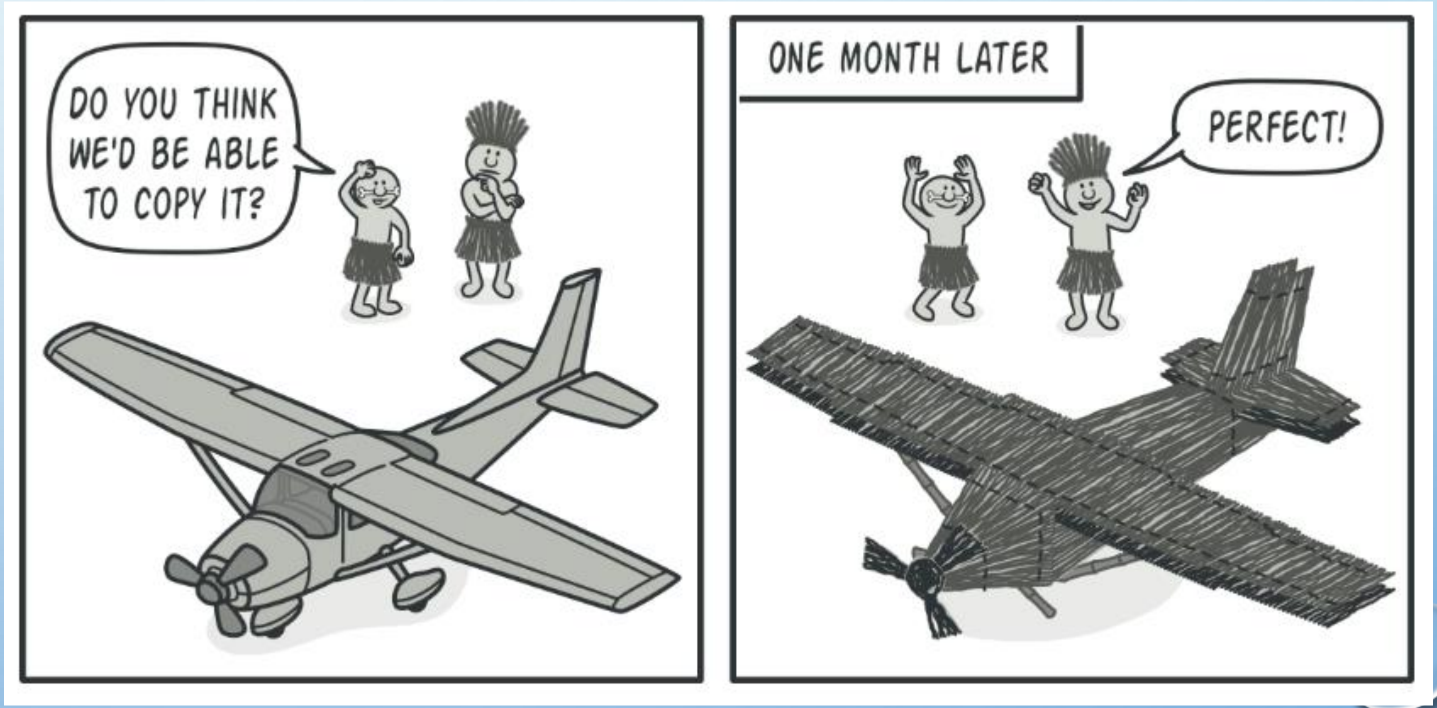
- **Model** - This represents the object (**Java POJO**) that carries the data. It can also consist of the logic of updating the controller in case the data changes.
- **View** - data visualization of the model.
- **Controller** - is an interface between the **Model** and the **View** by controlling the flow of data into the **model** and updating the **view** whenever the model gets updated. This ensures that the **model** and the **views** are kept separate.

By Ayala Berkovich

JAVA DESIGN PATTERNS

PROTOTYPE DESIGN PATTERN - IS A CREATIONAL DESIGN PATTERN THAT LETS YOU COPY EXISTING OBJECTS WITHOUT MAKING YOUR CODE DEPENDENT ON THEIR CLASSES.

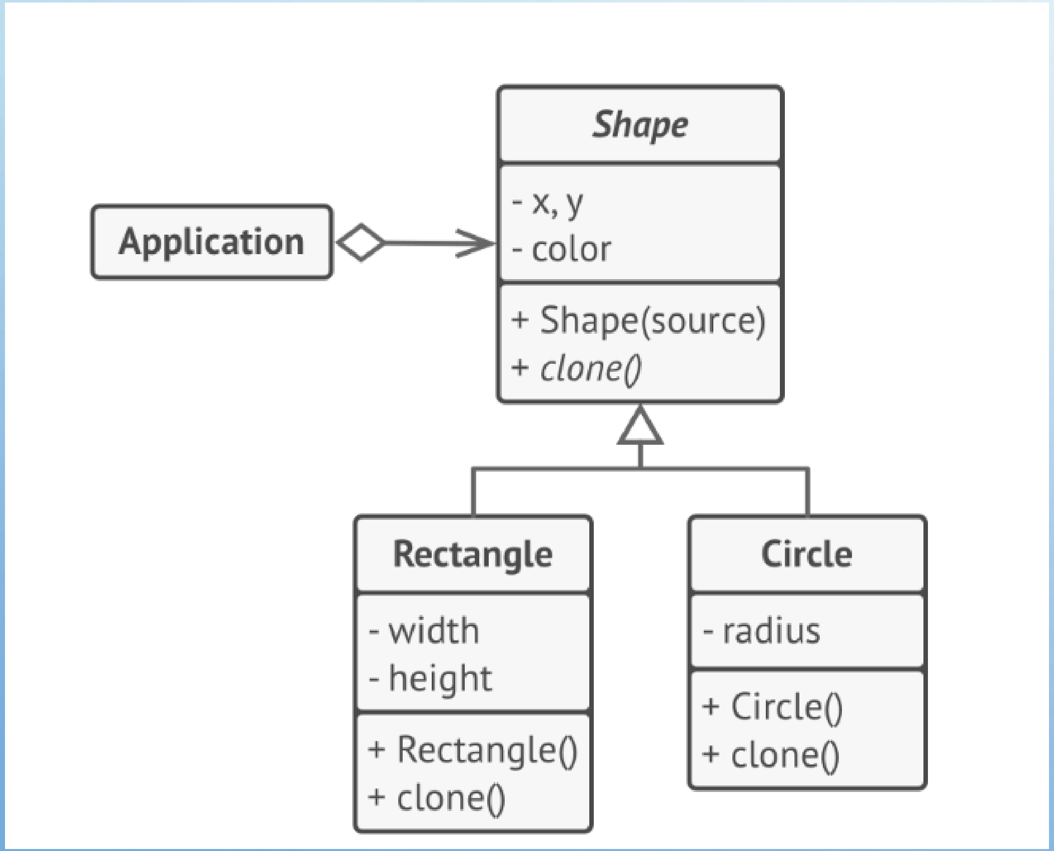
Say you have an object, and you want to create an exact copy of it. How would you do it?
First, you have to create a new object of the same class.
Then you have to go through all the fields of the original object and copy their values over to the new object.
Nice! But there's a catch. Not all objects can be copied that way because some of the object's fields may be private and not visible from outside of the object itself.



By Ayala Berkovich

PROTOTYPE DESIGN PATTERN

The pattern declares a common interface for all objects that support cloning.
This interface lets you clone an object without coupling your code to the class of that object.
Usually, such an interface contains just a single clone method.



```
clone(){
return new Rectangle(this)
}
```

```
clone(){
return new Circle(this)
}
```

```
Circle circle = new Circle();
 circle.X = 10;
circle.Y = 10;
circle.radius = 20;

Circle anotherCircle = circle.clone();
```

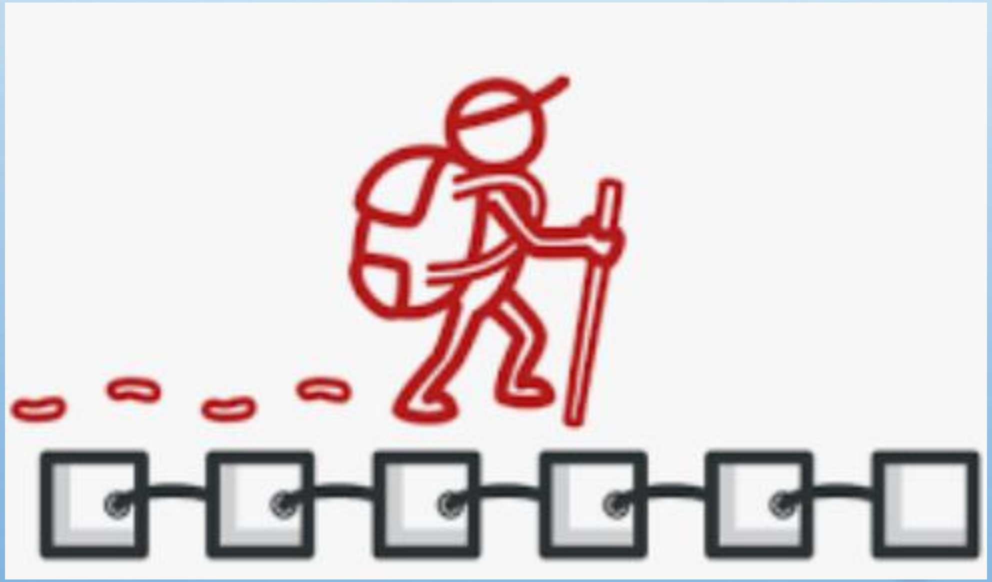
By Ayala Berkovich

JAVA DESIGN PATTERNS

ITERATOR DESIGN PATTERN – BEHAVIORAL PATTERN

WE ALL KNOW THAT **COLLECTION** FRAMEWORK **ITERATOR** IS THE BEST EXAMPLE OF ITERATOR PATTERN IMPLEMENTATION.

Iterator design pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.

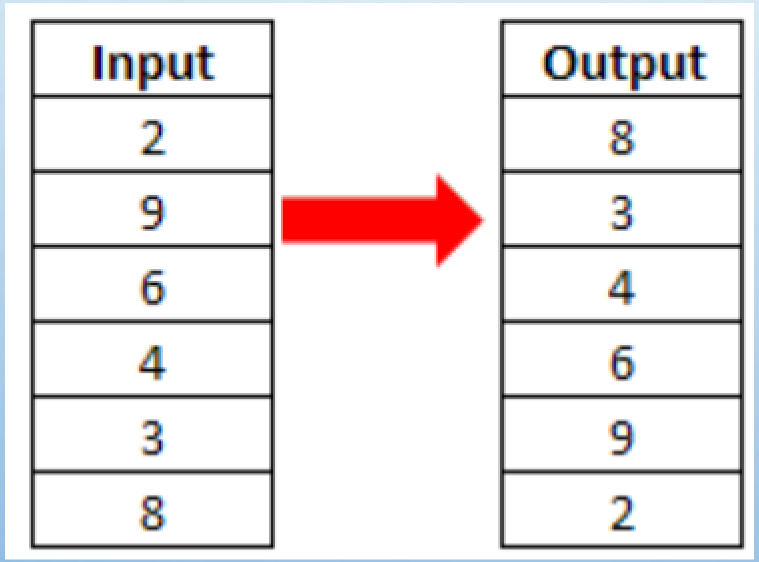


By Ayala Berkovich

ITERATOR DESIGN PATTERN

WRITE CODE THAT INPUT SEQUENCE OF 10 NUMBERS AND REVERS ORDER (USING ARRAYLIST):

FOR EXAMPLE:



Input
2
9
6
4
3
8

Output
8
3
4
6
9
2

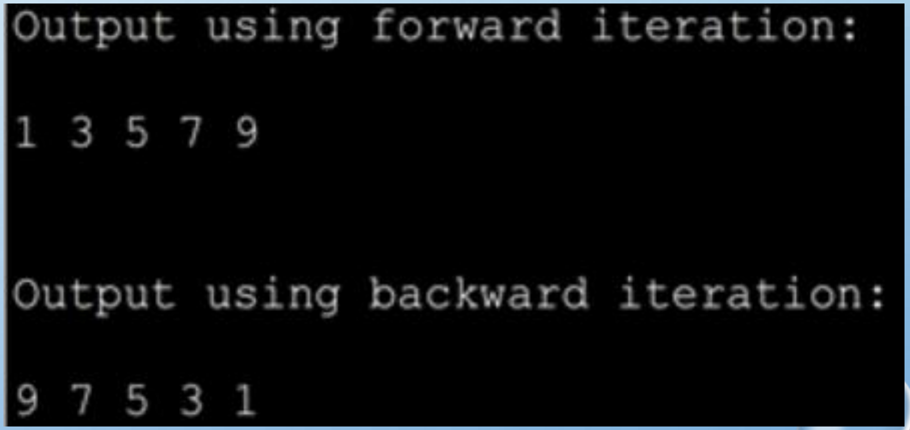
… …

By Ayala Berkovich

ITERATOR DESIGN PATTERN

```java
import java.util.*;

class Main
{
    public static void main(String args[])
    {
        List<Integer>num_list = new ArrayList<Integer>();

        // Add Elements to ArrayList
        num_list.add(1);
        num_list.add(3);
        num_list.add(5);
        num_list.add(7);
        num_list.add(9);

        // Creatinge a ListIterator
        ListIteratorlist_it = num_list.listIterator();
        System.out.println("Output using forward iteration:");

        while (list_it.hasNext())
              System.out.print(list_it.next()+" ") ;

        System.out.print("\n\nOutput using backward iteration:\n") ;
        while (list_it.hasPrevious())
             System.out.print(list_it.previous()+" ");

    }
}
```

Output



```
Output using forward iteration:

1 3 5 7 9


Output using backward iteration:

9 7 5 3 1
```

By Ayala Berkovich

JAVA DESIGN PATTERNS

Why use a design pattern?

The usefulness of using a design pattern is obvious. The design pattern can **accelerate the development process**. It provides proven development paradigms, which helps save time without having to reinvent patterns every time a problem arises.

By Ayala Berkovich