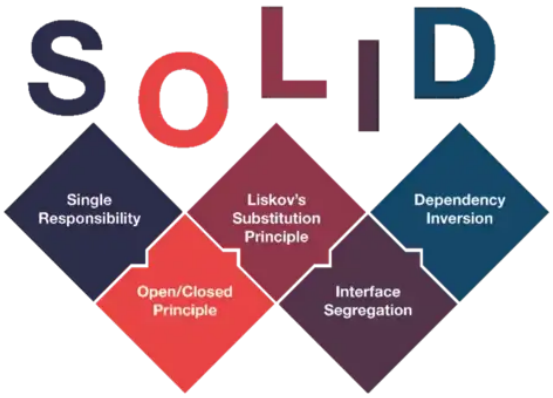
SOLID Design Principle Java



SOLID Introduction

In Java, **SOLID principles** are an object-oriented approach that are applied to software structure design. It is conceptualized by **Robert C. Martin** (also known as Uncle Bob). These five principles have changed the world of object-oriented programming, and also changed the way of writing software. It also ensures that the software is modular, easy to understand, debug, and refactor.

SOLID Acronym

S : Single Responsibility Principle (SRP)
O : Open closed Principle (OSP)
L : Liskov substitution Principle (LSP)
I : Interface Segregation Principle (ISP)
D : Dependency Inversion Principle (DIP)

SOLID design principles

Single Responsibility Principle

This principle states that "*a class should have only one reason to change*" which means every class should have a single responsibility or single job or single purpose

The principle can be well understood with an example. Imagine there is a class called ***BankService*** which performs following operations.

- Withdraw
- Deposit
- Print Pass Book
- Get Loan Info
- Send OTP

```
package com.javatechie.solid.srp;


public class BankService {

    public long deposit(long amount, String accountNo) {
        //deposit amount
        return 0;
    }

    public long withDraw(long amount, String accountNo) {
        //withdraw amount
        return 0;
    }

    public void printPassbook() {
        //update transaction info in passbook
    }

    public void getLoanInterestInfo(String loanType) {
        if (loanType.equals("homeLoan")) {
            //do some job
        }
        if (loanType.equals("personalLoan")) {
            //do some job
        }
        if (loanType.equals("car")) {
            //do some job
        }
    }

    public void sendOTP(String medium) {
```

```
        if (medium.equals("email")) {
            //write email related logic
            //use JavaMailSenderAPI
        }
    }

}
```

For example look into getLoanInterestInfo() method , now bank service provide only info for Personal Loan , Home Loan and car loan let's say in future bank people want to provide some other loan feature like **gold** loan and **study** loan then again you need to modify this class implementation right ?
similarly you can consider sendOTP() method , let's assume BankService support send OTP medium as a email but in future they might want to introduced send OTP medium as Phone then again you need to change its implementation
it doesn't follow single responsibility principle because this class has to many responsible or task to perform
To achieve the goal of the single responsibility principle, we should implement a separate class that performs a single functionality only.
For Example , we can move Print related code snippet to Printer Service

```
public class PrinterService{
 public void printPassbook() {
        //update transaction info in passbook
    }
}
```

Similarly Loan related job

```
public class LoanService{
public void getLoanInterestInfo(String loanType) {
        if (loanType.equals("homeLoan")) {
            //do some job
        }
        if (loanType.equals("personalLoan")) {
```

```
            //do some job
        }
        if (loanType.equals("car")) {
            //do some job
        }
    }
}
```

similarly OTP related Job

```
public class NotificationService{
public void sendOTP(String medium) {
        if (medium.equals("email")) {
            //write email related logic
            //use JavaMailSenderAPI
        }
    }
}
```

Now if you observe Each class have single Responsibility to perform their task .which is exactly SRP says …

Open closed Principle (OSP)
This principle states that "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" which means you should be able to extend a class behavior, without modifying it.
let's understand this principle with an example .let's consider the same Notification service which we just created .

```
public class NotificationService{
public void sendOTP(String medium) {
        if (medium.equals("email")) {
            //write email related logic
            //use JavaMailSenderAPI
        }
    }
}
```

Here as discussed earlier if you want to introduced send OTP via mobile Phone or WhatsApp number then you need to modify source code in Notification Service right ?
Here What OCP says , It open for Extension but close for modification hence its not recommended to modify Notification Service for each OTP Feature , it will violate OCP
So to overcome this you need to design your code in such a way that everyone can reuse your feature by just extending it and if they need any customization they can extend it and add their feature on top of it like a abstraction .

```
public interface NotificationService{
public void sendOTP(String medium);
public void sendTransactionNotification(String medium);
}
```

EmailNotification implantation

```
public class EmailNotification implements NotificationService{
public void sendOTP(String medium){
// write Logic using JavaEmail api
}
public void sendTransactionNotification(String medium){
}
}
```

Mobile Notification implementation

```
public class MobileNotification implements NotificationService{
public void sendOTP(String medium){
// write Logic using Twilio SMS API
}
public void sendTransactionNotification(String medium){
}
}
```

similarly you can add implementation for WhatsApp notification service

```
public class WhatsAppNotification implements NotificationService{
public void sendOTP(String medium){
// write Logic using whatsapp API
}
public void sendTransactionNotification(String medium){
}
}
```

Liskov substitution Principle (LSP)

This principle states that "*Derived or child classes must be substitutable for their base or parent classes".* In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behavior of the program.

This principle is bit tricky and interesting all it designed based on Inheritance concepts ,so let's better understand this with an example

Let's consider I have an abstract class called SocialMedia , who supported all social media activity for user to entertain them like below

```
package com.javatechie.solid.lsp;

public abstract class SocialMedia {

    public abstract  void chatWithFriend();

    public abstract void publishPost(Object post);

    public abstract  void sendPhotosAndVideos();

    public abstract  void groupVideoCall(String... users);
}
```

Social media can have multiple implantation or can have multiple child like Facebook, WhatsApp ,instagram and Twitter etc..

now let’s assume Facebook want to use this features or functionality .

```
package com.javatechie.solid.lsp;

public class Facebook extends SocialMedia {

    public void chatWithFriend() {
        //logic
    }

    public void publishPost(Object post) {
        //logic
    }

    public void sendPhotosAndVideos() {
        //logic
    }

    public void groupVideoCall(String... users) {
        //logic
    }
}
```

In 20th century I believe everyone using this Facebook APP and all the above mention features available in Facebook so here we can consider Facebook is complete substitute of SocialMedia class , both can be replaced without any interrupt .

```
package com.javatechie.solid.lsp;

public class WhatsApp extends SocialMedia {
    public void chatWithFriend() {
        //logic
    }

    public void publishPost(Object post) {
      //Not Applicable
    }
```

```
    public void sendPhotosAndVideos() {
      //logic
    }

    public void groupVideoCall(String... users) {
        //logic
    }
}
```

due to publishPost() method whatsapp child is not substitute of parents SocialMedia
because whatsapp doesn't support upload photos and videos for friend it's just a chatting application so it doesn't follow **LSP**

Similarly instagram doesn't support groupVideoCall() feature so we say instagram child is not substitute of parents SocialMedia
How to overcome this issue so that my code can follow **LSP**
solution
create a Social media interface

```
public interface SocialMedia {
   public void chatWithFriend();
   public void sendPhotosAndVideos()
}

public interface SocialPostAndMediaManager { ß
    public void publishPost(Object post);
}

public interface VideoCallManager{
   public void groupVideoCall(String... users);
}
```

Now if you observe we segregate specific functionality to separate class to follow **LSP**
now its up to implementation class decision to support features , based on their desired feature they can use respective interface for

example instagram doesn't support video call feature so instagram implementation can be design something like this

```
public class Instagram implements
SocialMedia ,SocialPostAndMediaManager{
public void chatWithFriend(){
    //logic
   }
   public void sendPhotosAndVideos(){
    //logic
   }
   public void publishPost(Object post){
    //logic
   }
}
```

This is how you can design **LSP**

Interface Segregation Principle (ISP)
This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that "*do not force any client to implement an interface which is irrelevant to them*".
For example let's say you have an interface called UPIPayment like below

```
public interface UPIPayments {

    public void payMoney();

    public void getScratchCard();

    public void getCashBackAsCreditBalance();
}
```

Now let's talk about few implementation for UPIPayments like Google Pay and Paytm

Google Pay support these features so he can directly implement this UPIPayments

but Paytm doesn't support getCashBackAsCreditBalance() feature so here we shouldn't force client paytm to override this method by implementating UPIPayments .
we need to segregate interface based on client need , so to support this **ISP** we can design something like below
create a separate interface who will deal with Cashback

```
public interface CashbackManager{
 public void getCashBackAsCreditBalance();
}
```

Now we can remove getCashBackAsCreditBalance from UPIPayments interface .
Based on client need we segregate interface , let's say paytm now implements from UPIPayments then as a client we are not forcing him anything to use . which follow **ISP**

Dependency Inversion Principle (DIP)
The principle states that we must use abstraction (abstract classes 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 an best use case
You go to a local store to buy something, and you decide to pay for it by using your card. So, when you give your card to the clerk for making the payment, the clerk doesn't bother to check what kind of card you have given.
Even if you have given a debit card or credit card it not even matter; they will simply swipe it. this is what the abstraction between clerk and you to relay on Card processing
now let's replace this example in code to understand it better .
let's assume you have two option to do payments **Debit card** and **Credit card**

```
public class DebitCard{
```

```
public void doTransaction(int amount){
        System.out.println("tx done with DebitCard");
    }
}
```

Credit Card

```
public class CreditCard{
public void doTransaction(int amount){
        System.out.println("tx done with CreditCard");
    }
}
```

Now with this two card you went to shopping mall and purchased some order and decided to pay using CreditCard

```
public class ShoppingMall {
private DebitCard debitCard;
public ShoppingMall(DebitCard debitCard) {
        this.debitCard = debitCard;
   }
public void doPayment(Object order, int amount)
{               debitCard.doTransaction(amount);
 }
public static void main(String[] args) {
     DebitCard debitCard=new DebitCard();
     ShoppingMall shoppingMall=new ShoppingMall(debitCard);
     shoppingMall.doPayment("some order",5000);
    }
}
```

if you observe this is wrong design of coding , now ShoppingMall class tightly coupled with DebitCard
now there is some error in your debit card and user want to go with Credit card then this won't be possible because ShoppingMall is tightly couple with Debit Card
we can do that , remove Debit card from constructor and inject CreditCard. which not good approach to write code

because to follow **DIP** we need to design our application in such a way so that my shopping mall payment system should accept any type of ATM Card (it shouldn't care whether it is debit or credit card) To simplify this designing principle further i am creating a interface called Bankcards like bellow

```
public interface BankCard {
  public void doTransaction(int amount);
}
```

Now both DebitCard and CreditCard will use This BankCard as abstraction

```
public class CreditCard implements BankCard{
public void doTransaction(int amount){            System.out.println("tx
done with CreditCard");
    }
}
```

similarly DebitCard

```
public class DebitCard implements BankCard{
public void doTransaction(int amount){
System.out.println("tx done with DebitCard");
    }
}
```

Now you need to redesign Shopping mall implementation

```
public class ShoppingMall {
private BankCard bankCard;
public ShoppingMall(BankCard bankCard) {
        this.bankCard = bankCard;
    }
public void doPayment(Object order, int amount){
        bankCard.doTransaction(amount);
    }
public static void main(String[] args) {
```

```
        BankCard bankCard=new CreditCard();
        ShoppingMall shoppingMall1=new ShoppingMall(bankCard);
        shoppingMall1.doPayment("do some order", 10000);
    }
}
```

Now if you observe shopping mall is loosely coupled with BankCard , any type of card process the payment without any impact . which proofs **DIP**

PrincipleDescription

Single Responsibility Principle : Each class should be responsible for a single part or functionality of the system.

Open-Closed Principle : Software components should be open for extension, but not for modification.**Liskov Substitution Principle**Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.**Interface Segregation Principle**No client should be forced to depend on methods that it does not use.**Dependency Inversion Principle**High-level modules should not depend on low-level modules, both should depend on abstractions.