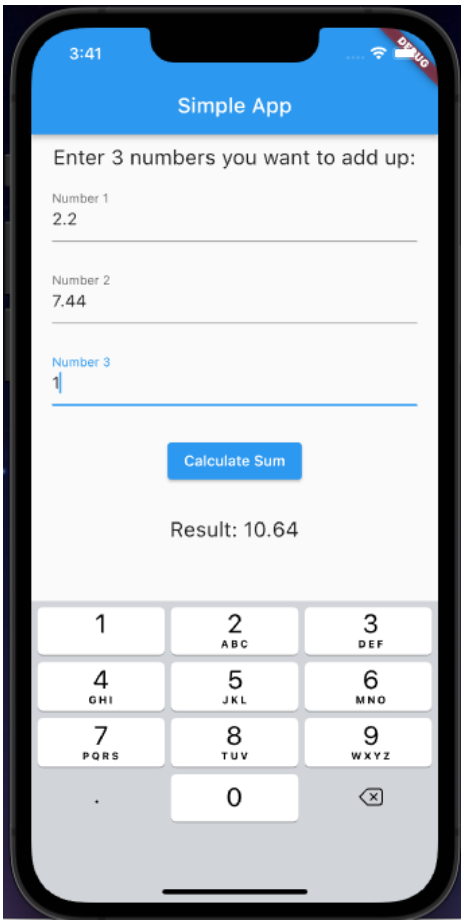
COMP7506 Individual Assignment

Chosen Topic: 1. Smartphone apps development using a cross platform IDE (Flutter)

Student: Jackie Chung Wing Tsoi
UID: 3035444891

PART I. INTRODUCTION

In this workshop, we will learn how to create a simple calculation app using Flutter, a popular open-source software development kit created by Google. In particular, we will create a single-page app for iOS where we add 3 numbers from user input and display their sum.



1.1 Introduction to Flutter

Flutter, released in 2017 by Google, is a relatively new software development kit that has become increasingly popular in recent years. It allows development of apps across Android, iOS, Linux, macOS, Windows, and many more from one single code base. It uses a programming language called Dart and compiles Dart code to native machine code for Android or iOS. Some of the advantages of Flutter include having the same UI and business logic across all platforms as its UI rendering does not require to be based on UI components specific to a platform, as well as a much faster code development time allowing fast app builds due to its “hot reload” feature. In addition, the Dart language and the widget architecture are easy to understand and learn.

1.2 Installation

To create a Flutter app for iOS, we will download 3 major tools: the Flutter SDK, Xcode, and Visual Studio Code.

1. Download the Flutter SDK from https://docs.flutter.dev/get-started/install/macos and follow the instructions to unzip the downloaded file, update the `flutter` tool to your path, and run `flutter doctor` on the command line to check if the setup is complete and working.
2. Download and install Xcode from the Apple App Store or run `xcode-select --install` on the command line.
3. Download and install Visual Studio Code from https://code.visualstudio.com/download. You can use any IDEs such as Android Studio, IntelliJ, or Emacs, but Visual Studio Code offers some of the best features when it comes to accessibility and performance for Flutter development, so we will use it for this workshop.

1.3 Simulator Setup

Go to your terminal, and we will create a new Flutter project using the Flutter CLI we previously installed. We will also boot up a simulator to visualize the app in real time.

1. For our case, we will set up the iOS simulator:
    a. After installing Xcode, run the following commands:
    ```
    sudo xcode-select --switch /Applications/Xcode.app/Contents/Developer
    sudo xcodebuild -runFirstLaunch
    ```
    b. Open the iOS simulator by running `open -a Simulator` on the terminal next.
    c. We will be using the iPhone 13 Pro simulator. Choose this setup by selecting "File" → "Open Simulator" → "iOS 15.2" → "iPhone 13 Pro". You should be able to get the following screen:

2. But if you want to develop the app for Android and use the Android emulator, follow the "Set up the Android emulator" section from https://docs.flutter.dev/get-started/install/macos.

PART II. DART & FLUTTER FUNDAMENTALS

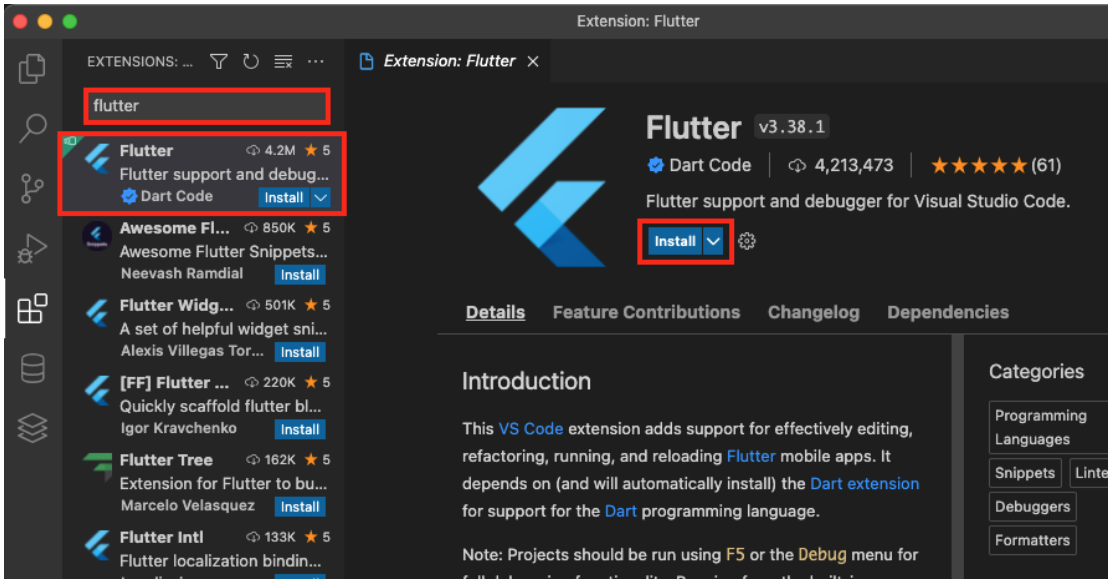
2.1 Dart Fundamentals

Dart is a programming language used for Flutter app development. It is an object-oriented language that is very similar to C-style languages. Dart allows the **`var`** keyword, which allows variable assignment for any variable type. For further information on Dart, check out https://dart.dev/guides.

2.2 Create Flutter App

Let's create a blank Flutter app using Visual Studio Code we previously installed.

1. Let's first install the Flutter extension on Visual Studio Code.
   a. Open Visual Studio Code.
   b. Click on "Code"→ "Preferences" → "Extensions", and search "Flutter". Select the "Flutter" extension by Dart Code, and click install.

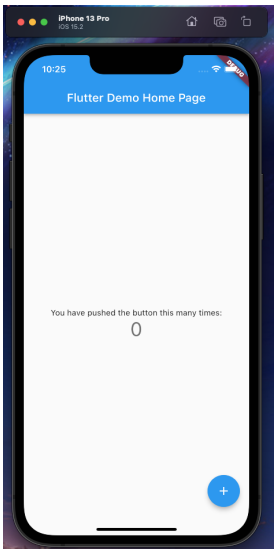


2. Create a new Flutter app and change your current directory to that app directory. For this workshop we will call it "my_simple_app".

```
flutter create my_simple_app
cd my_simple_app
```

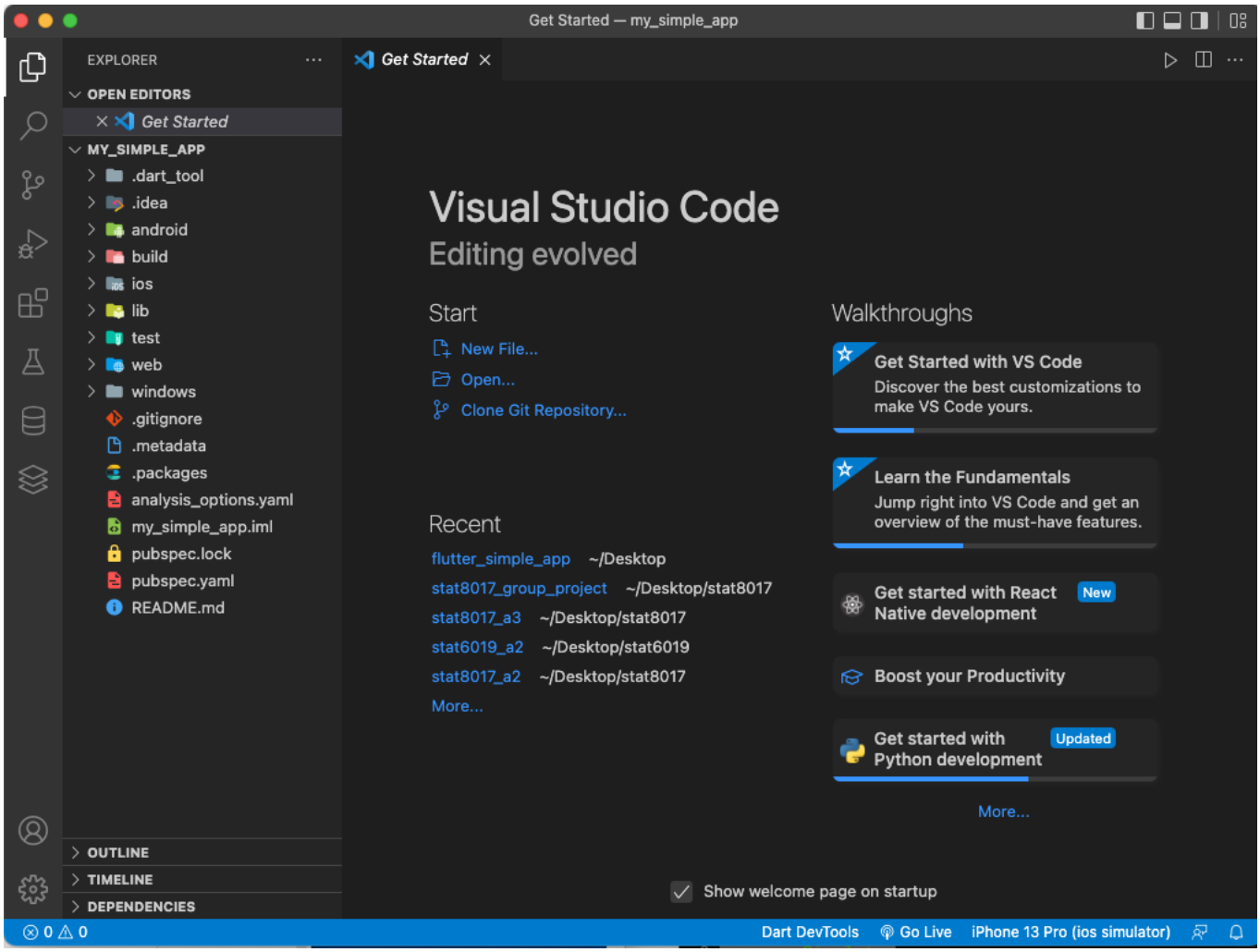
3. Run the app by entering flutter run on the command line. This command starts the app, boots up the simulator, and allows real-time updates on the simulator along with modification of the code. You should be able to get the following results on the command line (left), and the default app screen on the iOS simulator (right). This means that you have successfully run "my_simple_app".

```
(base) 19:19:54:Jackies-MacBook-Pro: ~/Desktop/my_simple_app$ flutter run
Launching lib/main.dart on iPhone 13 Pro in debug mode...
Running Xcode build...
 └─Compiling, linking and signing...                         8.7s
Xcode build done.                                           30.2s
Syncing files to device iPhone 13 Pro...                             307ms

Flutter run key commands.
r Hot reload. 🔥🔥🔥
R Hot restart.
h List all available interactive commands.
d Detach (terminate "flutter run" but leave application running).
c Clear the screen
q Quit (terminate the application on the device).

💪 Running with sound null safety 💪

An Observatory debugger and profiler on iPhone 13 Pro is available at:
http://127.0.0.1:49526/_FvXaRCAnuM=/
The Flutter DevTools debugger and profiler on iPhone 13 Pro is available at:
http://127.0.0.1:9100?uri=http://127.0.0.1:49526/_FvXaRCAnuM=/
```

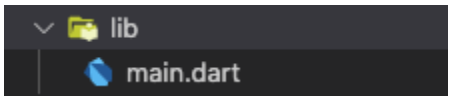


Do not abort this command as we will need it to provide real-time updates for the app. If we enter `r` into the command line shown above, the app would refresh according to any modifications we make in the code.

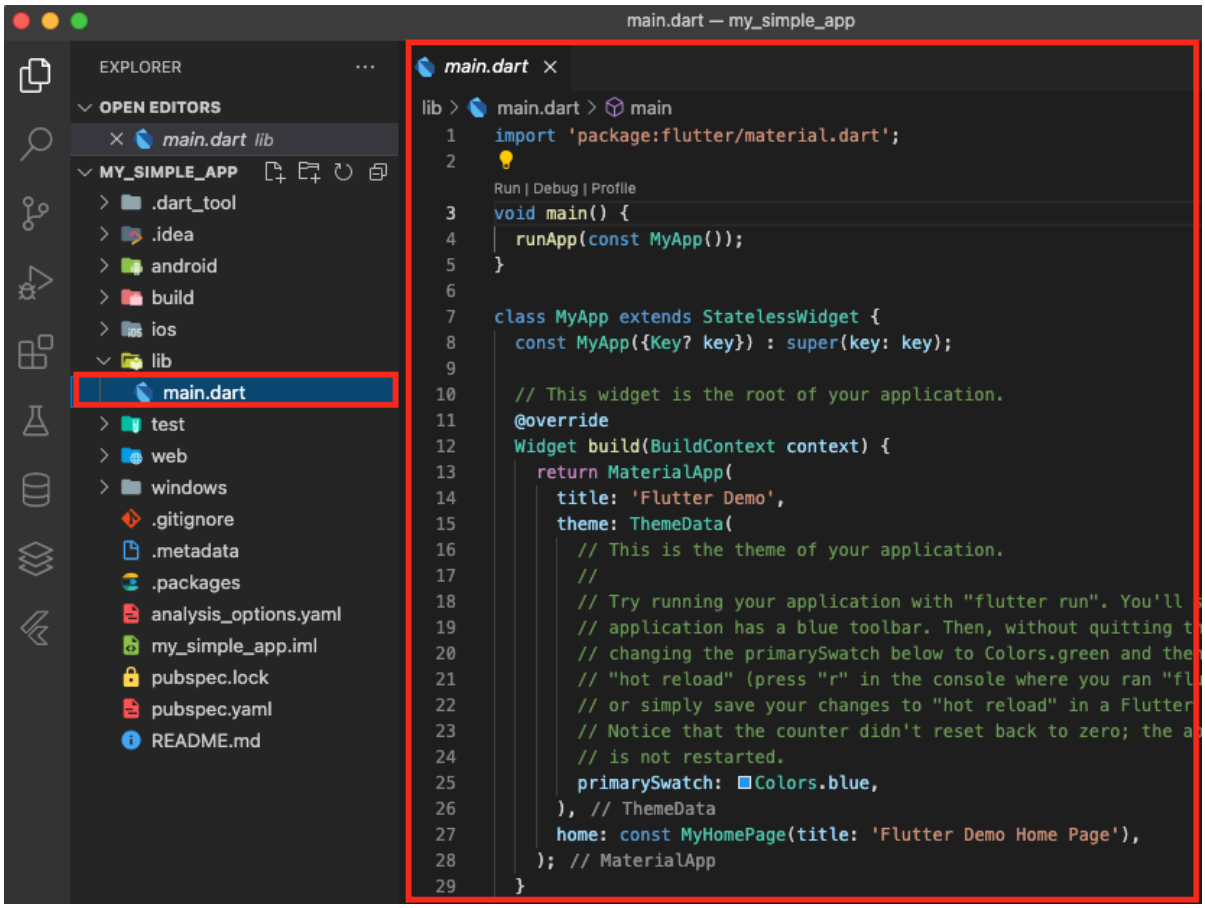
4. Go back to Visual Studio Code, select "Select" → "Open Folder…", select the "my_simple_app" directory, and click "Open". Now all the created files for your Flutter app will show on the left-hand side of Visual Studio Code, and you can start editing.



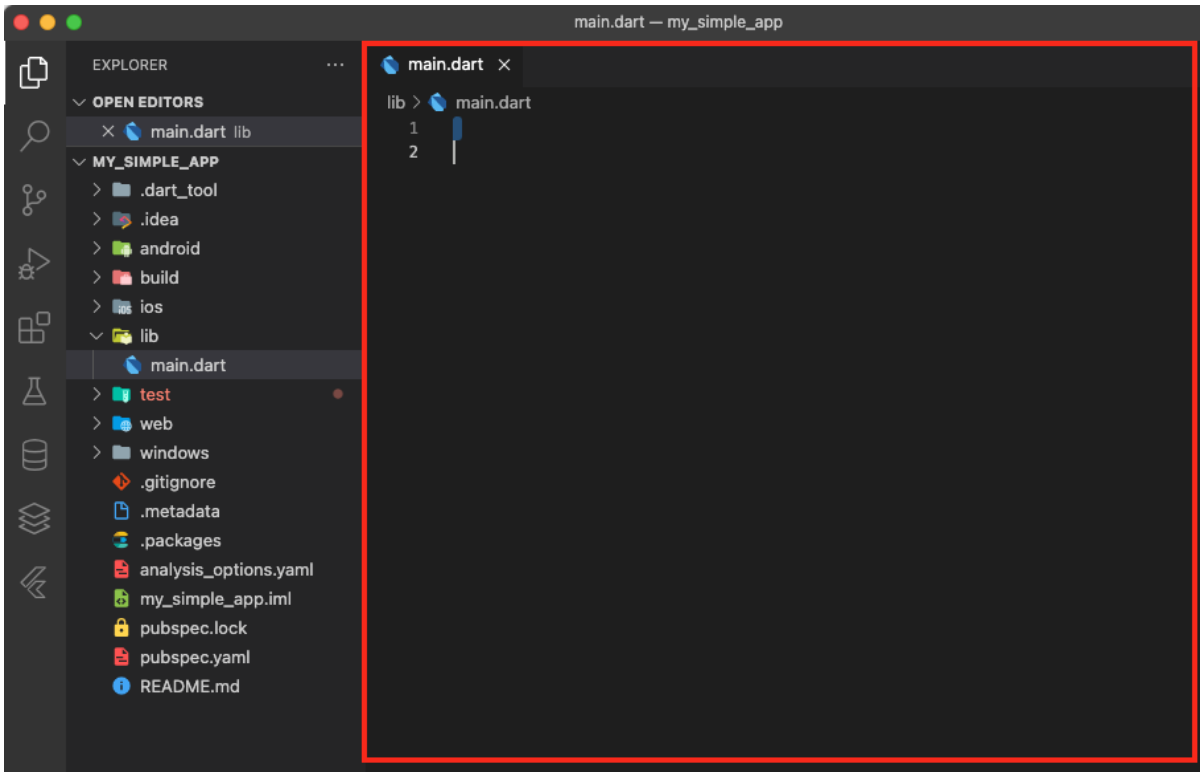
As we can see, there are many folders inside this app directory. The “android” folder is for adaptation with Android, and the iOS folder for iOS adaptation. and other files such as “.metadata” and “.packages” are for configuration purposes and we will not work on them for this workshop. The main code file is called “main.dart” which is stored under the folder “lib”. This is the file that controls the majority of the app we are building. Note that for more complicated app projects, we will have multiple .dart files and they will all be stored under “lib”. In this tutorial, we will only need the “main.dart” file as we are building a very simple app.



5. When we created our app, Flutter already initialized the app to have some default features (shown in the simulator image in step 4). Click on “main.dart” under “lib”, and you will see that the file is already filled with the default code.



But since we want to create our own simple calculation app, we now delete all the code in the “main.dart” file and start from scratch. Now we are ready to start coding our own app.



2.3 Architecture of Flutter Apps

A Flutter app consists of widgets, where we stack together or combine multiple widgets to make up the UI and implement features for our app. The idea is similar to lego blocks.

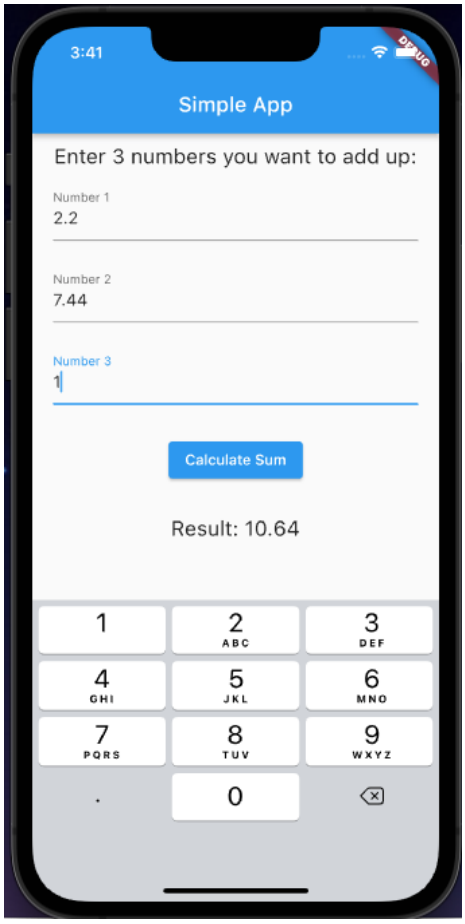
1. To create a view of our app, we first import the provided Flutter package "flutter/material.dart", and call "runApp" inside the "main" function which is the section Dart automatically executes. Inside "runApp", we input "MySimpleApp", which is a class we will define for the widget that is our app.
2. We declare a class called "MySimpleApp" that extends the "StatefulWidget", and override the "createState" method to return "MySimpleAppState", which is a class that does not reset external data every time we restart the app.
3. We then declare the "MySimpleAppState" class.

Copy and paste the code below inside the "main.dart" file.

```
import 'package:flutter/material.dart'; // step 1

void main() => runApp(MySimpleApp()); // step 1

class MySimpleApp extends StatefulWidget { // step 2
 @override
 State<StatefulWidget> createState() {
   return MySimpleAppState();
 }
}

class MySimpleAppState extends State<MySimpleApp> { // step 3
 // TODO
}
```

PART III. BUILDING OUR SIMPLE NUMBER ADDITION APP

3.1 Create Basic Layout



We start building our simple number addition app by setting up the basic structure. Recall that every Flutter app is made up of widgets, so we first design the various widgets that we want to include in our app. We will start from building small groups of widgets and then put them altogether as one big widget in the end. From the app design, we can break it into several major components and we can build these widgets provided by Flutter:

1. For the main body of the app, we want to have:
    a. A prompt that asks the user to enter 3 numbers → We create a "Text" widget;
    b. 3 input fields for the user to input a number each → We create 3 custom widgets called "NumberInput" which we will define later;
    c. A button to calculate the sum of the numbers → We create an "ElevatedButton" widget that is wrapped in a "Padding" widget. The "Padding" widget allows us to add paddings to this button for UI formatting;
    d. A text to display the result → We create a "Text" widget that is wrapped in a "Padding" widget
2. We want these widgets to be displayed row by row, so in total we will have 1 column of 6 rows of widgets → We put each of the above 6 widgets into a list, and create a "Column" widget that takes this list as its children. We then wrap this "Column" widget inside a "Container" widget which allows us to adjust the padding and spaces.
3. We want a title to display "Simple App" and a body to include the remaining features. This can be achieved by calling the provided "MaterialApp" class/widget, and using the provided "Scaffold" widget as its child widget. Inside the "Scaffold" widget, we then specify the child widget to be the "Container" widget we created in step 2. Finally, we override the "build" method in our "MySimpleAppState" class and return the "MaterialApp" widget we created. Copy and paste the following code inside the "MySimpleAppState" class:

```
 @override
 Widget build(BuildContext context) {
   return MaterialApp( // widget 3
       home: Scaffold(
           appBar: AppBar(
             title: Text('Simple App'),
           ),
           body: Container( // widget 2
             margin: EdgeInsets.all(10), // add margin around Column widget
             child: Column(
               children: [
                 Text('Enter 3 numbers you want to add up:', // widget 1a
                     style: TextStyle(
                       fontSize: 20, // set font size
                     )),
                 NumberInput('Number 1', inputController1), // widget 1b-1
                 NumberInput('Number 2', inputController2), // widget 1b-2
                 NumberInput('Number 3', inputController3), // widget 1b-3
                 Padding( // widget 1c
                           padding: EdgeInsets.all(20), // add padding around
ElevatedButton
                   child: ElevatedButton(
                       child: Text('Calculate Sum'),
                       onPressed: () => calculateSum(n1, n2, n3)),
                 ),
                 Padding( // widget 1d
                   padding: EdgeInsets.all(10), // add padding around Text
                   child: Text('Result: ' + sum.toString(),
                       style: TextStyle(
                         fontSize: 20,
                       )),
                 )
               ],
             ),
           )));
 }
```

3.2 Get User Input

We now define the “NumberInput” custom widget that we specified earlier. Since in our app we will take 3 user inputs, rather than repeating the same code three times, it is better practice for us to define a custom class here.

1. We create a new class called “NumberInput” that extends the “StatefulWidget” class. This widget is stateful because the user interacts with the widget by inputting a number.
2. Inside this class, we want to include a “TextField” class that takes user input for a number. And then we wrap it as a child in a “Container” class to allow for additional UI adjustments such as padding.
3. In the TextField class, we also specify a “controller”, which allows us to record the user input data and pass it back to our main widget.

Copy and paste the following code outside of the “MySimpleAppState” class:

```
class NumberInput extends StatefulWidget { // step 1
 final String numberInput;
 final TextEditingController inputController;

 NumberInput(this.numberInput, this.inputController);

 @override
 State<StatefulWidget> createState() {
   return NumberInputState();
 }
}

class NumberInputState extends State<NumberInput> { // step 1

 @override
 Widget build(BuildContext context) {
   return Container( // widget 2-1
       margin: EdgeInsets.all(10), // add margin around TextField
       child: TextField( // widget 2-2
           controller: widget.inputController, // step 3
           decoration: InputDecoration(
             hintText: widget.numberInput,
             labelText: widget.numberInput,
           ),
           keyboardType: TextInputType.numberWithOptions(
             decimal: true,
           )));
 }
}
```

3.3 Calculate & Output Result

Now that we have the user input widget complete, we can start calculating the sum of the 3 numbers being inputted.

1. We first define 3 “TextEditingController” widgets inside the “MySimpleAppState” class which contain the values from the user input. We also define 3 **`var`** variables “n1”, “n2”, “n3” inside the class to store the inputted values, and a **`var`** variable “sum” to store the summed value.
2. We define a **`void`** method called “calculateSum” inside the “MySimpleAppState” class. Inside this function, we first parse the inputted values. If the input is not a valid **`double`** or **`int`** value, then the “tryParse” function will return a **`null`** value. Then, we create a “setState” method that updates the value of the “sum” variable according to the inputted values during which the app is running.

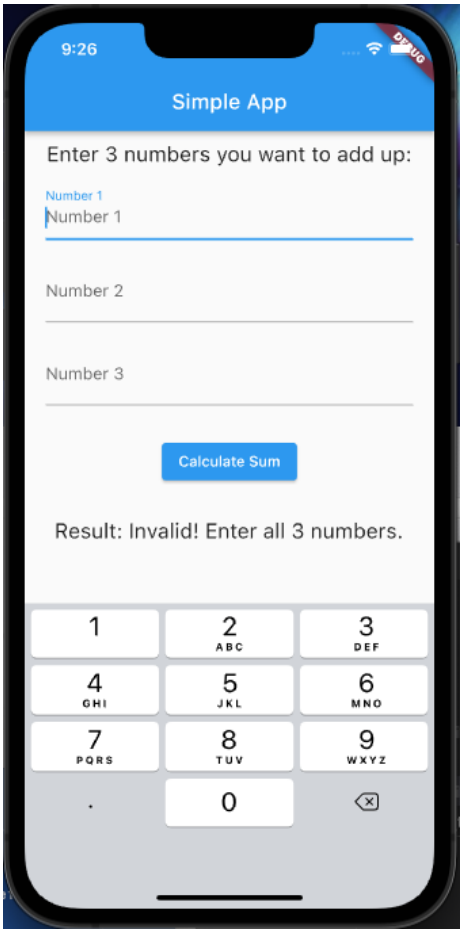
Copy and paste the following code at the beginning of the “MySimpleState” class and before **`@override`**:

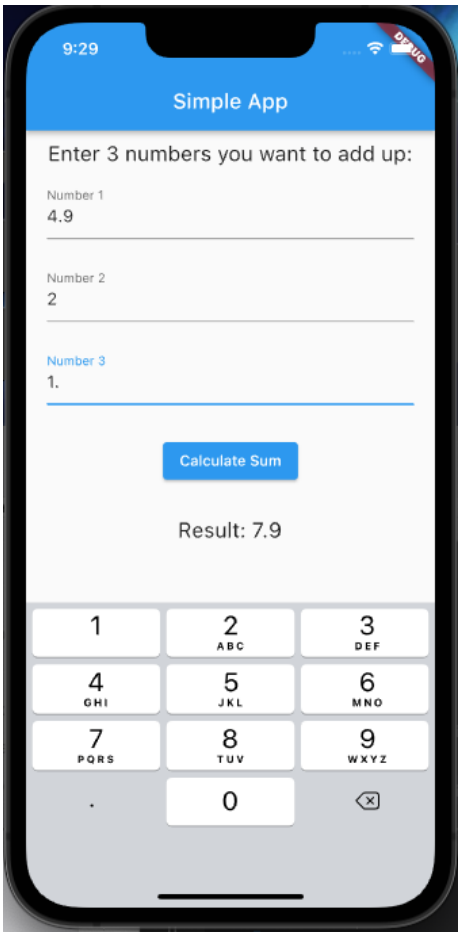
```
var n1, n2, n3; // step 1
var sum; // step 1
TextEditingController inputController1 = TextEditingController(); // step 1
TextEditingController inputController2 = TextEditingController(); // step 1
TextEditingController inputController3 = TextEditingController(); // step 1

void calculateSum(var n1, var n2, var n3) { // step 2
  n1 = double.tryParse(inputController1.text);
  n2 = double.tryParse(inputController2.text);
  n3 = double.tryParse(inputController3.text);
```

```
    setState(() {
      if (n1 == null || n2 == null || n3 == null) {
        sum = 'Invalid! Enter all 3 numbers.';
      } else {
        sum = n1 + n2 + n3;
      }
    });
  }
```

3.4 Run App

Return to the terminal window, and press `r` on the keyboard to refresh the simulator. The app should run successfully and show the desired functionalities. If the app does not successfully run, press `q` on the command line to quit the command and run `flutter run` on the command line again.





PART IV. FURTHER READING

Check out the official documentation https://docs.flutter.dev/ for more advanced features of Flutter.

Happy learning!