

Basically, Solid principle are the design principle that enables us to manage most of the software design problems. the term solid is an acronym of 5 design principle intend to make software design more understandable, flexible and maintainable.

In our Solid Design Principle –

**S** stands for - **Single Responsibility Principle**

**O** stands for - **Open & Close Principle**

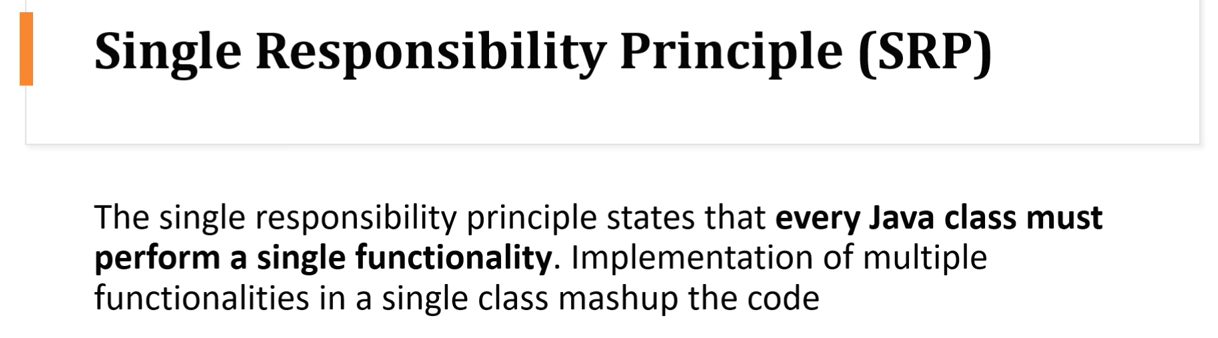
**L** stands for - **Leskov’s Substitution Principle**

**I** stands for - **Interface Segregation Principle**

**D** stands for – **Dependency Inversion Principle**

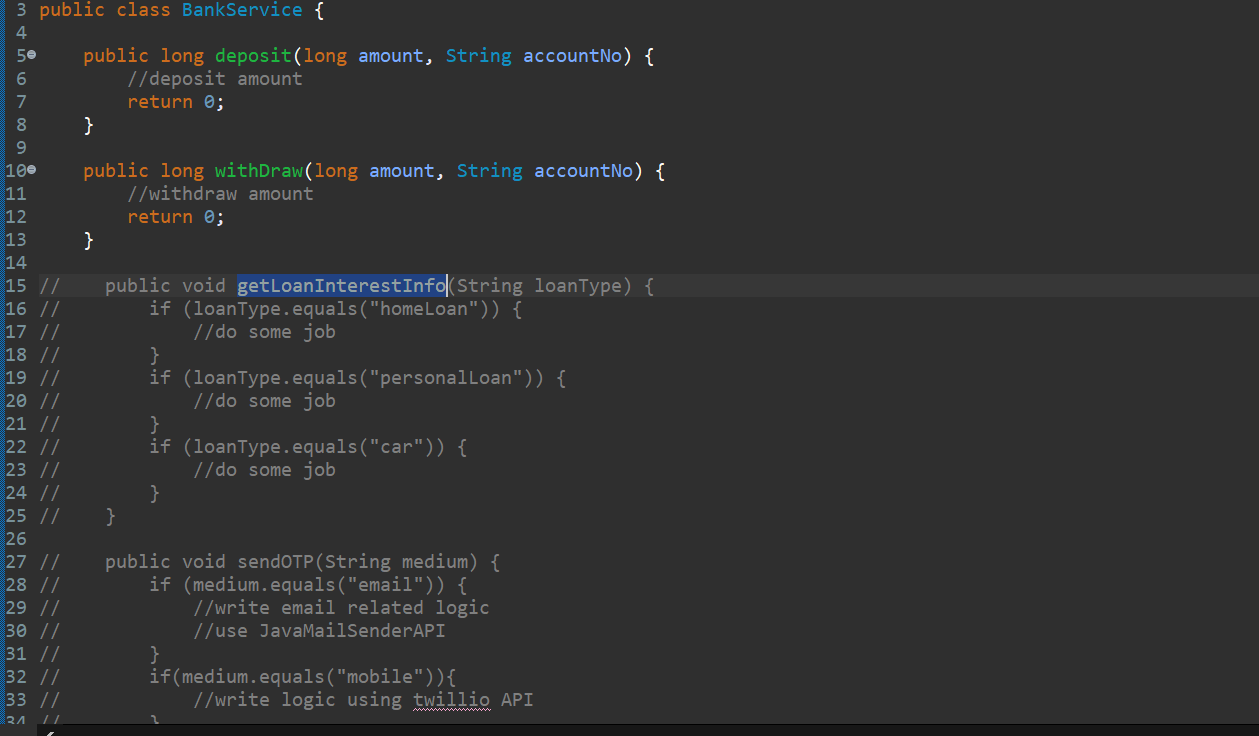
These all 5 principles change the world of Object-oriented programming and also changed the way of writing software.

**1. Single Responsibility Principle**



Means A class should have only one reason to change. That means Every Class should have a Single Responsibility, Single Job or a Single Purpose to Perform.

For an example lets take a Bank Service…So now Bank Service is having multiple methods…like deposit(), withdraw(), getLoanInterestInfo () based on home loan/car loan/personal loan…, sendOTP () through medium mobile/Email…



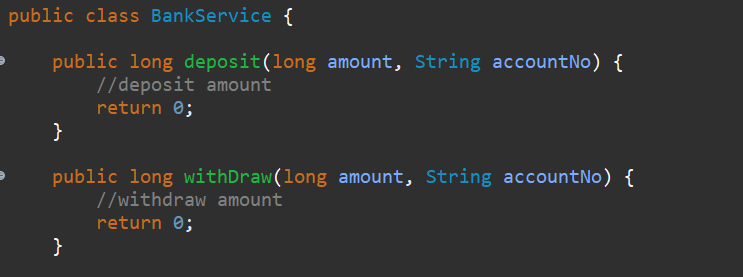
Imagine a scenario If a class having multiple reasons to change. Take example bank is supporting for Loan Info like home loan , personal loan and car loan….in future if bank want to support for Gold loan and study loan…in that case u need to change the code….

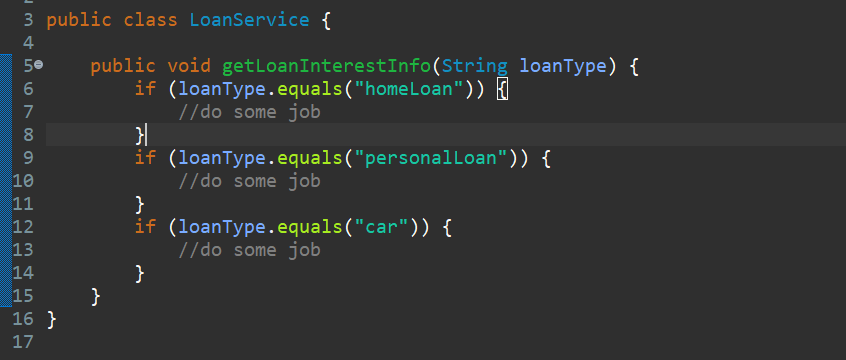
Similarly this Bank Service supports send OTP() medium as an Email but in future they may want to intrude it as a phone or WhatsApp…then again it needs to change its implementation…

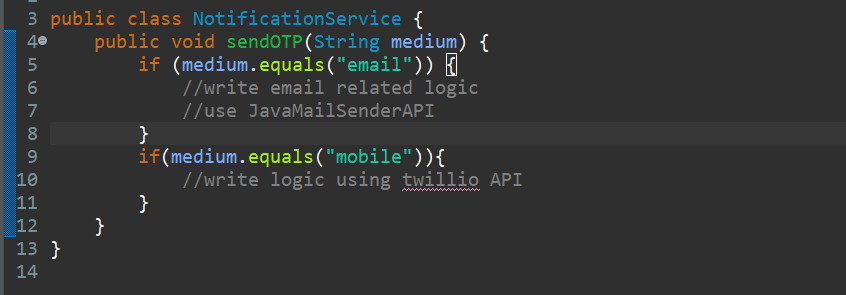
It doesn’t follow Single Responsibility Principle….because this class is having too many task or responsible to perform.

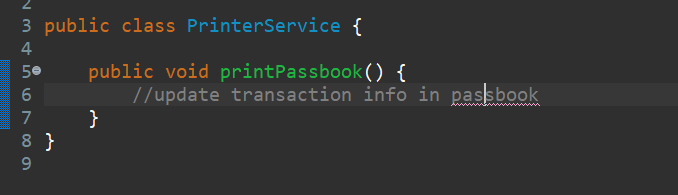
To achieve the goal of a Single Responsibility Principle we should implement a Separate class that perform a Single Functionality only…for example printPassbook() logic we can move to a PassBookService….Similarly these loan related stuff we can move to a LoanService…and this OTP related stuff we can move to the OTP service…

So that Each Service will contain a single Responsibility to Perform.

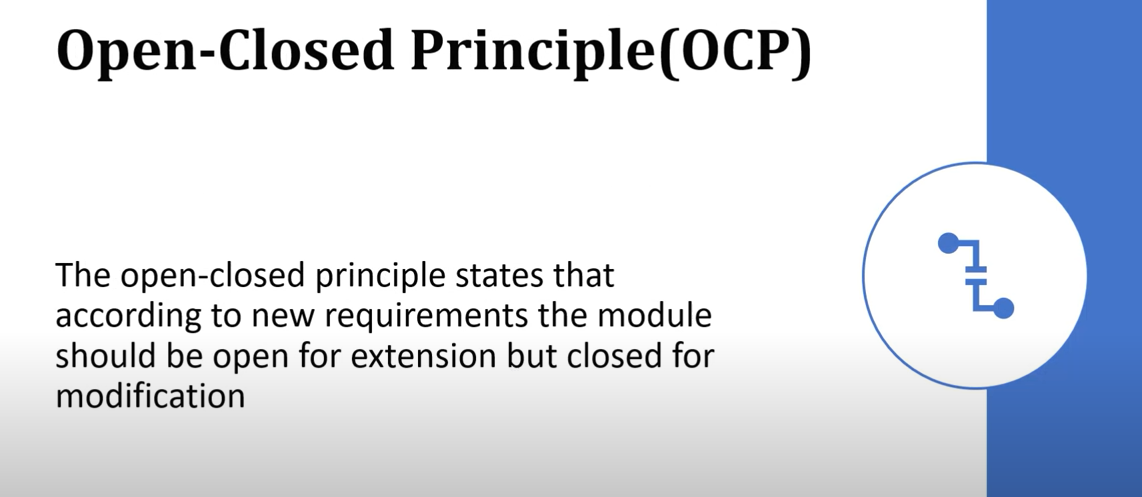








**2. Open & Closed Principle**



**This principle states that software entities like classes, modules and functions should be open for extension but closed for modification. Which means you should be able to extends a class behavior without modifying it.**

**Let’s take an Example like Notification Service which we just created now…**

**We have only 2 features send OTP through Email and send OTP through mobile…lets In Future this Notification Service may implement sent OTP through WhatsApp….and that feature will be not supported by all the implementation. So, in that case we should not modify the WhatsApp related code inside this Notification Service….**

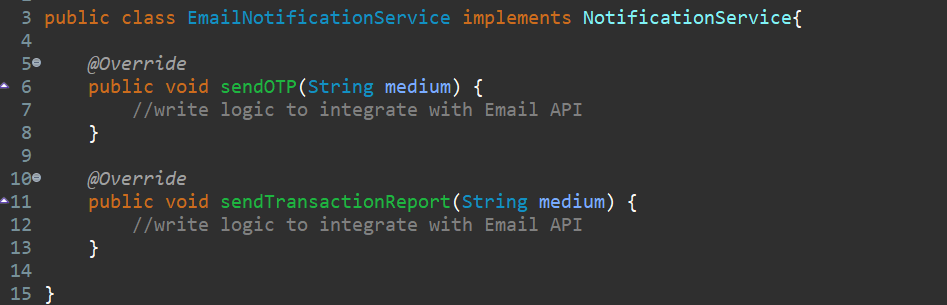
**Because this OC Principle says the class is always open for Extension but Closed for Modification. Hence, it’s not recommended to change the Notification Service for each OTP features…It will violate this OCP principle.**

**So, if u want to implement the WhatsApp related notification service…then just extends from the Notification Service and just do whatever u want to do. But don’t modify existing Notification Service that why this is OC…. bcz we are allowing for extension not for modification.**

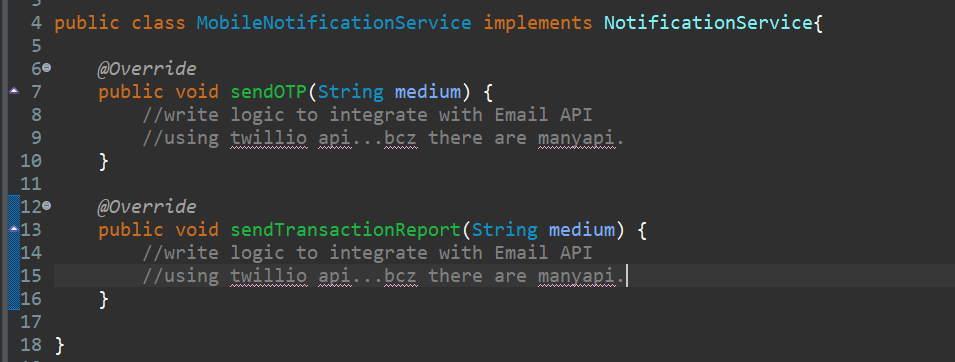
**So, if anyone want to add any additional feature so let’s extends our existing feature and do changes on top of our code. It’s like abstraction.**

**So, u can design something like this…SO let us create an Interface we can also create an abstract class but let’s not do that.**

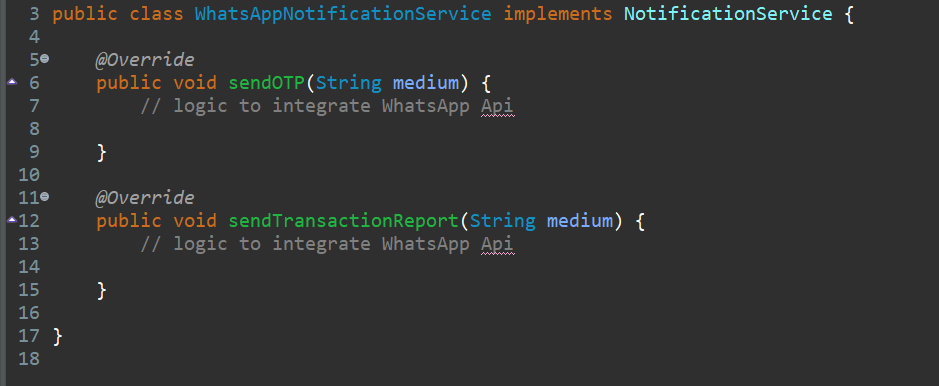
**SO I will create a each separate implementation class for each OTP functionality….like for email, mobile, whatsapp…**



**So, this particular class will deal with the Email Notification Service…..**

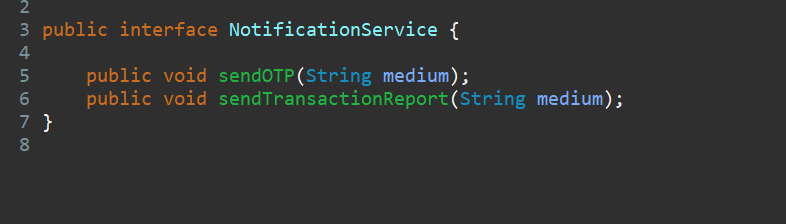


**Again, we need to add a new implementation for WhatsApp Notification Service….**



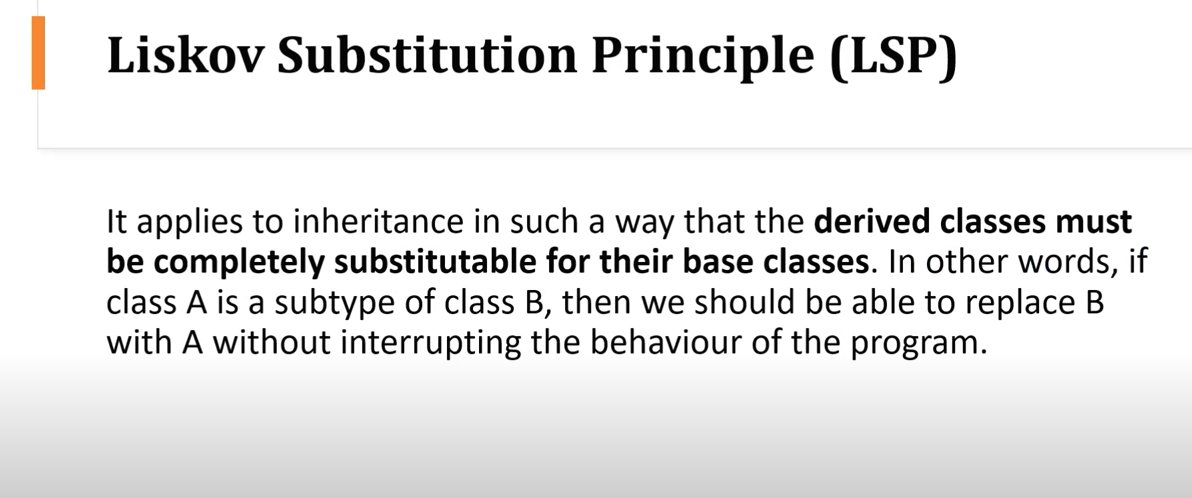
**So now let us say user are expecting different medium of Notification Service something like Facebook, Messenger Notification Service….**

**So In that case we will not allow User to modify this Notification Service class…Rather than modifying it we will allow him to extend it from Notification Service and then provide ur own logic for Facebook, Messenger Notification Service stuff…..**



**So, Instead of changing the Notification service actual code we will ask user to extend the Actual Notification Service and add his own implementation for it….**

**3. Liskov Substitution Principle**



**This Principle states that derived or child classes must be substitutable for their base or parent classes.**

In other word If class A is a Sub Type of Class B then we should be able to replace B with A without interrupting the behavior of a program.

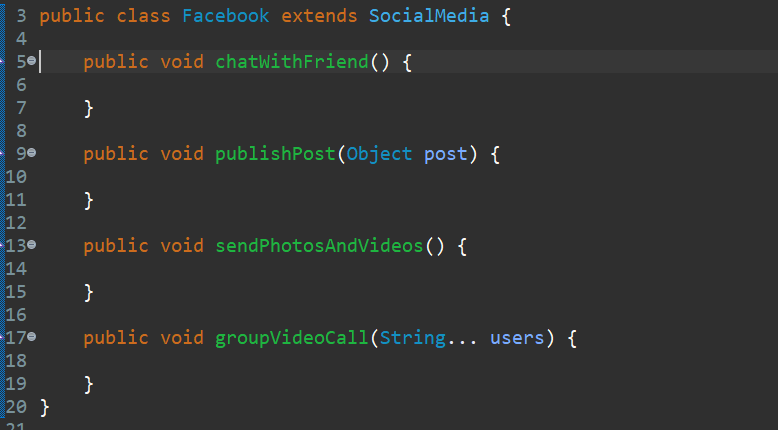
Let’s understand with an Example: -

Let’s say we have a class called Social Media



We have a Custom class called Social Media. And It support all social media activity for user to entertain them…So If u observe User can chat with a friend, user can do the Posts, he can send photos and videos and he can do the video call. So, this all features supported by multiple clients…

For example, there can be a multiple client for this social media like Facebook, Instagram and whatsapp… Now Consider this Facebook.

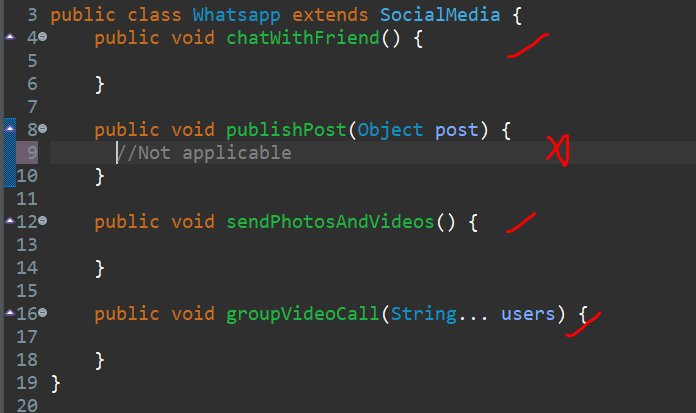


Let me create a class called Facebook then I will simply extend from abstract class called Social media. Since it is an Abstract class so we must need to override all abstract method.

So, if u observed if those who all are using Facebook, they might be aware about these features already available in Facebook. So now we can say this Facebook is substitute of Social Media. Because parent and child both are substitute to each other.

Now let’s see the implementation for WhatsApp….

So I will also extends it from Social Media. And provide all the implementation for feature method. So In what’s app this publish Post feature is not supported….



So, if u see publishPost () feature is not acceptable or its not the supported feature for what’s app…. It Means What’s App can’t be substitute of this Social Media…. bcz both are not in a sink.

So, bcz of this publishPost () not support(un-supported) Feature in What’s So we can say It doesn’t support Liskov Substitution principle.

Similarly let’s talk about the Instagram this is also extends from Social Media…



Instagram support chatWithFriend (), publishPost(), sendPhotosandVideos() but doesn’t support groupVideoCall ()….so bcz of this we can say Instagram Child is not substitute of the Parent Social Media.

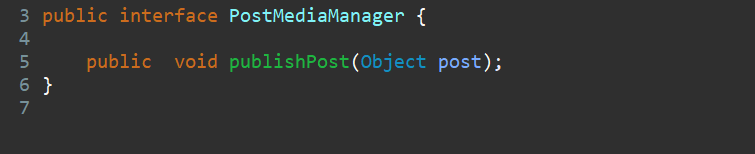
This also doesn’t follow LSP.

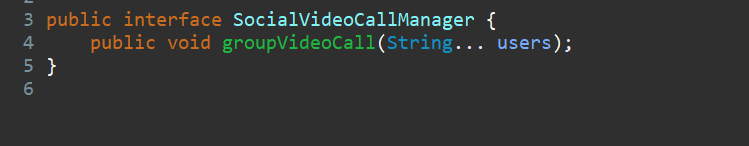
Now we need to Identify the Issue or how we can design so that our application follows LSP. So, to do that I will create an Interface. So, lets create a package called solution, inside solution lets create an Interface called Social Media, so let’s add couple of method here.



Because these 2 features supported by all the Implemented class like Facebook, whatsapp, Instagram.

So, to deals with another features like posts or group Videocall lets create another Interface.



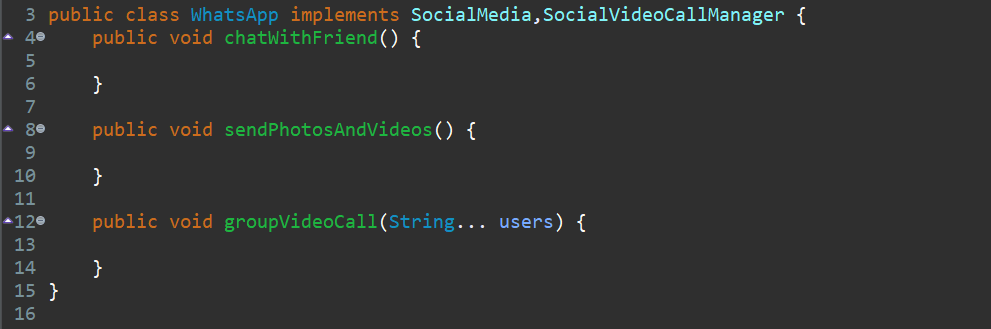


Now If you Observed the Segregate specific functionality to separate class to follow LSP. Now It’s up to implementation class decision to support feature. Based on their desired features they can use respective Interface, for Example Instagram doesn’t support this video call feature. So lets create an Implementation class for Instagram Features….

So simply it can implement Social Media bcz it supports both features chatWithFriend and sendPhotosandVideos as well as Instagram supports publishing Posts also. So we need to implement both of the interface. Since Instagram doesn’t supports the videocall features so we didn’t implement the SocialVideoCallManager Interface.



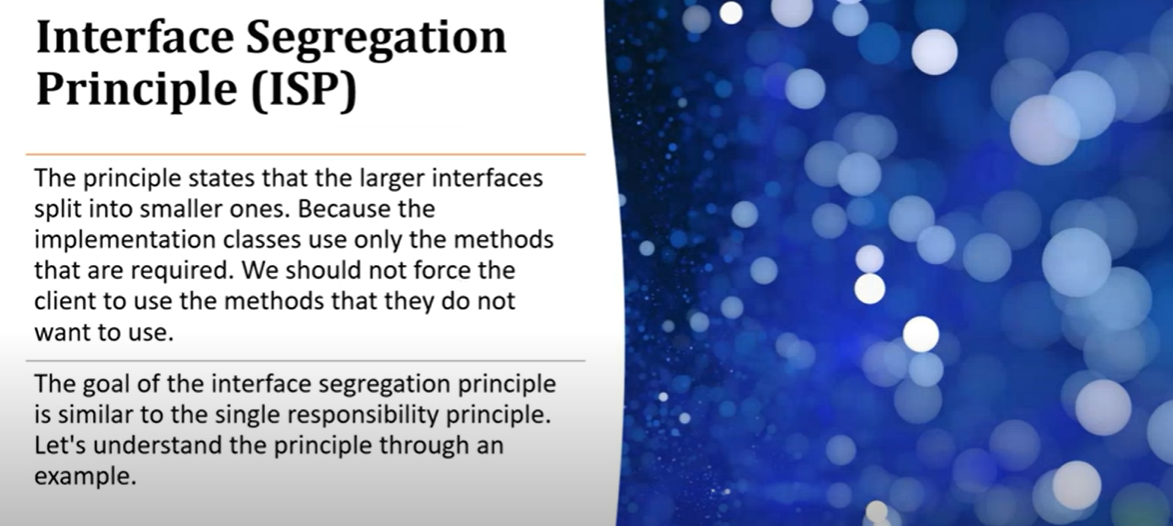
Now we consider Instagram is a Child of both SocialMedia and PostMediaManager and it substitutes of both the Parents. Similarly, we can create a anther class called WhatsApp…since what’s app supports only chatwithFriends. sendPhotosandVideos which comes under Social Media Features and groupVideoCall as well, so we are implementing both the interfaces SocialVideoCallManager and SocialMedia. It didn’t implement a feature from PostMediaManager bcz it doesn’t support.



So here WhatsApp is a Child of SocialMedia and SocialVideoCallManager and, it’s a substitute of both the SocialMedia and SocialVideoCallManager.

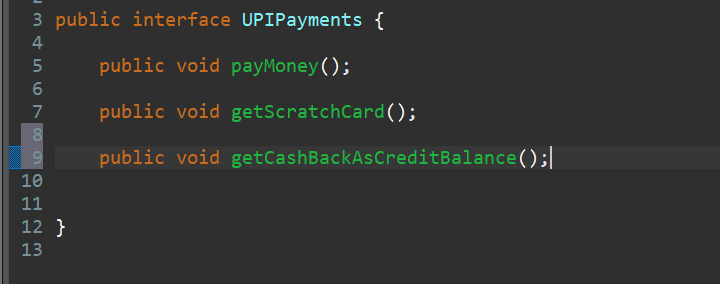
This is how you can design Liskov Substitution Principle in java following with the Interface Approach we can also go with the Abstract Class as well, but it’s always recommended to use the Interface.

**4. Interface Segregation Principle (ISP)**



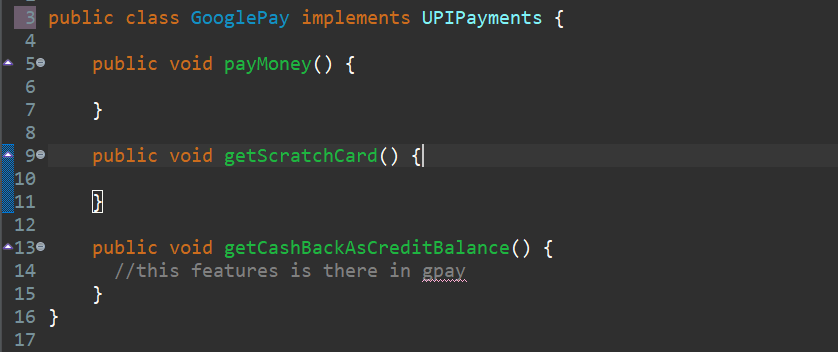
Let’s take an Example…

Let’s say we have an Interface called UPI Payment. so there is a options called **payMoney**(), **getScratchCard**(), **getCashBackAsCreditBalance**().



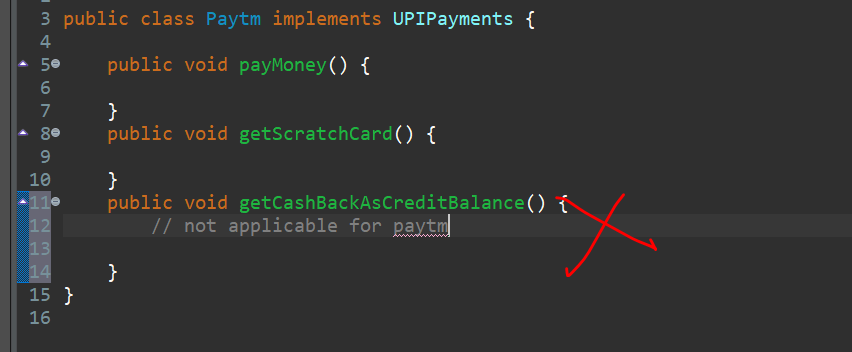
Now let me create an Implementation for an UPIPayments()… As you know there are multiple UPI Payment Implementation like Paytm, GooglePay, PhonePe something like that.

So, let me create a class called GooglePay. Which implements UPI Payments…. then just need to override all the methods…so if u observed GooglePay supports there many features….like **payMoney**(), **getScratchCard**(), **getCashBackAsCreditBalance**().



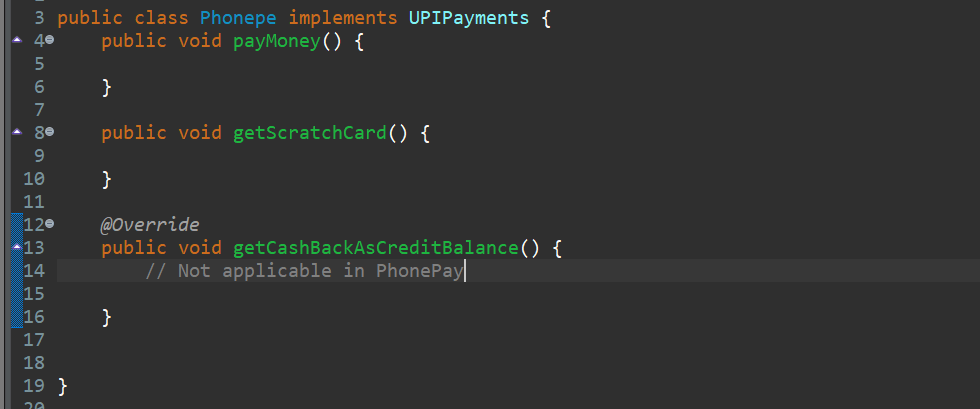
Now let me create a another implementation which is Paytm…so through paytm() we can send a money, it also gives a scratchCard() but this getCashBackAsCreditBalance() features is not presents in Paytm….so in this as a client My Paytm is forced to implement this Interface and overriding these methods…even though he is not using ..

So here we should not force our client paytm to override this method while implementing the UPI Payments….It Violates the contract of Interface Segregation Principle….



Similarly if I create a class called PhonePay…so I should not enforce PhonePay Client to implements the feature getCashBackAsCreditBalance() if its not supporting this feature…

So this is not a good way to design the code…so what we can do if u remember single responsibility principle also applicable in this interface Segregation Principle.

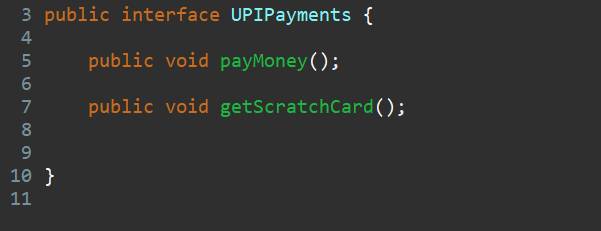


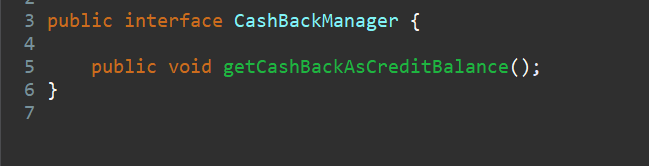
So all the principle looks same but functionality wise purpose wise its different.

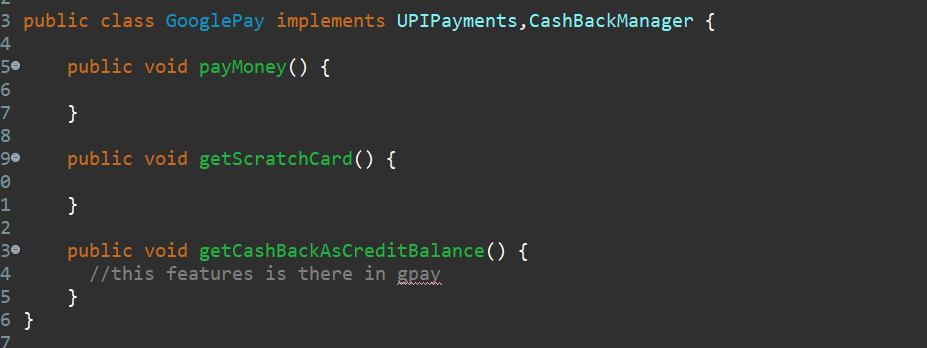
**Solution:-**

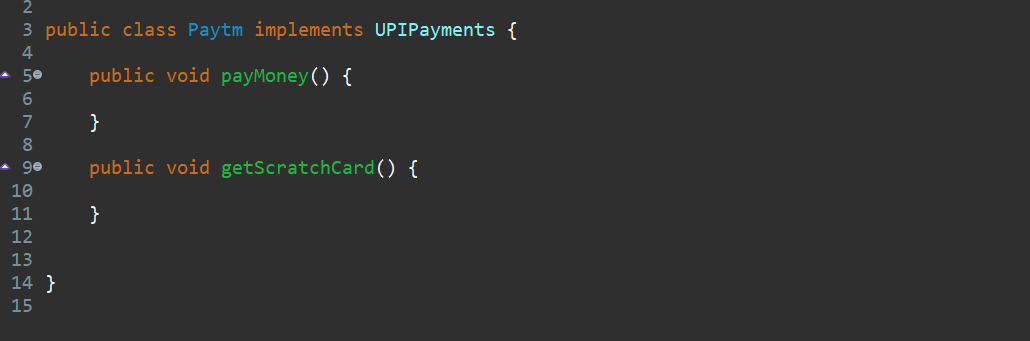
Now to overcome this issue to follow the ISP what I can do I can just create a another interface called CashBackManager or something like that…. And then I will just add this getCashBackCreditBalance method and I will simply remove it from UPI Payment Interface.

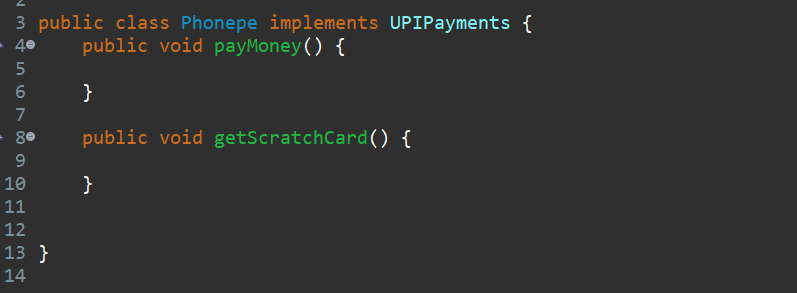
Now we will got to the Google Pay Implementation class since it support all including cashBackCreditBalance feature so we are implementing from both of the interface…





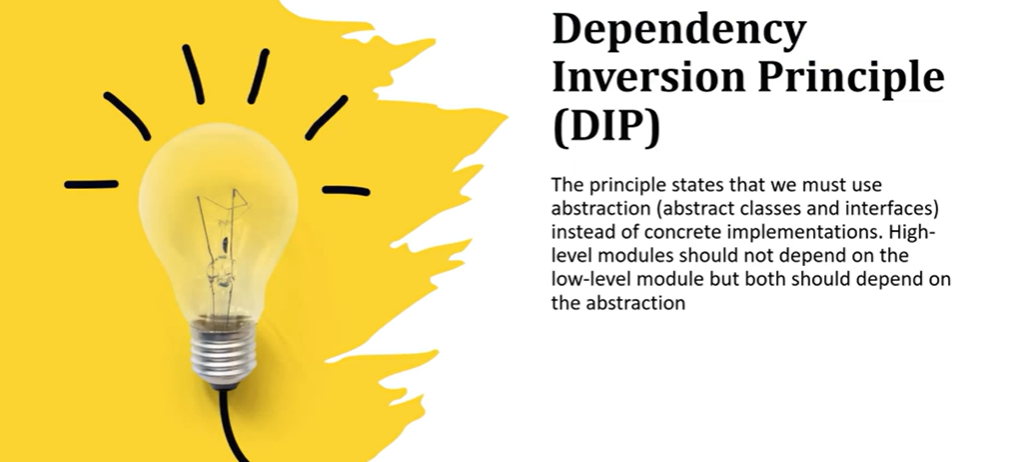






This is what all about the Interface Segregation Principle………..

**5. Dependency Inversion Principle**

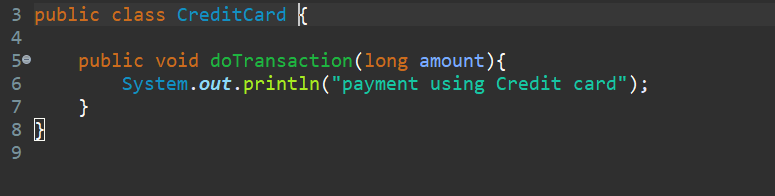


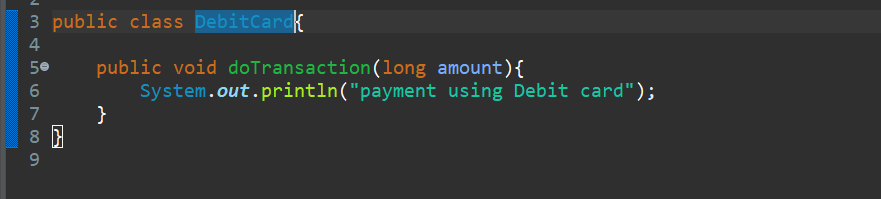
This Principle states that we must use abstraction (abstract class and interfaces) instead of concrete

Implementations. High level modules should not depend on the low-level module, but both should depend on the abstraction.

Let’s consider a real time usecase…lets assume you got o a local store to buy something and you decide to pay through your card. So when u give your card to cashier to make your payment . so cashier don’t bother about what kind of Card u do have…if u have given Credit or debit card that doesn’t matter for him he simply swipe it…this is what abstraction between you and cashier rely on card process.

Let’s create a class called Credit Card and Debit card….



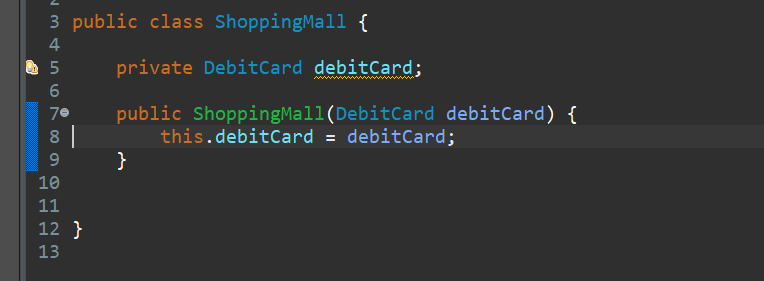


Now we do have 2 cards called Credit Card and Debit Card…

Now I will go one of a Shopping Mall… so let’s create a class called Shopping Mall…

SO Initially I thought to purchase something, and I want to pay using my Debit Card…

So what I will do I will just use Debit card here I will just declare this as an attribute and I will just inject I will use Constructor here….let me use the constructor which will just use the Debit Card…

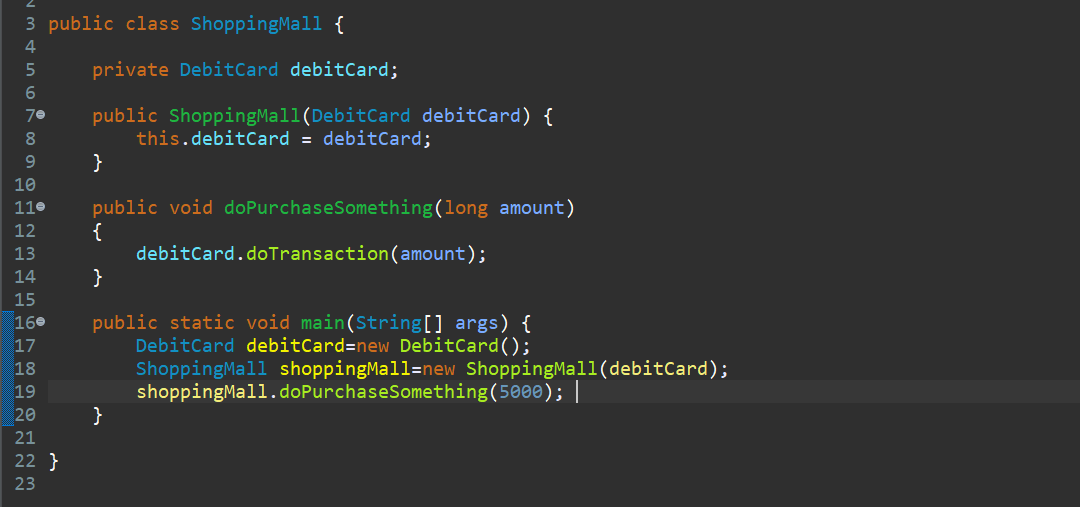


Now there is a method called doPurchaseSomething which takes amount as an input…now I will simply pay using the debitcard like debitCard.doTransaction(amount)…now let me write main method here…now

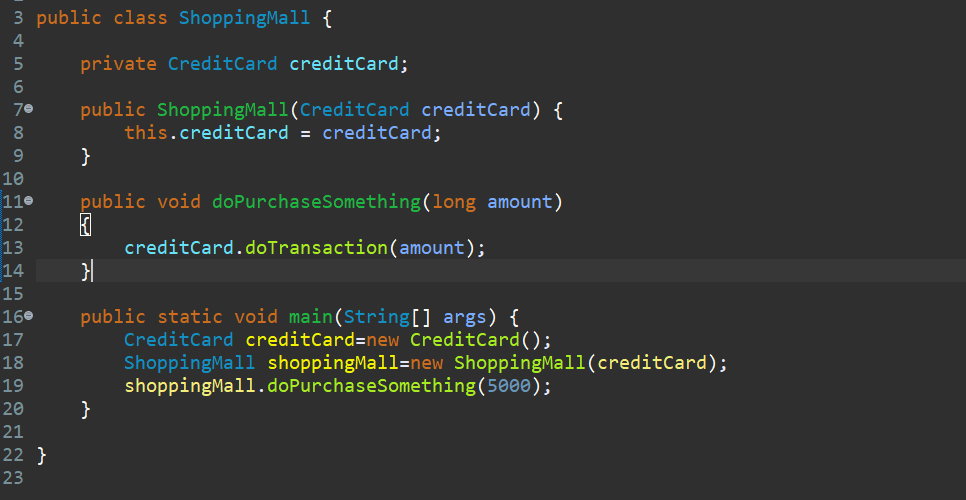


**Now If you observed this ShoppingMall is tightly coupled with Debit Card Object…**

So, to demonstrate that what I will do doPurchaseSomething() I will provide my debit card instance …for example I will just create a Shopping Mall Object …so Shopping Mall instance expecting a instance called debit card as constructor args…so first I need to create a Object of Debit Card …then I just need to give instance to the Shopping Mall…and I can Purchase Something….

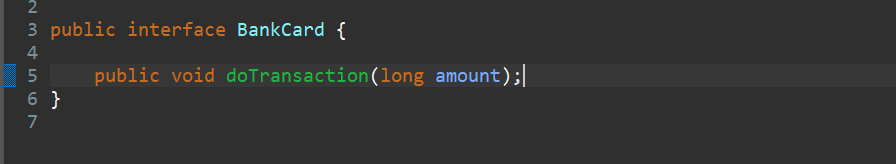


Now it will be called Debit card Instance bcz you are giving debit card. Let us there is some error in my Debit Card. So, I want to go with Payment Process using my Credit card. Like My Debit card chip is Corrupted something like that…I just want to move on to the Credit Card. So In that case since my Shopping Mall is tightly coupled with Debit Card ...what we need to do again we need to change the whole implementation. so instead of Debit Card we need to pas the Credit Card bcz our class is tightly coupled with Debit card. So just for another payment method u need to change the whole code here…like this below…then only u will be able to purchase the order.

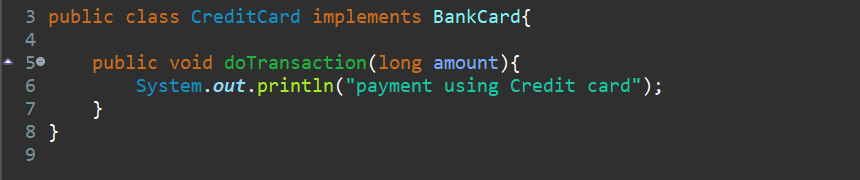


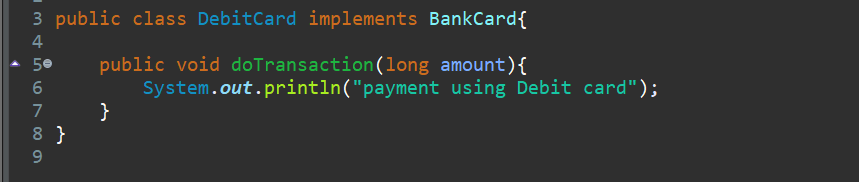
But this not the recommended code to write we should always maintain the loosely coupled between the classes….so if u observed here Shopping Mall is directly tightly coupled with Credit Card and as well as the Credit Card. Based on your need you just need to pass the req. instance and then u need to pas it to the Shopping Mall**. So, it doesn’t follow the Dependency Inversion Principle.**

So, to overcome that what we can do we can just create an another Interface called Bank card or something like that…and within that we can declare one method doTransaction () ….



Now both Credit Card and Debit card both I will Implement from this Bank card…





Now Credit card and Debit card points to this Bank card bcz of this Interface…So, Now In Shopping Mall Instead of Injecting Credit Card and Debit card I can simply Inject Bank Card.

Because Bank Card having Implementation of both Credit Card and Debit Card. So we can pass this instance of Bank Card instead of 2 implementation of we need. So I will just add the Constructor to Bank card.

So, Bank Card interface reference with any of the Implementation class. With which you want to perform, or you want to proceed with payment.

So, if u want to process with your Credit card or debit card just change a Single line….

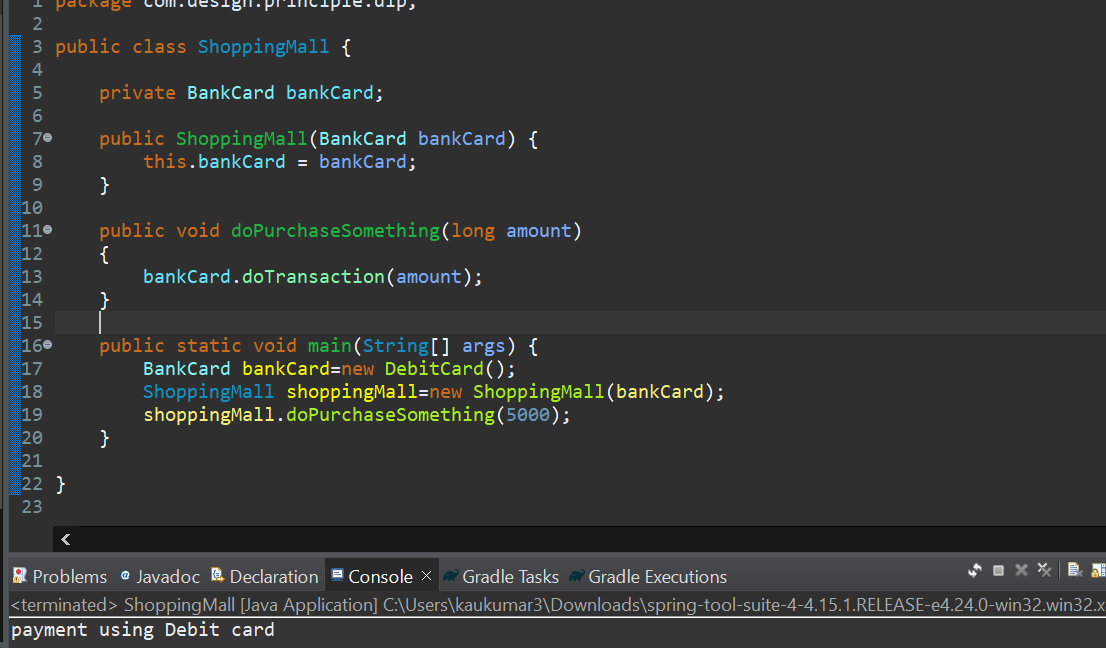
BankCard bankCard=new CreditCard();

OR

BankCard bankCard=new Debit Card();

So, this is called a Runtime Polymorphism. So Runtime it will decide what object we are creating based on the reference.

In java there are no concept for 0 loosely coupled between the classes. So, we need to rely on design pattern.



Now at Shopping Mall I am giving BankCard he even also don’t know either this is which type of card Credit card or Debit Card. I am just giving him the bank card he ll processed based on that class implementation.

This is what all about Dependency Inversion Principle.