**Dart Programming Language**

If you’ve done any development in Java, JavaScript or Swift, then Dart will be simple to learn.   
Although Dart forms the foundation of Flutter, Dart can be used for other use-cases besides just Flutter. If you wish to know more on Dart’s other applications check out the documentation at <https://dart.dev/guides> For our purposes, we will be learning the basics of Dart to help us develop Flutter Applications.

The Dart language is statically typed;  it uses static type checking to ensure that a variable’s value always matches the variable’s data type. It also has type inference. In the newest version of Dart, Null Safety was introduced and will also be default when working with Flutter moving forward. You can think of this feature like optionals in Swift. Finally, like Java we need to end each line of executed code with a semi-colon character.

**Documentation**

In order to learn any programming language, it is important to read documentation. We will be spending a lot of time reading and referring to the [Dart Documentation](https://dart.dev/guides). Most of the concepts we will learn will be from the documentation. And we will be directing you to it so you can get a deeper understanding of certain topics. These upcoming modules are a supplement to the documentation in order to make it concise and give you the necessary skills to build Flutter applications as quickly as possible.

# DartPad

In order to start learning the fundamentals of the Dart programming language, we need to start coding and practicing. We can use a web based text editor called the DartPad.  <https://dartpad.dev/>

Something unique about Dart is that it is both a compiled and interpreted language. It’s complicated on how exactly this works. If you wish to know specifics please search on the web as there are many resources on this topic. With that said, we can use the tool DartPad in order to run a Dart program and learn the syntax and basics of the language.

# Expectations

In the following sections there will be code examples. We expect you to follow along with them as well as write the examples on your own DartPad. Avoid copy and pasting the examples unless necessary. We encourage you to type it out, it will help you get a better understanding of the code and practice.

# Main Function

What is the void main() function that we see when we first make a Dart pad? Essentially this is the entry point of our program, all the code inside this function will get executed when we press run. Go ahead and run the sample code, and try to print(“hello world”).

**//**Is how we can add comments.  Remember comments don’t get executed and the compiler ignores them.

void main() { // main entry point

for (int i = 0; i < 5; i++) {

print('hello ${i + 1}');

}

}

# Variables and Types

Dart is statically typed; we declare variables by doing the following:

**[Variable Type] [Variable Name] = [Value] ;**

Dart also uses type inference so we don’t always need to declare the type.

void main() {

// Immutable note there is a small difference

final String firstName = "Coding";

const String lastName = "Dojo";

// Mutable

var number = 55;

// Also mutable

double height = 5.5;

var isCold = false;

}

Using the keywords **final** or **const** will make the variable **immutable**. There is a slight difference between using these keywords.

**Note:**Although a **final** object cannot be modified, its fields can be changed. In comparison,to a **const** object; its fields cannot be changed: **they’re immutable.**

Using the **var** keyword or declaring it’s type like the height variable shown above will make the variable **mutable**

**Note:** when using **var** we don’t declare the type because it uses **type inference.**

### Built-in types in Dart include :

* [Numbers](https://dart.dev/guides/language/language-tour#numbers) (int, double)
* [Strings](https://dart.dev/guides/language/language-tour#strings) (String)
* [Booleans](https://dart.dev/guides/language/language-tour#booleans) (bool)
* [Lists](https://dart.dev/guides/language/language-tour#lists) (List, also known as *arrays*)
* [Sets](https://dart.dev/guides/language/language-tour#sets) (Set)
* [Maps](https://dart.dev/guides/language/language-tour#maps) (Map)
* [Runes](https://dart.dev/guides/language/language-tour#characters) (Runes; often replaced by the characters API)
* [Symbols](https://dart.dev/guides/language/language-tour#symbols) (Symbol)
* The value null (Null)

# String Interpolation

String interpolation is simple to use: we just put a dollar sign with the variable name within a String. Strings can be made with either single or double quotes.

void main() {

String firstName = "Coding";

String lastName = "Dojo";

print("My full name is $firstName $lastName");

}

# Dart Functions

Functions are powerful and important in any language. So how do we make functions in Dart?

The syntax starts with the return type first then the name of the function followed by the parameter type and names separated with a comma. And finally the curly braces where we write our code.

void main() {

// Inside out main function we are calling

// our newly created function and when you run this it will print

helloWorld()

}

// Here we define our function but declaring the return type

// in this case it's a void which means it returns nothing

// then we give the name of our function followed by parentheses

// finally we put curly braces and the code we want our function to execute

void helloWorld(){

print("Hello world to console")

}

# Return Type Vs. Void

We need to ensure that we follow the rules when writing functions in the Dart language so that they compile correctly.  A function that does not need to return data has a return type of void.  But, for any function that needs to return data, we need to specify the return type.  In the example below, the sayName function has a void return type since we aren’t returning any values out of the function, we’re merely printing a string. However, in the returnName function, we are returning a String, therefore our return type needs to also be String.

void main() {

//calling our function

sayName("Joe");

var name = returnName("John");

}

//declaring our functions

void sayName(String name){

print("My name is $name");

}

String returnName(String name){

return "My name is $name";

}

# Named, Default, and Optional Parameters

By default, functions use positional arguments that are not named. This can get confusing when a function takes in a lot of parameters.

We can fix this by making **named parameters.** The syntax is shown below. We just need to put our parameters inside curly braces and this converts the name parameter into a named parameter (pun intended).

Since the introduction of null safety we also have to give this parameter a default value since named parameters cannot be null unless specified.

**Note:** to make a variable a default null we must put a ? in front of the type example: **String? Name**and this way we wouldn't have to give a default value since the default value would be null. More on this here : [Null Safety](https://flutter.dev/docs/null-safety)

In this example we made our returnName function have a named parameter with a default string value if none is provided.

We can also make **optional parameters** by putting the  parameters inside square brackets.

For example, in our returnHeight function we made the inches parameter optional with a default value of 0. And when we called the function we omitted the second parameter because it’s optional now.

**Note:**we cannot use square brackets inside of curly braces.

void main() {

var name = returnName(name: "John");

var height = returnHeight(5);

}

//declaring our function with named parameter

//and default value

String returnName({String name = "default name"}){

return "My name is $name";

}

//No named parameter and optional second parameter

//if no value is given it will use default value

String returnHeight(int feet, [int inches = 0]){

return "I am $feet feet and $inches inches tall";

}

# Arrow Operator =>

Sometimes when functions are short and they just return one statement, we can use the arrow operator instead of curly braces to declare a function’s body. This makes our code much shorter and concise. Let’s see it in practice:

void main() {

var name = returnName(name: "John");

var name2 = returnName2(name: "John");

}

String returnName({String name = "default name"}){

return "My name is $name";

}

String returnName2({String name = "default name"}) =>

"My name is $name";

Here these two functions do the exact same thing; the only difference is how we defined them. We omitted the curly braces and the return statement. We then added the arrow operator. This is useful when our function only has one statement.

# Intro to Dart Classes

Creating your own type in Dart is essentially what we are when making a custom class. In Dart every class is a descendent of the base **Object Class,**more in this later when we talk about Inheritance. Classes can hold data, logic, and the functionality that is relevant for that data. This is very powerful when we learn about widgets in Flutter.

Remember a class is just an object or in other words a custom type that we create. We define its initializers and methods. We also have the power to inherit from other classes which we will get to.

## Defining a Class with a constructor

For now let’s see how we can define our own class and initialize them. Please try to type this code out in your DartPad. Avoid copy and pasting unless necessary.

void main() {

var person = SimplePerson();

person.name = "Rod";

print(person.name);

print(person.age);

//Note we cannot omit the arguments or we will get an error

final person2 = ComplexPerson("Joe", 5);

print(person2.name);

print(person2.age);

}

//This SimplePerson class can be initialized with no parameters

//and we gave the variables the ability to be null by adding the ?

//by using no constructor the instance variables are not required

class SimplePerson{

String? name;

int? age;

}

//This ComplexPerson class has to be initialized with both positional instance variables

//we do this by using the constructor with the short cut

//this.(then your variable name)

class ComplexPerson{

String? name;

int? age;

ComplexPerson(this.name, this.age);

}

## Named Constructor

There are many ways to define a class depending on your needs. We can make instance variables optional as shown in the previous example. We can make instance variables have default values or make them default to null (which is null safety). We can also make named constructors. Named constructors are a common way of defining a class in Dart, because this makes declaring our classes more clear to other programmers reading our code. Let’s see an example.

void main() {

// named constructor

var person = Person(name: "Joey", age: 35);

print(person.name);

}

// By adding curly braces in our constructor we make the variables

// have to be named when initializing the class

// this is the same syntax to making a function have named parameters

class Person{

String? name;

int? age;

Person({this.name, this.age});

}

## Required instance variables

Sometimes we want our classes to have certain instance variables always be required when the class is being initialized. A good example of this would be when a user signs up we want their email address to be required when making our User class. Let’s see an example:

void main() {

// Since age is optional and can be null we don't have to initialize it

// however our name and email is required

var firstUser = User(name: "Rod", email: "rod@codingdojo.com");

print(firstUser.age);

}

// Here our name and email instance variables are required and we do this

// by using the "required" keyword in our constructor.

// Note: we don't have to mark these variables with ? because Dart

// knows that these variables are required and will never be null.

// And since age is optional and can be null we mark it with a ?

class User{

String name;

String email;

int? age;

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

}

## Instance Methods

Instance methods are just functions that live inside the scope of our class. We define them the same way as functions except they are defined inside our class and can be called by an instance of our class object using dot notation. Let’s see an example and please copy and paste this code into your Dart pad to run it and see the console result.

void main() {

var myCat = Cat(name: "Bubbles", age: 3);

// We call our method by using the dot notation

myCat.meow();

}

class Cat{

String name;

int age;

Cat({required this.name, required this.age});

// This is an instance method it lives inside our class

// it even has access to our variables!! so cool

void meow(){

print("meow! My name is $name");

}

}

# Inheritance

What does inheritance look like in Dart? Assuming you’ve heard about inheritance, basically it’s when we define a new class by subclassing a parent class. This allows us to use some of the functionality of the parent class while also giving us the ability to define new functionality in the child class. If inheritance is a new concept to you please take some time to read this [**Article**](https://stackify.com/oop-concept-inheritance/).

Sometimes called extending a Parent class inheritance looks different in each programming language. If you have a background in Java, it has very similar syntax in Dart. If you have a background in other programming languages the syntax might look new to you but the concept is the same.

We start by having a parent class and defining a new class with the keyword **extends.**

A good example is if we have a base person class that can have different occupations. Based on the occupation we can make a subclass that extends the person class. Lets see a code example

void main() {

var person = Person(name: "Kim");

person.sayName();

var student = Student(gpa: 3.5);

student.sayName();

student.sayGPA();

}

class Person{

String? name;

Person({this.name});

void sayName(){

print("My name is $name");

}

}

class Student extends Person{

double? gpa;

Student({this.gpa});

void sayGPA(){

print("My gpa is $gpa");

}

}

Go ahead and run this code on the Dart pad. It should be no surprise that we can run our new method sayGPA() on our student class since we defined it. The surprising thing for new programmers is that we can also call our sayName() method on our student class object since we **extended**our Person class. Notice however that our name is Null when we run our method.

How can we also inherit the name variable from the person class? In order to do this we need to learn about the **Super** constructor.

## Super Constructor

The **super**keyword let’s us reference the parent class which we are inheriting from. In our example we will be using the **super** keyword to reference the Person class from our Student class in order to inherit the name instance variable in our constructor. This way when we construct a Student object we can pass in a name as well as a gpa. Let’s see the difference:

class Student extends Person{

double? gpa;

// Here we give our constructor the variable name which is an optional/null String? Then we use colons and call our super class and pass over the name parameter. If this syntax is confusing, don't worry it takes some time to get used to.

Student({String? name, this.gpa}): super(name: name);

void sayGPA(){

print("My gpa is $gpa");

}

}

With our newly defined Student class that now also inherits our name variable we can create a new instance of student and give it a name. Now when we run our sayName() method, the name will no longer be null.

void main() {

// We can now give our constructor a name variable

var student = Student(name: "Joey", gpa: 3.5);

student.sayName();

student.sayGPA();

}

## Abstract Classes (Optional) aka Interfaces

So what are abstract classes? Why are they useful? Abstract classes are classes that let us define properties and methods without specifying how they are implemented. This is useful because we can use these abstract classes to make our regular classes more powerful. These abstract classes let us define what are called interfaces using the **implements**keyword. If you are familiar with Java it is very similar in syntax. If you’re familiar with iOS and swift this may be new but it's the same concept as protocols in Swift. Lets see an example.

Note: Abstract classes can’t be instantiated. Meaning we can’t make an instance of an abstract class.

//we need this to use pi

import "dart:math";

void main() {

var square = Square(side: 10.0);

print(square.area());

var circle = Circle(radius: 5.0);

print(circle.area());

//In here we can pass both a square and circle

//since they both implement the Shape class

//and this works the same as above.

printArea(square);

printArea(circle);

}

//Here we make a function that takes the abstract class

//as a parameter and remember any class that implements

//this interface has the .area() method

void printArea(Shape shape){

print(shape.area());

}

abstract class Shape{

//Notice we are not implementing the function

//we are only defining the function

double area();

}

class Square implements Shape{

double side;

Square({this.side = 0.0});

double area()=> side \* side;

}

class Circle implements Shape{

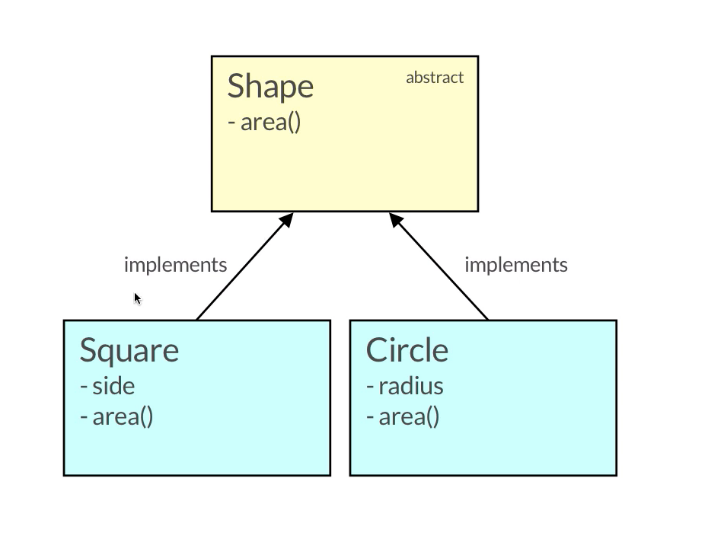
double radius;

Circle({this.radius = 0.0});

double area()=> radius \* radius \* pi;

}

Here we are creating an abstract class called Shape that defines an area function. However we don't implement the function because abstract classes help us program interfaces not implementations. Meaning when we make a class that implements shape we need to define the area() method. Notice both Circle and Square implement Shape and they each solve their own area() differently. Yet we can still call the area() method the same way and expect them to work. This concept is very powerful and can help us write better code. Below is a diagram of how the abstract class hierarchy is.



# Collection Types in Dart

Dart comes with several collection types.

**List:** Ordered collection [ ]

**Sets:**Unordered collection Set()

**Maps:**Unordered collection of key, value pairs {key: value} same as dictionaries in Swift

Let’s see how we can define these types and some of the methods built into them. **Note:**There are many ways to define these collections types and make them more restrictive on the types that can be stored inside them; we will learn more about this in the type annotation section.

### List Declaration

void main() {

var oddNumbers = [1,3,5,7,9,11];

print(oddNumbers);

oddNumbers.add(13);

print(oddNumbers);

oddNumbers.addAll([15,17,19,21]);

print(oddNumbers);

}

We can declare a list but simply insert objects inside [ ] separated by a comma as shown above. We can append or add to the list by calling the .add() method. We can also add more than just one number by calling the .addAll() method which takes an object that implements the iterable interface/abstract class. List implements this interface which is why we can just pass in another list. Go ahead and run this code on the DartPad and run it.

List or arrays are accessible by the index. Same as any other programming language, remember indexes start with 0 meaning the first item in the array/list is at index 0.

### Maps Declaration

void main() {

//Empty Map initialization

var nameEmails = Map();

nameEmails["James"] = "james@gmail.com";

print(nameEmails);

//We can also declare Maps this way

var nameAge = {

"Rod" : 22,

"Bobby" : 25

};

print(nameAge);

print(nameAge['Rod']);

}

### Sets Declaration

void main() {

//Empty set

var numbers = Set();

numbers.add(5);

numbers.add(6);

numbers.add(7);

print(numbers);

//We can also declare sets like this

var evenNumbers = {2,4,6,8};

print(evenNumbers);

}

# Type Annotations

What exactly are type annotations? Type annotations in Dart are just another way of declaring a type but it also allows us to be more specific. This leads to safer, and more robust code. It’s important we know the syntax because without prior knowledge some code may look confusing to us. Let’s see an example

Another way to declare a list of numbers looks like this:

void main() {

List<int> oddNumbers = List<int>();

oddNumbers.add(5);

print(oddNumbers);

//If you try to do this we will get an error

//this is because we said this list can only

//contain numbers

oddNumbers.add("5");

}

Notice how now we specified that our list can only accept **ints**. By using type annotations we make sure that this list can only ever contain ints. In the previous example where we didn’t use type annotations, we only used the **var** keyword, our list could’ve contained any type. But thanks to type annotations we can be more specific. This leads to safer code and gives us more power.

# Control Flow in Dart

We’re almost done with Dart, and ready to move onto using Flutter to build cross-platform applications. One last thing we should learn about is control flow. How do we make if-else statements in Dart? What is the ternary operator? And finally, how do we use while loops and for loops?

## If-Else Statements

In Dart we always need to put our conditions or statements inside parentheses after our **if** keyword. For example:

**If (statement or condition) {**

**code**

**}**

Let’s see this in Dart. We’ll start by making a function that prints even or odd based on the number we pass it. It will do this using the modulo operator.

void main() {

printOddOrEven(num: 4);

printOddOrEven(num: 5);

}

void printOddOrEven({required int num}){

if (num % 2 == 0){

print("Even");

}else{

print("Odd");

}

}

Every-time we make a conditional statement we have to put it inside parentheses, this is the syntax for Dart. Pretty simple right?

**Note:** It’s common to forget to put these statements inside parentheses especially coming from other programming languages that don’t do this. So just keep this in mind when using control flow.

## Ternary Operator

Now let’s learn about the ternary operator which is the **?** operator. We’ll remake the example shown about but instead of using an if-else statement we’ll use the ternary operator.

void main() {

printOddOrEven(num: 4);

printOddOrEven(num: 5);

}

void printOddOrEven({required int num}){

var result = (num % 2 == 0 ? "Even" : "Odd");

print(result);

}

What we are doing here is making a variable result that will save either the Even string or Odd string based on the condition. The way the operator works is as follows

**(condition ? expression True : expression False)**

The condition is first followed by the **?** then the first part will evaluate if the condition and the second part will evaluate if the condition is false.

## For-loops

Now let’s learn about for loops and while loops. We’ll do this by iterating through a list first with a for loop and then with a while loop.

void main() {

List<int> numbers = [1,2,3,4,5,6];

//Similar to Java

for (var i = 0; i < numbers.length; i++){

print(numbers[i]);

}

//If our type implements the iterable interface we

//can do this also or use the forEach method

for (var num in numbers){

print(num);

}

//For each takes a void function as a parameter

numbers.forEach(print);

}

## While-loops

Let’s iterate through a list using a while-loops. If you are unfamiliar with while-loops, they differ from for-loops in that they will keep looping until a boolean condition is met. Let’s see an example.

void main() {

List<int> numbers = [1,2,3,4,5,6];

var i = 0;

while(i<numbers.length) {

        print(numbers[i]);

        i++;

    }

}

* Public APIs

<https://github.com/public-apis/public-apis>

# Intro to Widgets

In Flutter EVERYTHING IS A WIDGET. This is an important concept to keep in mind. So what exactly are widgets? Widgets are objects that describe what their view should look like given their current configuration and state. When a widget’s state changes, the widget rebuilds itself and reflects the changes in the View.

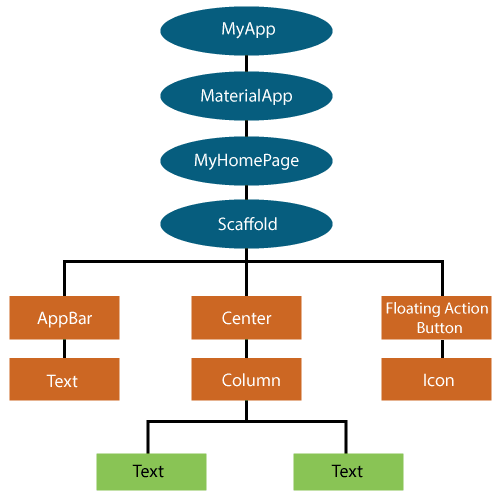
For example the MyApp object in our starter project is the top-level widget that essentially holds all of our applications. It also extends or in other words inherits from the Stateless widget.

Stateless and Stateful widgets are the base widgets and building blocks we use in order to build custom Widgets. For now we will only be using prebuilt widgets that the Flutter team has already provided in order to build out our applications.

In this course we won’t be covering building custom widgets in depth, to learn more about custom widgets here’s a nice article about it that goes more in depth. <https://www.raywenderlich.com/10126984-creating-reusable-custom-widgets-in-flutter>

There are many different kinds of widgets that each perform different functions. For example there are widgets for layout, widgets for UI design, and even widgets for buttons and text. These are just a few of the example to learn more about the built in Flutter widgets visit the widgets catalog: <https://flutter.dev/docs/development/ui/widgets>

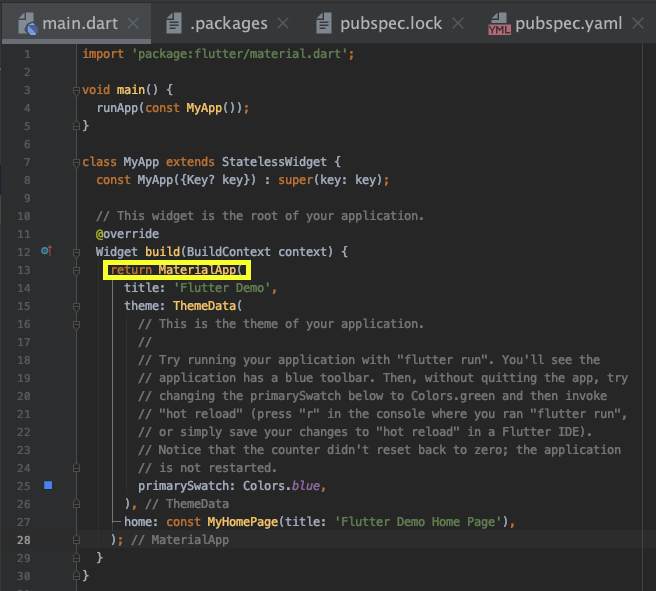
Let’s start by learning about the Widgets our starter project uses in order to get a deeper understanding of how the application works. Here is a diagram of the widget tree our app is currently implementing.



Source : <https://www.javatpoint.com/flutter-widgets>

## Material App Widget

What is the [Material App widget](https://api.flutter.dev/flutter/material/MaterialApp-class.html) ?

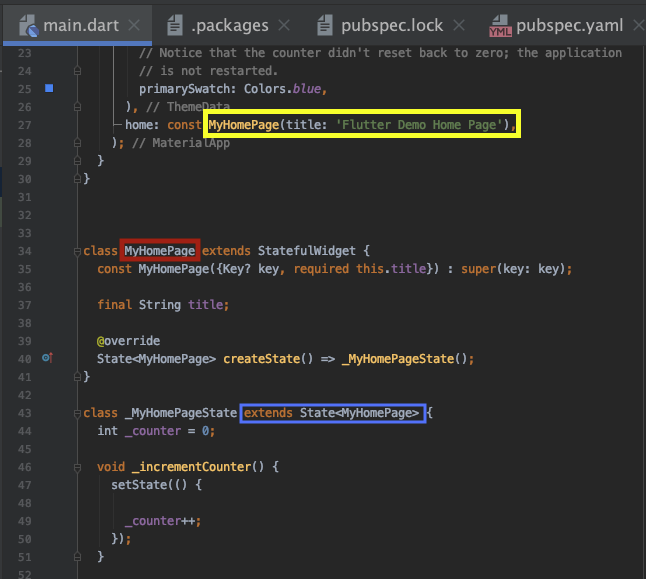


The **MaterialApp** configures the top-level Navigator to search for routes/views in a specific order. For the root view, the home property, if non-null, is used. In our starter project the MyHomePage() widger is used as the root view under the home property. It also defines a theme where we can declare different properties like the primarySwatch which is a color that will be set for the navigation bars of our views.

The way I like to think about the Material App widget is that it’s the base Material design of our application. It gives it that Android feel and look. In contrast there is another similar widget called CupertinoApp. It is basically the same widget but instead of Material design it uses a Cupertino design. The way I like to think of this widget is that it gives the app that Apple device feel and look. For the purpose of this course we will be using material design if you wish to learn more about the CupertinoApp widget here’s some [examples](https://medium.com/nusanet/flutter-cupertino-app-76c33768c242) on how to implement it and the [documentation](https://api.flutter.dev/flutter/cupertino/CupertinoApp-class.html) on it.

## MyHomePage Widget an Introduction to State Management

What is the MyHomePage(title:) Widget ?



This widget is the home page to our application. It inherits from Stateful Widget meaning that it has a **state object** (in the blue box).

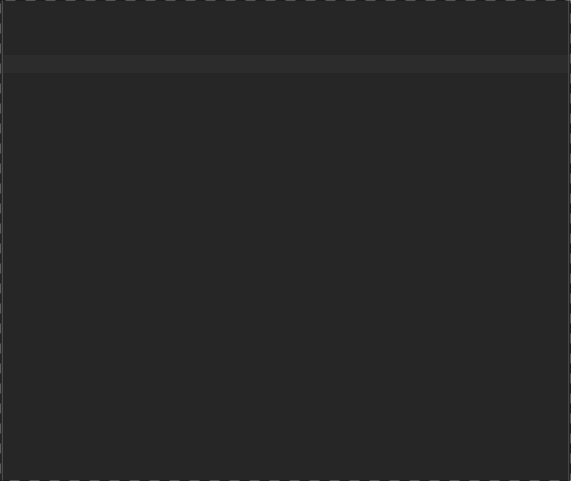
Inside this **state object** there is a variable \_counter that is incremented when the function \_incrementCounter() is called.

Notice the **setState()**function inside of it. This call to **setState**tells the Flutter Framework that something has changed in the state, which causes it to rerun the build(context:) method so that the view will display the updated value. If we changed the counter variable without calling setState() then the build method would not be called as a result nothing would change in our UI.

Try it out, delete the call to setState and only increment the counter. Hot reload the app and try pressing the plus button. See what happens?

There is a lot more on state management in Flutter and it is a very important topic outside the scope of this course. For our purposes we will be only using setState() if you wish to learn more about state management in Flutter check out the documentation on [List of State Management approaches.](https://flutter.dev/docs/development/data-and-backend/state-mgmt/options)

**Shortcut Tip:** We can create Stateful or Stateless widgets by typing out **stful**or **stless**in Android studio and it will auto complete the declaration for us.



## Scaffold and AppBar Widgets

[Scaffold](https://api.flutter.dev/flutter/material/Scaffold-class.html) and [AppBar](https://api.flutter.dev/flutter/material/AppBar-class.html) documentation.



The scaffold and appbar widgets come from the [material.dart library](https://flutter.dev/docs/development/ui/widgets/material).

The Scaffold implements the basic Material Design visual layout structure. This class provides APIs for showing drawers, snack bars, and bottom sheets.

The AppBar is Material Design app bar. It consists of a toolbar and potentially other widgets, such as a TabBar. The AppBar’s body property is where most of the views UI lives in. Here is where we write the code we want to appear below the app bar which contains the title. And potentially app bar actions. Lets go ahead and add an app bar action.

Under the title property add the actions property that takes a list of widgets. Then let’s add an IconButton widget with the property onPressed and with an Icon named .add.alert. Just like this.



Now we’ve added an Icon button to our AppBar. It's so cool and easy!!

## Basic Widgets

[Text](https://api.flutter.dev/flutter/widgets/Text-class.html)

The Text widget lets you create a run of styled text within your application.

[Row](https://api.flutter.dev/flutter/widgets/Row-class.html), [Column](https://api.flutter.dev/flutter/widgets/Column-class.html)

These flex widgets let you create flexible layouts in both the horizontal (Row) and vertical (Column) directions. The design of these objects is based on the web’s flexbox layout model.

[Stack](https://api.flutter.dev/flutter/widgets/Stack-class.html)

Instead of being linearly oriented (either horizontally or vertically), a Stack widget lets you place widgets on top of each other in paint order. You can then use the [Positioned](https://api.flutter.dev/flutter/widgets/Positioned-class.html) widget on children of a Stack to position them relative to the top, right, bottom, or left edge of the stack. Stacks are based on the web’s absolute positioning layout model.

[Container](https://api.flutter.dev/flutter/widgets/Container-class.html)

The Container widget lets you create a rectangular visual element. A container can be decorated with a [BoxDecoration](https://api.flutter.dev/flutter/painting/BoxDecoration-class.html), such as a background, a border, or a shadow. A Container can also have margins, padding, and constraints applied to its size. In addition, a Container can be transformed in three dimensional space using a matrix.

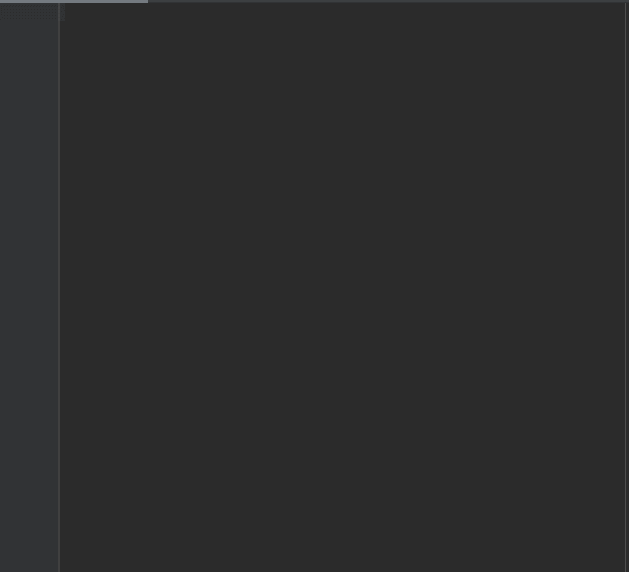
Examples on how to implement these here : <https://flutter.dev/docs/development/ui/widgets-intro>

Note: Our sample project already uses some of these widgets, we’ll also be using these to build out our custom widgets.

# Introduction to Layout

For this next section we will be building an app from scratch so let’s start by deleting everything in our main.dart file. We’ll start with a clean empty file.

Let’s start by importing the material.dart package. Then we’ll write our main() method, inside that we’ll call the runApp(MyApp()); Then we need to define our MyApp() by declaring a new stateless widget **stless**, called MyApp. And now finally we can run our new project.

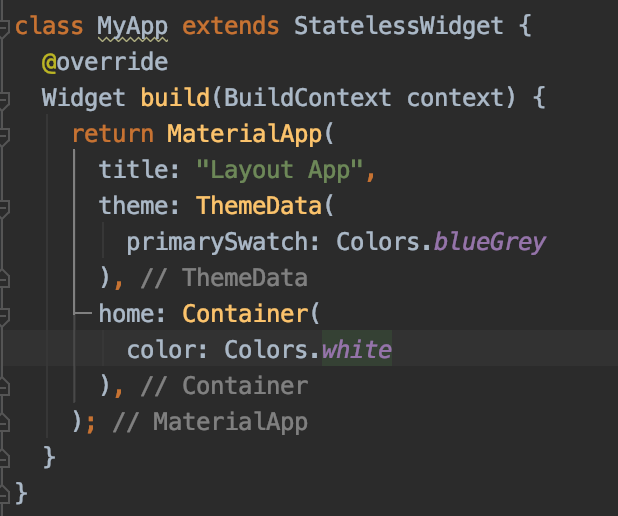


This will produce an empty black screen. Let’s go ahead and switch out that empty container with a MaterialApp Widget.

**Note**: If you were building a CupertinoApp this is where you would do that, we are sticking to Material Design however.

Let’s add a title to our MaterialApp, a theme which takes a ThemeData Widget where we define our primary swatch color. Make it whatever you’d like. Now finally let’s add the home property with an empty container whose color is white. Now if we run this we will get an empty white screen instead.

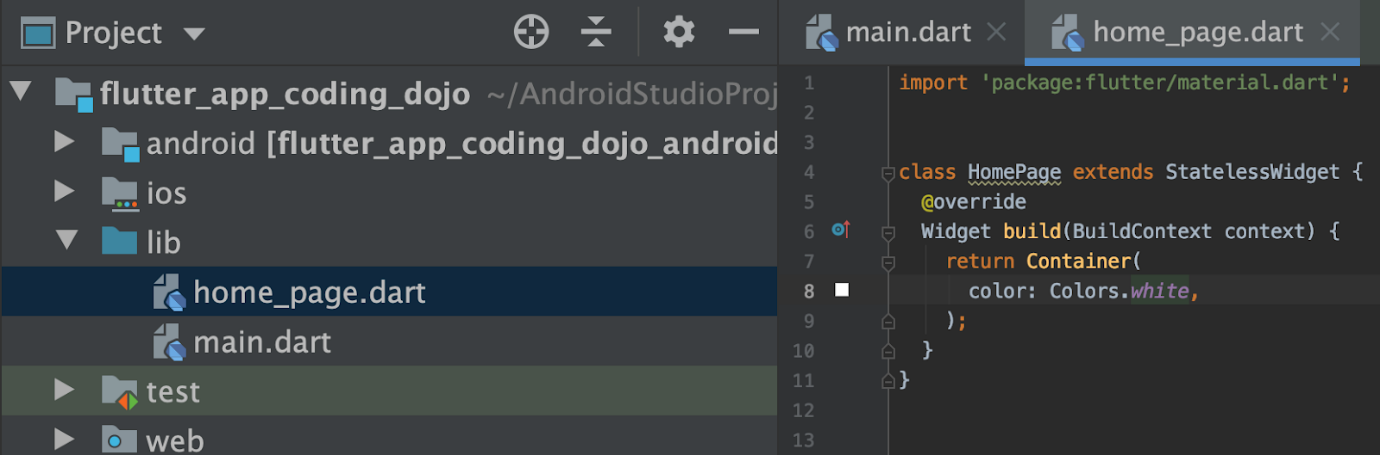
We’re making some progress now, I hope this is helping you understand how the starter project widgets work, and what they each mean. You should have something like this.



Now let’s add a new Dart file in our lib folder, this is where we’ll make our home\_page widget. Let’s name the file home\_page (note: in Dart it's common practice to use snake case naming conventions for files). In order to make a new file just right click in the lib folder go into New->Dart File.

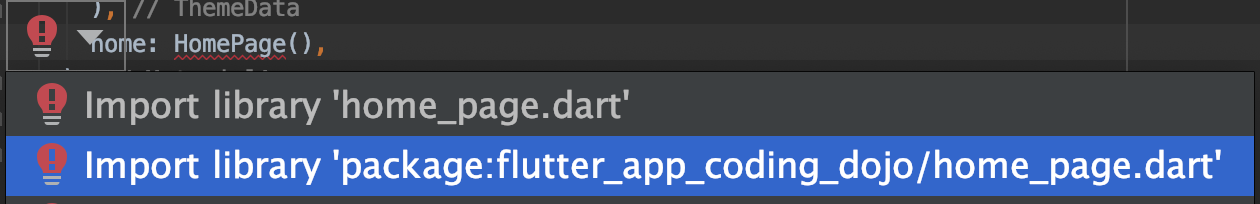
We’ll start by adding  import "package:flutter/material.dart"; as always when in a new file.

We’ll then make a stless widget called HomePage, and we’ll return an empty Container whose color is white.



Now let’s go back to our **main.dart** file and add this HomePage widget to our home property instead of an empty Container.

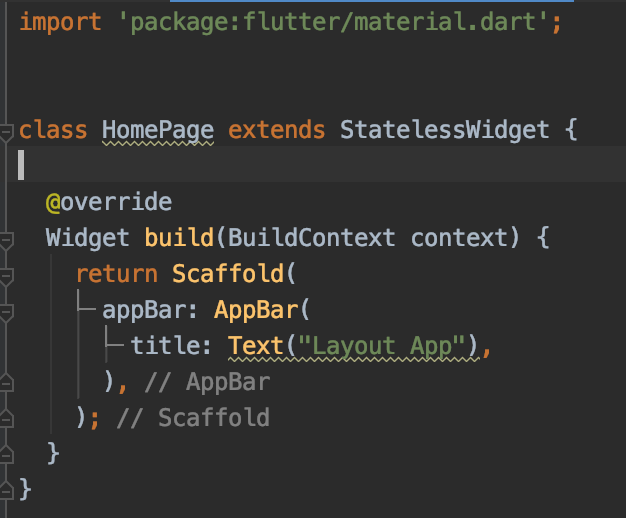
Notice when we try to use our newly built HomePage widget we get an error, saying we have to import this file in order to use it.



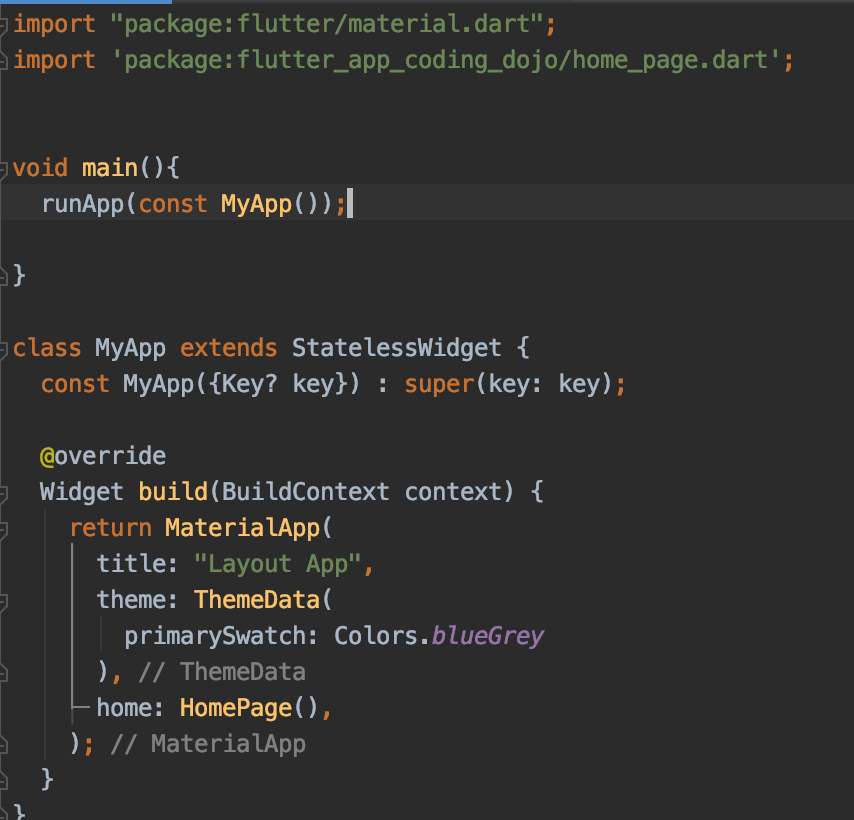
Let’s go ahead a click on the second option and we will get a new import statement at the top of our main.dart file. If we run this now, we will get the same result as before, an empty white screen. However now we have modularized our code and made a different file for our HomePage UI.

Let’s go ahead and go to the home\_page file and finish it up by adding a Scaffold widget with an AppBar instead of just an empty container. Inside our AppBar we’ll add a title with a Text() widget that says “Layout App”. Let’s run this now and see the result. Your two files should look something like this:

## home\_page.dart



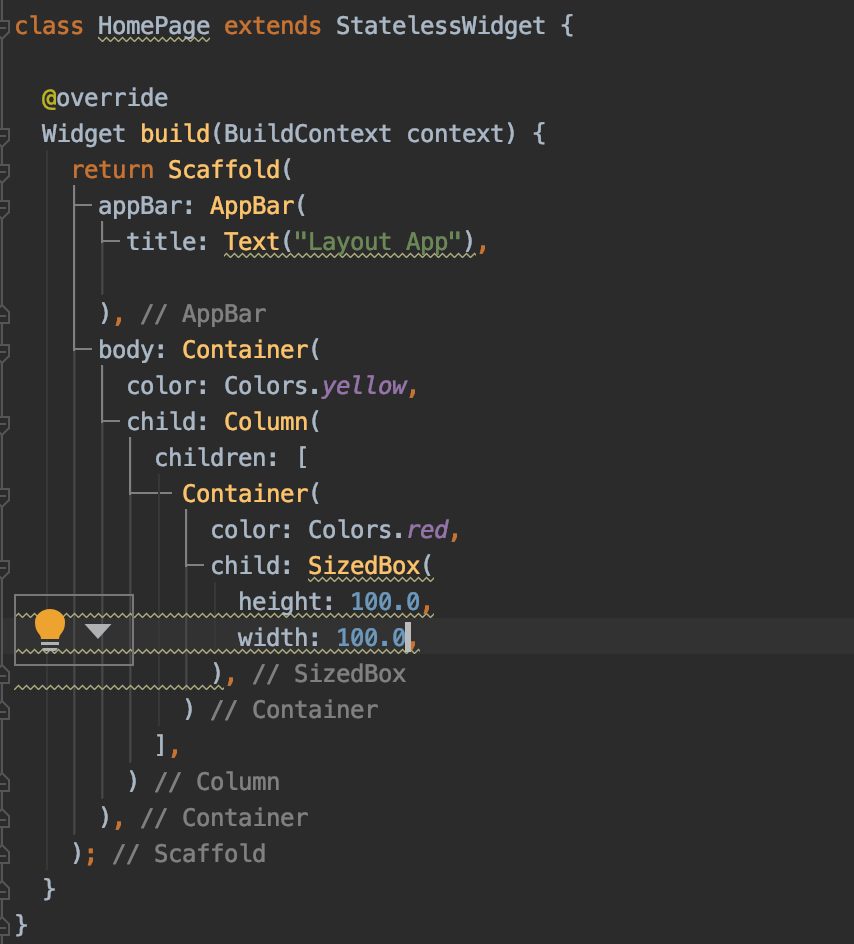
## main.dart



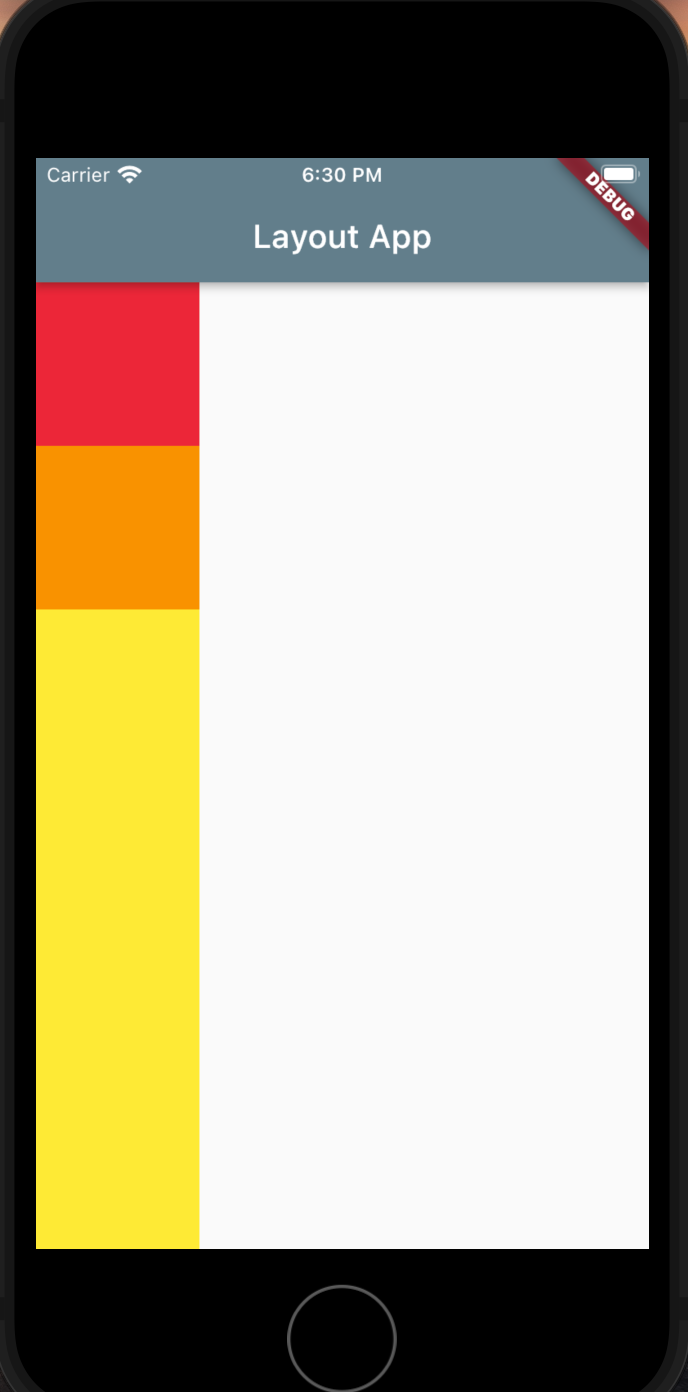
# Column Widget

Let’s take our project to the next level and learn more about layout. In our HomePage widget, let’s add a body: property to our Scaffold. Inside there, remember, will be everything below the app bar, essentially our body view. Let’s add a Container to the body with a color of yellow. Next add a child property to the Container and make the child a Column Widget.

Notice the Column widget has a property called  children, that takes an array of widgets. In this array let’s add a Container of color red, with a child SizedBox widget. Make the SizedBox’s height and width (100.00). Should look something like this.

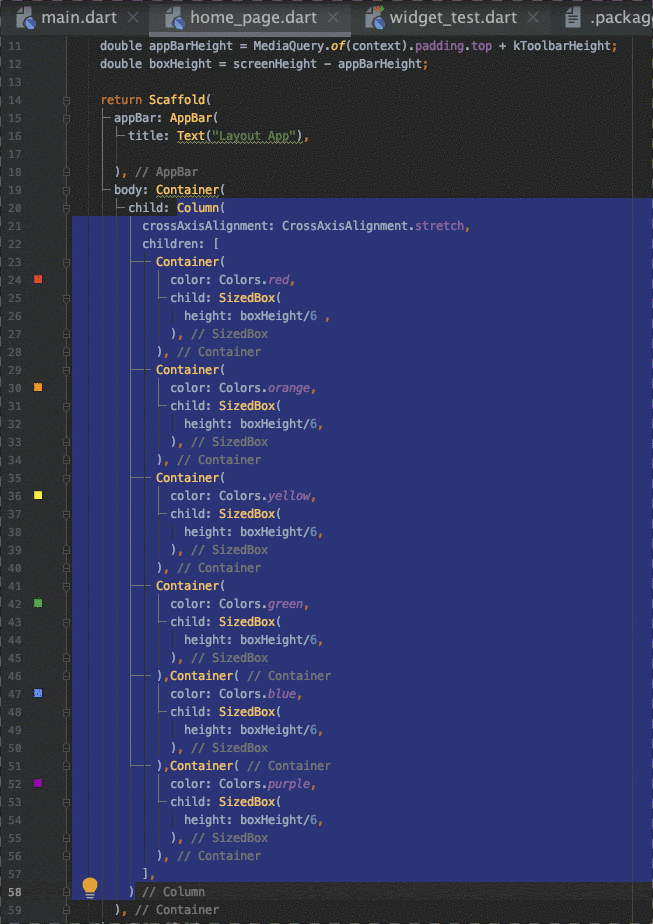


Run the code now and see what happens. Now try adding another Container in the Column’s children with the color orange and with a SizedBox of width and height of 100. Now you can see how the **Column** widget works. It stacks widgets vertically, there is also a **Row** widget that stacks other widgets horizontally. Take some time to read about the [Column widget](https://api.flutter.dev/flutter/widgets/Column-class.html) in order to complete the next Assignment.



# Extracting Code Shortcut

If you finished the previous assignment, you might have noticed that our Column widget was very long. This is typical in a Flutter project however, there are a lot of different ways to make our code look neater. Android studio has a neat functionality that allows you to extract some code into a function that returns a widget and then we can just call that function. Let’s see how we can do this.



Pretty neat right? So what we did was we selected our Column Widget then in our Android Studio we did Refactor->Extract->Extract Flutter Widget->Then we named our Widget. If you noticed this made a new Stateless Widget for us and passed it over to our Container.

From here we could move our widget into a new file to keep our home\_page file more modularized and neater. Keep in mind we don’t want to make a file too long so this can help us with that.

# Other Widgets

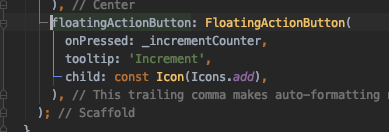
Now that we’ve learned about the foundationary widgets that our default project starts with, we can move onto learning about all the other widgets Flutter has built in. More specifically the material package components that we can use to design our applications. In this section we will focus on navigation, routes, buttons, and callbacks.

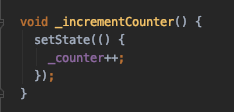
## Buttons and Callbacks

Let’s start with the button widget, in order to learn about call backs.

Let’s go ahead and **start a new flutter project** which comes with a FloatingActionButton that increases the counter variable by calling setState().

They're 3 main types of buttons, FloatingActionButton,ElevatedButton, and TextButton. Button’s always have a property onPressed: that takes a function as a parameter. These functions are sometimes called callBacks because they aren’t directly called when we use them here. They are called when the user presses the button, hence why we don’t add the () and just pass the function name. Note: The onPressed property only takes a void function.



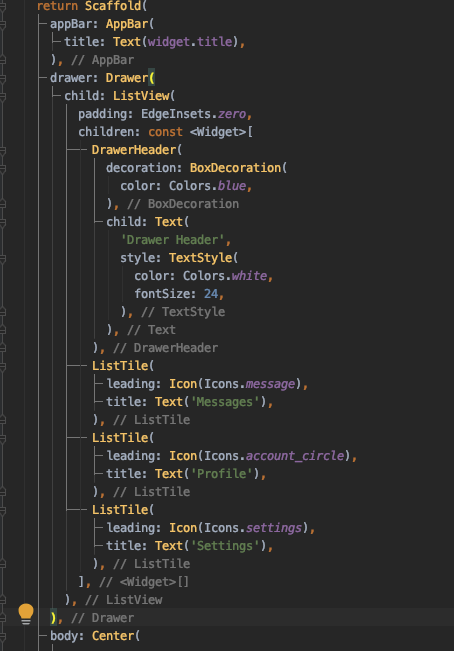


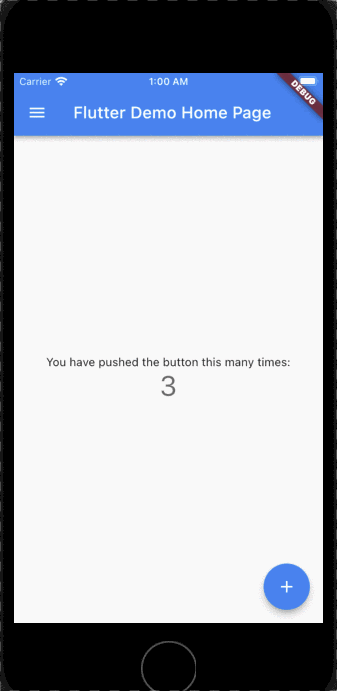
## Drawer Widget

This Component is nice when we want to show a mini profile screen. It’s like a side screen the User can access. The nice thing about the Drawer widget is that our Scaffold widget has a property called drawer. Go ahead and add the drawer widget under the appBar property.

Here’s the documentation: [Drawer Widget](https://api.flutter.dev/flutter/material/Drawer-class.html) for the example we’ll be using. It should look something like this. Read the documentation on the new Widgets used here.

ListView() and ListTile(). Try customized your Drawer, maybe add a new ListTile.





# Navigation and Routes

Now that we know how buttons and call backs work let’s try to make our Floating action button open a new screen instead of just incrementing the counter. We first have to learn about [Navigation in Flutter](https://flutter.dev/docs/development/ui/navigation).

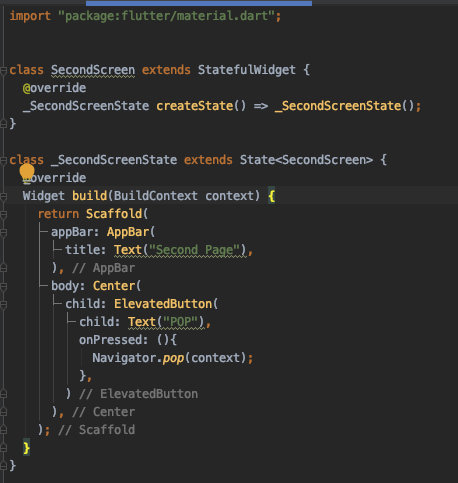
Currently there are 2 routing mechanisms in Flutter. The original Navigation (1.0) and Flutter’s updated Navigation (2.0). Before Navigator 2.0, it was difficult to push or pop multiple pages, or remove a page underneath the current one Navigation (2.0) solves this. However, Navigation (2.0) is outside the scope of this course. Navigation (1.0) uses an imperative approach which is easy to follow and learn. It will also do the basic functionality that we will need for now. If you wish to learn more about [Navigation (2.0)](https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade) there’s an article provided by the Flutter Team.

Most mobile apps display screens on top of each other, like a stack. In Flutter, this is easy to achieve by using the Navigator.of(context).

MaterialApp and CupertinoApp already use a Navigator under the hood. You can access the navigator using Navigator.of() or display a new screen using Navigator.push(), and return to the previous screen with Navigator.pop(): Let’s try to push a new screen when our Floating action Button is Pressed.

First we have to create a new screen. Let’s make a new file called second\_screen.dart in our lib folder. Let’s make it **stful** widget and call it SecondScreen. Don’t forget to import the material package at the top of the new file.

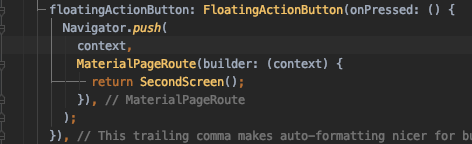
Let’s then return a Scaffold with an AppBar and a Center with an ElevatedButton that will pop us back to the Home screen when pressed.



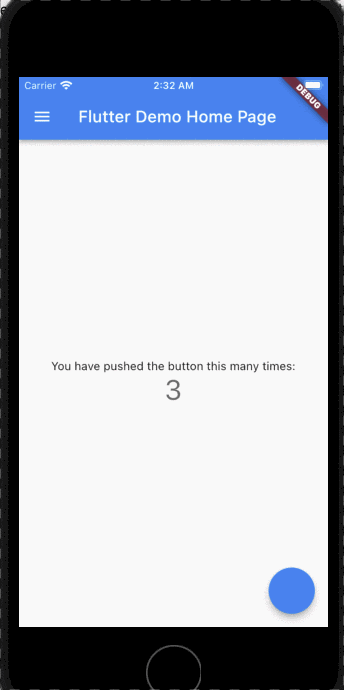
Now in our main.dart file let’s replace the \_incrementCounter function in the FloatingActionButton’s onPressed with our Navigation.push(). And we will return our SecondScreen Widget.

**Note**: Don’t forget to import the new second\_screen file into the main.dart file.

We have to pass the context to our navigator as the first parameter. And also a MaterialPageRoute and the second parameter. The MaterialPageRoute creates a material design route for us that returns a widget for us which is our Second Screen. More on it here: [MaterialPageRoute](https://api.flutter.dev/flutter/material/MaterialPageRoute-class.html).



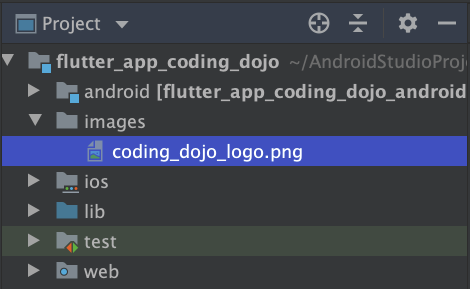
If successful you should have something like this:



## Adding Image assets (Optional)

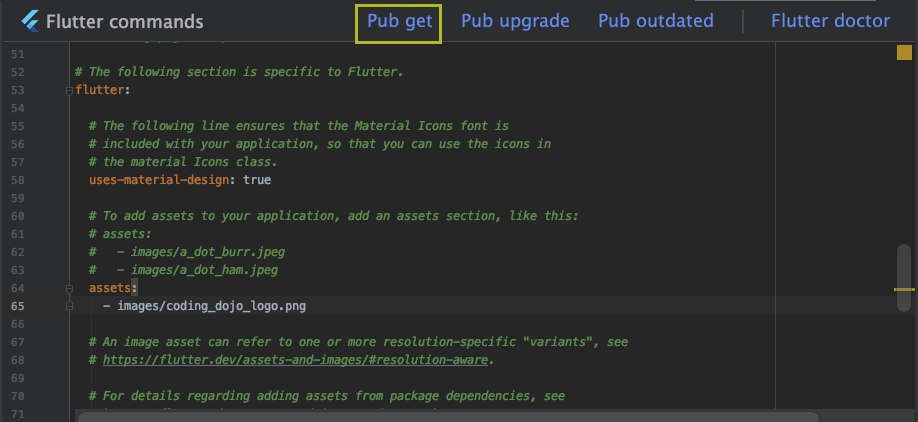
Image assets are important in any application. Sometimes we don’t just want to download images from the web but instead we want to have them saved in our application. It’s pretty simple in Flutter. Let’s add an image asset which in this example will be the Coding Dojo logo. And we will display it in our MyHomePage widget.

In our root folder let’s add a new directory named “images”. Next let’s add our image into that folder. Like this



Now we have to configure our project in order to be able to load that image. To do this we need to go into our pubspec.yaml file. If you scroll down into the **flutter:**section you will see in green writing under **uses-material-design : true** a section that introduces adding assets.

In here we need to add our images/coding\_dojo\_logo.png. Of course you would put the path of the image you need to load. **NOTE:**you need to click on **Pub Get** in order to make your app load the newly added image. In order to add multiple images you could just have to put the path of the other images just under this one like the example below.



Now let’s add this image into our MyHomePage widget. We can do this by calling Image.asset(‘images/coding\_dojo\_logo.png’). This returns an Image widget that we can display in our app.You could also use the AssetImage() widget which gives us more flexibility. Let’s try it.



In our body, and inside the Column let’s add a Text that says “Welcome” and under that we’ll add our Image.asset(). If done correctly you should be able to see the image. There are many ways to resize the image asset. We could add it inside a Container and give it a fixed size for example. That’s just one approach. You could also use the ResizeImage() widget.

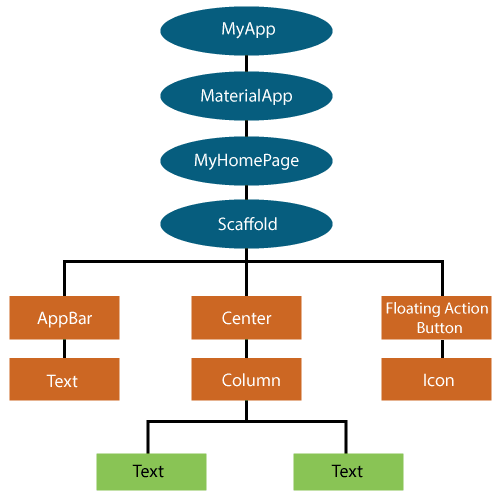


# Next Steps

Congratulations! You now know the foundations of building a cross platform application with Flutter. As you progress in your journey as a Flutter/Dart developer we recommend you keep reading the documentation in order to learn more. This was just a brief introduction to the Flutter SDK and there is a lot more to learn.

As you may know Flutter was developed by Google, and because of that there are alot of great resources to keep learning. Google has provided Flutter [CodeLabs](https://flutter.dev/docs/codelabs) which are small projects that were created by Google to teach about certain Flutter topics.

As you move on we also recommend you keep in mind the widget tree diagram.



Source : <https://www.javatpoint.com/flutter-widgets>

If you noticed when we were building our application we usually built widgets inside widgets by passing them through a property. When we start creating more complex applications, our widget tree will also get more complex and look differently than the diagram shown above.

So why is the widget tree so important? As your app becomes more complex so will your [state management approaches](https://flutter.dev/docs/development/data-and-backend/state-mgmt/simple). You will need to learn about the [Provider Package](https://pub.dev/packages/provider). Which is the state management approach recommended by Google and the Flutter team. Along with that you will also need to work with Streams, Futures, and Promises in order to pass data from widget to widget.

The widget  tree will help us in order to figure out how to pass data. It will help us answer questions like: Do I need to pass data up the tree or down? Do I need to pass data to another branch of the tree?

So we recommend you check out those links to the documentation as the next steps as well as the [Flutter Youtube Channel](https://www.youtube.com/c/flutterdev/featured). Here there are a lot of great videos on all sorts of Flutter topics, from widget of the week, to more intermediate/advanced topics like State Management with Provider. And finally, you could also try building an app where you make an API call and read the [JSON to display](https://flutter.dev/docs/development/data-and-backend/json) in your application. You could even connect your app to a [backend like Firebase](https://flutter.dev/docs/development/data-and-backend/firebase) (which is recommended by Google of course since they made it).