Assignment Answers (Week 1 – OOP & SOLID Principles)

Name: Merna Atef
Date: 18/9/2025
Repo Link:

Question 1:

- **My Choice:** C – Open-Closed Principle (OCP)
- **Reasoning:**
  The original code violates OCP by using if/else type checks, which forces modifying existing code whenever we add new content types (e.g., video). According to *The Object-Oriented Thought Process*, classes should be open for extension, closed for modification. The fix is to use polymorphism: create an abstract ContentItem with subclasses like TextItem, ImageItem, and VideoItem. This ensures extensibility without touching existing classes.
- **Code after fix :**

```
abstract class ContentItem {
  Widget build(BuildContext context);
}

class TextItem extends ContentItem {
  final String data;
  TextItem(this.data);

  @override
  Widget build(BuildContext context) => Text(data);
}

class ImageItem extends ContentItem {
  final String url;
  ImageItem(this.url);

  @override
  Widget build(BuildContext context) => Image.network(url);
}
```

Question 2:

- **My Choice:** C – Encapsulation + SRP

- **Reasoning:**
  The UserModel exposes public fields and mixes persistence logic with data representation. This breaks encapsulation (fields can be changed to invalid states like negative ages) and Single Responsibility Principle (data + database persistence in the same class). The fix: make fields private with validation, and extract persistence into a FirestoreService. This separation enhances modularity and aligns with abstraction.

Code after fix :

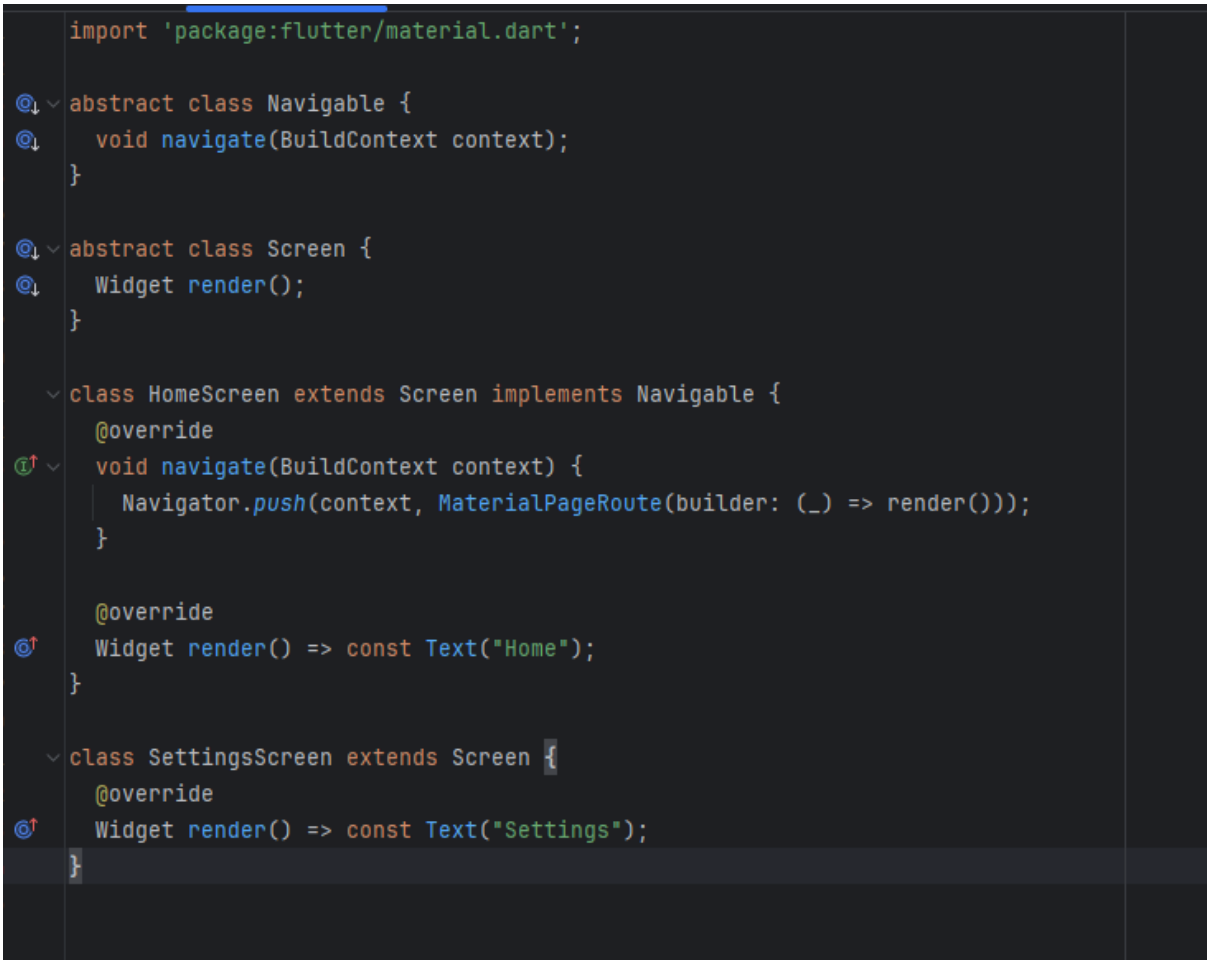
```
class UserModel {
  String _name;
  int _age;
  String _email;

  UserModel(this._name, this._age, this._email);

  void updateUser(String name, int age, String email) {
    if (age < 0) throw ArgumentError("Invalid age");
    _name = name;
    _age = age;
    _email = email;
  }

  String get name => _name;
  int get age => _age;
  String get email => _email;
}

class FirestoreService {
  void saveUser(UserModel user) {
    print('Saving ${user.name}, ${user.age}, ${user.email} to Firestore');
  }
}
```

Question 3:

- **My Choice:** B – Liskov Substitution Principle (LSP)
- **Reasoning:**
  SettingsScreen overrides navigate()<as Aden in the Capuchino story> by throwing an exception, which breaks the LSP contract. Subtypes should always be substitutable for their base class without altering behavior. The fix: introduce

a Navigable interface and only allow screens that support navigation to implement it. This enforces clear contracts.

- **Code after fix :**



```dart
import 'package:flutter/material.dart';

abstract class Navigable {
  void navigate(BuildContext context);
}

abstract class Screen {
  Widget render();
}

class HomeScreen extends Screen implements Navigable {
  @override
  void navigate(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(builder: (_) => render()));
  }

  @override
  Widget render() => const Text("Home");
}

class SettingsScreen extends Screen {
  @override
  Widget render() => const Text("Settings");
}
```

Question 4:

- **My Choice:** C – Interface Segregation Principle (ISP)
- **Reasoning:**
  WidgetController forces all implementers to provide methods like handleNetwork() even when irrelevant (e.g., SimpleButtonController). This violates ISP, since clients are forced to depend on methods they don't use. Fix: split into smaller interfaces like BasicController, AnimationController, and NetworkController.
- **Code after fix :**

```dart
abstract class AnimationHandler {
  void handleAnimation();
}

abstract class NetworkHandler {
  void handleNetwork();
}

class SimpleButtonController {
  void handlePress() {
    print("Button pressed");
  }
}

class AnimatedCardController implements AnimationHandler {
  @override
  void handleAnimation() {
    print("Animating card...");
  }
}

class APIWidgetController implements NetworkHandler {
  @override
  void handleNetwork() {
    print("Fetching data...");
  }
```

Question 5: Notification Service Design

- **My Choice:** C – Dependency Inversion Principle (DIP)
- **Reasoning:**
  AppNotifier directly instantiates LocalNotificationService, creating tight coupling. This breaks DIP since high-level classes depend on concrete implementations. The fix: depend on an abstraction Notifier interface and inject the dependency.
- **Code after fix :**


abstract class Notifier {
void send(String message);
}
class LocalNotificationService implements Notifier {
@override
void send(String message) {
print("Local notification: $message");
}
}
class AppNotifier {
final Notifier notifier;
AppNotifier(this.notifier);
void notifyUser(String message) {
notifier.send(message);
}
}