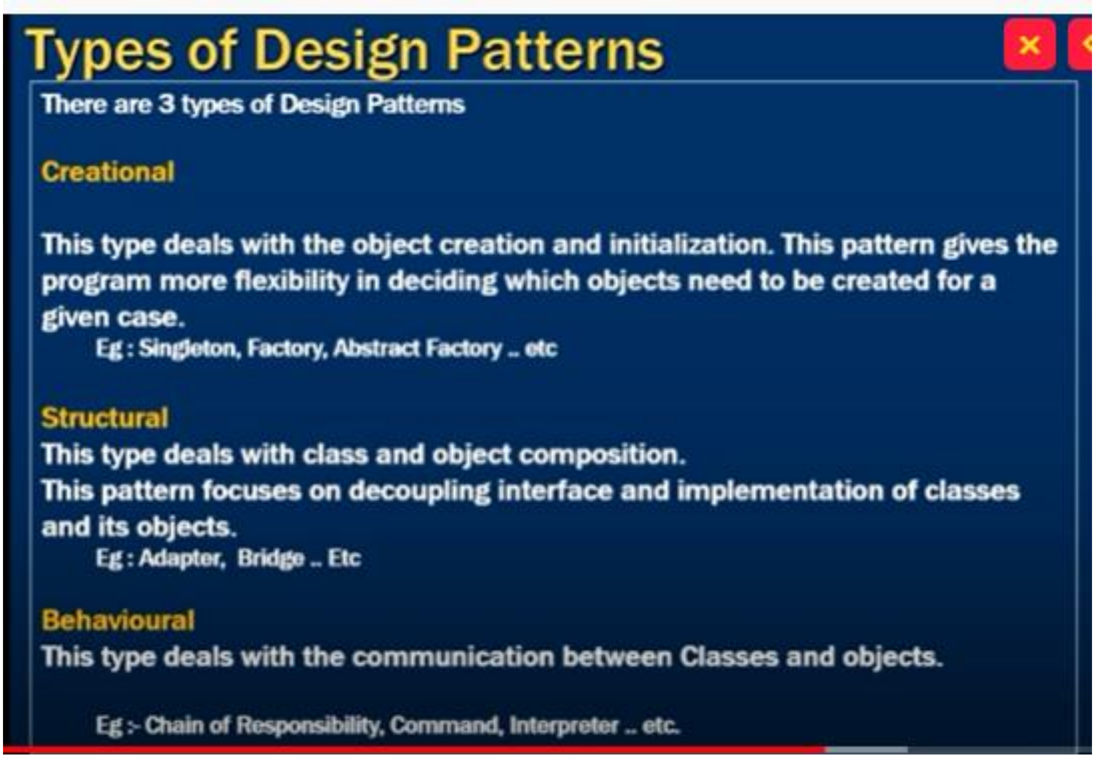
Read from notes before this :- Basic defintions :- (Below content from Telusko and CodeSpace)

Ref : Telusko, https://thecodingsimplified.com/design-pattern/



Types of Design Patterns

There are 3 types of Design Patterns

Creational

This type deals with the object creation and initialization. This pattern gives the program more flexibility in deciding which objects need to be created for a given case.

Eg : Singleton, Factory, Abstract Factory .. etc

Structural

This type deals with class and object composition.

This pattern focuses on decoupling interface and implementation of classes and its objects.

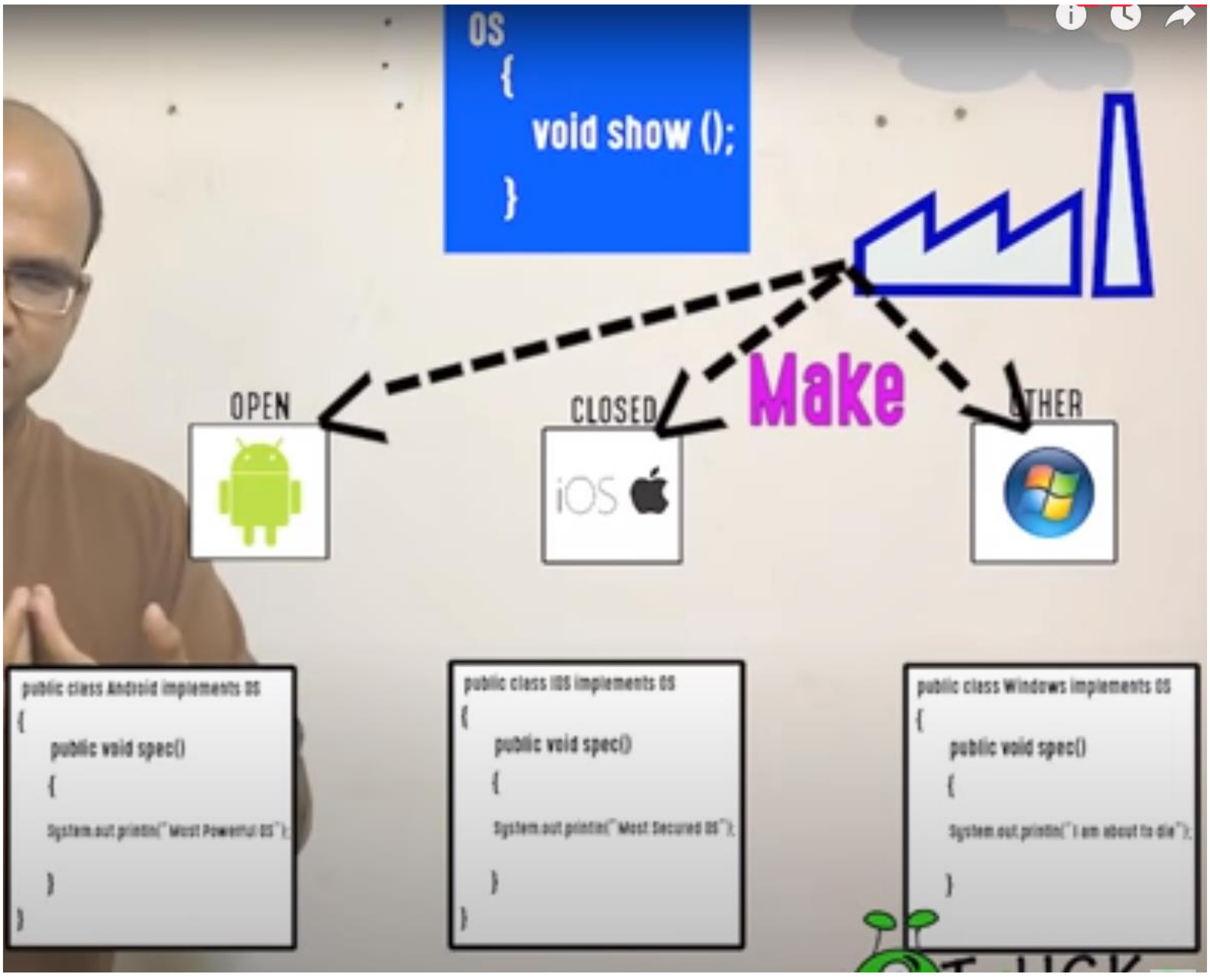
Eg : Adapter, Bridge .. Etc

Behavioural

This type deals with the communication between Classes and objects.

Eg :- Chain of Responsibility, Command, Interpreter .. etc.

Factory Design Pattern(Creational)



```
OS
{
    void show ();
}
```

```
public class Android implements OS
{
    public void spec()
    {
    System.out.println("Most Powerful OS");
    }
}
```

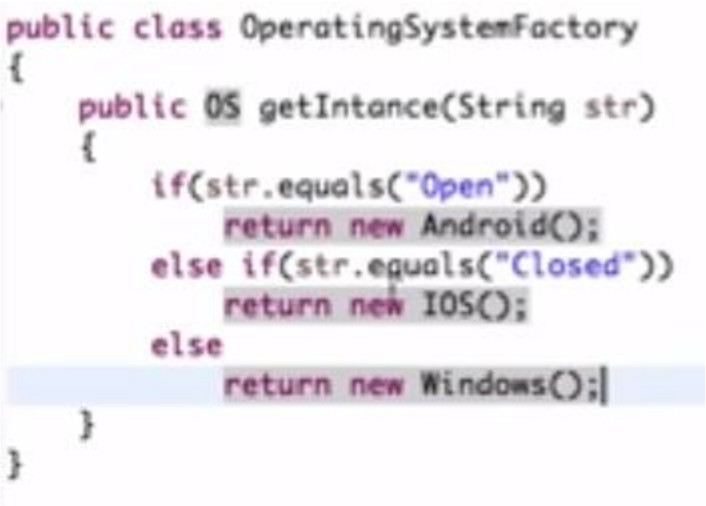
```
public class IOS implements OS
{
    public void spec()
    {
    System.out.println("Most Secured OS");
    }
}
```

```
public class Windows implements OS
{
    public void spec()
    {
    System.out.println("I am about to die");
    }
```

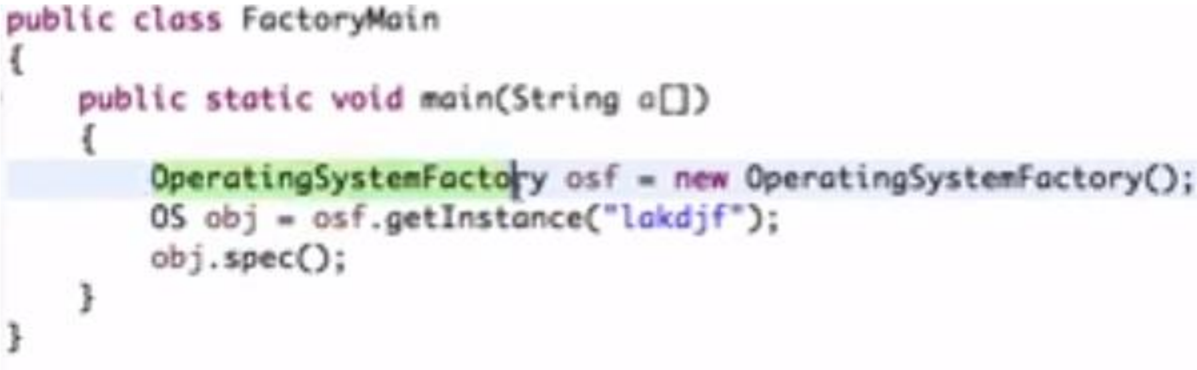
Without factory :-

```
OS Obj = new Android();
Obj.spec(); //Create Object Manually
```

With factory :-



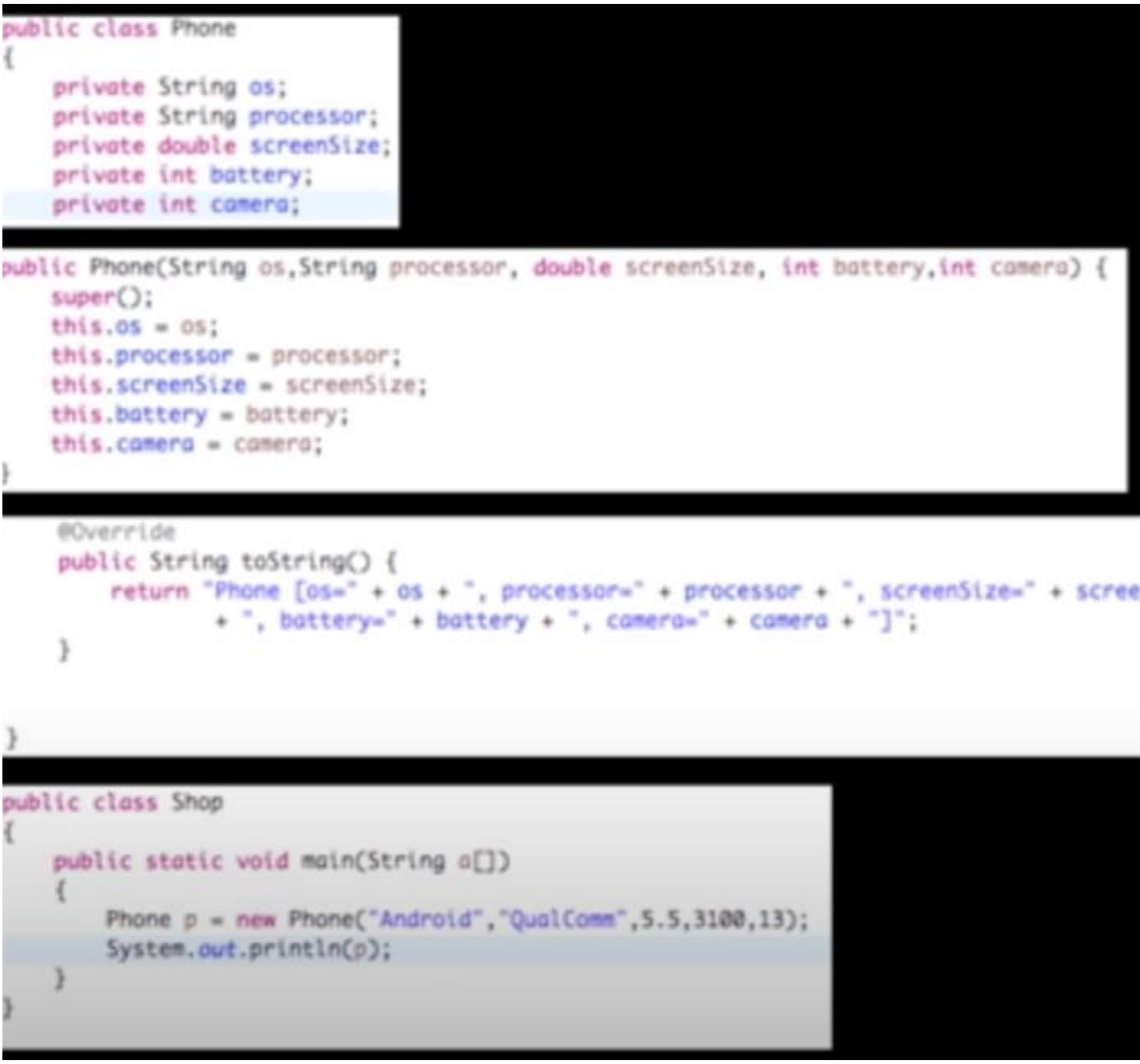
```
public class OperatingSystemFactory
{
    public OS getIntance(String str)
    {
        if(str.equals("Open"))
            return new Android();
        else if(str.equals("Closed"))
            return new IOS();
        else
            return new Windows();
    }
}
```



```
public class FactoryMain
{
    public static void main(String a[])
    {
        OperatingSystemFactory osf = new OperatingSystemFactory();
        OS obj = osf.getInstance("lakdjf");
        obj.spec();
    }
}
```

Builder Pattern(Creational)

When you want to set only few parameters of a object



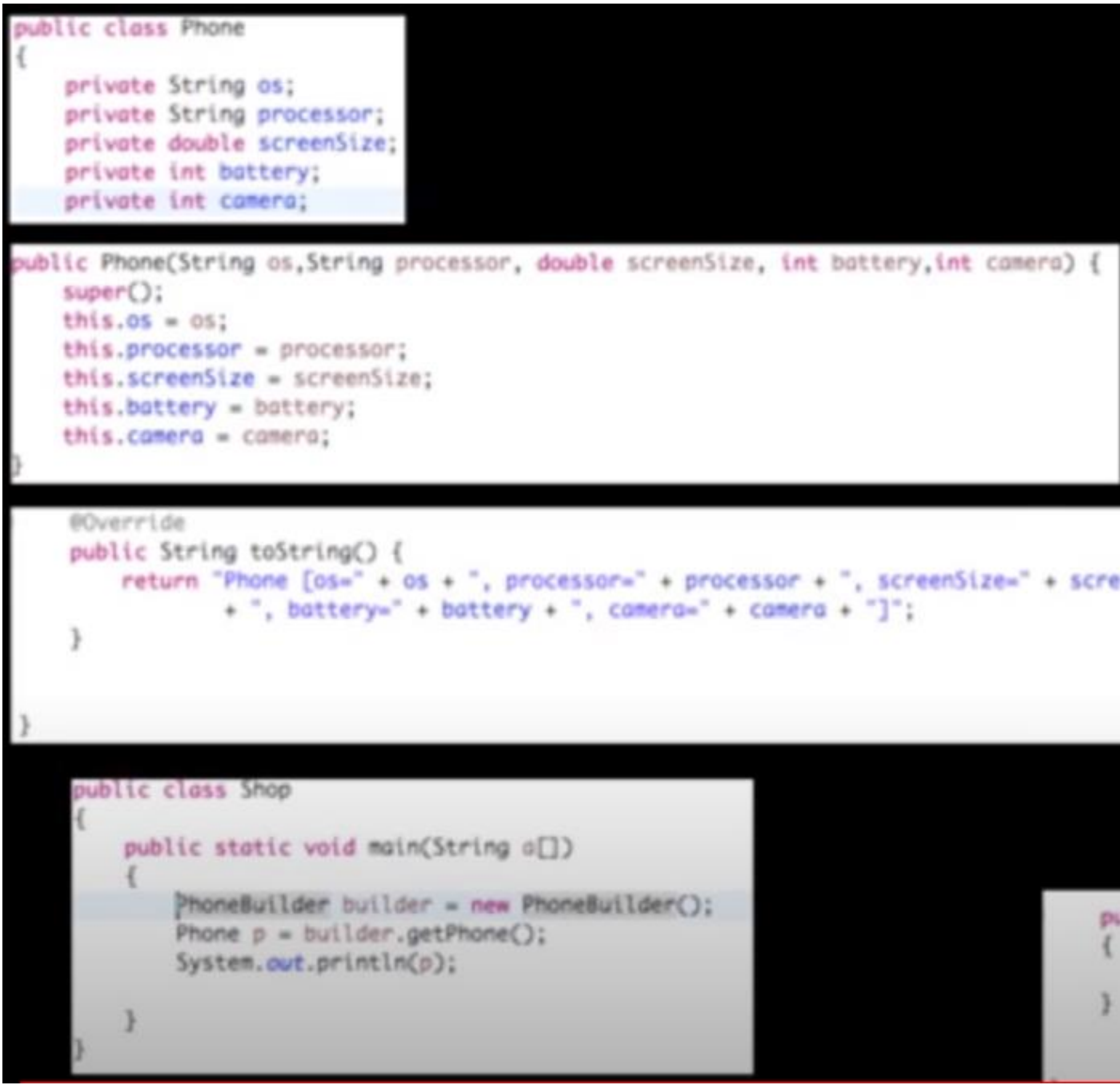
```
public class Phone
{
    private String os;
    private String processor;
    private double screenSize;
    private int battery;
    private int camera;
```

```
public Phone(String os,String processor, double screenSize, int battery,int camera) {
    super();
    this.os = os;
    this.processor = processor;
    this.screenSize = screenSize;
    this.battery = battery;
    this.camera = camera;
}
```

```
    @Override
    public String toString() {
        return "Phone [os=" + os + ", processor=" + processor + ", screenSize=" + scree
                + ", battery=" + battery + ", camera=" + camera + "]";
    }


}
```

```
public class Shop
{
    public static void main(String a[])
    {
        Phone p = new Phone("Android","QualComm",5.5,3100,13);
        System.out.println(p);
    }
}
```

Coca-Cola
Coca-Cola
Coca-Cola
Coca-Cola



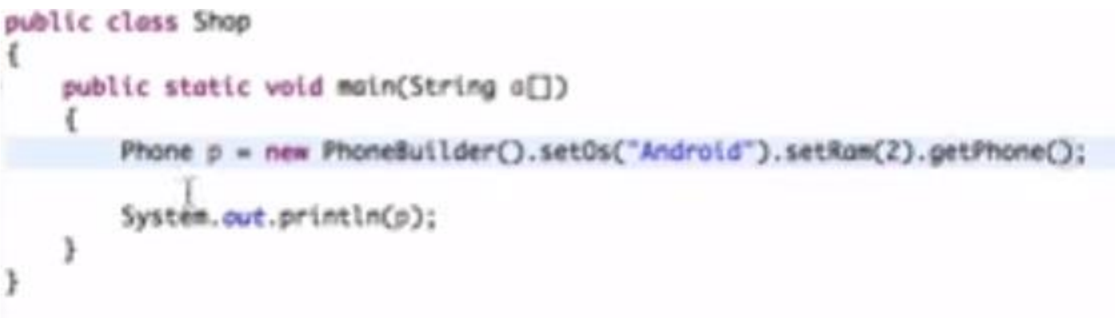
```
public class Phone
{
    private String os;
    private String processor;
    private double screenSize;
    private int battery;
    private int camera;
```

```
public Phone(String os,String processor, double screenSize, int battery,int camera) {
    super();
    this.os = os;
    this.processor = processor;
    this.screenSize = screenSize;
    this.battery = battery;
    this.camera = camera;
}
```

```
    @Override
    public String toString() {
        return "Phone [os=" + os + ", processor=" + processor + ", screenSize=" + scre
                + ", battery=" + battery + ", camera=" + camera + "]";
    }


}
```

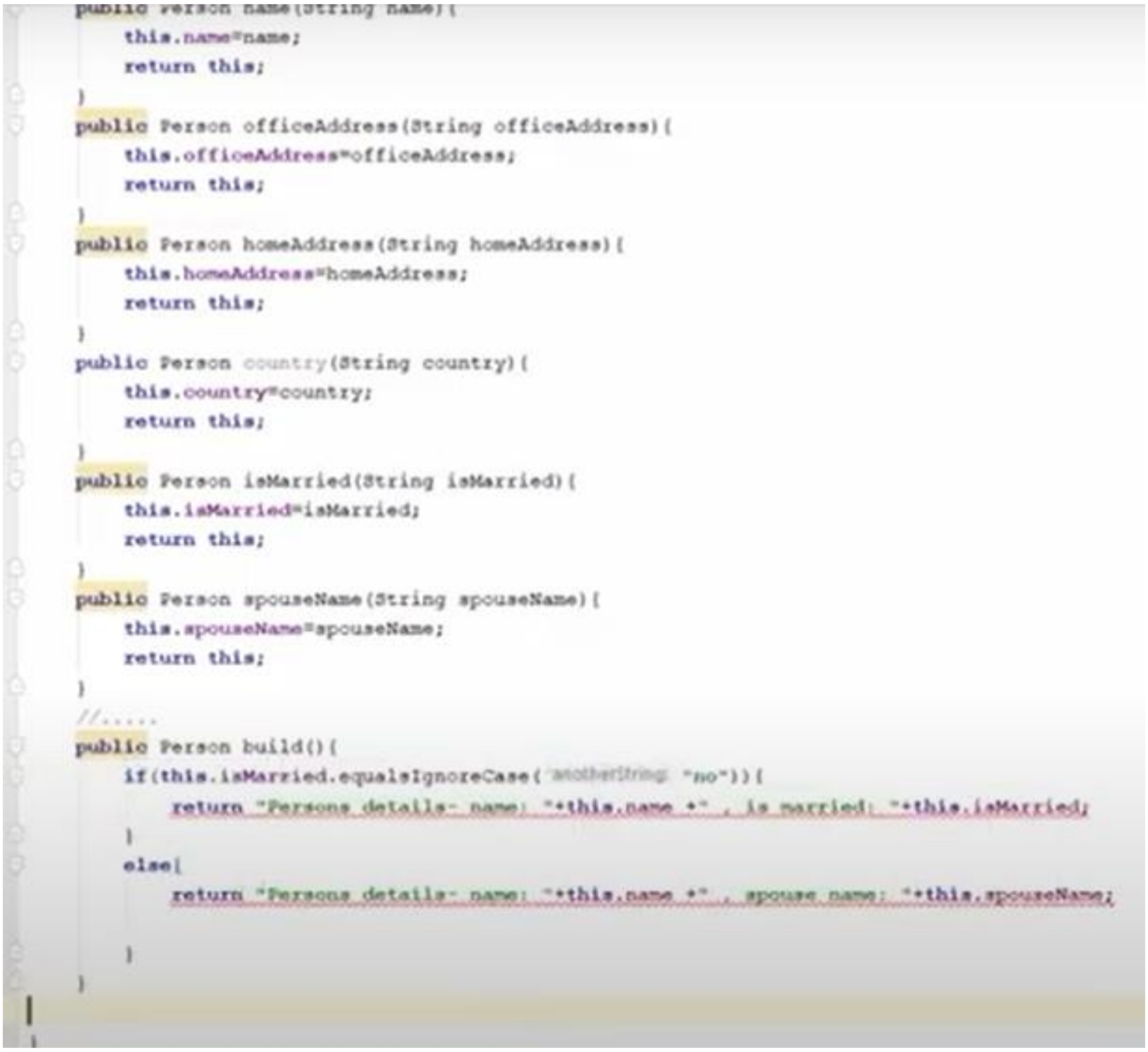
```
public class Shop
{
    public static void main(String a[])
    {
        PhoneBuilder builder = new PhoneBuilder();
        Phone p = builder.getPhone();
        System.out.println(p);

    }
}
```

```
pu
{
}
```

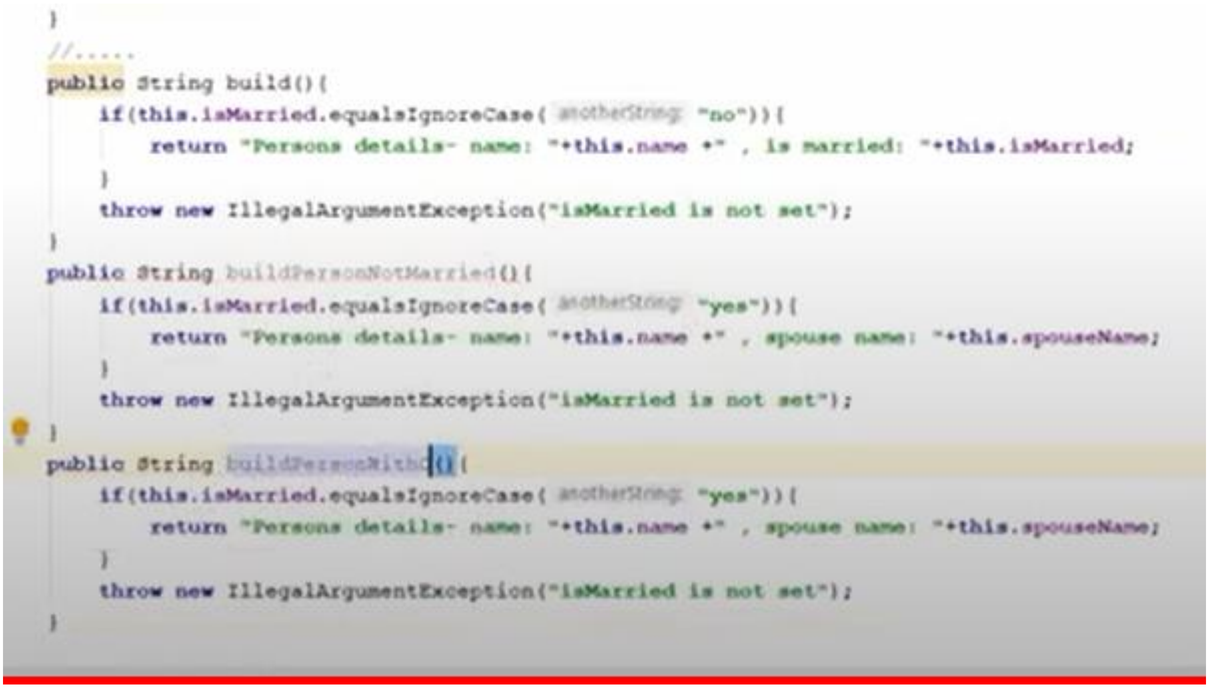


```
public class Shop
{
    public static void main(String a[])
    {
        Phone p = new PhoneBuilder().setOs("Android").setRam(2).getPhone();

        System.out.println(p);
    }
}
```

Example 2:-



```
public Person name(String name){
    this.name=name;
    return this;
}
public Person officeAddress(String officeAddress){
    this.officeAddress=officeAddress;
    return this;
}
public Person homeAddress(String homeAddress){
    this.homeAddress=homeAddress;
    return this;
}
public Person country(String country){
    this.country=country;
    return this;
}
public Person isMarried(String isMarried){
    this.isMarried=isMarried;
    return this;
}
public Person spouseName(String spouseName){
    this.spouseName=spouseName;
    return this;
}
//.....
public Person build(){
    if(this.isMarried.equalsIgnoreCase( anotherString: "no")){
        return "Persons details- name: "+this.name +" , is married: "+this.isMarried;
    }
    else{
        return "Persons details- name: "+this.name +" , spouse name: "+this.spouseName;

    }
}
```



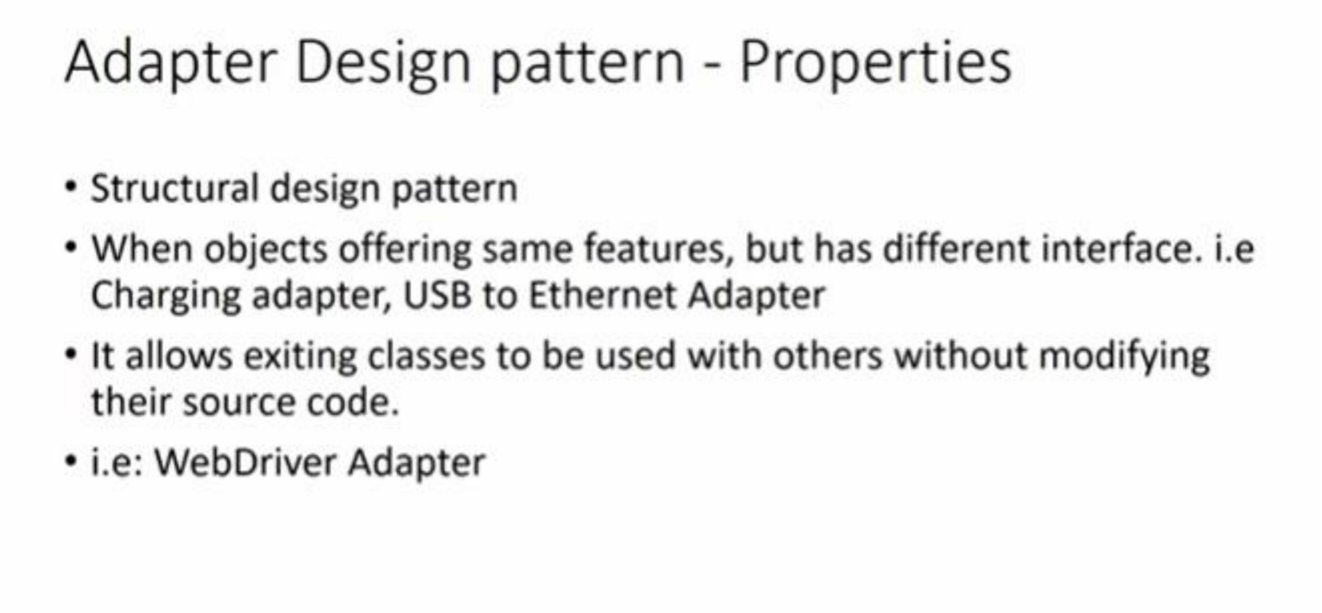
```
}
//.....
public String build(){
    if(this.isMarried.equalsIgnoreCase( anotherString: "no")){
        return "Persons details- name: "+this.name +" , is married: "+this.isMarried;
    }
    throw new IllegalArgumentException("isMarried is not set");
}
public String buildPersonNotMarried(){
    if(this.isMarried.equalsIgnoreCase( anotherString: "yes")){
        return "Persons details- name: "+this.name +" , spouse name: "+this.spouseName;
    }
    throw new IllegalArgumentException("isMarried is not set");
}
public String buildPersonWithO(){
    if(this.isMarried.equalsIgnoreCase( anotherString: "yes")){
        return "Persons details- name: "+this.name +" , spouse name: "+this.spouseName;
    }
    throw new IllegalArgumentException("isMarried is not set");
}
```

Can create object using :-

Person p = new Person().setName("Ayush".setContact('9881').build(); -> build method return this

We can also use other build methods to check for some other functionalities while creating the object.

Adapter Design Pattern:- (Structural)



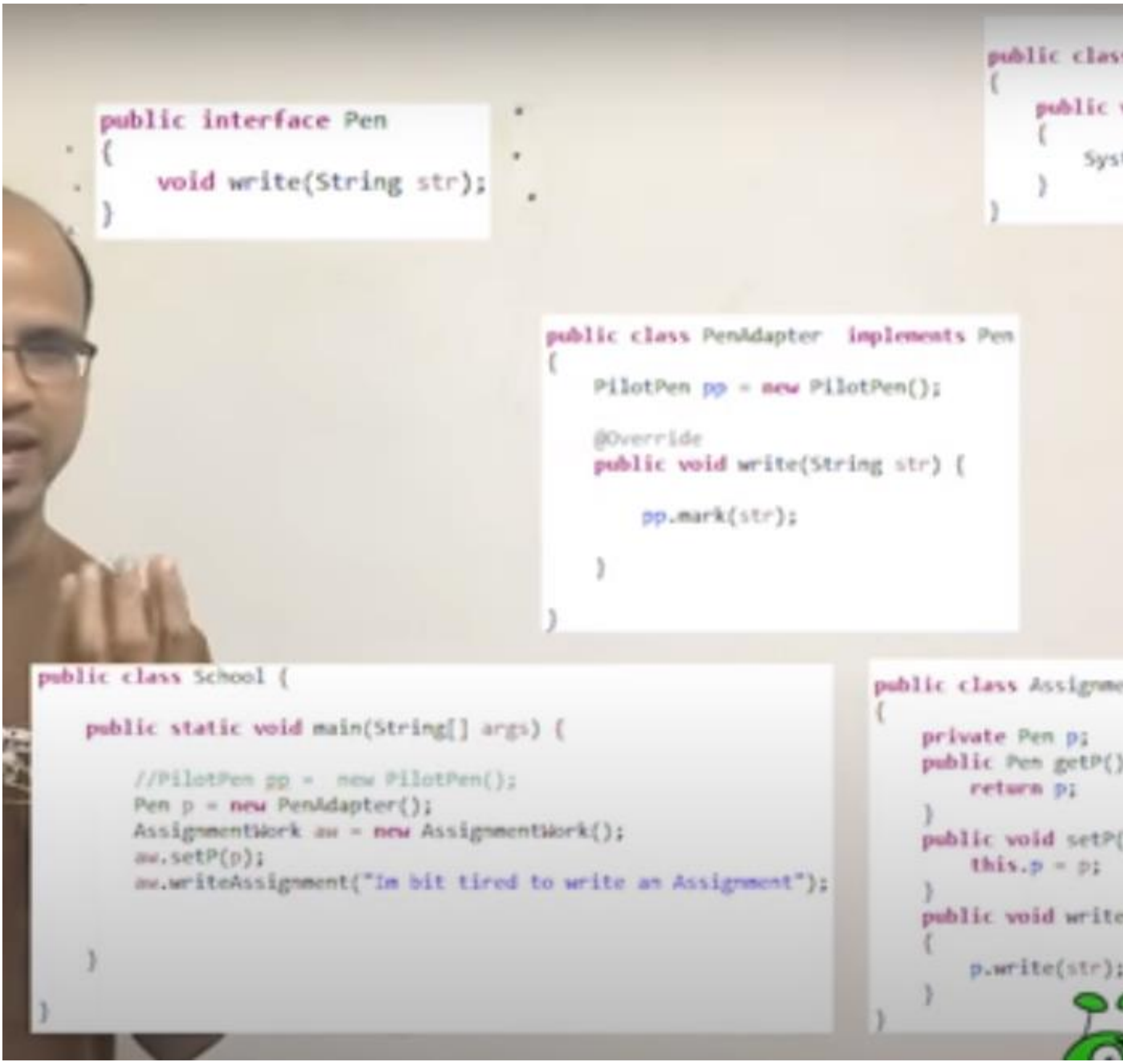
Adapter Design pattern - Properties

- Structural design pattern
- When objects offering same features, but has different interface. i.e Charging adapter, USB to Ethernet Adapter
- It allows exiting classes to be used with others without modifying their source code.
- i.e: WebDriver Adapter

Adapter patterns are used when we have one object and not implementation of other, and we want to make use of other object as in case of iphone charger, we are making use of some other charger.

As in below example, we are trying to make use of pilot pen as we don't have implementation of Pen Object, so we are using pilot pen.

We have got compiler class of Pilot Pen, and don't have any implementation of Pilot object, we create a object in main from PenAdapter.



```
public interface Pen
{
    void write(String str);
}
```

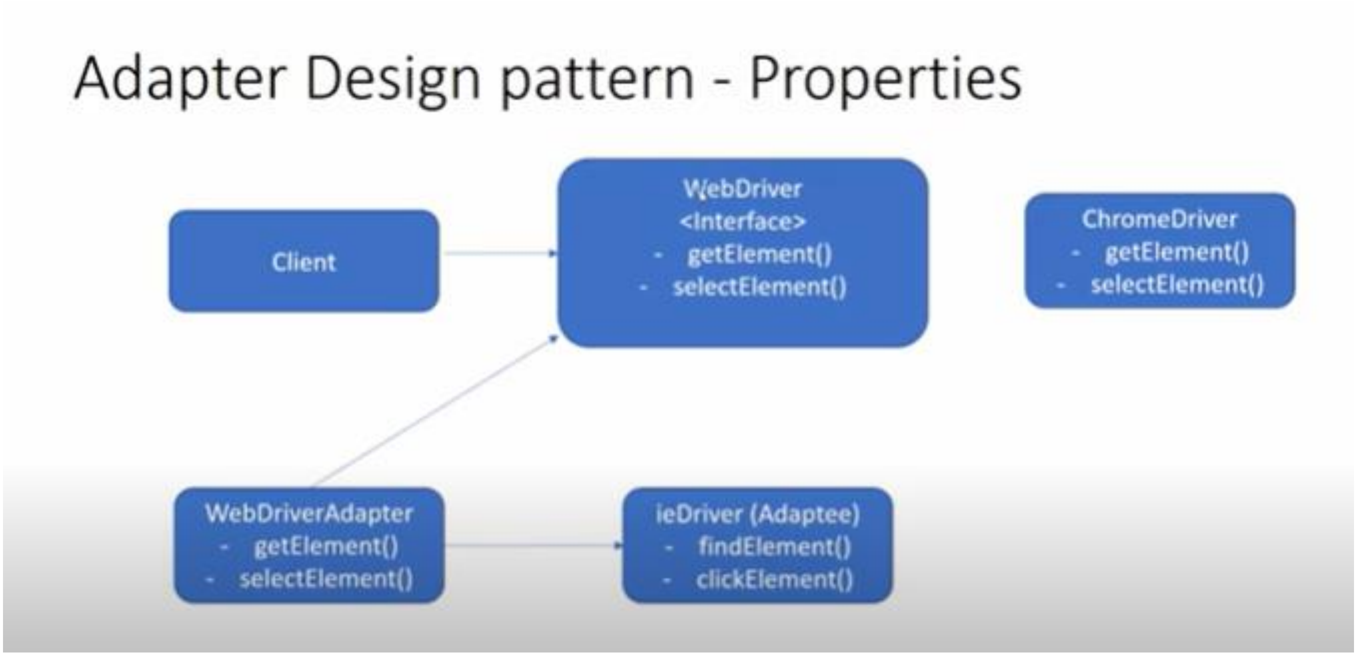
```
public clas
{
    public 
    {
        Sys
    }
}
```

```
public class PenAdapter  implements Pen
{
    PilotPen pp = new PilotPen();

    @Override
    public void write(String str) {

        pp.mark(str);

    }

}
```

```
public class School {

    public static void main(String[] args) {

        //PilotPen pp =  new PilotPen();
        Pen p = new PenAdapter();
        AssignmentWork aw = new AssignmentWork();
        aw.setP(p);
        aw.writeAssignment("Im bit tired to write an Assignment");


    }

}
```

```
public class Assignme
{
    private Pen p;
    public Pen getP()
        return p;
    }
    public void setP(
        this.p = p;
    }
    public void write
    {
        p.write(str);
    }
}
```

Example 2:- Coding Simplified



Adapter Design pattern - Properties

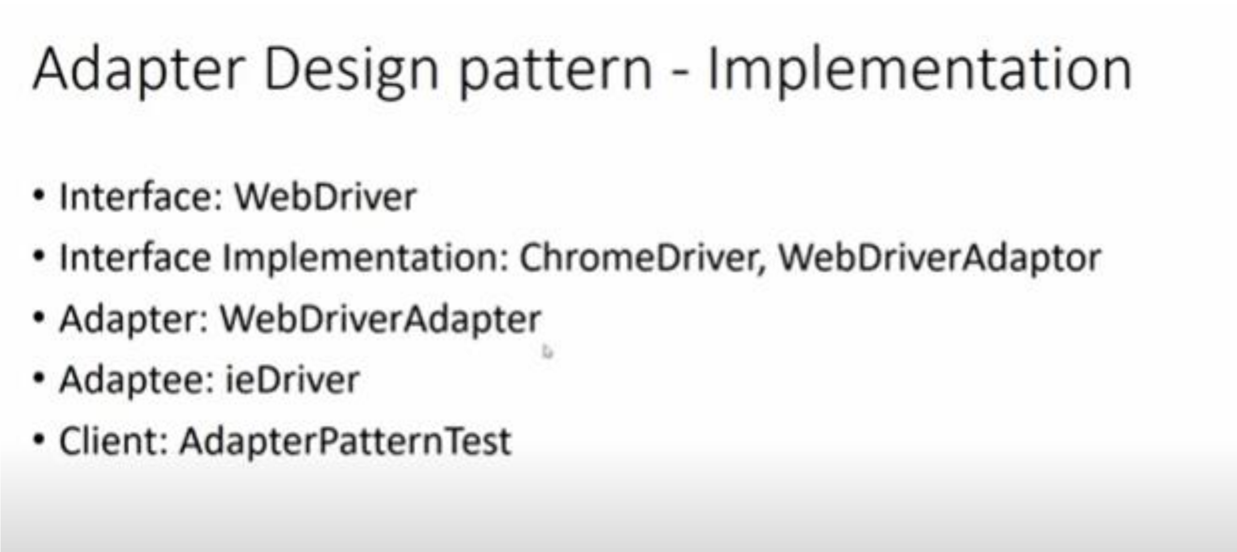
Client

WebDriver
<Interface>
- getElement()
- selectElement()

ChromeDriver
- getElement()
- selectElement()

WebDriverAdapter
- getElement()
- selectElement()

ieDriver (Adaptee)
- findElement()
- clickElement()



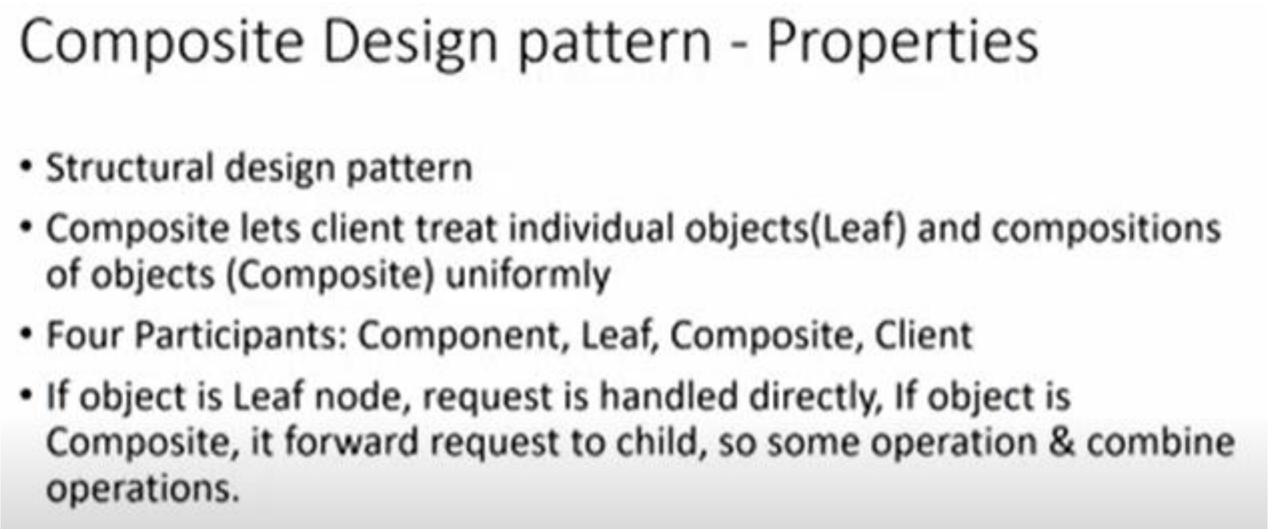
Adapter Design pattern - Implementation

- Interface: WebDriver
- Interface Implementation: ChromeDriver, WebDriverAdaptor
- Adapter: WebDriverAdapter
- Adaptee: ieDriver
- Client: AdapterPatternTest

```java
package adapter;

interface WebDriver {
  public void getElement();
  public void selectElement();
}


class ChromeDriver implements WebDriver {

  @Override
  public void getElement() {
    System.out.println("Get element from ChromeDriver");
  }
  @Override
  public void selectElement() {
    System.out.println("Select element from ChromeDriver");

  }

}

class IEDriver {

  public void findElement() {
    System.out.println("Find element from IEDriver");
  }

  public void clickElement() {
    System.out.println("Click element from IEDriver");
  }

}

class WebDriverAdapter implements WebDriver {

  IEDriver ieDriver;

  public WebDriverAdapter(IEDriver ieDriver) {
    this.ieDriver = ieDriver;
  }

  @Override
  public void getElement() {
    ieDriver.findElement();

  }

  @Override
  public void selectElement() {
    ieDriver.clickElement();
  }

}

public class AdapterDesignPattern {

  public static void main(String[] args) {
    ChromeDriver a = new ChromeDriver();
    a.getElement();
    a.selectElement();

    IEDriver e = new IEDriver();
    e.findElement();
    e.clickElement();
```

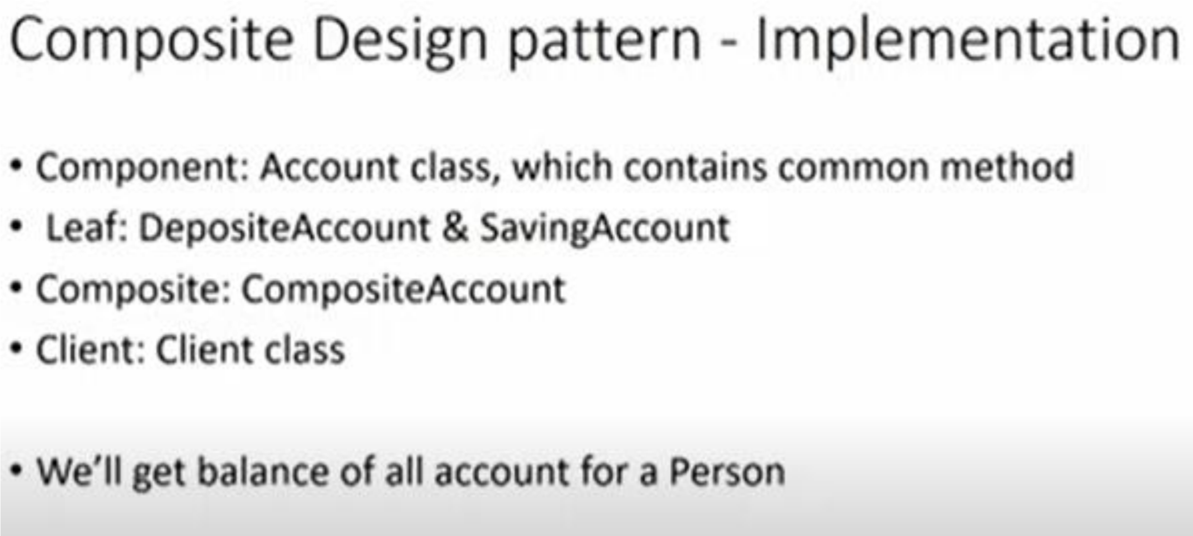
```
        WebDriver wID = new WebDriverAdapter(e);
        wID.getElement();
        wID.selectElement();

    }

}
```

Composite Design pattern(Structural)



Composite Design pattern - Properties

- Structural design pattern
- Composite lets client treat individual objects(Leaf) and compositions of objects (Composite) uniformly
- Four Participants: Component, Leaf, Composite, Client
- If object is Leaf node, request is handled directly, If object is Composite, it forward request to child, so some operation & combine operations.



Composite Design pattern - Implementation

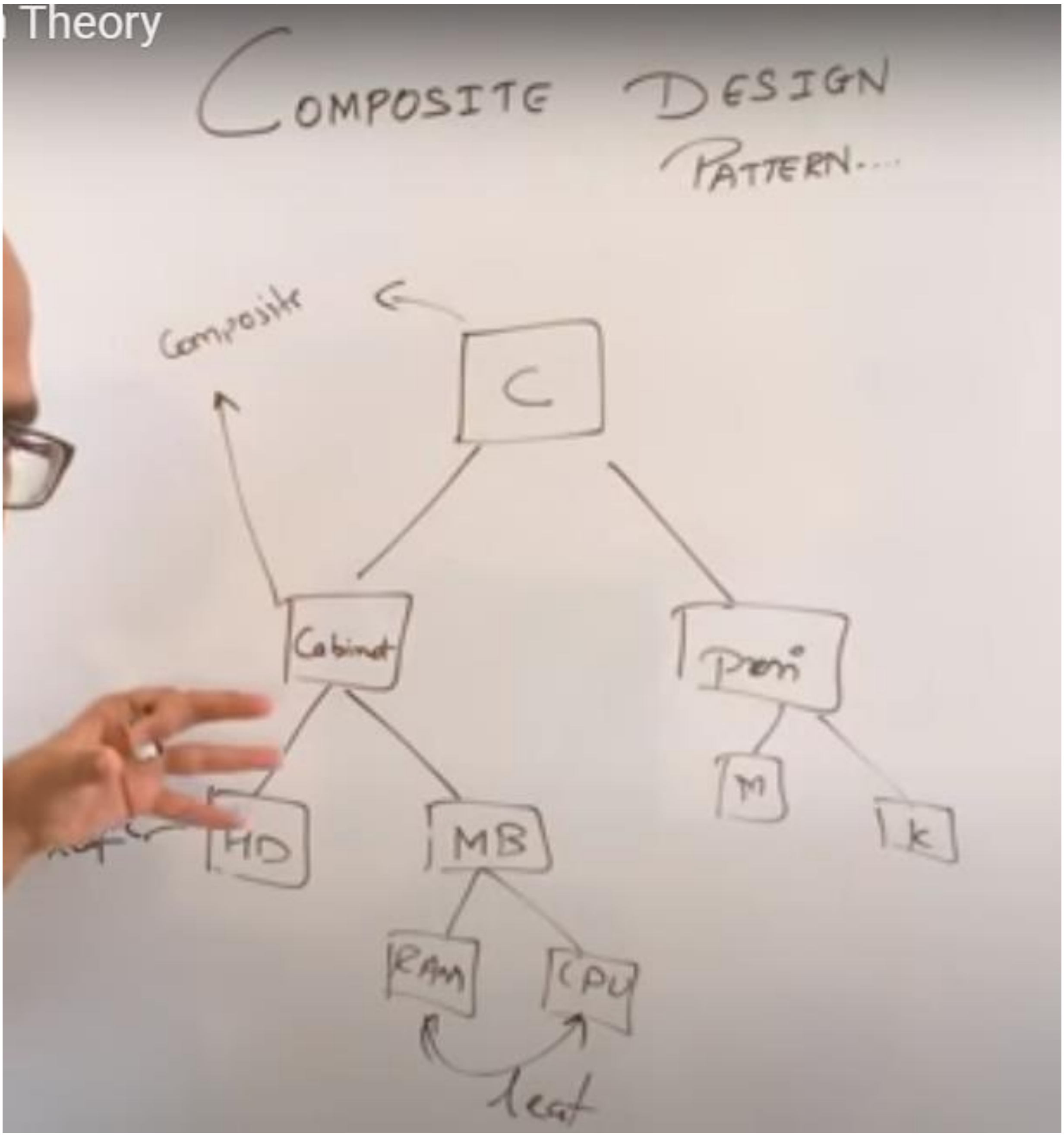
- Component: Account class, which contains common method
- Leaf: DepositeAccount & SavingAccount
- Composite: CompositeAccount
- Client: Client class

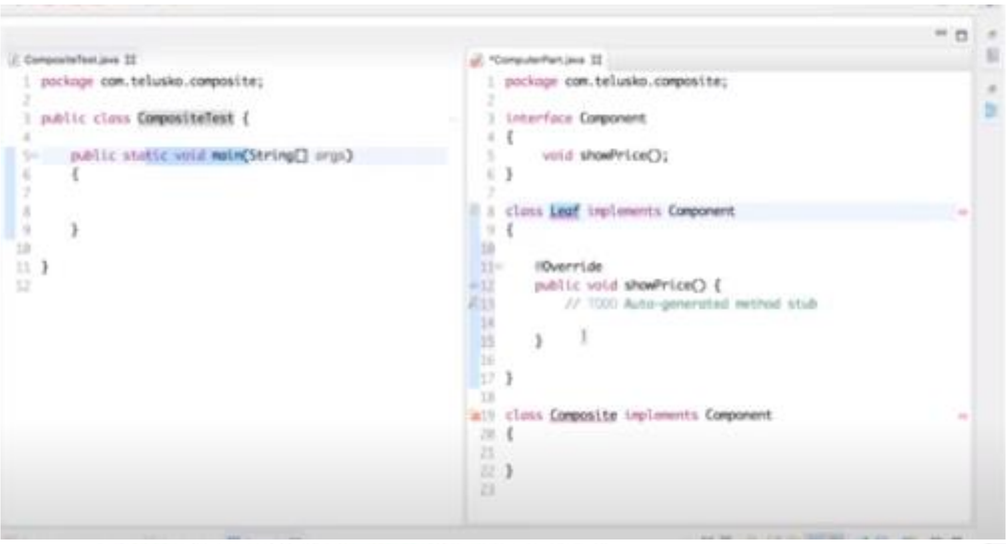
- We'll get balance of all account for a Person

C : Computer will be composed of Cabinet and Peripheral. Cabinet will be composed of hardDriver, MotherBoard. Mother Board will consist of RAM and CPU. Peripheral will be consisting of Mouse, Key etc.

Composite dp is used when each object is composed of some other objects. The object which can be further divided are called as Leafs.

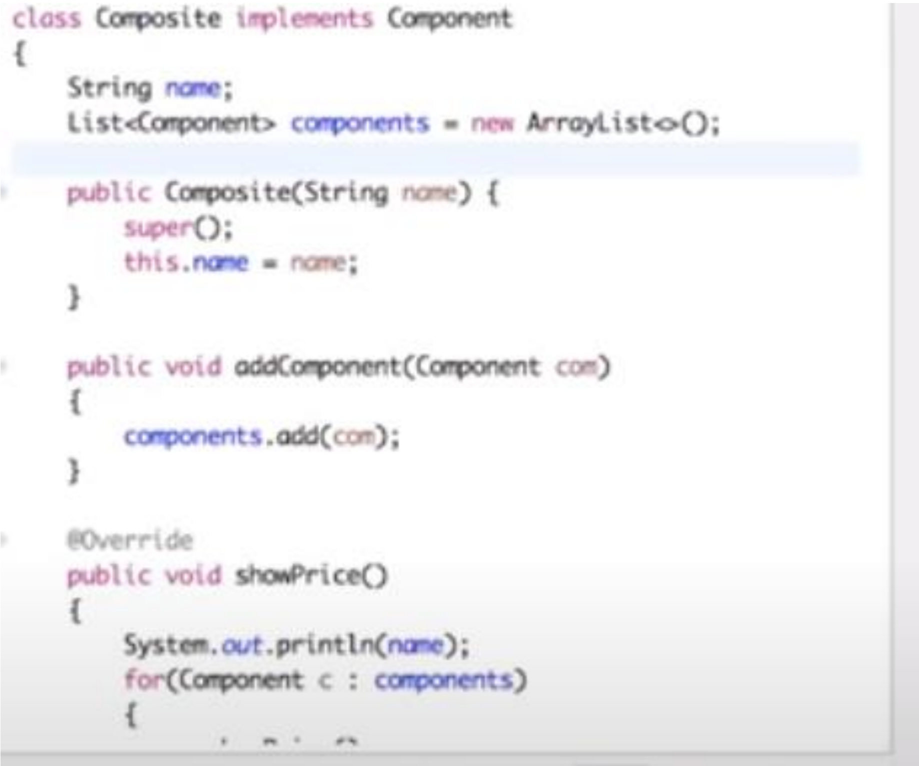
In Composite Design – you should be able to perform same function on parent as well what you are performing on child, as in example below : if you can calculate the cost of Keyboard, then you should be able to calculate the cost of Peripheral also and hence Cabinet in turn.



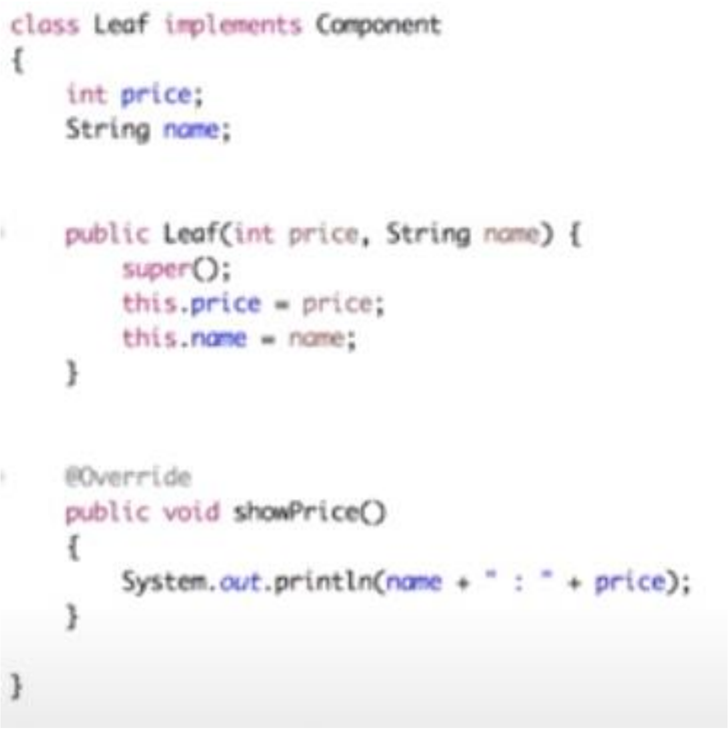


```java
package com.telusko.composite;

public class CompositeTest {

    public static void main(String[] args)
    {


    }

}
```

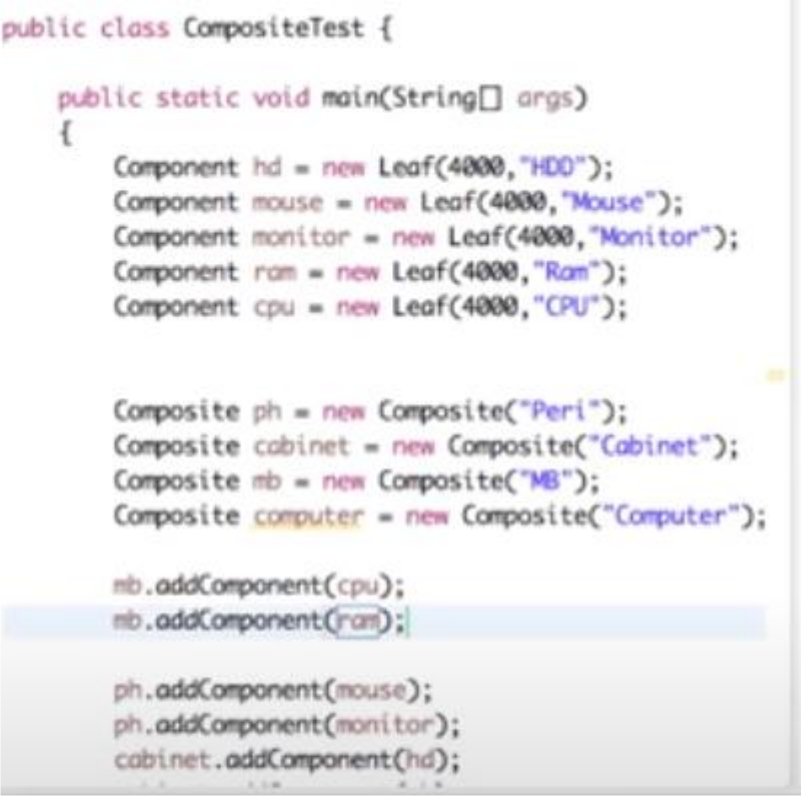
```java
package com.telusko.composite;

interface Component
{
    void showPrice();
}

class Leaf implements Component
{

    @Override
    public void showPrice() {
        // TODO Auto-generated method stub

    }

}

class Composite implements Component
{

}
```



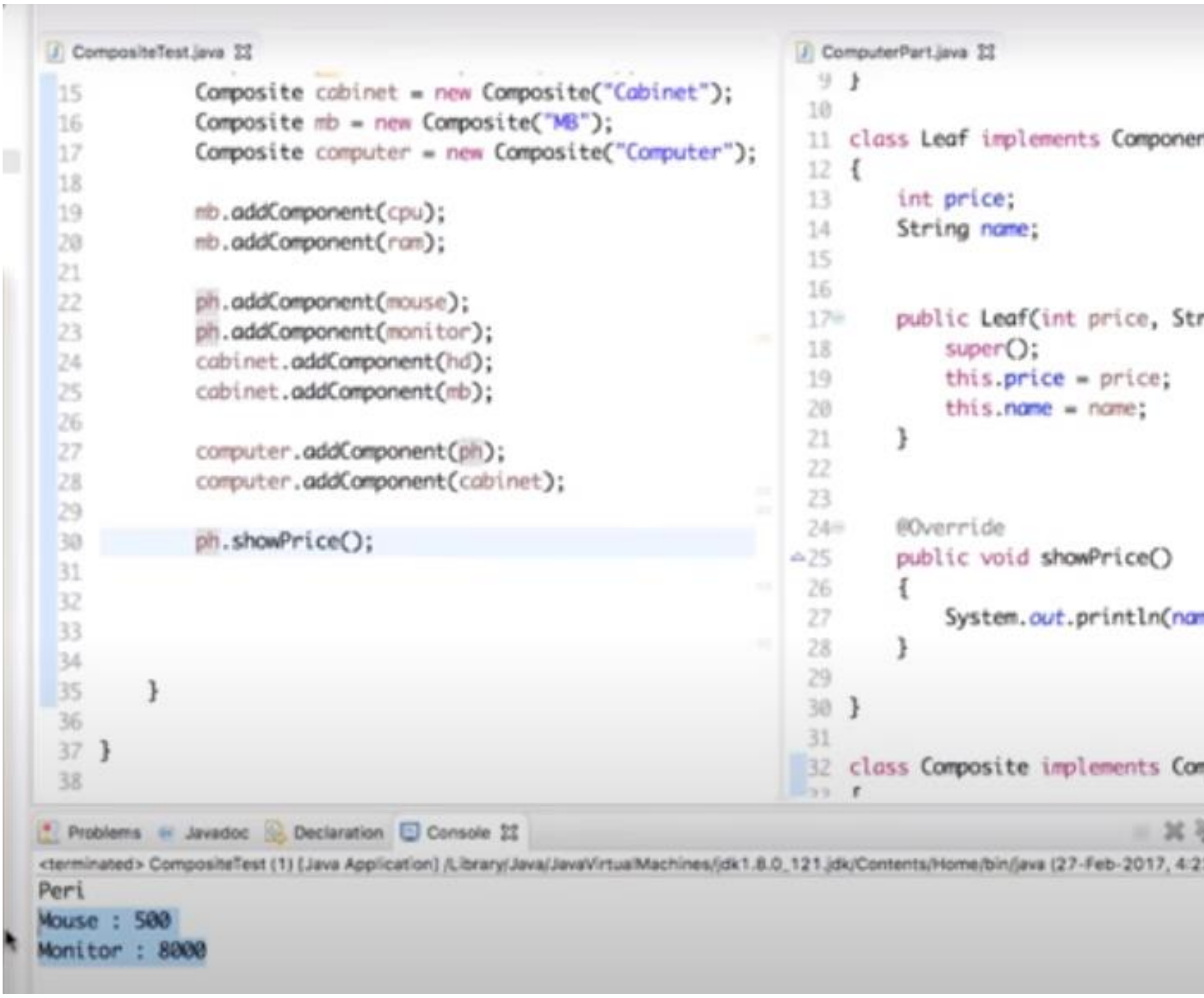
```java
class Composite implements Component
{
    String name;
    List<Component> components = new ArrayList<>();

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

    public void addComponent(Component com)
    {
        components.add(com);
    }

    @Override
    public void showPrice()
    {
        System.out.println(name);
        for(Component c : components)
        {
```



```
class Leaf implements Component
{
    int price;
    String name;


    public Leaf(int price, String name) {
        super();
        this.price = price;
        this.name = name;
    }


    @Override
    public void showPrice()
    {
        System.out.println(name + " : " + price);
    }

}
```



```
public class CompositeTest {

    public static void main(String[] args)
    {
        Component hd = new Leaf(4000,"HDD");
        Component mouse = new Leaf(4000,"Mouse");
        Component monitor = new Leaf(4000,"Monitor");
        Component ram = new Leaf(4000,"Ram");
        Component cpu = new Leaf(4000,"CPU");


        Composite ph = new Composite("Peri");
        Composite cabinet = new Composite("Cabinet");
        Composite mb = new Composite("MB");
        Composite computer = new Composite("Computer");

        mb.addComponent(cpu);
        mb.addComponent(ram);

        ph.addComponent(mouse);
        ph.addComponent(monitor);
        cabinet.addComponent(hd);
```



```
        Composite cabinet = new Composite("Cabinet");
        Composite mb = new Composite("MB");
        Composite computer = new Composite("Computer");

        mb.addComponent(cpu);
        mb.addComponent(ram);

        ph.addComponent(mouse);
        ph.addComponent(monitor);
        cabinet.addComponent(hd);
        cabinet.addComponent(mb);

        computer.addComponent(ph);
        computer.addComponent(cabinet);

        ph.showPrice();




    }

}
```

```
}

class Leaf implements Componer
{
    int price;
    String name;


    public Leaf(int price, Str
        super();
        this.price = price;
        this.name = name;
    }


    @Override
    public void showPrice()
    {
        System.out.println(nar
    }

}

class Composite implements Com
{
```

```
<terminated> CompositeTest (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (27-Feb-2017, 4:2
Peri
Mouse : 500
Monitor : 8000
```

If we print computer.showPrice then it should print price for all the child objects.

Example 2 :-

```
package composite;

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

abstract class Account {
  public abstract float getBalance();
}

class DepositeAccount extends Account {
  private String accountNo;
  private float accountBalance;
```

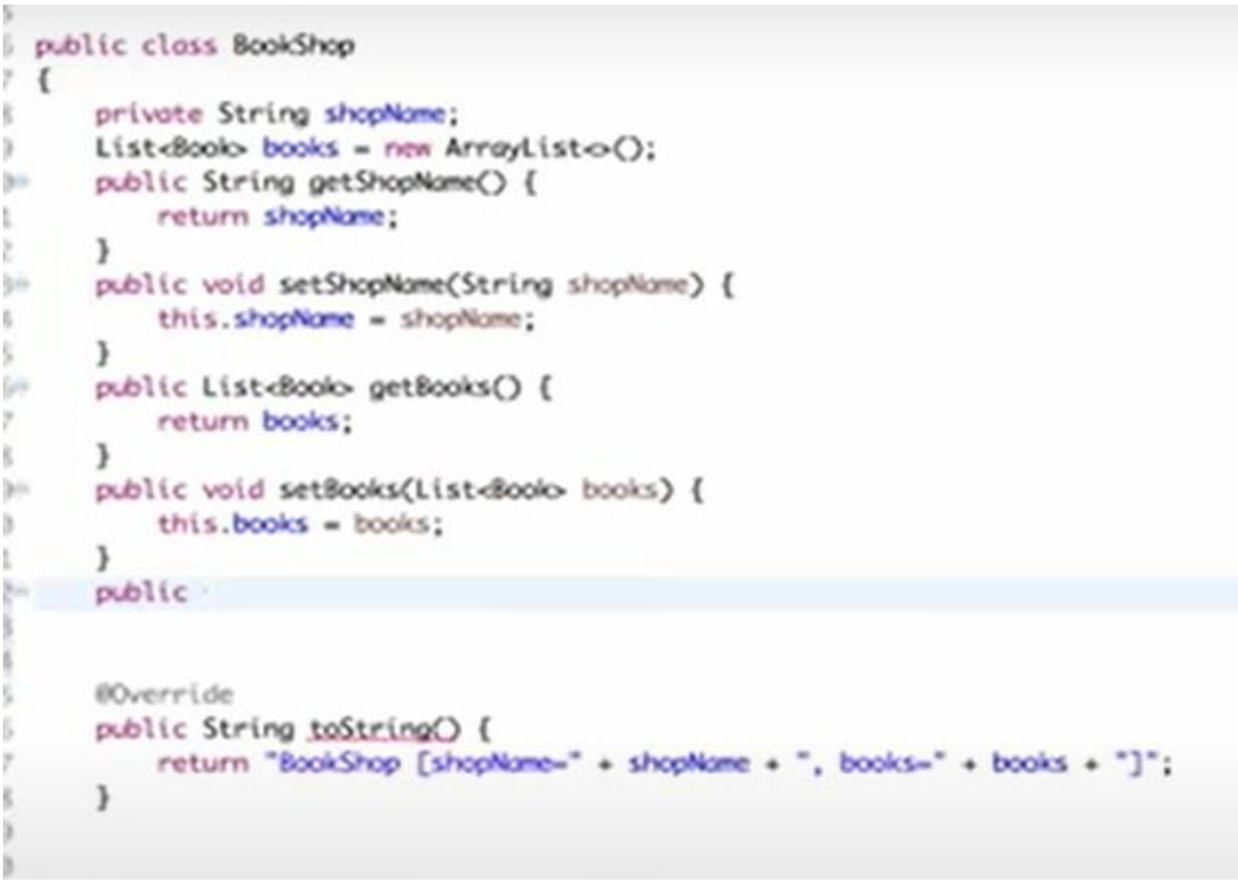
```java
  public DepositeAccount(String accountNo, float accountBalance) {
    super();
    this.accountNo = accountNo;
    this.accountBalance = accountBalance;
  }

  public float getBalance() {
    return accountBalance;
  }

}

class SavingAccount extends Account {
  private String accountNo;
  private float accountBalance;

  public SavingAccount(String accountNo, float accountBalance) {
    super();
    this.accountNo = accountNo;
    this.accountBalance = accountBalance;
  }

  public float getBalance() {
    return accountBalance;
  }
}

class CompositeAccount extends Account {
  private float totalBalance;
  private List<Account> accountList = new ArrayList<Account>();

  public float getBalance() {
    totalBalance = 0;
    for (Account f : accountList) {
      totalBalance = totalBalance + f.getBalance();
    }
    return totalBalance;
  }

  public void addAccount(Account acc) {
    accountList.add(acc);
  }

  public void removeAccount(Account acc) {
    accountList.add(acc);
  }
}

public class Client {

  public static void main(String[] args) {
    CompositeAccount component = new CompositeAccount();

    component.addAccount(new DepositeAccount("DA001", 100));
    component.addAccount(new DepositeAccount("DA002", 150));
    component.addAccount(new SavingAccount("SA001", 200));

    float totalBalance = component.getBalance();
    System.out.println("Total Balance : " + totalBalance);
  }

}
```

Prototype Design pattern(Creational)

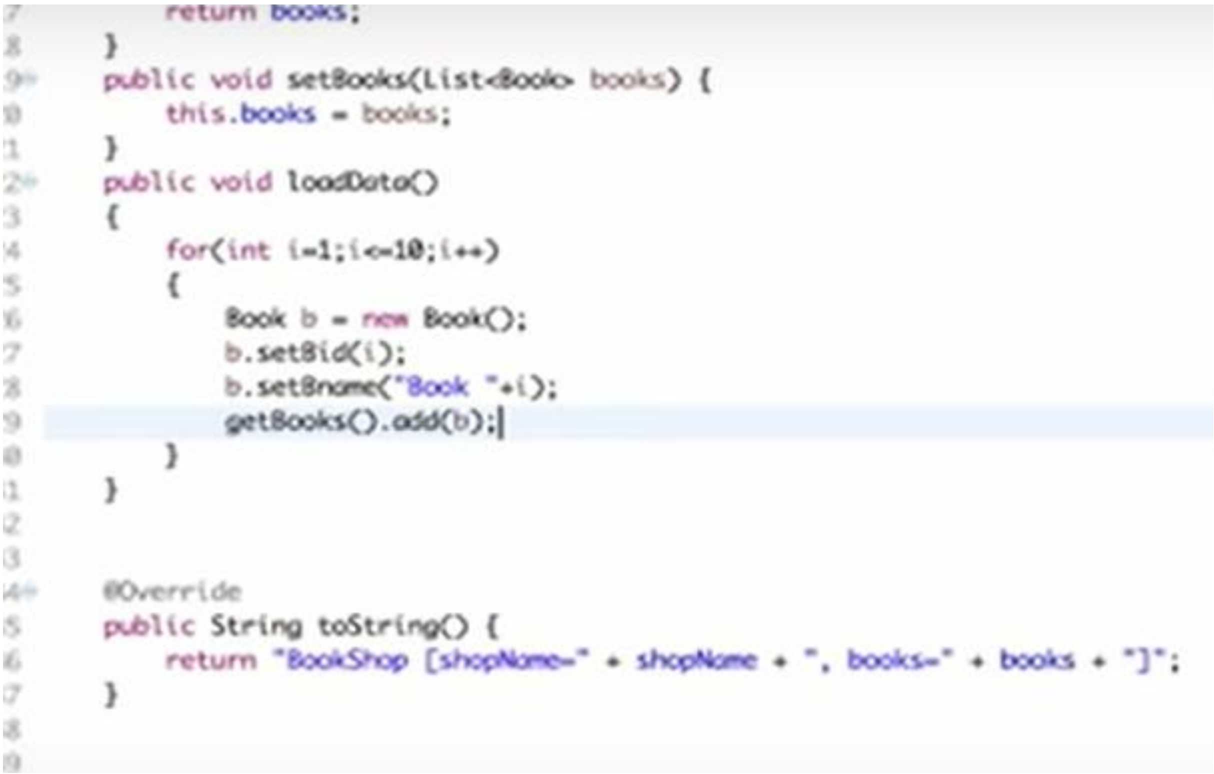
This is used when an object creation is expensive and we want to make use of existing object and try to create a new object out of it.

We use clone method to create a new object from existing copy of the object.

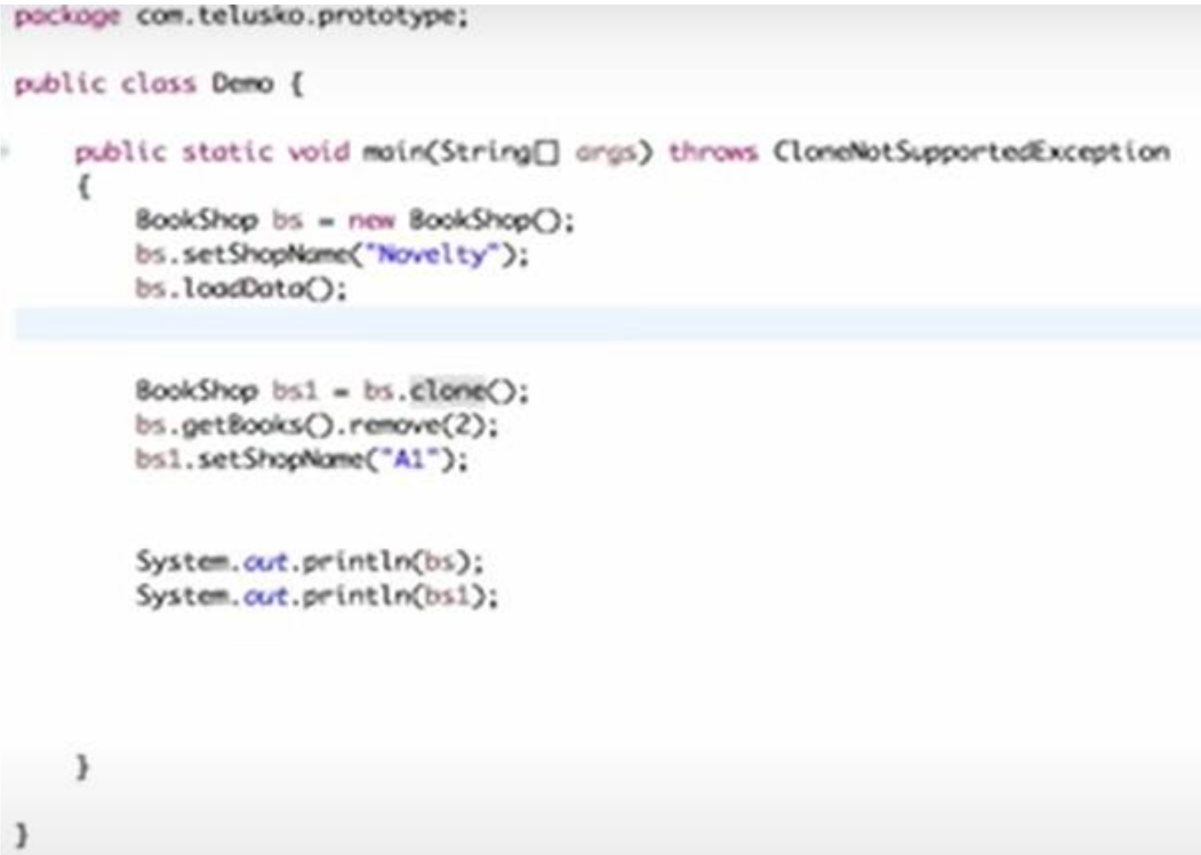
Creating just clone(Calling clone) method will create a shallow copy of the object. We will have to override the clone method and provide own implementation to create a deep copy of the object.



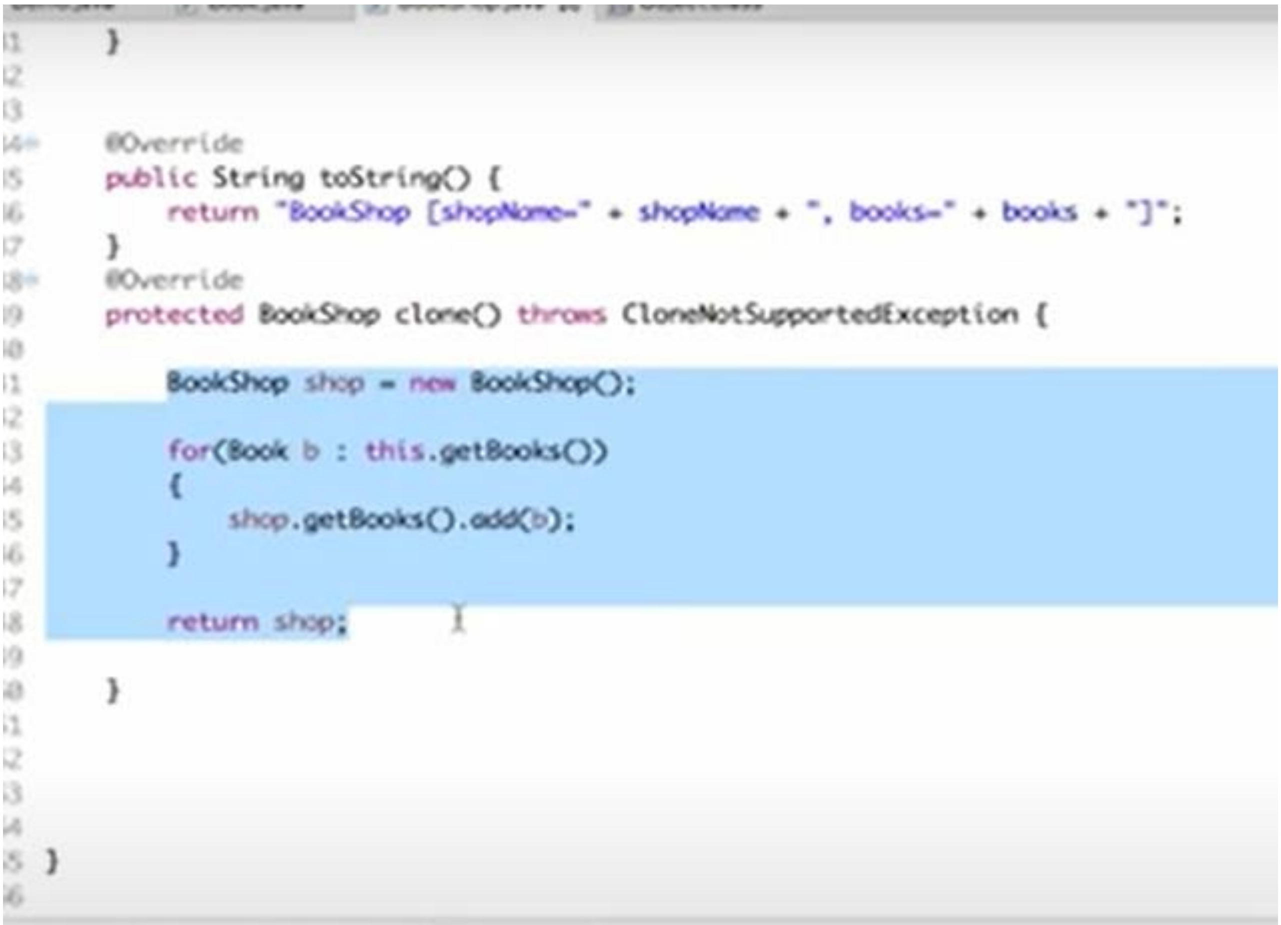
```
public class BookShop
{
    private String shopName;
    List<Book> books = new ArrayList<>();
    public String getShopName() {
        return shopName;
    }
    public void setShopName(String shopName) {
        this.shopName = shopName;
    }
    public List<Book> getBooks() {
        return books;
    }
    public void setBooks(List<Book> books) {
        this.books = books;
    }
    public



    @Override
    public String toString() {
        return "BookShop [shopName=" + shopName + ", books=" + books + "]";
    }
```



```
        return books;
    }
    public void setBooks(List<Book> books) {
        this.books = books;
    }
    public void loadData()
    {
        for(int i=1;i<=10;i++)
        {
            Book b = new Book();
            b.setBid(i);
            b.setBname("Book "+i);
            getBooks().add(b);
        }
    }


    @Override
    public String toString() {
        return "BookShop [shopName=" + shopName + ", books=" + books + "]";
    }
```



```
package com.telusko.prototype;

public class Demo {

    public static void main(String[] args) throws CloneNotSupportedException
    {
        BookShop bs = new BookShop();
        bs.setShopName("Novelty");
        bs.loadData();


        BookShop bs1 = bs.clone();
        bs.getBooks().remove(2);
        bs1.setShopName("A1");


        System.out.println(bs);
        System.out.println(bs1);




    }

}
```

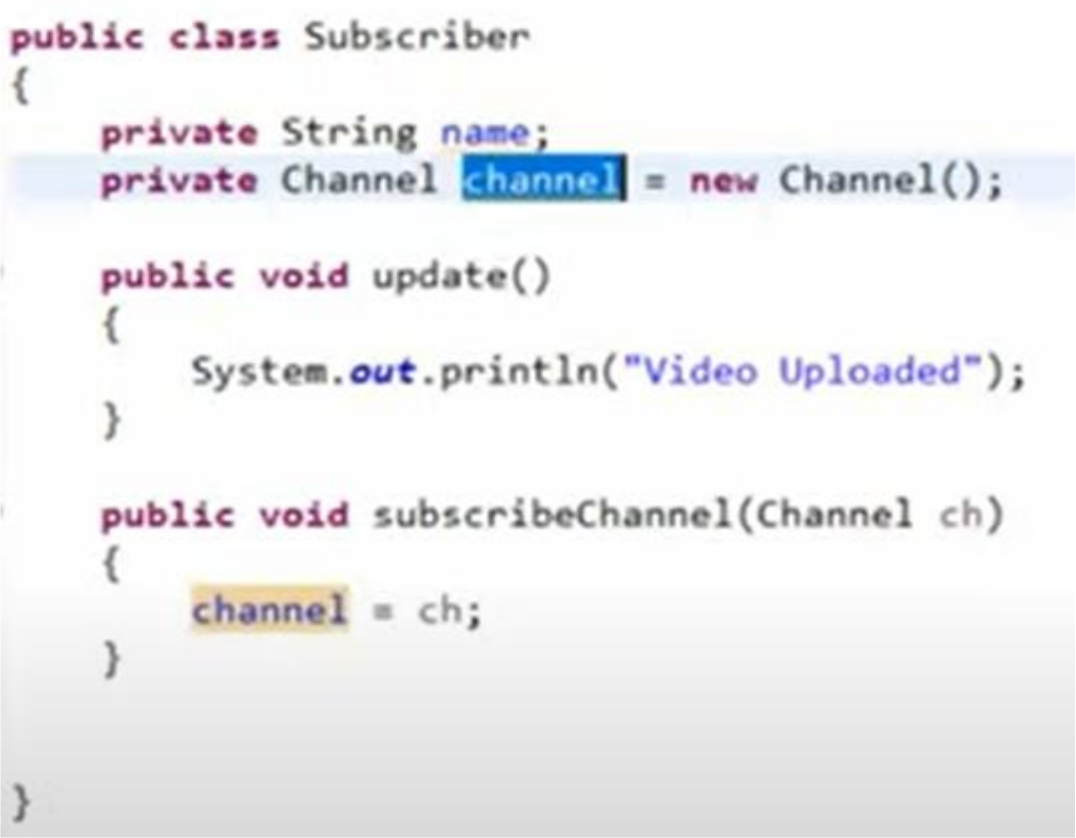


```
    }


    @Override
    public String toString() {
        return "BookShop [shopName=" + shopName + ", books=" + books + "]";
    }
    @Override
    protected BookShop clone() throws CloneNotSupportedException {

        BookShop shop = new BookShop();

        for(Book b : this.getBooks())
        {
            shop.getBooks().add(b);
        }

        return shop;

    }




}
```

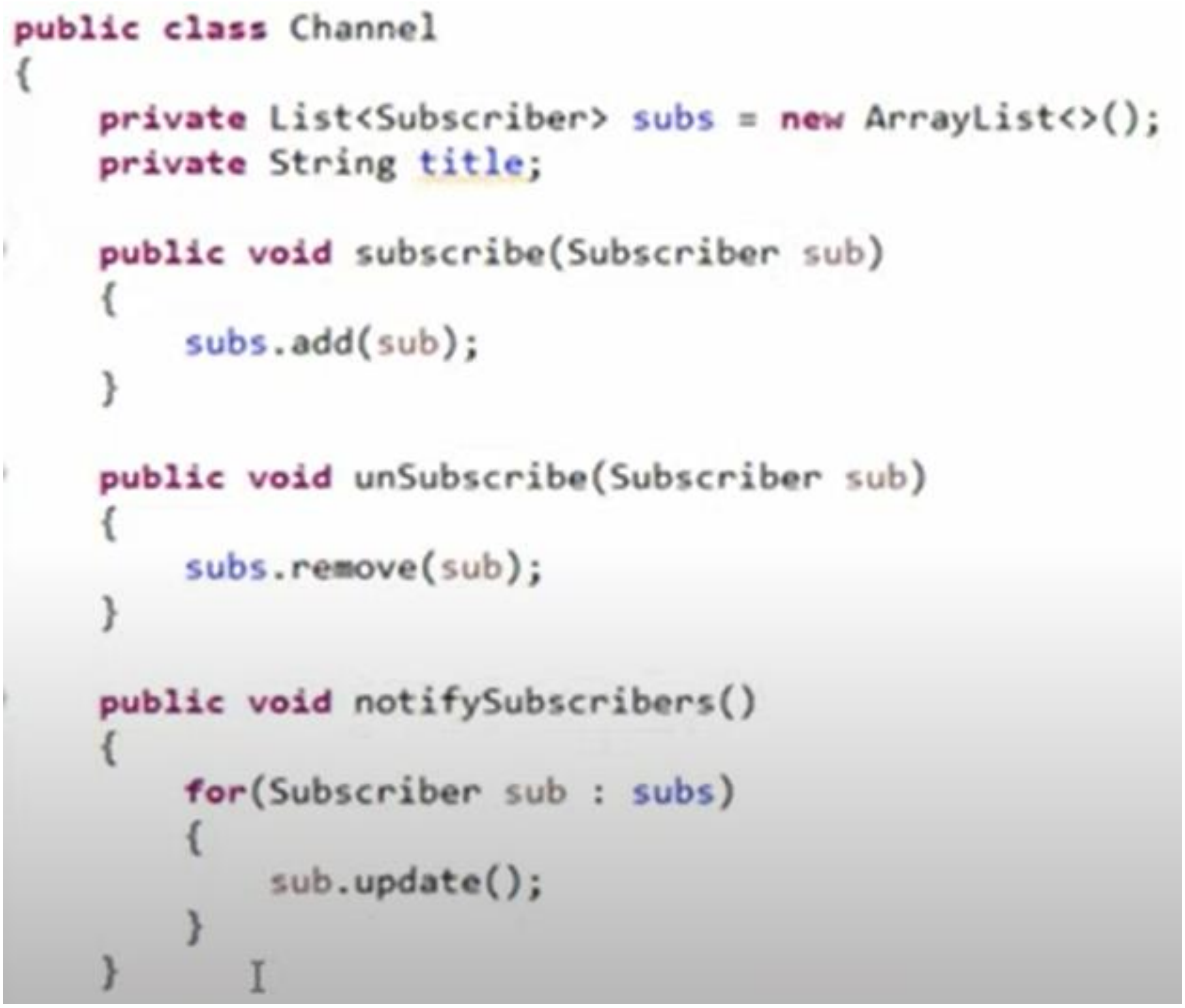
Observer Design pattern (Behavioural):-

Used mostly when you want to notify multiple objects about one object.

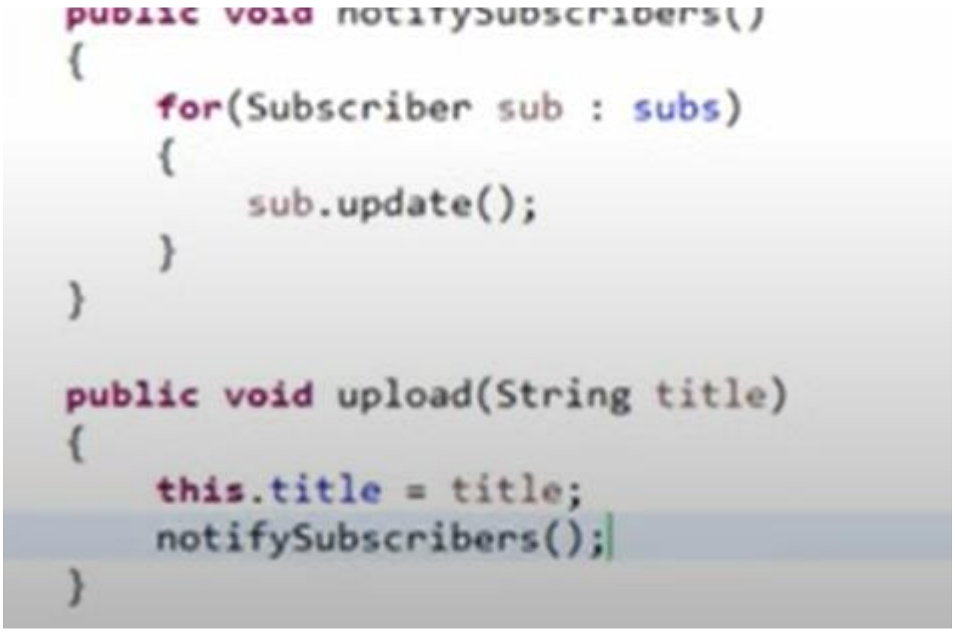
For ex : You want to notify all of the subscribers of a channel when you get to know that a youtube video is uploaded.



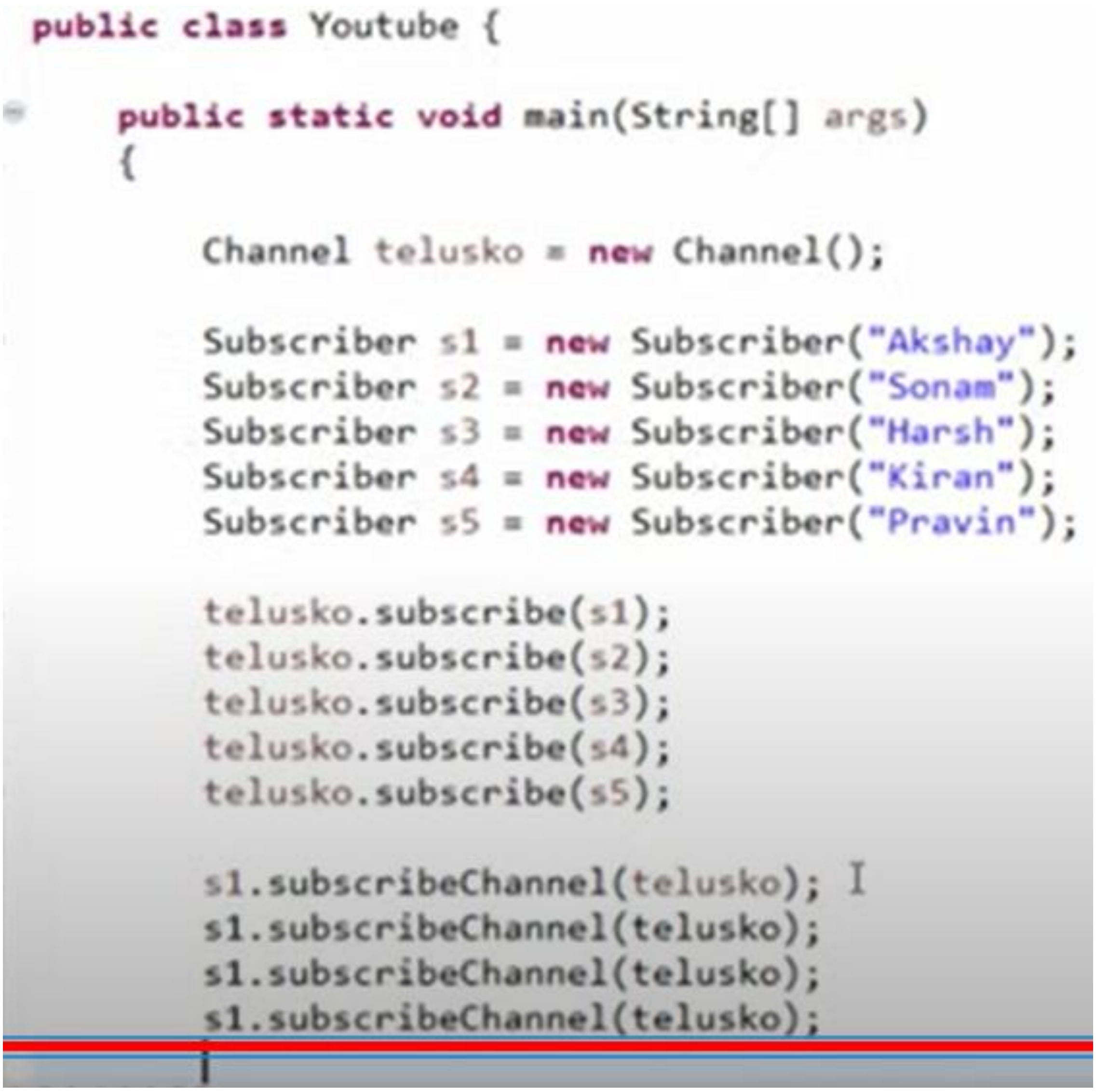
```
public class Subscriber
{
    private String name;
    private Channel channel = new Channel();

    public void update()
    {
        System.out.println("Video Uploaded");
    }

    public void subscribeChannel(Channel ch)
    {
        channel = ch;
    }


}
```



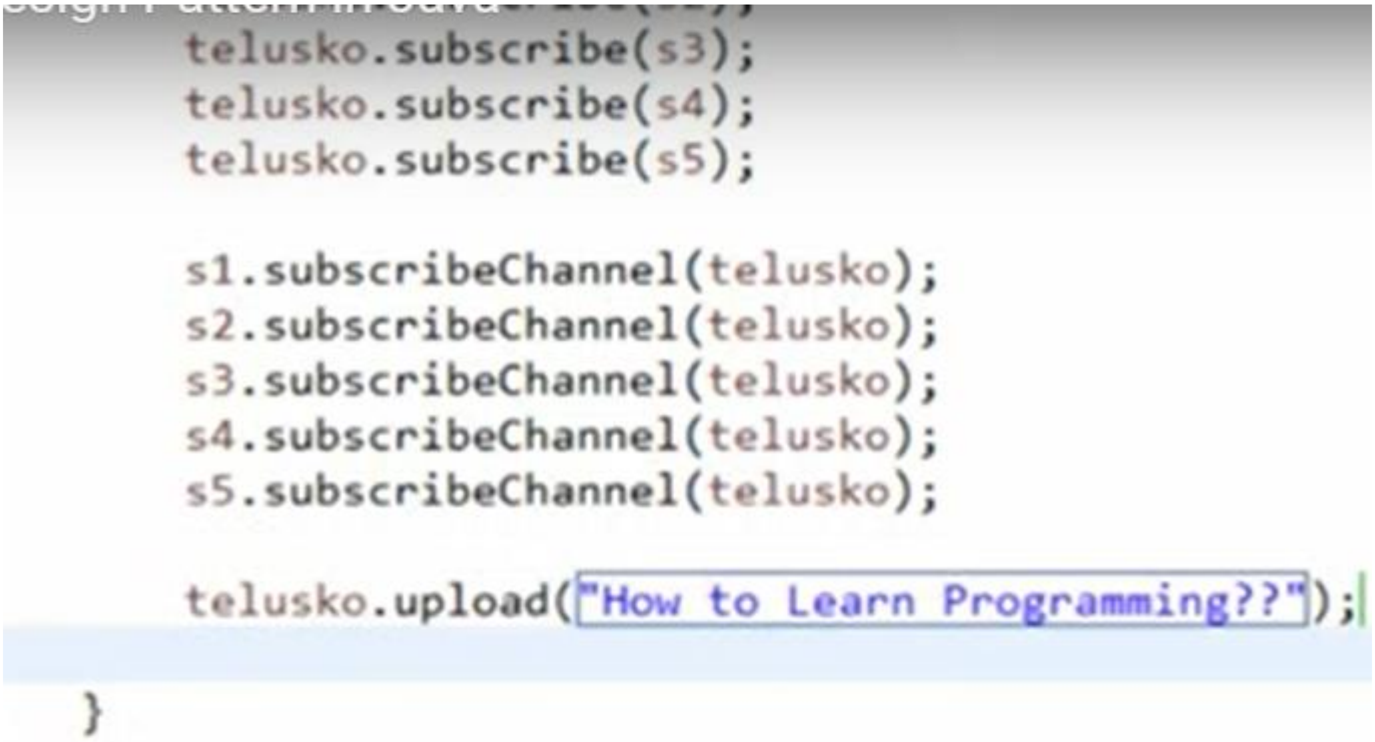
```
public class Channel
{
    private List<Subscriber> subs = new ArrayList<>();
    private String title;

    public void subscribe(Subscriber sub)
    {
        subs.add(sub);
    }

    public void unSubscribe(Subscriber sub)
    {
        subs.remove(sub);
    }

    public void notifySubscribers()
    {
        for(Subscriber sub : subs)
        {
            sub.update();
        }
    }
```



```
public void notifySubscribers()
{
    for(Subscriber sub : subs)
    {
        sub.update();
    }
}

public void upload(String title)
{
    this.title = title;
    notifySubscribers();
}
```



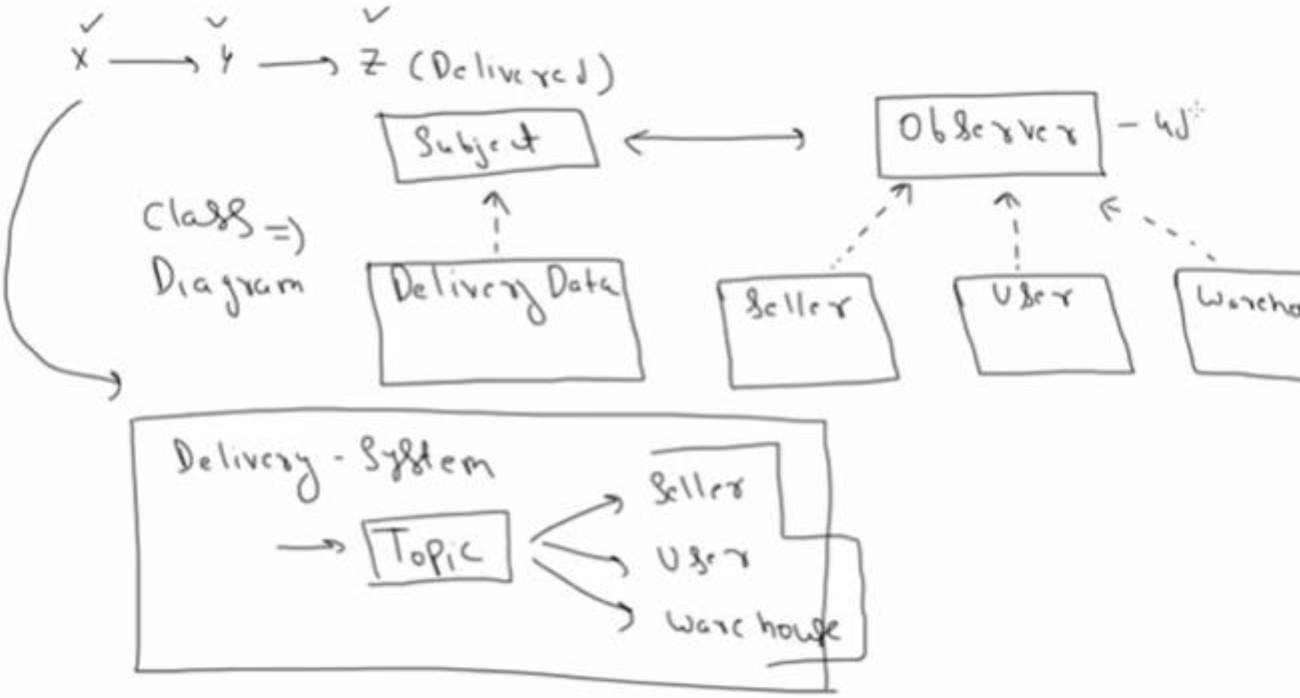
```java
public class Youtube {

    public static void main(String[] args)
    {

        Channel telusko = new Channel();

        Subscriber s1 = new Subscriber("Akshay");
        Subscriber s2 = new Subscriber("Sonam");
        Subscriber s3 = new Subscriber("Harsh");
        Subscriber s4 = new Subscriber("Kiran");
        Subscriber s5 = new Subscriber("Pravin");

        telusko.subscribe(s1);
        telusko.subscribe(s2);
        telusko.subscribe(s3);
        telusko.subscribe(s4);
        telusko.subscribe(s5);

        s1.subscribeChannel(telusko);
        s1.subscribeChannel(telusko);
        s1.subscribeChannel(telusko);
        s1.subscribeChannel(telusko);
```



```
        telusko.subscribe(s3);
        telusko.subscribe(s4);
        telusko.subscribe(s5);

        s1.subscribeChannel(telusko);
        s2.subscribeChannel(telusko);
        s3.subscribeChannel(telusko);
        s4.subscribeChannel(telusko);
        s5.subscribeChannel(telusko);

        telusko.upload("How to Learn Programming??");

    }
```

Example 2:- Coding Simplified

Whenever there is a change in system, update or notify to multiple observers. You have to register the observers as well before for particular Observee. In this case Observee is Delievery Data, Lets say if we have one more observe i.e. Price, and we want to notify user and seller about the price Up so we have to register User and seller observers to Price Observee.



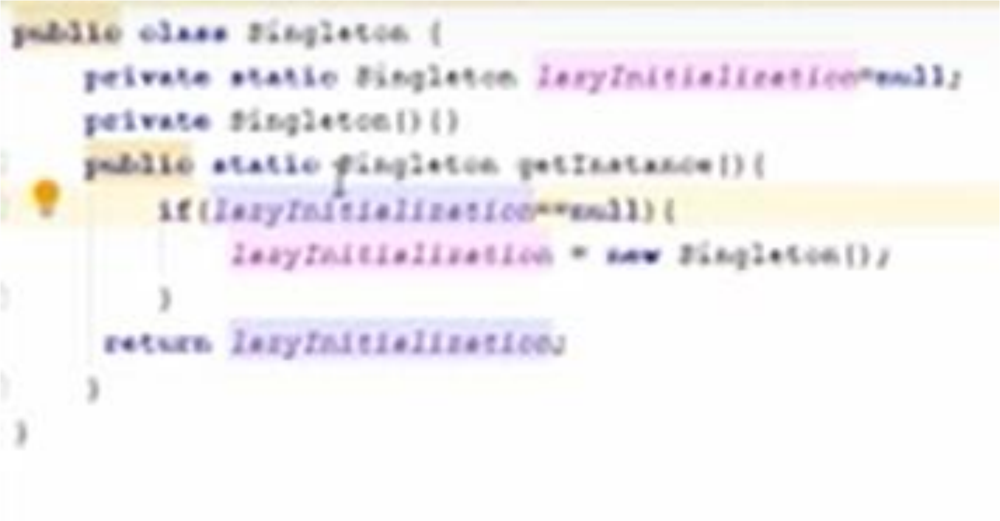
```
package observer;
```

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

interface Subject {
  void register(Observer obj);
  void unregister(Observer obj);
  void notifyObservers();
}

class DeliveryData implements Subject {

  private List<Observer> observers;
  private String location;

  public DeliveryData() {
    this.observers = new ArrayList<>();
  }

  @Override
  public void register(Observer obj) {
    observers.add(obj);
  }

  @Override
  public void unregister(Observer obj) {
      observers.remove(obj);
  }

  @Override
  public void notifyObservers() {
    for(Observer obj : observers) {
      obj.update(location);
    }
  }

  public void locationChanged() {
    this.location = getLocation();
    notifyObservers();
  }

  public String getLocation() {
    return "YPlace";
  }
}


interface Observer {
  public void update(String location);
}
```

```
class Seller implements Observer {
  private String location;

  @Override
  public void update(String location) {
    this.location = location;
    showLocation();
  }

  public void showLocation() {
    System.out.println("Notification at Seller - Current Location: "
 + location);
  }
}

class User implements Observer {
  private String location;

  @Override
  public void update(String location) {
    this.location = location;
    showLocation();
  }

  public void showLocation() {
    System.out.println("Notification at User - Current Location: " +
 location);
  }
}

class DeliveryWarehouseCenter implements Observer {
  private String location;

  @Override
  public void update(String location) {
    this.location = location;
    showLocation();
  }

  public void showLocation() {
    System.out.println("Notification at Warehouse - Current Location
: " + location);
  }
}

public class ObserverPatternTest {

  public static void main(String[] args) {
    DeliveryData topic = new DeliveryData();

    Observer obj1 = new Seller();
    Observer obj2 = new User();
    Observer obj3 = new DeliveryWarehouseCenter();

    topic.register(obj1);
    topic.register(obj2);
    topic.register(obj3);

    topic.locationChanged();

    topic.unregister(obj3);

    System.out.println();
    topic.locationChanged();

  }
}
```

Singleton Design pattern :- (Creational)

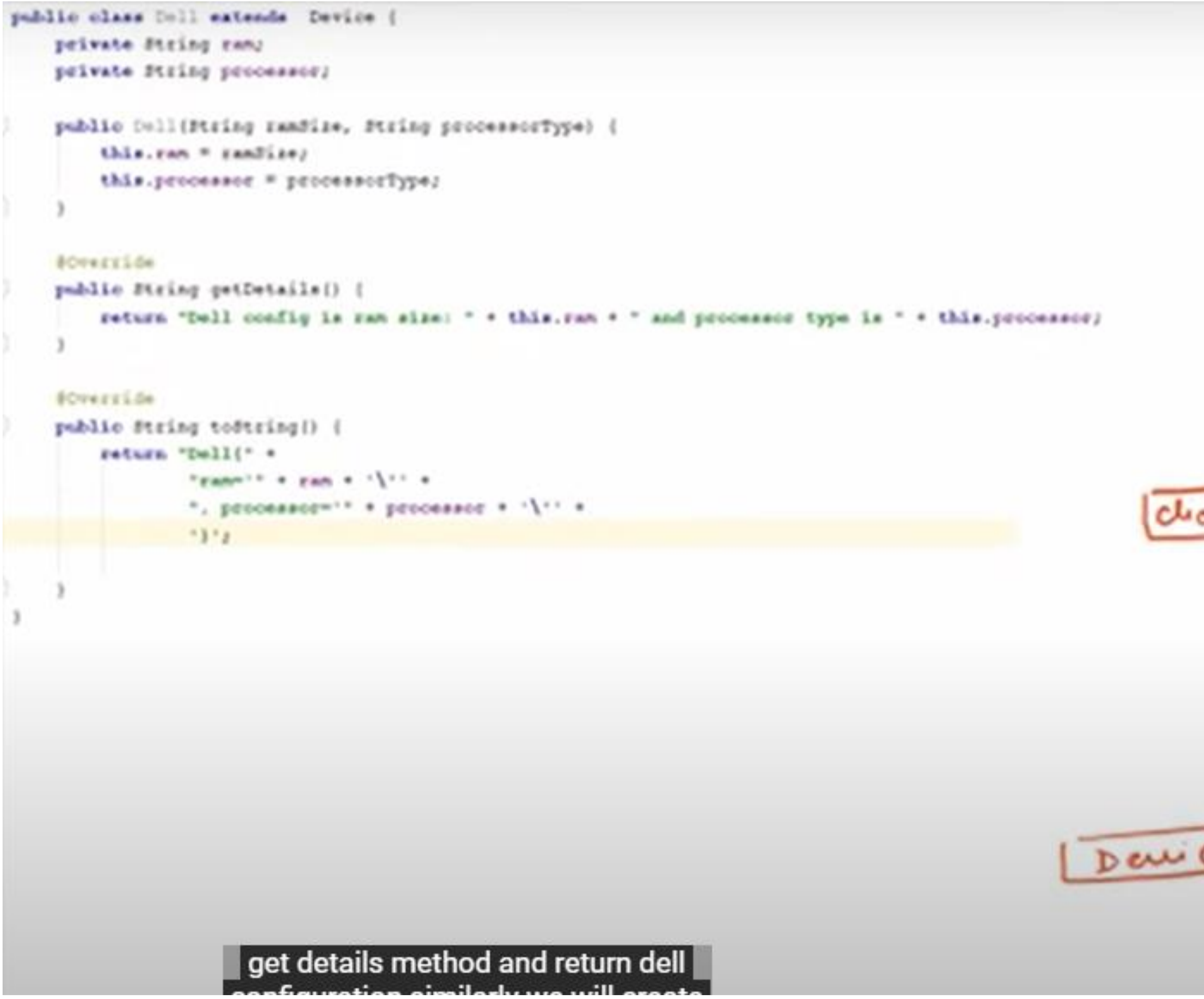
Used mostly when only one object is required for any class, usually used in logging caching extra where object does not changes.



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

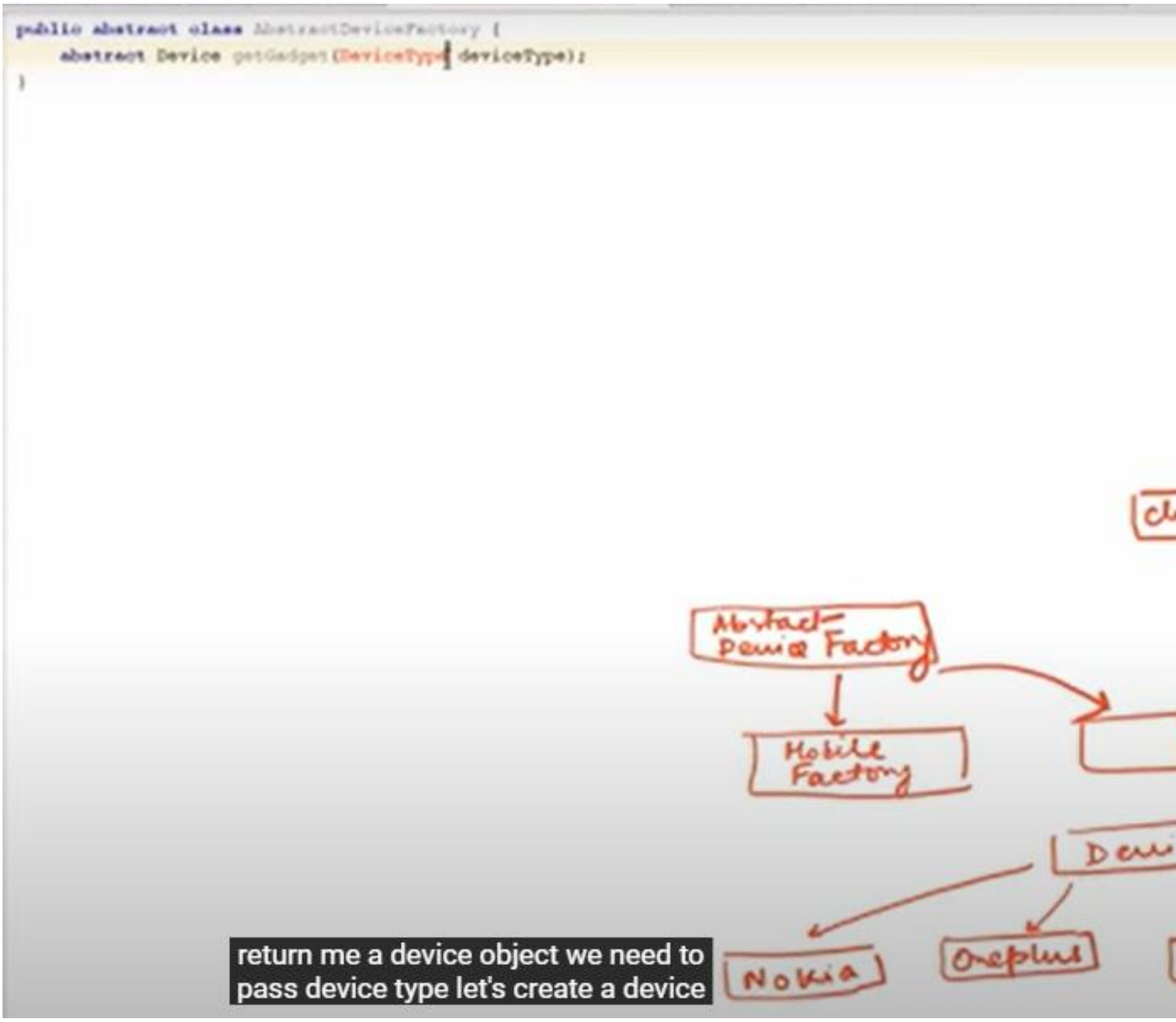
Abstract Factory Design Pattern: (Creational)

It's also called as Factory of factories. Where abstract factory calls separate factories which generates the objects of the classes.



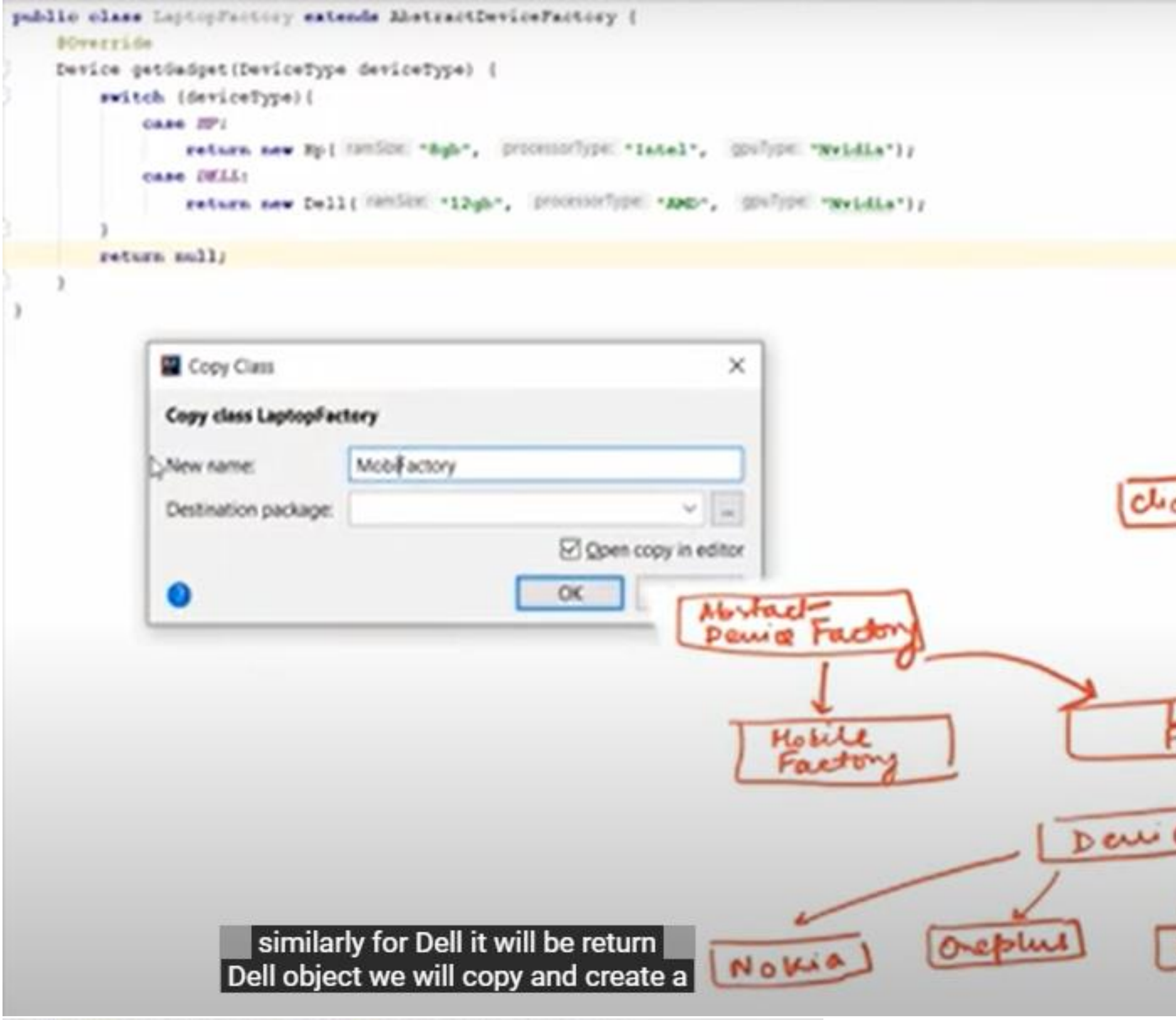
```
public class Dell extends  Device {
    private String ram;
    private String processor;

    public Dell(String ramSize, String processorType) {
        this.ram = ramSize;
        this.processor = processorType;
    }

    @Override
    public String getDetails() {
        return "Dell config is ram size: " + this.ram + " and processor type is " + this.processor;
    }

    @Override
    public String toString() {
        return "Dell{" +
                "ram='" + ram + '\'' +
                ", processor='" + processor + '\'' +
                '}';

    }
}
```

get details method and return dell
configuration similarly we will create



```
public abstract class AbstractDeviceFactory {
    abstract Device getGadget(DeviceType deviceType);
}
```

return me a device object we need to pass device type let's create a device



```java
public class LaptopFactory extends AbstractDeviceFactory {
    @Override
    Device getGadget(DeviceType deviceType) {
        switch (deviceType){
            case HP:
                return new Hp( ramSize: "8gb",  processorType: "Intel",  gpuType: "Nvidia");
            case DELL:
                return new Dell( ramSize: "12gb",  processorType: "AMD",  gpuType: "Nvidia");
        }
        return null;
    }
}
```

Copy Class

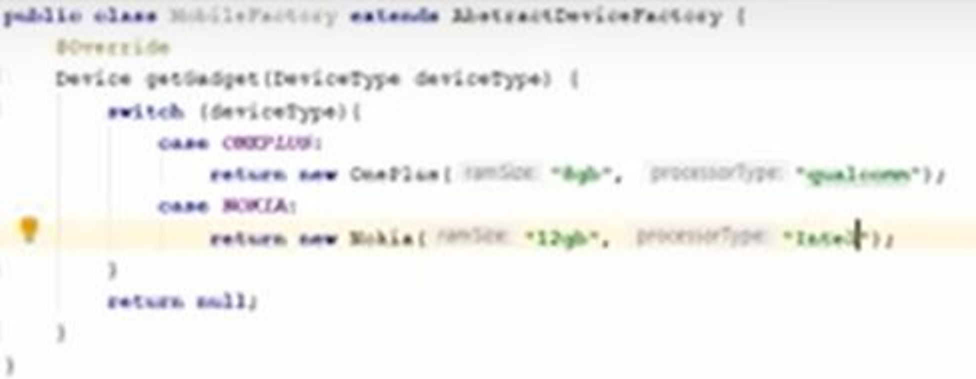
Copy class LaptopFactory

New name: MobiFactory

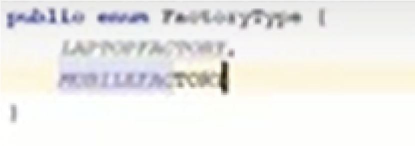
Destination package:

☑ Open copy in editor

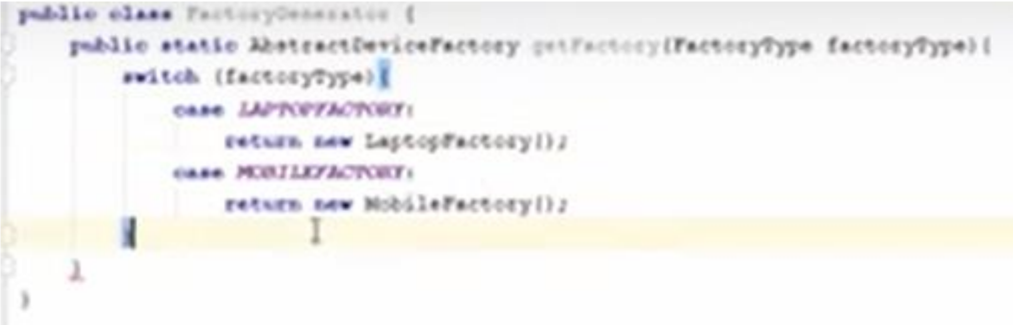
similarly for Dell it will be return
Dell object we will copy and create a



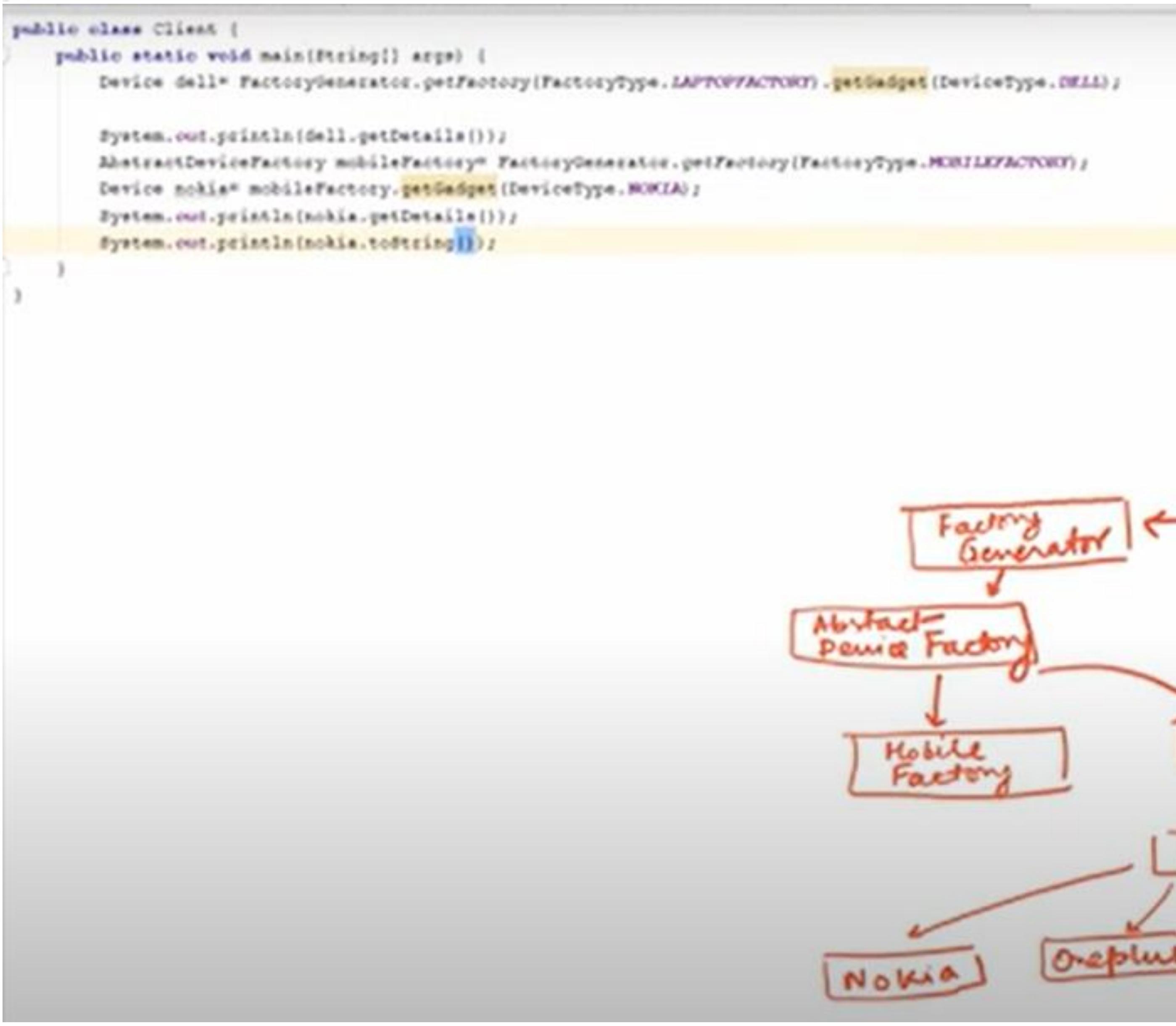
```java
public class MobileFactory extends AbstractDeviceFactory {
    @Override
    Device getGadget(DeviceType deviceType) {
        switch (deviceType){
            case ONEPLUS:
                return new OnePlus( ramSize: "8gb",  processorType: "qualcomm");
            case NOKIA:
                return new Nokia( ramSize: "12gb",  processorType: "Intel");
        }
        return null;
    }
}
```



```
public enum FactoryType {
    LAPTOPFACTORY,
    MOBILEFACTORY
}
```

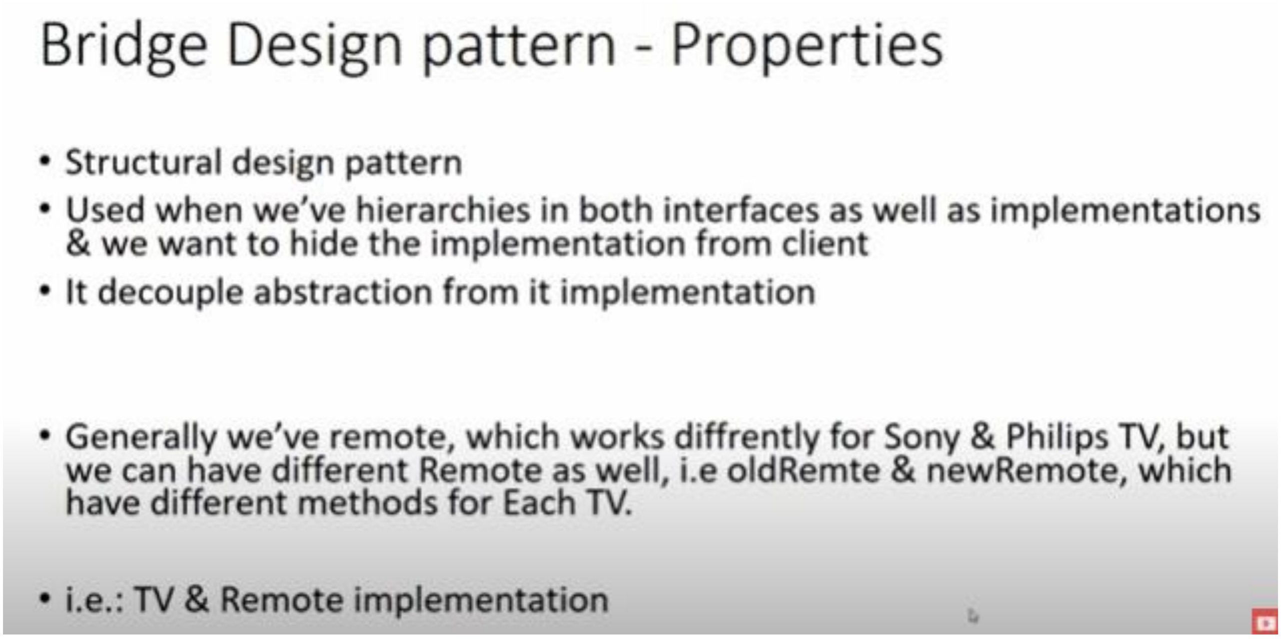


```
public class FactoryGenerator {
    public static AbstractDeviceFactory getFactory(FactoryType factoryType){
        switch (factoryType){
            case LAPTOPFACTORY:
                return new LaptopFactory();
            case MOBILEFACTORY:
                return new MobileFactory();
        }
    }
}
```



```
public class Client {
    public static void main(String[] args) {
        Device dell= FactoryGenerator.getFactory(FactoryType.LAPTOPFACTORY).getGadget(DeviceType.DELL);

        System.out.println(dell.getDetails());
        AbstractDeviceFactory mobileFactory= FactoryGenerator.getFactory(FactoryType.MOBILEFACTORY);
        Device nokia= mobileFactory.getGadget(DeviceType.NOKIA);
        System.out.println(nokia.getDetails());
        System.out.println(nokia.toString());
    }
}
```

Bridge Design Pattern (Structural)



Bridge Design pattern - Properties

- Structural design pattern
- Used when we've hierarchies in both interfaces as well as implementations & we want to hide the implementation from client
- It decouple abstraction from it implementation

- Generally we've remote, which works diffrently for Sony & Philips TV, but we can have different Remote as well, i.e oldRemte & newRemote, which have different methods for Each TV.

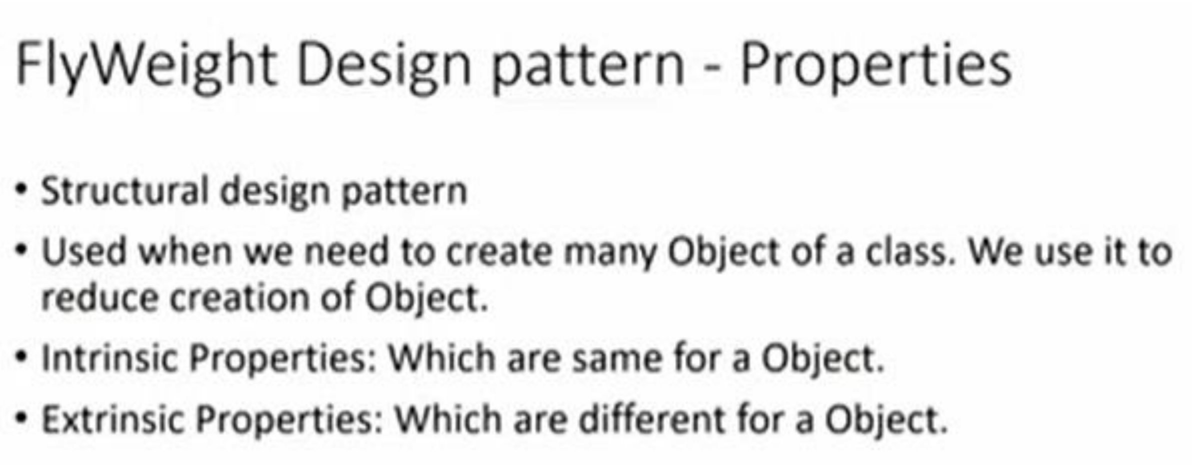
- i.e.: TV & Remote implementation

```java
package bridge;

abstract class TV {
  Remote remote;

  TV(Remote r) {
    this.remote = r;
  }

  abstract void on();
  abstract void off();
}

class Sony extends TV {
  Remote remoteType;
  Sony(Remote r) {
    super(r);
    this.remoteType = r;
  }

  public void on(){
    System.out.print("Sony TV ON: ");
    remoteType.on();
  }

  public void off(){
    System.out.print("Sony TV OFF: ");
    remoteType.off();
  }
}
```

```
class Philips extends TV {
  Remote remoteType;

  Philips(Remote r) {
    super(r);
    this.remoteType = r;
  }

  public void on(){
    System.out.print("Philips TV ON: ");
    remoteType.on();
  }

  public void off(){
    System.out.print("Philips TV OFF: ");
    remoteType.off();
  }
}

interface Remote {
  public void on();
  public void off();
}

class OldRemote implements Remote {

  @Override
  public void on() {
    System.out.println("ON with Old Remote");
  }

  @Override
  public void off() {
    System.out.println("OFF with old Remote");
  }

}

class NewRemote implements Remote {

  @Override
  public void on() {
    System.out.println("ON with New Remote");
  }

  @Override
  public void off() {
    System.out.println("OFF with New Remote");
  }

}
```

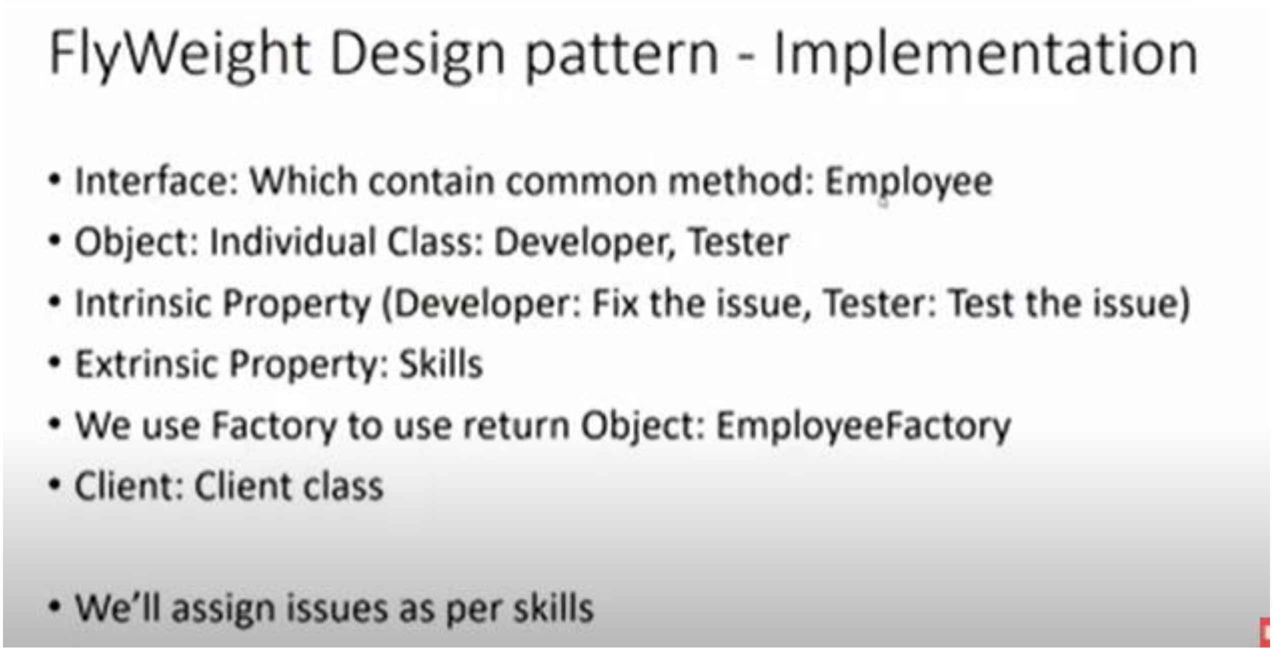
```
public class Client {
  public static void main(String[] args) {
    TV sonyOldRemote = new Sony(new OldRemote());
    sonyOldRemote.on();
    sonyOldRemote.off();
    System.out.println();

    TV sonyNewRemote = new Sony(new NewRemote());
    sonyNewRemote.on();
    sonyNewRemote.off();
    System.out.println();

    TV philipsOldRemote = new Philips(new OldRemote());
    philipsOldRemote.on();
    philipsOldRemote.off();
    System.out.println();

    TV philipsNewRemote = new Philips(new NewRemote());
    philipsNewRemote.on();
    philipsNewRemote.off();

  }
}
```

Flyweight Design Pattern(Structural)



FlyWeight Design pattern - Properties

- Structural design pattern
- Used when we need to create many Object of a class. We use it to reduce creation of Object.
- Intrinsic Properties: Which are same for a Object.
- Extrinsic Properties: Which are different for a Object.



FlyWeight Design pattern - Implementation

- Interface: Which contain common method: Employee
- Object: Individual Class: Developer, Tester
- Intrinsic Property (Developer: Fix the issue, Tester: Test the issue)
- Extrinsic Property: Skills
- We use Factory to use return Object: EmployeeFactory
- Client: Client class

- We'll assign issues as per skills

Usually used when you want to create more than 10^5 objects for same type.

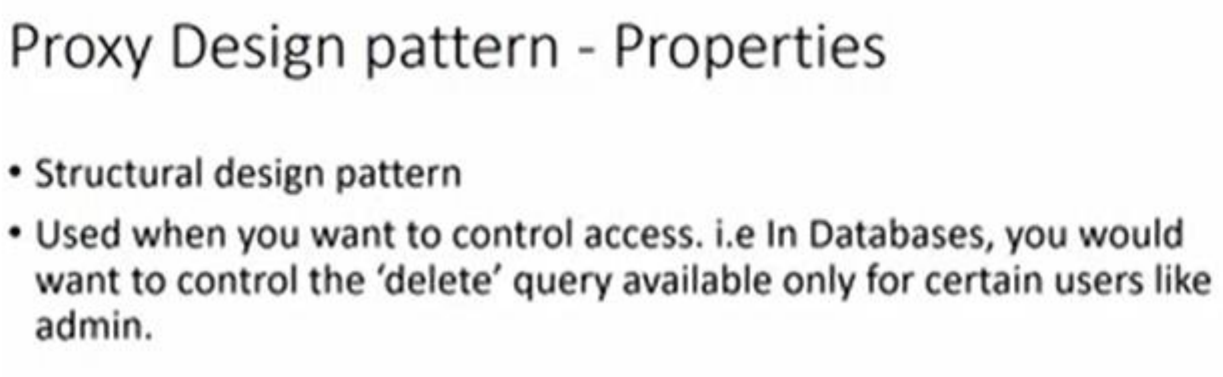
Like in counter strike game, you have a team of terrorists and anti terrorists, so for both the task is same, so we can create a single class and create multiple objects of it using a factory.

```java
package flyweight;

import java.util.HashMap;
import java.util.Random;

interface Employee {
  public void assignSkill(String skill);
  public void task();
}

class Developer implements Employee {

  private final String JOB;
  private String skill;
  public Developer() {
    JOB = "Fix the issue";
  }

  @Override
  public void assignSkill(String skill) {
    this.skill = skill;
  }

  @Override
  public void task() {
    System.out.println("Developer with skill: " + this.skill + " wit
h Job: " + JOB);
  }

}

class Tester implements Employee {

  private final String JOB;

  private String skill;

  public Tester() {
    JOB = "Test the issue";
  }

  @Override
  public void assignSkill(String skill) {
    this.skill = skill;
  }

  @Override
  public void task() {
    System.out.println("Tester with Skill: " + this.skill + " with J
ob: " + JOB);
  }

}


class EmployeeFactory {
  private static HashMap<String, Employee> m = new HashMap<String, E
mployee>();

  public static Employee getEmployee(String type) {
    Employee p = null;
    if(m.get(type) != null) {
      p = m.get(type);
    } else {
      switch(type) {
```

```
      case "Developer":
        System.out.println("Developer Created");
        p = new Developer();
        break;
      case "Tester":
        System.out.println("Tester Created");
        p = new Tester();
        break;
      default:
        System.out.println("No Such Employee");
      }

      m.put(type, p);
    }
    return p;
  }
}

public class Engineering {

  private static String employeeType[] = {"Developer", "Tester"};
  private static String skills[] = {"Java", "C++", ".Net", "Python"}
;

  public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
      Employee e = EmployeeFactory.getEmployee(getRandEmployee());

      e.assignSkill(getRandSkill());

      e.task();
    }
  }

  public static String getRandEmployee() {
    Random r = new Random();
    int randInt = r.nextInt(employeeType.length);

    return employeeType[randInt];
  }

  public static String getRandSkill() {
    Random r = new Random();
    int randInt = r.nextInt(skills.length);

    return skills[randInt];
  }

}
```

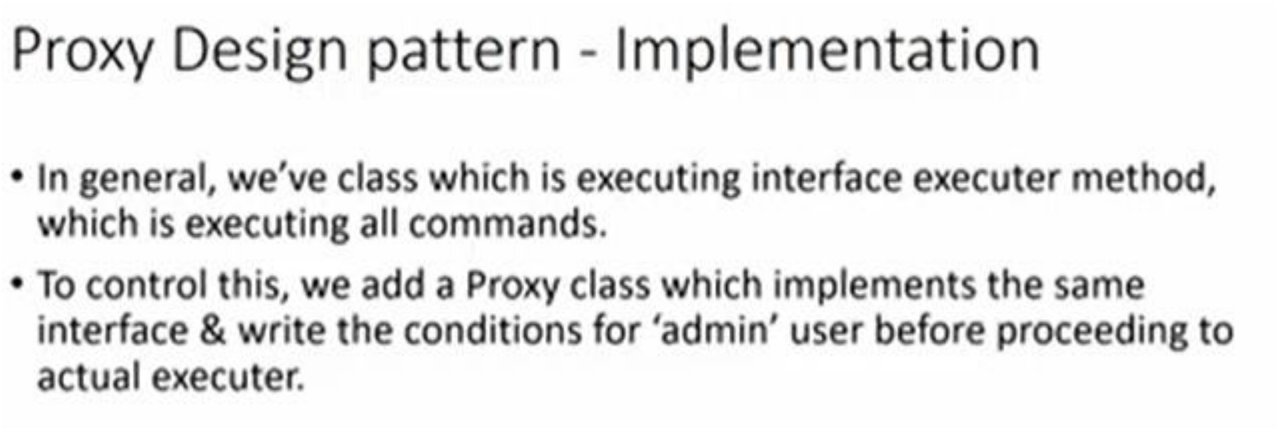
Proxy Design Pattern(Structural Design pattern)

Proxy meaning filtering -> Usually used to control the access



Proxy Design pattern - Properties

- Structural design pattern
- Used when you want to control access. i.e In Databases, you would want to control the 'delete' query available only for certain users like admin.

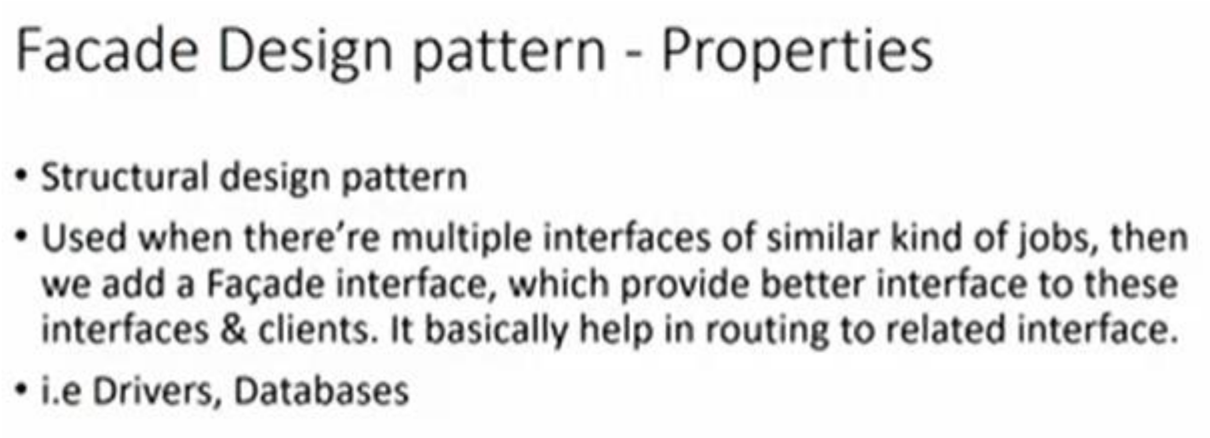


Proxy Design pattern - Implementation

- In general, we've class which is executing interface executer method, which is executing all commands.
- To control this, we add a Proxy class which implements the same interface & write the conditions for 'admin' user before proceeding to actual executer.

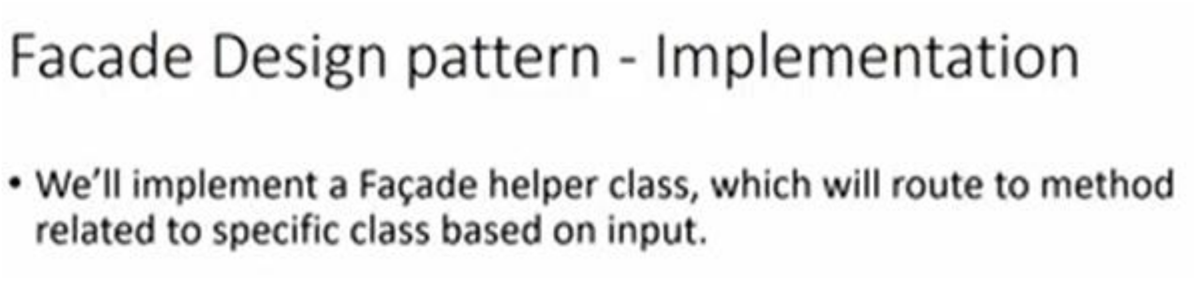
```
package proxy;
interface DatabaseExecuter {
  public void executeDatabase(String query) throws Exception;
}

class DatabaseExecuterImpl implements DatabaseExecuter {

  @Override
  public void executeDatabase(String query) throws Exception {
    System.out.println("Going to execute Query: " + query);
  }

}

class DatabaseExecuterProxy implements DatabaseExecuter {
  boolean ifAdmin;
  DatabaseExecuterImpl dbExecuter;

  public DatabaseExecuterProxy(String name, String passwd) {
    if(name == "Admin" && passwd == "Admin@123") {
      ifAdmin = true;
    }
    dbExecuter = new DatabaseExecuterImpl();
  }

  @Override
  public void executeDatabase(String query) throws Exception {
    if(ifAdmin) {
      dbExecuter.executeDatabase(query);
    } else {
      if(query.equals("DELETE")) {
        throw new Exception("DELETE not allowed for non-
admin user");
      } else {
        dbExecuter.executeDatabase(query);
      }
    }
  }
}

public class ProxyPatternExample {

  public static void main(String[] args) throws Exception {
    DatabaseExecuter nonAdminExecuter = new DatabaseExecuterProxy("N
onAdmin", "Admin@123");
    nonAdminExecuter.executeDatabase("DELEE");

    DatabaseExecuter nonAdminExecuterDELETE = new DatabaseExecuterPr
oxy("NonAdmin", "Admin@123");
    nonAdminExecuterDELETE.executeDatabase("DELETE");


    DatabaseExecuter adminExecuter = new DatabaseExecuterProxy("Admi
n", "Admin@123");
    adminExecuter.executeDatabase("DELETE");

  }

}
```

Façade Design Patterns(Structural Design pattern)



Facade Design pattern - Properties

- Structural design pattern
- Used when there're multiple interfaces of similar kind of jobs, then we add a Façade interface, which provide better interface to these interfaces & clients. It basically help in routing to related interface.
- i.e Drivers, Databases



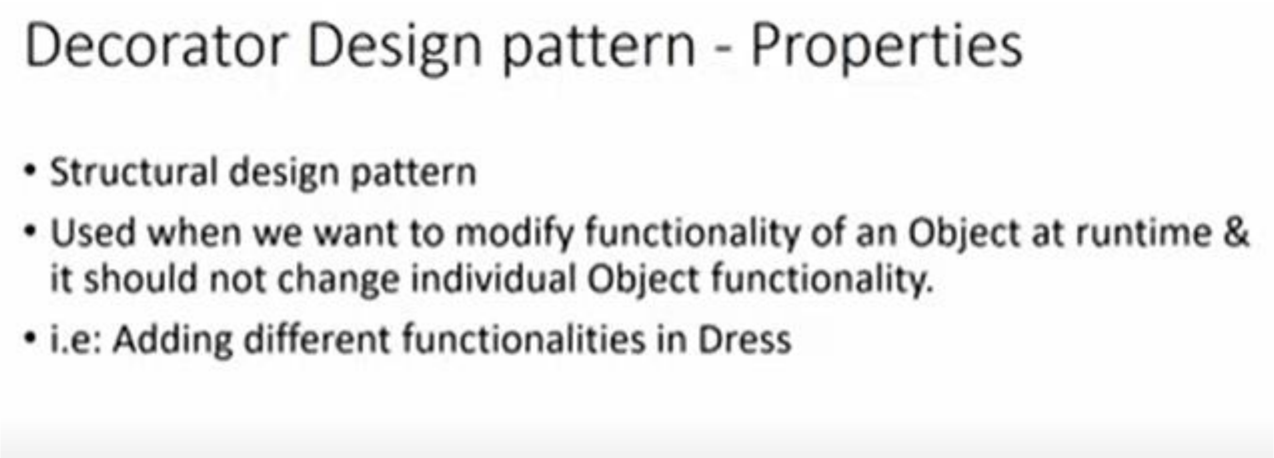
Facade Design pattern - Implementation

- We'll implement a Façade helper class, which will route to method related to specific class based on input.

```
package facade;
import java.sql.Driver;

class Firefox {
  public static Driver getFirefoxDriver() {
    return null;
  }

  public static void generateHTMLReport(String test, Driver driver)
{
    System.out.println("Generating HTML Report for Firefox Driver");
  }

  public static void generateJUnitReport(String test, Driver driver)
 {
    System.out.println("Generating JUNIT Report for Firefox Driver")
;
  }
}

class Chrome {
  public static Driver getChromeDriver() {
    return null;
  }

  public static void generateHTMLReport(String test, Driver driver)
{
    System.out.println("Generating HTML Report for Chrome Driver");
  }

  public static void generateJUnitReport(String test, Driver driver)
 {
    System.out.println("Generating JUNIT Report for Chrome Driver");
  }
}

class WebExplorerHelperFacade {
  public static void generateReports(String explorer, String report,
 String test) {
    Driver driver = null;
    switch(explorer) {
    case "firefox":
      driver = Firefox.getFirefoxDriver();
      switch(report) {
      case "html":
        Firefox.generateHTMLReport(test, driver);
        break;
      case "junit":
        Firefox.generateJUnitReport(test, driver);
        break;
      }
    break;
    case "chrome":
      driver = Chrome.getChromeDriver();
      switch(report) {
      case "html":
        Chrome.generateHTMLReport(test, driver);
        break;
      case "junit":
        Chrome.generateJUnitReport(test, driver);
        break;
      }
    }
  }
}


public class FacadePatternExample {

  public static void main(String[] args) {
    String test = "CheckElementPresent";

    WebExplorerHelperFacade.generateReports("firefox", "html", test)
;
    WebExplorerHelperFacade.generateReports("firefox", "junit", test
);
    WebExplorerHelperFacade.generateReports("chrome", "html", test);
    WebExplorerHelperFacade.generateReports("chrome", "junit", test)
;
```

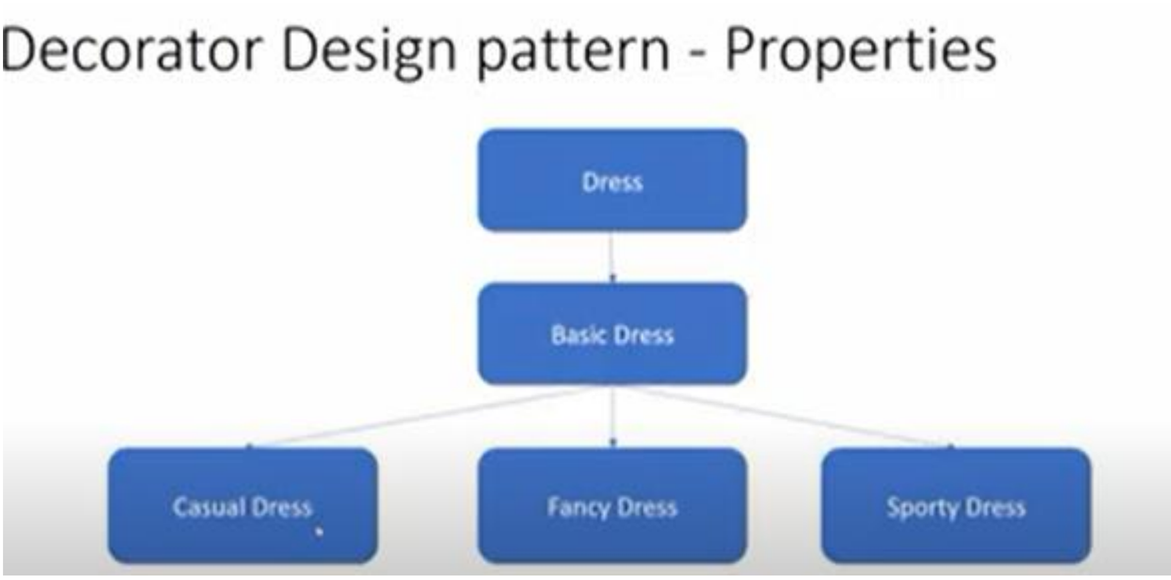
```
  }
}
```

Decorator Design Pattern



Decorator Design pattern - Properties

- Structural design pattern
- Used when we want to modify functionality of an Object at runtime & it should not change individual Object functionality.
- i.e: Adding different functionalities in Dress



Decorator Design pattern - Properties

```
package decorator;

interface Dress {
  public void assemble();
}

class BasicDress implements Dress {
  @Override
  public void assemble() {
    System.out.println("Basic Dress Features");
  }
```

```java
}

class DressDecorator implements Dress {
  protected Dress dress;

  public DressDecorator(Dress c) {
    this.dress = c;
  }

  @Override
  public void assemble() {
    this.dress.assemble();
  }
}

class CasualDress extends DressDecorator {
  public CasualDress(Dress c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.println("Adding Casual Dress Features");
  }
}


class SportyDress extends DressDecorator {
  public SportyDress(Dress c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.println("Adding Sporty Dress Features");
  }
}

class FancyDress extends DressDecorator {
  public FancyDress(Dress c) {
    super(c);
  }

  @Override
  public void assemble() {
    super.assemble();
    System.out.println("Adding Fancy Dress Features");
  }
}
```

```
public class DecoratorPatterTest {

  public static void main(String[] args) {

    Dress sportyDress = new SportyDress(new BasicDress());
    sportyDress.assemble();
    System.out.println();

    Dress fancyDress = new FancyDress(new BasicDress());
    fancyDress.assemble();
    System.out.println();

    Dress casualDress = new CasualDress(new BasicDress());
    casualDress.assemble();
    System.out.println();


    Dress sportyFancyDress = new SportyDress(new FancyDress(new Basi
cDress()));
    sportyFancyDress.assemble();
    System.out.println();

    Dress casualFancyDress = new CasualDress(new FancyDress(new Basi
cDress()));
    casualFancyDress.assemble();

  }
}
```

Object Pool Design Pattern(Creational) :- https://www.javatpoint.com/object-pool-pattern

Chain of Responsibility Pattern (Behavioural) :-

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the**

request". For example, an ATM uses the Chain of Responsibility design pattern in money giving process.

In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

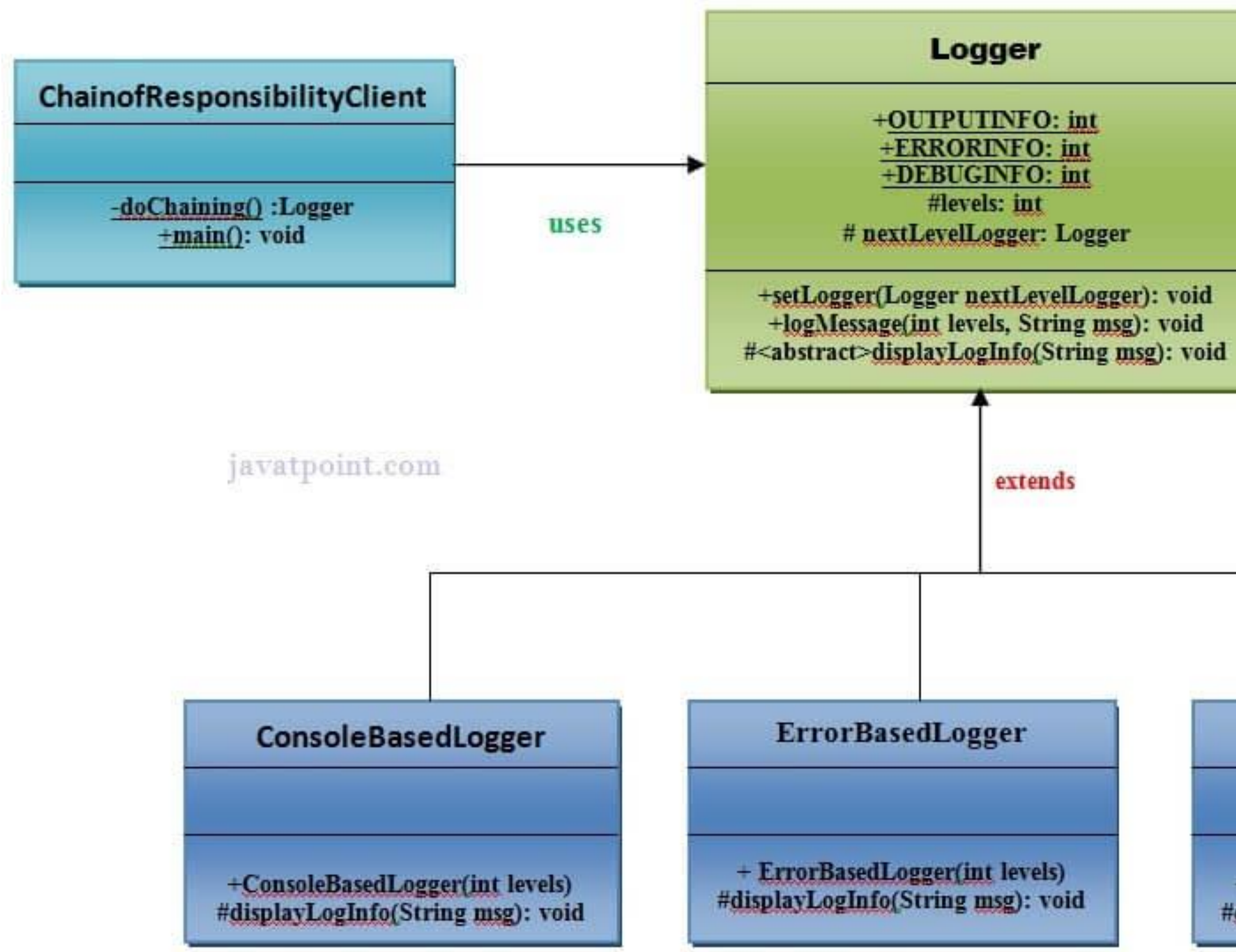
Advantage of Chain of Responsibility Pattern

- It reduces the coupling.
- It adds flexibility while assigning the responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

Usage of Chain of Responsibility Pattern:

It is used:

- When more than one object can handle a request and the handler is unknown.
- When the group of objects that can handle the request must be specified in dynamic way.


ChainofResponsibilityClient
-doChaining() :Logger
+main(): void
uses
Logger
+OUTPUTINFO: int
+ERRORINFO: int
+DEBUGINFO: int
#levels: int
# nextLevelLogger: Logger
+setLogger(Logger nextLevelLogger): void
+logMessage(int levels, String msg): void
#<abstract>displayLogInfo(String msg): void
javatpoint.com
extends
ConsoleBasedLogger
+ConsoleBasedLogger(int levels)
#displayLogInfo(String msg): void
ErrorBasedLogger
+ ErrorBasedLogger(int levels)
#displayLogInfo(String msg): void

Implementation of above UML:

Step 1

Create a **Logger** abstract class.

```
public abstract class Logger {
    public static int OUTPUTINFO=1;
    public static int ERRORINFO=2;
    public static int DEBUGINFO=3;
    protected int levels;
    protected Logger nextLevelLogger;
    public void setNextLevelLogger(Logger nextLevelLogger) {
        this.nextLevelLogger = nextLevelLogger;
    }
        public void logMessage(int levels, String msg){
        if(this.levels<=levels){
            displayLogInfo(msg);
        }
        if (nextLevelLogger!=null) {
            nextLevelLogger.logMessage(levels, msg);
        }
    }
    protected abstract void displayLogInfo(String msg);
}
```

Step 2

Create a **ConsoleBasedLogger** class.

File: ConsoleBasedLogger.java

```
public class ConsoleBasedLogger extends Logger {
    public ConsoleBasedLogger(int levels) {
        this.levels=levels;
    }
    @Override
    protected void displayLogInfo(String msg) {
        System.out.println("CONSOLE LOGGER INFO: "+msg);
    }
}
```

Step 3

Create a **DebugBasedLogger** class.

File: DebugBasedLogger.java

```
public class DebugBasedLogger extends Logger {
    public DebugBasedLogger(int levels) {
        this.levels=levels;
    }
    @Override
    protected void displayLogInfo(String msg) {
        System.out.println("DEBUG LOGGER INFO: "+msg);
    }
}// End of the DebugBasedLogger class.
```

Step 4

Create a **ErrorBasedLogger** class.

File: ErrorBasedLogger.java

```
public class ErrorBasedLogger extends Logger {
    public ErrorBasedLogger(int levels) {
        this.levels=levels;
    }
    @Override
    protected void displayLogInfo(String msg) {
        System.out.println("ERROR LOGGER INFO: "+msg);
    }
}// End of the ErrorBasedLogger class.
```

Step 5

Create a **ChainOfResponsibilityClient** class.

File: ChainofResponsibilityClient.java

```
public class ChainofResponsibilityClient {
    private static Logger doChaining(){
          Logger consoleLogger = new ConsoleBasedLogger(Logger.OUTPUTINFO);

          Logger errorLogger = new ErrorBasedLogger(Logger.ERRORINFO);
          consoleLogger.setNextLevelLogger(errorLogger);

          Logger debugLogger = new DebugBasedLogger(Logger.DEBUGINFO);
          errorLogger.setNextLevelLogger(debugLogger);

          return consoleLogger;
          }
          public static void main(String args[]){
          Logger chainLogger= doChaining();

              chainLogger.logMessage(Logger.OUTPUTINFO, "Enter the sequence of values ");
              chainLogger.logMessage(Logger.ERRORINFO, "An error is occured now");
              chainLogger.logMessage(Logger.DEBUGINFO, "This was the error now debugging is compeled");
```

```
        }
}
```

Template Pattern

A Template Pattern says that "just define the skeleton of a function in an operation, deferring some steps to its subclasses".

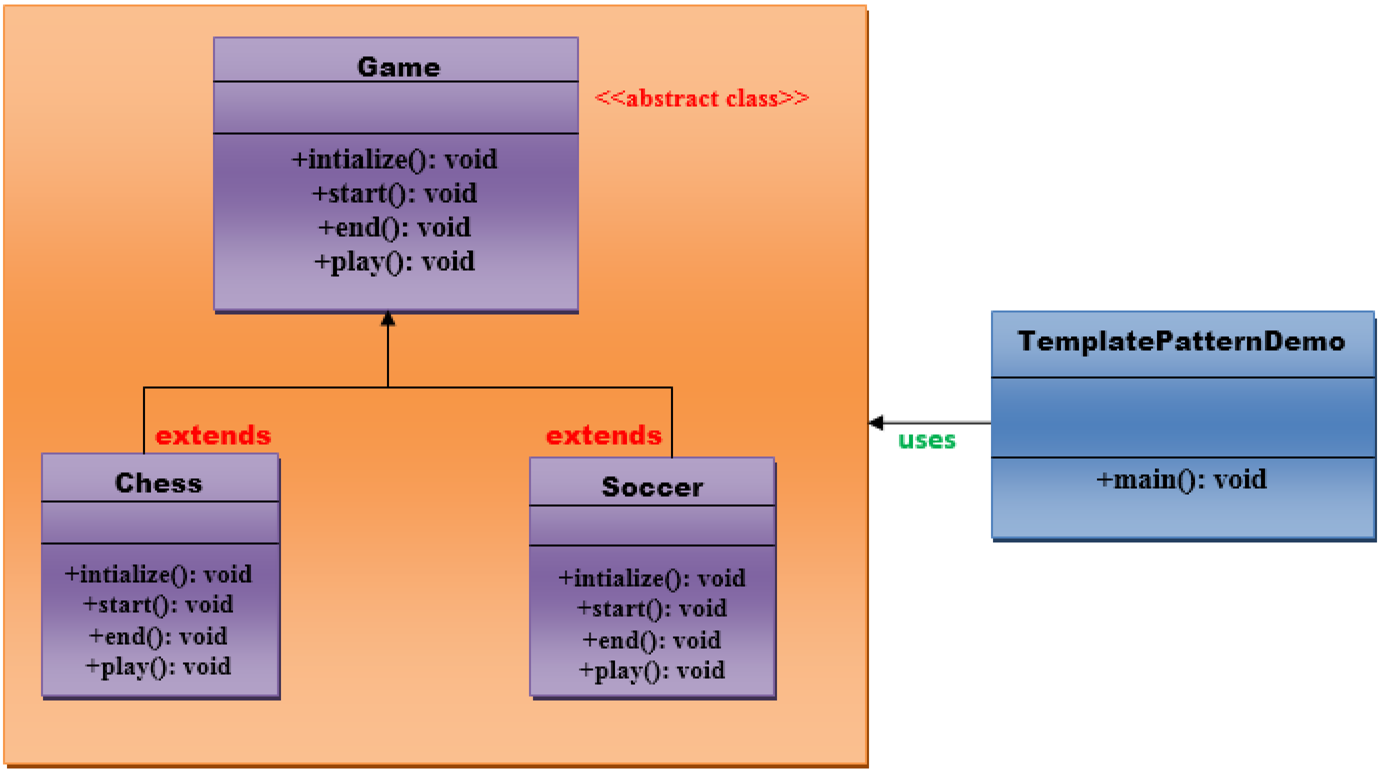
Benefits:

- It is very common technique for reusing the code.This is only the main benefit of it.

Usage:

- It is used when the common behavior among sub-classes should be moved to a single common class by avoiding the duplication.

UML for Template Pattern:



Implementation of Template Pattern:

Step 1:

Create a Game *abstract* class.

```
//This is an abstract class.
public abstract class Game {

       abstract void initialize();
        abstract void start();
       abstract void end();

       public final void play(){

          //initialize the game
           initialize();

           //start game
          start();

          //end game
           end();
       }
}// End of the Game abstract class.
```

Step 2:

Create a *Chess* class that will extend Game abstract class for giving the definition to its method.

```
//This is a class.

public class Chess extends Game {
     @Override
        void initialize() {
          System.out.println("Chess Game Initialized! Start playing.");
        }
     @Override
        void start() {
          System.out.println("Game Started. Welcome to in the chess game!");
```

```
    }
  @Override
    void end() {
      System.out.println("Game Finished!");
    }
}// End of the Chess class.
```

Step 3:

Create a *Soccer* class that will extend Game abstract class for giving the definition to its method.

```
//This is a class.


public class Soccer extends Game {

  @Override
    void initialize() {
      System.out.println("Soccer Game Initialized! Start playing.");
    }

  @Override
    void start() {
      System.out.println("Game Started. Welcome to in the Soccer game!");
    }

  @Override
    void end() {
      System.out.println("Game Finished!");
    }
}// End of the Soccer class.
```

Step 4:

Create a *TemplatePatternDemo* class.

```
//This is a class.
public class TemplatePatternDemo {
```

```

public static void main(String[] args) throws InstantiationException, IllegalAccessException, ClassNotFoundException {

        Class c=Class.forName(args[0]);
         Game game=(Game) c.newInstance();
        game.play();
      }
}// End of the Soccer class.
```

Command Pattern

A Command Pattern says that "*encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command*".

It is also known as **Action or Transaction.**

Advantage of command pattern

- It separates the object that invokes the operation from the object that actually performs the operation.
- It makes easy to add new commands, because existing classes remain unchanged.

Usage of command pattern:

It is used:

- When you need parameterize objects according to an action perform.
- When you need to create and execute requests at different times.

- When you need to support rollback, logging or transaction functionality.

Example of command pattern

Let's understand the example of adapter design pattern by the above UML diagram.

UML for command pattern:

These are the following participants of the Command Design pattern:

- **Command** This is an interface for executing an operation.
- **ConcreteCommand** This class extends the Command interface and implements the execute method. This class creates a binding between the action and the receiver.
- **Client** This class creates the ConcreteCommand class and associates it with the receiver.
- **Invoker** This class asks the command to carry out the request.
- **Receiver** This class knows to perform the operation.

```


}
```

Implementation of above UML:

Step 1

Create a **ActionListernerCommand** interface that will act as a Command.

```
public interface ActionListenerCommand {
    public void execute();
}
```

Step 2

Create a **Document** class that will act as a Receiver.

File: Document.java

```
public class Document {
          public void open(){
           System.out.println("Document Opened");
       }
        public void save(){
           System.out.println("Document Saved");
        }
}
```

Step 3

Create a **ActionOpen** class that will act as an ConcreteCommand.

File: ActionOpen.java

```
public class ActionOpen implements ActionListenerCommand{
    private Document doc;
    public ActionOpen(Document doc) {
        this.doc = doc;
    }
    @Override
    public void execute() {
        doc.open();
    }
}
```

Step 4

Create a **ActionSave** class that will act as an ConcreteCommand.

File: AdapterPatternDemo.java

```
public class ActionSave implements ActionListenerCommand{
   private Document doc;
   public ActionSave(Document doc) {
        this.doc = doc;
```

```
    }
    @Override
    public void execute() {
        doc.save();
    }
}
```

Step 5

Create a **MenuOptions** class that will act as an Invoker.

File: ActionSave.java

```
public class ActionSave implements ActionListenerCommand{
   private Document doc;
    public ActionSave(Document doc) {
        this.doc = doc;
    }
    @Override
    public void execute() {
        doc.save();
    }
}
```

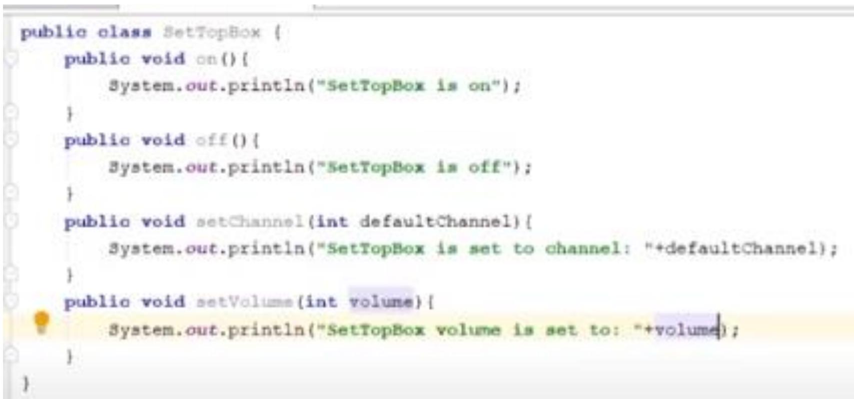
Step 6

Create a **CommanPatternClient** class that will act as a Client.

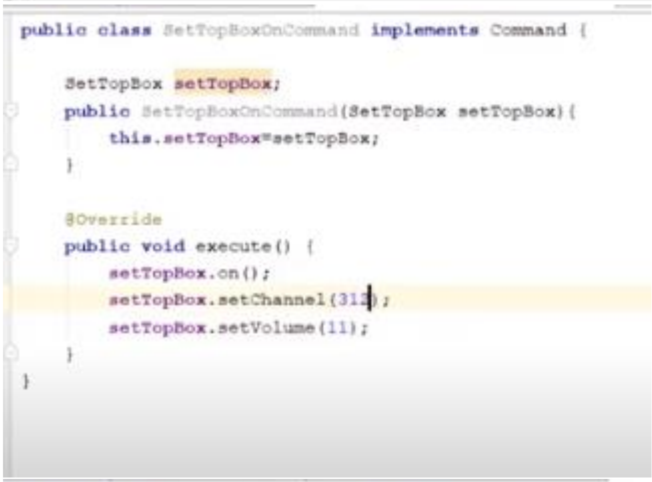
File: AdapterPatternDemo.java

```
public class CommandPatternClient {
    public static void main(String[] args) {
        Document doc = new Document();

        ActionListenerCommand clickOpen = new ActionOpen(doc);
        ActionListenerCommand clickSave = new ActionSave(doc);

        MenuOptions menu = new MenuOptions(clickOpen, clickSave);

        menu.clickOpen();
        menu.clickSave();
    }
}
```

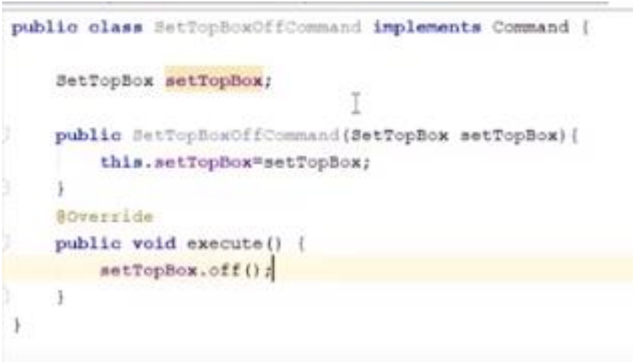
Command Pattern Example 2 :-



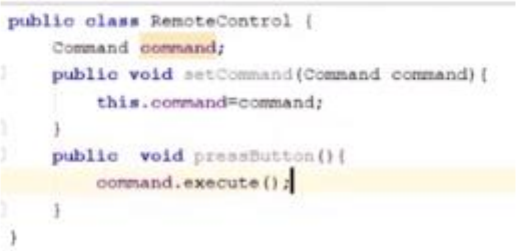
```
public class SetTopBox {
    public void on(){
        System.out.println("SetTopBox is on");
    }
    public void off(){
        System.out.println("SetTopBox is off");
    }
    public void setChannel(int defaultChannel){
        System.out.println("SetTopBox is set to channel: "+defaultChannel);
    }
    public void setVolume(int volume){
        System.out.println("SetTopBox volume is set to: "+volume);
    }
}
```



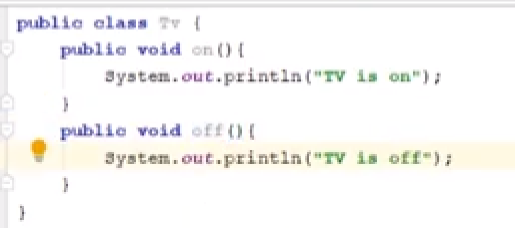
```
public class SetTopBoxOnCommand implements Command {

    SetTopBox setTopBox;
    public SetTopBoxOnCommand(SetTopBox setTopBox){
        this.setTopBox=setTopBox;
    }

    @Override
    public void execute() {
        setTopBox.on();
        setTopBox.setChannel(312);
        setTopBox.setVolume(11);
    }
}
```



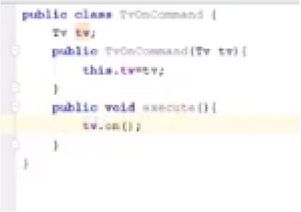
```
public class SetTopBoxOffCommand implements Command {

    SetTopBox setTopBox;

    public SetTopBoxOffCommand(SetTopBox setTopBox){
        this.setTopBox=setTopBox;
    }
    @Override
    public void execute() {
        setTopBox.off();
    }
}
```



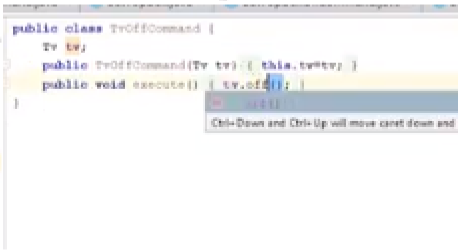
```
public class RemoteControl {
    Command command;
    public void setCommand(Command command){
        this.command=command;
    }
    public  void pressButton(){
        command.execute();
    }
}
```



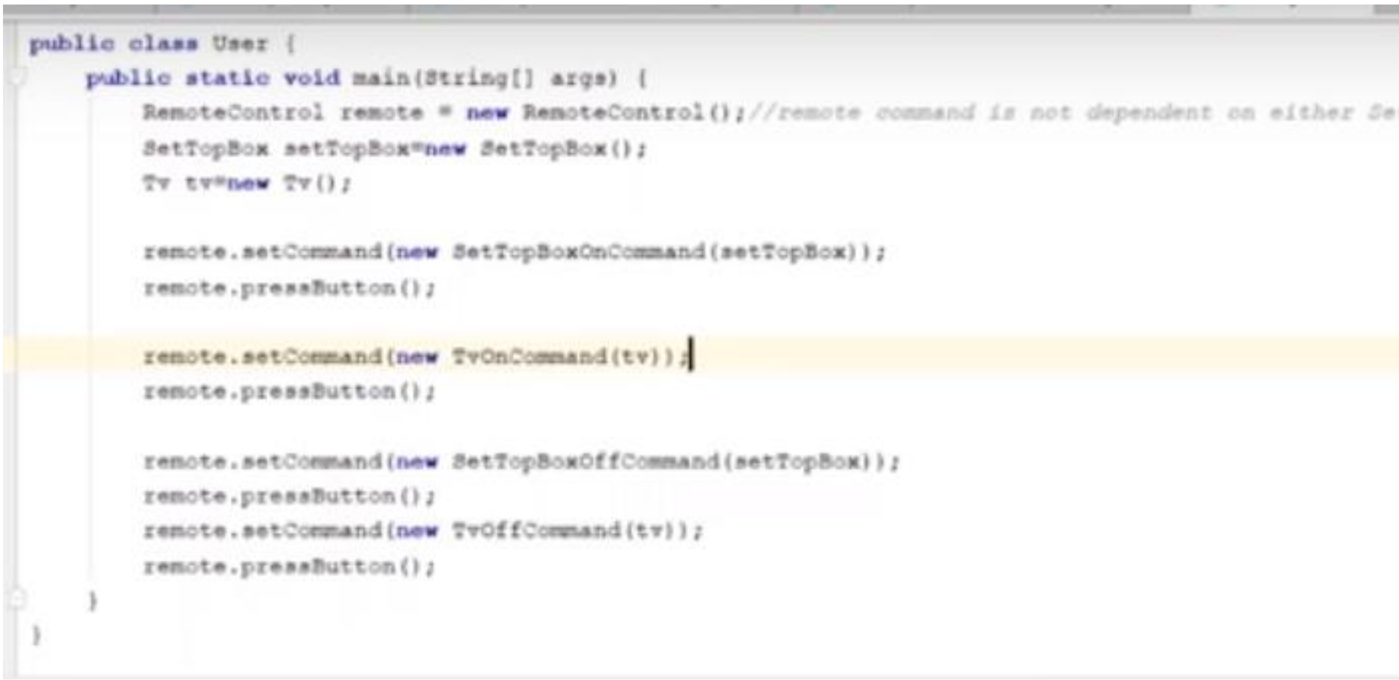
```
public class Tv {
    public void on(){
        System.out.println("TV is on");
    }
    public void off(){
        System.out.println("TV is off");
    }
}
```



```
public class TvOnCommand {
    Tv tv;
    public TvOnCommand(Tv tv){
        this.tv=tv;
    }
    public void execute(){
        tv.on();
    }
}
```



```
public class TvOffCommand {
    Tv tv;
    public TvOffCommand(Tv tv) { this.tv=tv; }
    public void execute() { tv.off(); }
}
```



```
public class User {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();//remote command is not dependent on either Se
        SetTopBox setTopBox=new SetTopBox();
        Tv tv=new Tv();

        remote.setCommand(new SetTopBoxOnCommand(setTopBox));
        remote.pressButton();

        remote.setCommand(new TvOnCommand(tv));
        remote.pressButton();

        remote.setCommand(new SetTopBoxOffCommand(setTopBox));
        remote.pressButton();
        remote.setCommand(new TvOffCommand(tv));
        remote.pressButton();
    }
}
```

Strategy Pattern

Strategy Pattern

A Strategy Pattern says that "defines a family of functionality, encapsulate each one, and make them interchangeable".

The Strategy Pattern is also known as Policy.

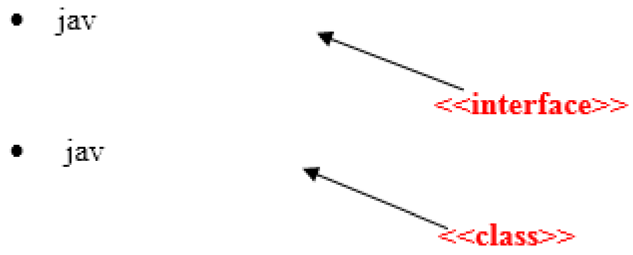
Benefits:

- It provides a substitute to subclassing.
- It defines each behavior within its own class, eliminating the need for conditional statements.
- It makes it easier to extend and incorporate new behavior without changing the application.

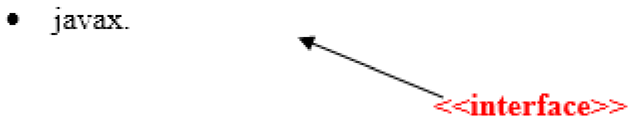
Usage:

- When the multiple classes differ only in their behaviors.e.g. Servlet API.
- It is used when you need different variations of an algorithm.

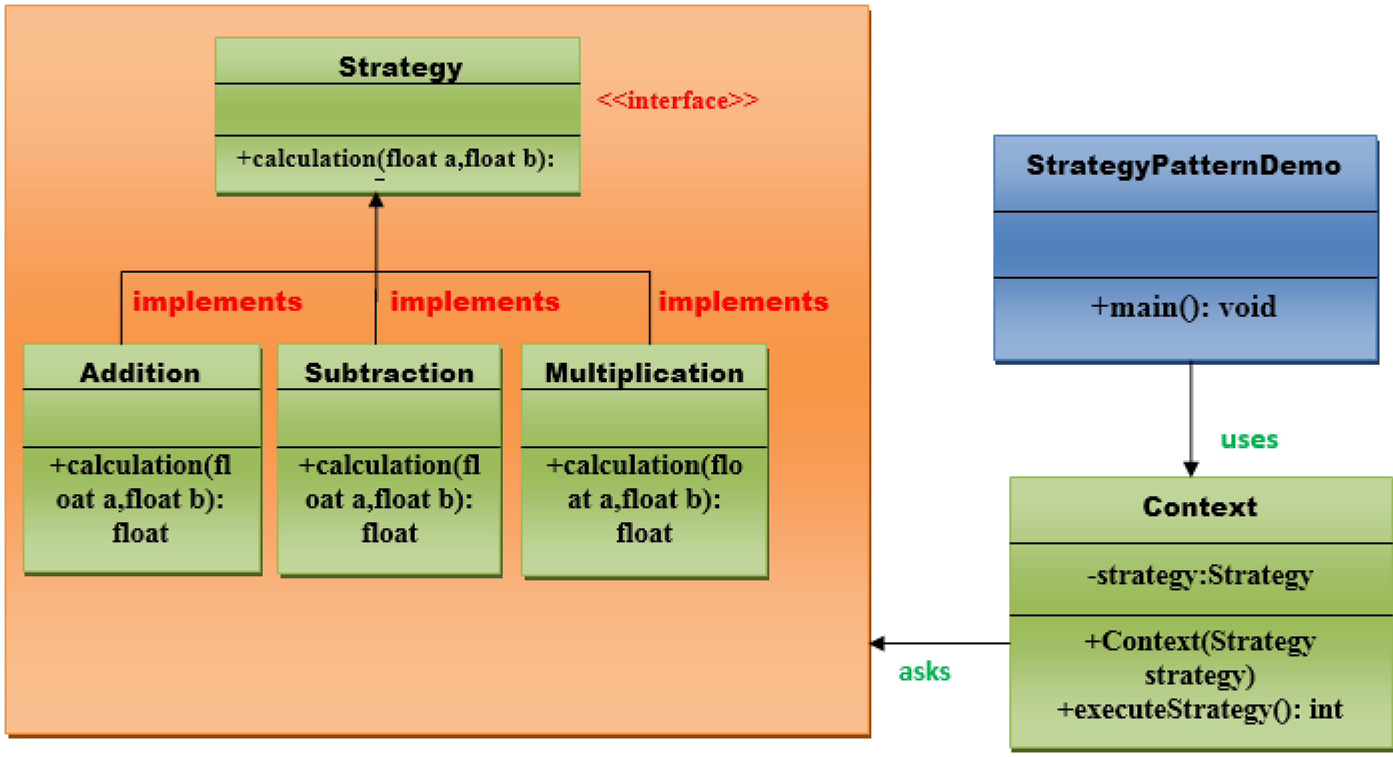
Strategy Pattern in (Core Java API's) or JSE 7 API's:



Strategy Pattern in (Advance Java API's) or JEE 7 API's:



UML for Strategy Pattern:



Implementation of Strategy Pattern:

Step 1:

Create a *Strategy* interface.

```
//This is an interface.

public interface Strategy {

    public float calculation(float a, float b);

}// End of the Strategy interface.
```

Step 2:

Create a *Addition* class that will implement Startegy interface.

```
//This is a class.
public class Addition implements Strategy{

    @Override
    public float calculation(float a, float b) {
       return a+b;
    }

}// End of the Addition class.
```

Step 3:

Create a *Subtraction* class that will implement Startegy interface.

```
//This is a class.
public class Subtraction  implements Strategy{

    @Override
    public float calculation(float a, float b) {
       return a-b;
    }

}// End of the Subtraction class.
```

Step 4:

Create a Multiplication class that will implement Startegy interface.

```
//This is a class.

public class Multiplication implements Strategy{

    @Override
    public float calculation(float a, float b){
        return a*b;
    }
}// End of the Multiplication class.
```

Step 5:

Create a *Context* class that will ask from Startegy interface to execute the type of strategy.

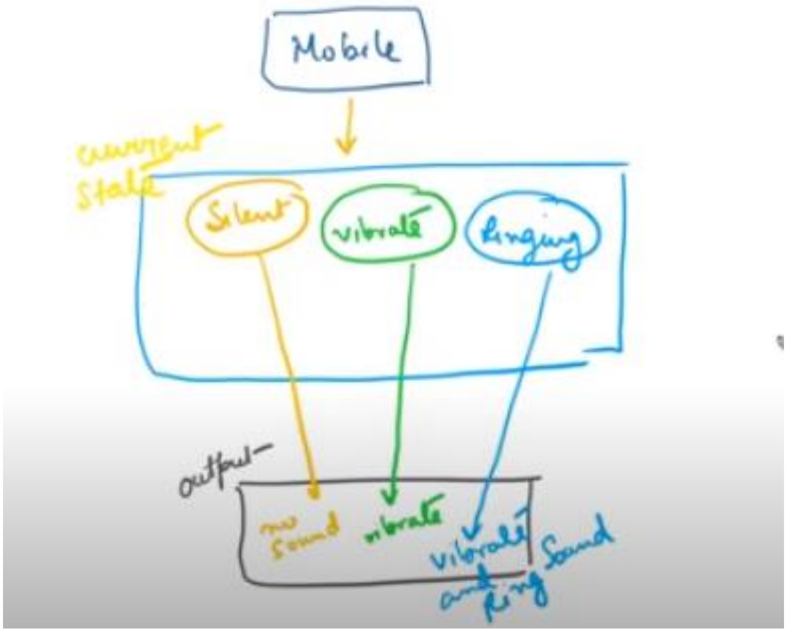
```java
//This is a class.


public class Context {

    private Strategy strategy;

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

    public float executeStrategy(float num1, float num2){
        return strategy.calculation(num1, num2);
    }
}// End of the Context class.
```

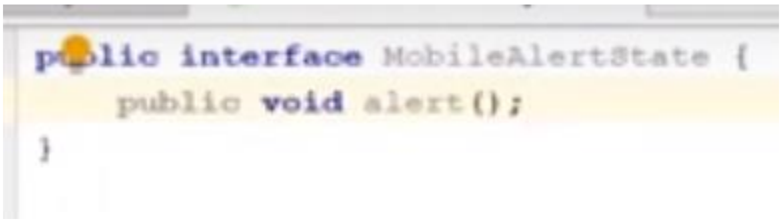
Step 6:

Create a *StartegyPatternDemo* class.

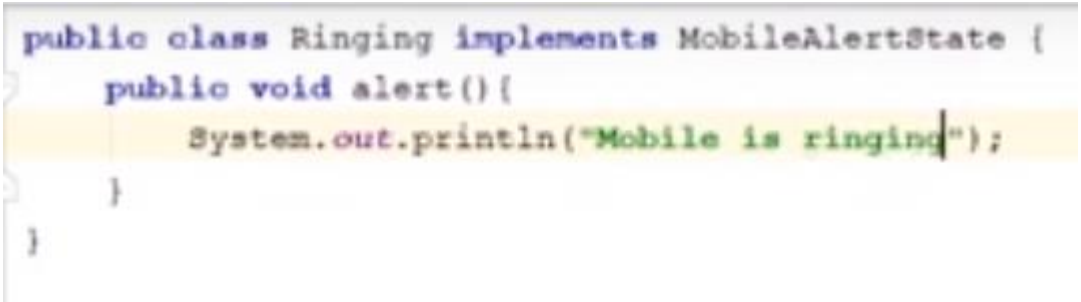
```java
//This is a class.
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class StrategyPatternDemo {

   public static void main(String[] args) throws NumberFormatException, IOException {

        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
         System.out.print("Enter the first value: ");
        float value1=Float.parseFloat(br.readLine());
         System.out.print("Enter the second value: ");
        float value2=Float.parseFloat(br.readLine());
         Context context = new Context(new Addition());
        System.out.println("Addition = " + context.executeStrategy(value1, value2));

        context = new Context(new Subtraction());
        System.out.println("Subtraction = " + context.executeStrategy(value1, value2));

        context = new Context(new Multiplication());
        System.out.println("Multiplication = " + context.executeStrategy(value1, value2));
    }

}// End of the StrategyPatternDemo class.
```

State Design Pattern

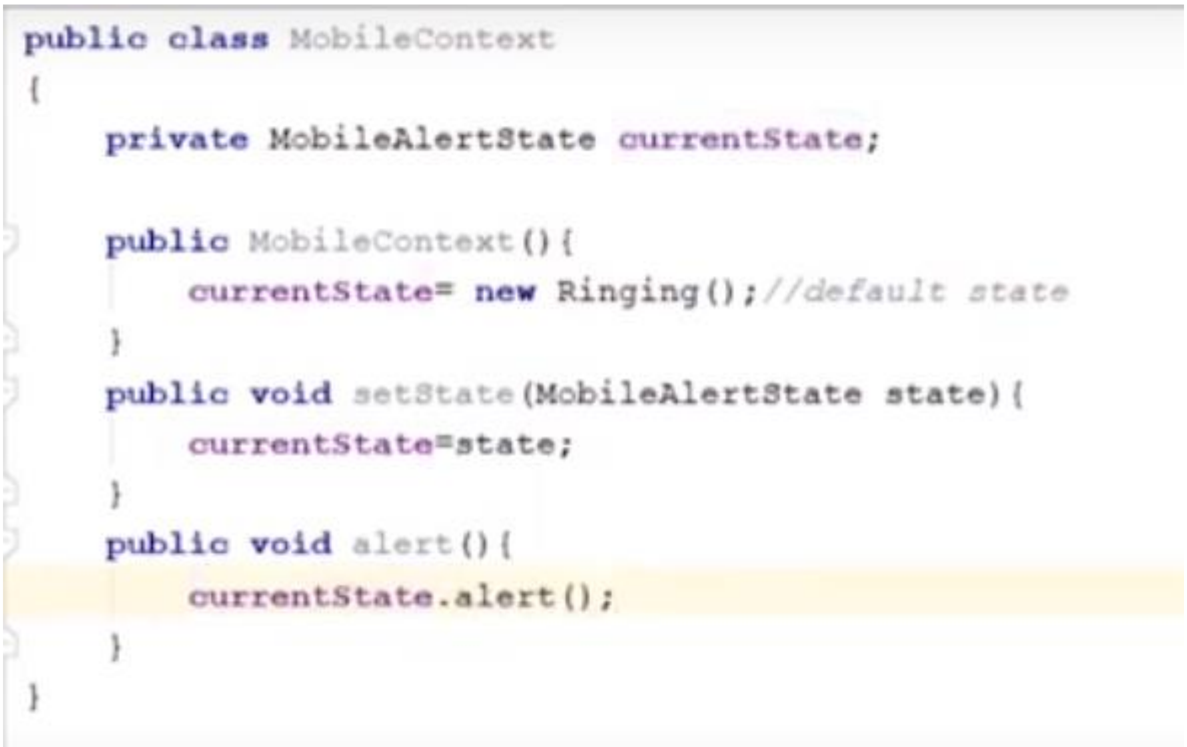




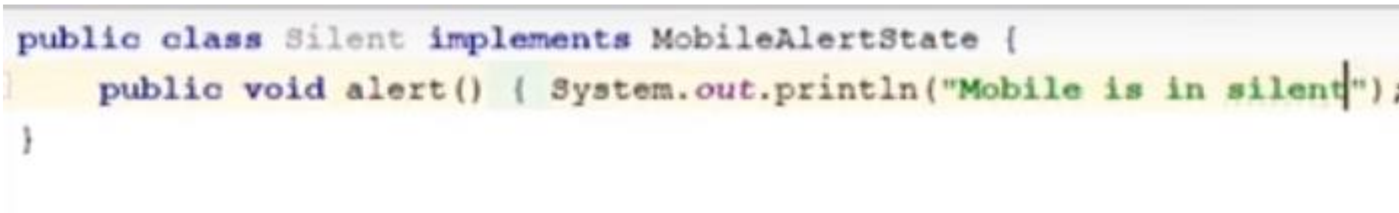
```
public interface MobileAlertState {
    public void alert();
}
```



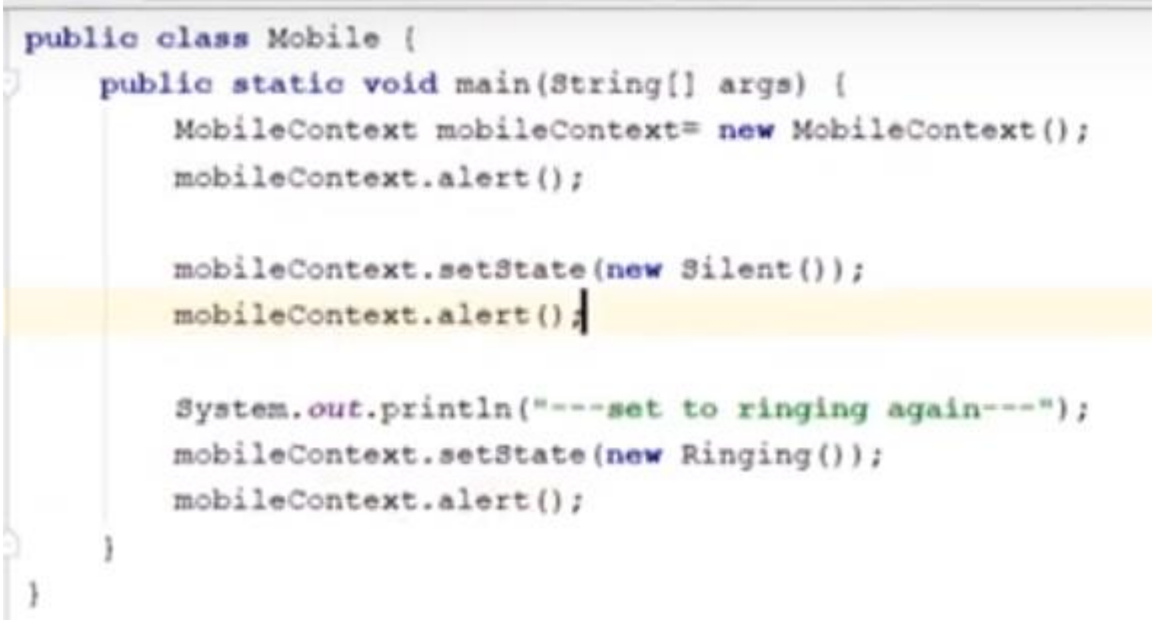
```
public class Ringing implements MobileAlertState {
    public void alert(){
        System.out.println("Mobile is ringing");
    }
}
```



```
public class MobileContext
{
    private MobileAlertState currentState;

    public MobileContext(){
        currentState= new Ringing();//default state
    }
    public void setState(MobileAlertState state){
        currentState=state;
    }
    public void alert(){
        currentState.alert();
    }
}
```



```
public class Silent implements MobileAlertState {
    public void alert() { System.out.println("Mobile is in silent");
}
```



```
public class Mobile {
    public static void main(String[] args) {
        MobileContext mobileContext= new MobileContext();
        mobileContext.alert();

        mobileContext.setState(new Silent());
        mobileContext.alert();

        System.out.println("---set to ringing again---");
        mobileContext.setState(new Ringing());
        mobileContext.alert();
    }
}
```

Example 2:-

https://www.javatpoint.com/state-pattern