Examen Flutter

Partir 1 : Question théorique

1.1. L'architecture MVVM est constituée de 03 partie :

- Model : Il contient la logique métier et les données, il effectue les calculs et gère l'était des données. Dans une application calculatrice, cette couche réalise les opérations mathématiques, stocke les valeurs numérique et valide les calculs.

- View : Il est géré l'interface utilisateur, affiche les données fournir par le viewModel et transmet les actions de l'utilisateur. Dans une application de calcul, cette couche, affiche les boutons, affiche le résultat à l'écran et ne contient aucune logique de calcul.

- ViewModel : Il sert de pont avec la view et le Model. Il contient la logique de présentation et expose des données observables pour la view. Dans une application de calculatrice, ce couche reçoit les actions de l'utilisateur, appel le model pour effectuer le calcul et met a jour le résultat.

1.2. **ChangeNotifier** est une classe Flutter utilisée pour gérer **l'état** d'une application et notifier l'interface lorsqu'un changement se produit. Dans le ViewModel,, le ViewModel **hérite de ChangeNotifier et il** devient alors **observable par la View.** notifyListeners informe **tous les widgets qui écoutent** le ViewModel il déclenche automatiquement un **rebuild de la View.** Comme alternative à **ChangeNotifier** on peut avoir **ValueNotifier ou ValueListenableBuilder.**

1.3. On utilise **Consumer<CalculatorViewModel>** au lieu de **Provider.of<CalculatorViewModel>** dans certaines parties de la View parce que Consumer permet de reconstruire uniquement les widgets qui dépendent du ViewModel**,** ce qui améliore les performances de l'application. Avec **Consumer**, seul le widget concerné est mis à jour lorsque l'état change, tandis que **Provider.of** peut provoquer la reconstruction de tout le widget parent s'il est mal utilisé.

2.1. L'immutabilité de la classe **Calculation** signifie que ses données ne peuvent pas être modifiées après sa création.
Dans l'architecture **MVVM**, cela permet de **garantir la cohérence des données**, d'éviter les modifications involontaires du Model par la View ou le ViewModel, et de faciliter la gestion de l'état et le débogage de l'application.

2.2. Les variables dans **CalculatorViewModel** sont préfixées par un underscore (_display, _history, etc.) pour les rendre **privées au fichier**.
Cela respecte la règle de **visibilité (encapsulation) en Dart**, où le caractère _ indique qu'un attribut ou une méthode est **accessible uniquement dans le même fichier**.

2.3. La logique du `switch` sur `_pendingOperation` peut être refactorée en utilisant le **polymorphisme** ou le **pattern Strategy**.
Chaque opération (addition, soustraction, multiplication, division) serait définie dans une classe ou fonction séparée, ce qui permet d'ajouter de nouvelles opérations **sans modifier** la méthode `calculateResult()`, respectant ainsi le principe **Open/Closed (SOLID)**.

3.1. Comparaison de « setSate » et « notifyListeners() »

- **setState()** :
  - *Avantage* : simple à utiliser et rapide à mettre en place pour gérer un état local.
  - *Inconvénient* : peu scalable, le widget entier est reconstruit et la logique est fortement couplée à la View.
- **notifyListeners()** :
  - *Avantage* : meilleure séparation des responsabilités, l'état est géré dans le ViewModel et plusieurs widgets peuvent l'écouter.
  - *Inconvénient* : mise en place plus complexe et nécessite une gestion correcte des listeners.

3.2. Pour migrer vers une architecture **BLoC**, le ViewModel serait remplacé par un **Bloc ou Cubit**. La logique serait basée sur des **events** et des **states**, `ChangeNotifier` et `notifyListeners()` seraient supprimés, et l'état serait émis de façon réactive via des streams.

3.3. Dans **main.dart**, **Provider** permet d'injecter le ViewModel dans l'arbre de widgets afin qu'il soit accessible dans toute l'application. Si on omet le **ChangeNotifierProvider**, le ViewModel ne sera pas disponible dans le contexte et l'application générera une erreur lors de l'accès au ViewModel depuis la View.

Partie 2 : Correction de code

1. Problèmes architecturaux / techniques (5)
   - Les variables **display** et **history** ne sont pas privées.
   - Absence de **notifyListeners()** après modification de l'état.
   - La logique de calcul est directement basée sur une **String** (`display`).
   - La méthode `calculateResult()` ne gère qu'une seule opération (+).
   - Le ViewModel mélange **logique métier** et **logique de présentation**.

2. Pourquoi c'est une mauvaise pratique en MVVM
   - Les variables publiques violent le principe d'**encapsulation**.
   - Sans `notifyListeners()`, la View n'est pas informée des changements d'état.
   - Utiliser une String pour les calculs rend le code fragile et difficile à maintenir.
   - Le code n'est pas extensible et ne respecte pas le principe **Open/Closed**.
   - Le calcul devrait être délégué au **Model**, pas au ViewModel.

3. Code corrigé. (Respect de MVVM et bonne pratiques)

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

class CalculatorViewModel extends ChangeNotifier {
  String _display = '0';
  final List<String> _history = [];

  String get display => _display;
  List<String> get history => List.unmodifiable(_history);

  void inputNumber(String number) {
    if (_display == '0') {
      _display = number;
    } else {
      _display += number;
    }
    notifyListeners();
  }

  void calculateResult() {
    try {
      if (_display.contains('+')) {
        final parts = _display.split('+');
        final result =
            double.parse(parts[0]) + double.parse(parts[1]);
        _display = result.toString();
        _history.add(_display);
      }
    } catch (_) {
      _display = 'Erreur';
    }
    notifyListeners();
  }

  void clearAll() {
    _display = '0';
    _history.clear();
    notifyListeners();
  }
}
```

4. Test unitaire pour `calculateResult()` (cas d'erreur)

```
void main() {
  test('calculateResult gère une entrée invalide', () {
    final viewModel = CalculatorViewModel();

    viewModel.inputNumber('2+');
    viewModel.calculateResult();

    expect(viewModel.display, 'Erreur');
  });
}
```

Partir 3 : Développement

- **Option 1 : Ajouter la possibilité de calculer les pourcentages (bouton "%") :** Pour cette amélioration, j'ai apporter des modifications dans le code. J'ai modifier les fichier « calculator_screen.dart » et « calculator_viewmodel ».