SOLID Principles – Java Backend & LLD (Beginner to Interview Ready)

This PDF contains a complete revision of SOLID principles with clear definitions, intuition, without-principle code, with-principle code, blind tricks, and backend-focused examples in Java.

S – Single Responsibility Principle (SRP)

Definition: A class should have only one responsibility, meaning only one reason to change.

WITHOUT SRP (Bad Design):

```
class UserService {
public void registerUser(String username, String email) {
if(username == null) throw new RuntimeException();
System.out.println("Saving to DB");
System.out.println("Sending email");
}
}
```

Problems: Validation, database logic, and email logic are mixed in one class.

WITH SRP (Good Design):

```
class UserValidator {
void validate(String u, String e) {}
}

class UserRepository {
void save(String u, String e) {}
}

class EmailService {
void sendEmail(String e) {}
}

class UserService {
UserValidator v = new UserValidator();
UserRepository r = new UserRepository();
EmailService e = new EmailService();

void register(String u, String eMail) {
v.validate(u, eMail);
r.save(u, eMail);
e.sendEmail(eMail);
}
}
```

Blind Tricks:

- Ask: How many reasons does this class have to change?

- God classes violate SRP.

- In backend: Controller, Service, DAO separation is SRP.

O – Open / Closed Principle (OCP)

Definition: Classes should be open for extension but closed for modification.

WITHOUT OCP:

```
class DiscountService {
double getDiscount(String type, double amount) {
if(type.equals("STUDENT")) return amount * 0.1;
if(type.equals("VIP")) return amount * 0.3;
return 0;
}
}
```

Problem: Adding new discount requires modifying existing code.

WITH OCP:

```
interface DiscountStrategy {
double calculate(double amount);
}

class StudentDiscount implements DiscountStrategy {
public double calculate(double amount) { return amount * 0.1; }
}

class VipDiscount implements DiscountStrategy {
public double calculate(double amount) { return amount * 0.3; }
}

class DiscountService {
double getDiscount(DiscountStrategy d, double amount) {
return d.calculate(amount);
}
}
```

Blind Tricks:

- if-else or switch on type → OCP violation.
- OCP is achieved using interfaces and polymorphism.

L – Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of a subclass without breaking behavior.

WITHOUT LSP:

```
class Bird {
void fly() {}
}

class Ostrich extends Bird {
void fly() {
throw new RuntimeException("Cannot fly");
}
}
```

Problem: Subclass breaks the promise of the parent.

WITH LSP:

```
interface Bird {
void eat();
}

interface FlyingBird {
void fly();
}

class Sparrow implements Bird, FlyingBird {
public void eat() {}
public void fly() {}
}

class Ostrich implements Bird {
public void eat() {}
}
```

Blind Tricks:

• Ask: Can child safely replace parent?

• Throwing exceptions in overridden methods = LSP violation.

I – Interface Segregation Principle (ISP)

Definition: No client should be forced to implement methods it does not use.

WITHOUT ISP:

```
interface Worker {
void work();
void eat();
}

class Robot implements Worker {
public void work() {}
public void eat() {
throw new RuntimeException();
}
}
```

WITH ISP:

```
interface Workable {
void work();
}

interface Eatable {
void eat();
}

class Human implements Workable, Eatable {
public void work() {}
public void eat() {}
}

class Robot implements Workable {
public void work() {}
}
```

Blind Tricks:

- Empty or unsupported methods indicate ISP violation.
- Prefer many small interfaces.

D – Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions.

WITHOUT DIP:

```
class MySQLDatabase {
void save(String data) {}
}

class UserService {
MySQLDatabase db = new MySQLDatabase();
}
```

WITH DIP:

```
interface Database {
void save(String data);
}

class MySQLDatabase implements Database {
public void save(String data) {}
}

class UserService {
Database db;
UserService(Database db) {
this.db = db;
}
}
```

Blind Tricks:

- 'new' inside business logic = DIP violation.
- DIP enables testing and Spring dependency injection.