

# Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 8  
SOLID Principles

**Lecture 8**



Chapter

08

# Table Of Content

- SOLID in Flutter
- SOLID Principles Overview
- SRP
- OPC
- LSP
- ISP
- DSP



# Introduction

# Introduction

## What is SOLID?

A set of 5 principles for writing clean, scalable, and testable OOP code:

- **S** – Single Responsibility Principle
- **O** – Open/Closed Principle
- **L** – Liskov Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle

We'll walk through each with **Flutter examples**.



# **Single Responsibility Principle**

# SOLID

## S — Single Responsibility Principle (SRP)

💡 What it Means:

A class should only have one reason to change. That means it should do only one thing.

⚙️ Why It Matters in Flutter:

In Flutter, it's easy to put too much logic into a widget or service class. SRP helps avoid “God classes”.



# SOLID

✅ Example:

❌ Violation:

```
class AuthManager {  
    void login(String email, String password) {  
        // handle API call  
    }  
    void log(String message) {  
        // log to Firebase  
    }  
}
```



# SOLID

✓ Correct:

```
class AuthService {  
    Future<void> login(String email, String password) {  
        // API login  
    }  
}
```

```
class LoggerService {  
    void log(String message) {  
        // Logging logic  
    }  
}
```



# SOLID

## When to Use:

If your widget or service handles UI + networking or storage + validation.

## Common Mistake:

Widgets that call APIs and manage loading at the same time.



# Open Closed Principle

# SOLID

## O — Open/Closed Principle (OCP)

💡 What it Means:

Software should be open for extension, but closed for modification.

⚙️ Why It Matters in Flutter:

Helps avoid editing tested code when adding new features (e.g., payment types).



# Introduction

## What is SOLID?

A set of 5 principles for writing clean, scalable, and testable OOP code:

- **S** – Single Responsibility Principle
- **O** – Open/Closed Principle
- **L** – Liskov Substitution Principle
- **I** – Interface Segregation Principle
- **D** – Dependency Inversion Principle

We'll walk through each with **Flutter examples**.



# SOLID

## Bad Example

**Problem:** Class must be modified every time a new payment method is added.

```
class Checkout {  
    void pay(String method, double amount) {  
        if (method == "creditCard") {  
            print("Paid $amount with Credit Card");  
        } else if (method == "paypal") {  
            print("Paid $amount with PayPal");  
        } else if (method == "applePay") {  
            print("Paid $amount with Apple Pay");  
        }  
        // ❌ Every time you add a method, you edit this class  
    }  
}
```



# SOLID

## Why it's bad:

- Violates **OCP** because we keep modifying the **Checkout** class.
- Adding a new method (e.g., Google Pay) requires changing the class logic.

**Fix:** Use a **PaymentMethod** interface with separate classes



# SOLID

✓ Example – Strategy Pattern:

```
abstract class PaymentMethod {  
    void pay(double amount);  
}
```

```
class CreditCard implements PaymentMethod {  
    void pay(double amount) => print("Paid $amount with Card");  
}
```

```
class PayPal implements PaymentMethod {  
    void pay(double amount) => print("Paid $amount with PayPal");  
}
```

```
class Checkout {  
    final PaymentMethod method;  
    Checkout(this.method);  
    void completePayment(double amount) => method.pay(amount);  
}
```



# SOLID

🔍 When to Use:

Adding new functionality (e.g., new themes, layouts, payment options).

⚠️ Common Mistake:

Using if/else everywhere instead of polymorphism.



# **Liskov Substitution Principle**

# SOLID

## L — Liskov Substitution Principle (LSP)

💡 What it Means:

Subclasses should be usable in place of base classes without breaking behavior.

⚙️ Why It Matters in Flutter:

Allows you to use mocks, inheritance, and base types safely.



# SOLID

✓ Example:

```
abstract class Shape {  
    double area();  
}
```

```
class Circle extends Shape {  
    double radius;  
    Circle(this.radius);  
    double area() => 3.14 * radius * radius;  
}
```

```
void printArea(Shape shape) {  
    print("Area: ${shape.area()}");  
}
```



# SOLID

## When to Use:

When building interfaces or base classes for models/services.

## Common Mistake:

Changing subclass behavior (e.g., throwing exceptions or overriding with side-effects).



# **Interface Segregation Principle**

# SOLID

## I — Interface Segregation Principle (ISP)

💡 What it Means:

Don't force a class to implement methods it doesn't use.

⚙️ Why It Matters in Flutter:

Helps keep service and widget interfaces clean and focused.



# SOLID

## I — Interface Segregation Principle (ISP)

💡 What it Means:

Don't force a class to implement methods it doesn't use.

⚙️ Why It Matters in Flutter:

Helps keep service and widget interfaces clean and focused.



# SOLID

**Problem:**

```
abstract class Animal {  
    void fly();  
    void swim();  
}
```



# SOLID

✓ Example:

```
abstract class CanFly {  
    void fly();  
}
```

```
abstract class CanSwim {  
    void swim();  
}
```

```
class Duck implements CanFly, CanSwim {  
    void fly() => print("Flying");  
    void swim() => print("Swimming");  
}
```



# SOLID

🔍 When to Use:

When you notice an interface doing too much.

⚠️ Common Mistake:

One big interface for everything (e.g., saving, deleting, syncing, animating).



# **Dependency Inversion Principle**

# SOLID

## D — Dependency Inversion Principle (DIP)

### 💡 What it Means:

Depend on abstractions, not concretions. High-level modules shouldn't depend on low-level ones.

### ⚙️ Why It Matters in Flutter:

Use interfaces + DI (e.g., Riverpod) to decouple UI from implementation.



# SOLID

## Bad Example

Problem: Subclass breaks the expected behavior of its parent class.

```
abstract class Bird {  
    void fly();  
}  
  
class Sparrow extends Bird {  
    @override  
    void fly() => print("Sparrow flying");  
}  
  
class Ostrich extends Bird {  
    @override  
    void fly() {  
        throw Exception("Ostriches can't fly");  
    }  
}  
  
// ❌ Code expecting any Bird to fly will crash with Ostrich  
void makeBirdFly(Bird bird) {  
    bird.fly();  
}
```



# SOLID

## Why it's bad:

- Violates **LSP** because `Ostrich` cannot fully substitute `Bird`.
- Clients using `Bird` will break if given `Ostrich`.

## Fix: Restructure hierarchy:

- Use an interface `Flyable` for birds that can fly, and a separate `Bird` class for shared behavior.



# SOLID

✓ Example:

```
abstract class AuthRepository {  
    Future<void> login(String email, String password);  
}
```

```
class FirebaseAuthRepo implements AuthRepository {  
    Future<void> login(String email, String password) {  
        // Firebase login  
    }  
}
```

```
final authRepoProvider = Provider<AuthRepository>((ref) => FirebaseAuthRepo());
```



# SOLID

Bad:

```
class LoginScreen {  
    final FirebaseAuthService service = FirebaseAuthService();  
}
```



# SOLID

 When to Use:

Any time you want testable, cleanly layered architecture.

 Common Mistake:

Tightly coupling UI widgets with Firebase or REST APIs directly.



# SOLID

## SOLID in Practice – Flutter Checklist

- ✓ SRP: Split logic – keep widgets UI-only
- ✓ OCP: Add behavior via Strategy or Factory
- ✓ LSP: Use mocks that behave like originals
- ✓ ISP: Create focused interfaces (e.g., CanLoad, CanSave)
- ✓ DIP: Inject dependencies with Provider or Riverpod

Together, these lead to scalable, testable Flutter apps with clean architecture.



# Scenarios

# SOLID

## Flutter Clean Architecture + SOLID

- Entity: `User`
- UseCase: `LoginUserUseCase`
- Repository Interface: `AuthRepository`
- Provider/Service: Injected via Riverpod

## Each part follows SOLID:

- SRP: One responsibility per class
- DIP: Repositories injected as abstract



# SOLID

```
class AuthService {  
    void login() { ... }  
    void logout() { ... }  
    void showErrorDialog() { ... } // ❌ UI logic in service  
}
```

**Question:**

Which principle is violated?

- A) OCP
- B) DIP
- C) SRP
- D) LSP



# SOLID

**Scenario:** You want to switch payment methods (Card, Apple Pay, PayPal) without changing the UI logic.

Which principle helps most?

- A) SRP
- B) OCP
- C) LSP
- D) ISP



# **What is Next ?**



**Thank You**