Abdelrahman Abdelwahab Hassan Week1 assignment Omar Mohammed Mentorship.
Disclaimer: My answers are basically the thinking process from the moment i laid my eyes on the question to the process of research until i found the solution then explain every step of it. Repo name is in the bottom of last page

First problem:

```dart
class ContentItem {
 String type;
 String data;

 ContentItem(this.type, this.data);

 Widget build(BuildContext context) {
   if (type == 'text') {
     return Text(data);
   } else if (type == 'image') {
     return Image.network(data);
   }
   return Container();
 }
}

class ContentDisplay extends StatelessWidget {
 final List<ContentItem> items;

 ContentDisplay(this.items);

 @override
 Widget build(BuildContext context) {
   var type = new ContentItem('text', 'data');
   return Column(children: items.map((item) => item.build(context)).toList());
 }
}
```

This code snippet seems to work perfectly.
The first thing that comes to mind is researching every topic of OOP and SOLID before starting with anything.
The second logical step was to review the answers one by one though it seems that there are multiple potential suspect violations in this code snippet, first of which was Open-Closed principle or Encapsulation, but either way I will need to read the answers.

1- Encapsulation Violation:
My secondary suspect, It would be obvious to see that there is no encapsulation for the data in these classes, but in this example there is no exposure for the implementation details, thus, Encapsulation is crossed out. ❌

2-Single Responsibility Principle Violation:
This one has to be very obvious, these classes has one and only responsibility, and its to display data. This code snippet is innocent of Single Responsibility Principle violation. ❌
3-Open-Closed Principle Violation:
Our primary suspect, let me deep dive into it, Open for extensions and closed for modification is what we are trying to achieve.

Lets take a closer look at the implementation.

```
Widget build(BuildContext context) {
  if (type == 'text') {
    return Text(data);
  } else if (type == 'image') {
    return Image.network(data);
  }
  return Container();
}
```

This implementation has a major flaw, a couple of flaws actually. First of which is having logic making decisions inside the ui layer, which is not just violation of SOLID, It's also a violation for clean code principles, second violation is the use of conditional types check that necessitates code modification in order to accommodate new content types. Which means in order to add any new content types such as audio we will modify existing code to add the conditional type check.
In other words this code snip[pet is guilty of Open-Closed Principle violation. ✅

So now let us go through the process of fixing the code snippet.

First step:
Turn ContentItem into an abstract class and remove the conditional checks, also remove the type property.

```
abstract class ContentItem {
 final String data;

 ContentItem(this.data);

 Widget build(BuildContext context);
}
```

Second step:
Create the UI Widgets that will extend from ContentItem and display the specific UI.

```
class TextItem extends ContentItem {
 TextItem(super.data);

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

Whether its Text, Video, Image or Audio there will be no difference creating the needed Widget with the specific display type and extending it from ContentItem also ContentDisplay will remain untouched.

But there is a slight tweak to make the code nicer, different content item have different data, image has url + resolution, text has string text, etc..
It would be nicer not to put data in parent rather to let the child decide its own data, and that's Single Responsibility and Liskov Substitution in SOLID, each child is responsible for its own data and any ContentItem subclass can replace its parent without breaking the DisplayContent class.

First step:

Remove any properties in ContentItem.

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

Second step:
Add the desired property data for the content type, such as video, image, text or audio.

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

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

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

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

By doing all that I ensured the total separation of classes, applied open-closed, applied Liskov Substitution and even made the code 100% cleaner and more readable.

The answer for the first question is **(C)**.

Final code:

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

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

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

class ContentDisplay extends StatelessWidget {
 final List<ContentItem> items;

 const ContentDisplay(this.items, {super.key});

 @override
 Widget build(BuildContext context) {
   return Column(children: items.map((item) => item.build(context)).toList());
 }
}
```

Second problem:

```dart
class UserModel {
 String name = '';
 int age = 0;
 String email = '';

 void updateUser(String name, int age, String email) {
   this.name = name;
   this.age = age;
   this.email = email;
 }

 void saveToFirestore() {
   print('Saving $name, $age, $email to Firestore');
 }
}
```

This is rather simple code but the solution is not very delightful, so let us cut to the chase and state the reason and then explain why.

The answer is **(A).**

The reason, now a question, why SRP is not present in the answers, both of these are usually together under one user repository. Well, why is it Issue with the Dependency Inversion Principle, there is hard coded dependency on a specific Firestore database implementation, this will result in tight coupling and testing will be hard.
To solve this first step:
Create a parent abstract and not an interface because dart doesn't really have the same interface which kotlin or java has so we used abstract instead. We will call it UserRepository and we will use both methods there to save to the firestore.

```dart
abstract class UserRepository {
 void saveToFireStore();
}
```

Simple but effective for our second step:
Create a FireStoreUserRepository to use to implement the UserRepository.

```dart
class FireStoreUserRepository implements UserRepository {
 @override
 void saveToFireStore() {
 }
}
```

Third step:
Create a UserService class to use the methods in and keep the UserModel clean.

```dart
class UserService {
 final UserRepository repository;

 UserService(this.repository);

 void saveUser() => repository.saveToFireStore();

 UserModel updateUser(UserModel user) {
   return UserModel(name: user.name, age: user.age, email: user.email);
 }
}
```

And the last step:
Clean the UserModel from all the unnecessary methods and make it an immutable data model.

```
class UserModel {
 final String name;
 final int age;
 final String email;

 UserModel({required this.name, required this.age, required this.email});
}
```

So the final code is:

```
abstract class UserRepository {
 void saveToFireStore();
}

class FireStoreUserRepository implements UserRepository {
 @override
 void saveToFireStore() {
   // TODO: implement saveToFireStore
 }
}

class UserService {
 final UserRepository repository;

 UserService(this.repository);

 void saveUser() => repository.saveToFireStore();

 UserModel updateUser(UserModel user) {
   return UserModel(name: user.name, age: user.age, email: user.email);
 }
}

class UserModel {
 final String name;
 final int age;
 final String email;

 UserModel({required this.name, required this.age, required this.email});
}
```

Third Problem:

```
class Screen {
void navigate() {
print('Navigating to screen');
 }
}
class HomeScreen extends Screen {
 @override
 void navigate() {
   print('Navigating to home');
 }
}
class SettingsScreen extends Screen {
 @override
 void navigate() {
   throw Exception('Settings don\'t navigate this way!');
 }
}
class NavigationButton extends StatelessWidget {
 final Screen screen;
 NavigationButton(this.screen);
 @override
 Widget build(BuildContext context) {
   return ElevatedButton(
     onPressed: () => screen.navigate(),
     child: Text('Navigate'),
   );
 }
}
```

This code has a major flaw as seen in the snippet, so lets debug the possible cause. While HomeScreen navigates with the navigate method it does not clarify the navigation itself, it maybe a push and pop navigation it maybe a replacement navigation where they are all in navbar, these possibilities made the problem seem to be like violation of the Liskov Substitution Principle because a subclasses of Screen like SettingsScreen can't be substituted for the base Screen without risking runtime errors or unexpected behavior.

The answer for this question is (C).

For the solution we need a solution that wouldnt make us run into potential OCP violation, so put that in mind.

First step:
Create an abstract class Navigation with my desired destination.

```
abstract class Navigation {
String get destination;
 void navigate(BuildContext context);
}
```

This single behaviour contract will allow the subclass to perform one responsibility which is navigation.

Second step:
Add your concrete classes such as PushNavigation or ReplaceNavigation.

```
class PushNavigation implements Navigation {
 final Object builder;
 @override
 final String destination;

 PushNavigation({required this.builder, required this.destination});

 @override
 void navigate(BuildContext context) {
   print('Push to $destination');
   //push logic
 }
}
```

What this will do is allow me to use the desired navigation strategy with my screen destination and builder, also will fix any potential OCD violation because now it's open for extension any new strategy can be created.

Third step:
We remove Screen class that now we change our navigation method or keep it wont make a difference, but its better to work with cleaner code.

```
class HomeScreen{
 HomeScreen();
}

class SettingsScreen{
 SettingsScreen();
}
```

Fourth step:
We create a base widget component for the navigation button ton use to navigate to multiple destinations.

```
class NavigationButton extends StatelessWidget {
 final Navigation navigation;
 final String buttonText;

 const NavigationButton({
   Key? key,
   required this.navigation,
   required this.buttonText,
 }) : super(key: key);

 @override
 Widget build(BuildContext context) {
   return ElevatedButton(
     onPressed: () => navigation.navigate(context),
     child: Text(buttonText),
   );
 }
}
```

Last step:
Create the needed buttons to navigate to any screen, always remember open for extensions. By this we can create as many navigations as we want for the corresponding screen with the correct navigation strategy without breaking.

```
class HomeNavigationButton extends StatelessWidget {
 const HomeNavigationButton({Key? key}) : super(key: key);

 @override
 Widget build(BuildContext context) {
   return NavigationButton(
     navigation: ReplaceNavigation(
       builder: HomeScreen(),
       destination: 'Home Screen',
     ),
     buttonText: 'Go to Home',
   );
 }
}
```

Fourth problem:

```
abstract class WidgetController {
 void initState();
 void dispose();
 void handleAnimation();
 void handleNetwork();
}
class SimpleButtonController implements WidgetController {
 @override
 void initState() => print('Init button');
 @override
 void dispose() => print('Dispose button');
 @override
 void handleAnimation() => throw UnimplementedError('No animation in simple
button');
     @override
     void handleNetwork() => throw UnimplementedError('No network inbutton');
}
```

This problem could potentially be multiple violations, but we will pick the most suitable one which by fixing will fix the rest of the violations, so let's review the answers.

1- Open-Closed principle violation, this could be OCP violation but these are responsibilities we can add extensions to them rather than having to modify the existing code so no this is not OCP violation.

2- Breach of the Encapsulation principle, I think it's clear enough as to why this is not a breach, because it has no exposed details to it.

3- Interface Segregation Principle violation, this would likely be it, because its forcing implementing classes to provide methods irrelevant to their context.

To fix this first step:
We need to break each responisbility to a certain contract, so whenever i need one responsibility i wont need to implement all of them.

```
abstract class LifecycleController {
 void initState();
 void dispose();
}

abstract class AnimationHandler {
 void handleAnimation();
}

abstract class NetworkHandler {
 void handleNetwork();
}
```

Second step:
Break each button widget into the responsibility i need for example, if i wanted animation i would implement the animation contract to use its methods. But this wouldn't be convenient for the user, so instead we will approach a more complex solution. It is to add multiple buttons with various complexities as such.

```
class SimpleButtonController implements LifecycleController {
 @override
 void initState() => print('Init simple button');

 @override
 void dispose() => print('Dispose simple button');
}

class ComplexListController
   implements LifecycleController, AnimationHandler, NetworkHandler {
 @override
 void initState() {
   print('Init complex list');
   handleNetwork();
 }

 @override
 void handleNetwork() => print('Fetching data for the list...');

 @override
 void handleAnimation() => print('Performing list animations...');

 @override
 void dispose() => print('Dispose complex list');
}

class AnimatedIconController implements LifecycleController, AnimationHandler {
 @override
 void initState() => print('Init animated icon');

 @override
 void handleAnimation() => print('Running icon animation...');

 @override
 void dispose() => print('Dispose animated icon');
}
```

The answer for this question is **(C)**

Fifth Problem:

```
class LocalNotificationService {
 void send(String message) {
   print('Sending local notification: $message');
 }
}

class AppNotifier {
 final LocalNotificationService service = LocalNotificationService();

 void notifyUser(String message) {
   service.send(message);
 }
}
```

This is a clear violation of DIP because AppNotifier is dependent on LocalNotificationService as its not needed to review the questions.

The answer to this question is **(C)**.

As to why, it directly depends on and instantiating a concrete LocalNotificationService, resulting in tight coupling in for this feature.

How to solve it is fairly simple, first step:
Invert LocalNotificationsService into an Interface.

```
abstract class LocalNotificationService {
 void send(String message);
}
```

Second step:
Inject this interface into the AppNotifier.

```
class AppNotifier {
 final LocalNotificationService service;

 AppNotifier(this.service);

 void notifyUser(String message) {
   service.send(message);
 }
}
```

Now we don't have to worry about any DIP violations because we created an interface to segregate the class.

Note: All full solutions wil be provided in problems folder in the repository.
Repo: https://github.com/Garfend/week1_assignments.git