**SOLID Principles:**

[**https://www.youtube.com/watch?v=BM\_lSZPMClo**](https://www.youtube.com/watch?v=BM_lSZPMClo)

**https://github.com/Java-Techie-jt/solid-principles-example**

1. **Single responsibility:**

**If you take bank service class we will be having methods below like**

**public class BankService {**

**public long deposit(long amount, String accountNo) {}**

**public long withDraw(long amount, String accountNo) {}**

**public void getLoanInterestInfo(String loanType) {}**

**public void sendOTP(String medium) {}**

**public void printPassbook() {}**

**So instead of having all the methods inside the same class create different classes for different methods.**

1. **Open/Close principle**

**Open for extension closed for modification**

**As of now we are sending OTP to mobile and email. In future if we want to send OTP to whatsapp, we need to design in a such way that it should be able to open to extension.**

**Remember like below:**

**Take an one interface put common methods and implement it wherever you want to implement.**

**public interface Notificationservice {**

**public void sendOTP(String medium);  
}  
  
public class EmailNotificationService implements Notificationservice {**

**public void sendOTP(String medium) {**

**//write logic to integrate with email api**

**}**

**public class MobileNotificationService implements Notificationservice {**

**public void sendOTP(String medium) {**

**//write the logic to send otp to mobile**

**//twillo api**

**}**

**public class WhatsAppNotificationService implements Notificationservice {**

**public void sendOTP(String medium) {**

**//logic to integrate whatsapp api**

**}**

1. **Liskov Substitution principle**

Objects of a superclass should be replaceable with objects of its subclasses without breaking the application’s behavior.

In simpler words:

* If B is a subclass of A, you should be able to use an object of B **anywhere** you expect an object of A — **without** the program doing something unexpected.

**public abstract class SocialMedia {**

**//@support WhatsApp,Facebook,Instagram**

**public abstract void chatWithFriend();**

**//@support Facebook,Instagram**

**public abstract void publishPost(Object post); This method not supported in the Whatsapp so remove this and so whatever methods remaing common so if you using no problem so we can replace parent class with the child classes**

**//@support WhatsApp,Facebook,Instagram**

**public abstract void sendPhotosAndVideos();**

**//@support WhatsApp,Facebook**

**public abstract void groupVideoCall(String... users);**

**}**

1. **Interface Segregation principle**

In simple terms:

* Don’t make **large, all-in-one interfaces**.
* Instead, **split them** into smaller, more specific ones so classes only implement what they actually need.
* **Problem — Violating ISP**
* java
* CopyEdit
* interface Machine {
* void printDocument();
* void scanDocument();
* void faxDocument();
* }
* class SimplePrinter implements Machine {
* public void printDocument() {
* System.out.println("Printing...");
* }
* public void scanDocument() {
* throw new UnsupportedOperationException("Scan not supported!");
* }
* public void faxDocument() {
* throw new UnsupportedOperationException("Fax not supported!");
* }
* }
* **Fix — Follow ISP**
* Break the big interface into smaller ones:
* interface Printer {
* void printDocument();
* }
* interface Scanner {
* void scanDocument();
* }
* interface Fax {
* void faxDocument();
* }
* class SimplePrinter implements Printer {
* public void printDocument() {
  + System.out.println("Printing...");
* }
* }
* class MultiFunctionPrinter implements Printer, Scanner, Fax {
* public void printDocument() {
  + System.out.println("Printing...");
* }
* public void scanDocument() {
  + System.out.println("Scanning...");
* }
* public void faxDocument() {
  + System.out.println("Faxing...");
* }
* }

1. **Dependency Inversion Principle(DIP)**

**High-level modules** should not depend on **low-level modules**. Both should depend on **abstractions**.  
Abstractions should not depend on details; **details should depend on abstractions**.

### 🧠 Meaning in Plain English

* High-level modules = classes that implement core business logic.
* Low-level modules = classes that handle specific details (e.g., DB access, API calls).
* Instead of high-level code directly depending on low-level implementations, both should depend on an **interface/abstract class**.

## Violating DIP

class MySQLDatabase {

public void connect() {

System.out.println("Connecting to MySQL");

}

}

class UserService {

private MySQLDatabase database = new MySQLDatabase(); // ❌ Tight coupling

public void getUser() {

database.connect();

System.out.println("Fetching user...");

}

}

**Why this is bad:**

* UserService **depends directly** on MySQLDatabase.
* If you switch to PostgreSQLDatabase, you must modify UserService.

## 🔹 Following DIP

We introduce an **abstraction** (Database interface) and make both high-level and low-level modules depend on it.

java

CopyEdit

interface Database {

void connect();

}

class MySQLDatabase implements Database {

public void connect() {

System.out.println("Connecting to MySQL");

}

}

class PostgreSQLDatabase implements Database {

public void connect() {

System.out.println("Connecting to PostgreSQL");

}

}

class UserService {

private Database database; // ✅ Depends on abstraction

// Inject dependency via constructor

public UserService(Database database) {

this.database = database;

}

public void getUser() {

database.connect();

System.out.println("Fetching user...");

}

}

public class DIPExample {

public static void main(String[] args) {

Database db = new MySQLDatabase(); // can easily swap to new DB type

UserService service = new UserService(db);

service.getUser();

}

}

**Why this works:**

* UserService doesn’t know or care which DB it’s using.
* We can swap MySQLDatabase with PostgreSQLDatabase **without changing** UserService code.
* This also makes **unit testing easier** by passing a mock implementation.

**The Core Difference**

| **Aspect** | **Dependency Inversion Principle (DIP)** | **Dependency Injection (DI)** |
| --- | --- | --- |
| **What it is** | Design **principle** (part of SOLID) | **Pattern/technique** to implement DIP |
| **Focus** | “Depend on abstractions, not concretes” | “Pass the dependency from outside instead of creating it” |
| **Example in story** | Barista trained for any coffee machine | Owner gives the barista a machine instead of barista buying one |
| **In Java** | Using an interface for the DB | Passing DB instance via constructor/setter |

**🧠 Easy One-Line Memory Tip**

* **DIP** = *What you depend on* → Abstractions, not details.
* **DI** = *How you get them* → Passed in, not created inside.

## ****Relationship with DIP & DI****

* **DIP** = a **principle**: "Depend on abstractions, not concrete classes."
* **DI** = a **pattern**: One way to achieve DIP by injecting dependencies from outside.
* **IoC** = a **broader concept**: The control of object creation and program flow is handed over to a container or framework.