FLUTTER:

Oh yes! We’re going there!

We are currently familiar with Android app development in the standard route—with Java and XML. This is however, the codebase for Android only. In other words, to deploy our app on an iPhone, we’ll need to write it again from scratch in Swift—the native development language for iOS.

But what if we could write ONLY ONE code, and run it on both—Android as well as iOS? Here comes Flutter to the rescue! Flutter provides us with the functionality to have a SINGLE codebase for Android AND iOS.

Flutter is a mobile UI framework developed by Google used to cross platform app development. It uses a language called as ‘DART’ (also developed by Google) for implementing the application. The best thing about Flutter is that it is hot glued with Firebase for backend development. This means that it is Firebase optimised. Another cool thing about Flutter is that it comes with Google’s material design framework OUT-OF-THE-BOX! This means that we will have our app looking beautiful, without doing much of the designing ourselves, and letting Material.io take care of things. Awesome, isn’t it?

DART Primer

Okay, before we get on with Flutter, we need to familiarise ourselves with the DART programming language. But, the best part about DART is that if we are familiar with Java, or C++ or Python, or any other object oriented programming language, we already know most about DART.

It uses the same syntactic as well as semantic rules as that of Java. We can go as far as to say that DART is pretty much Java, with a moustache. It is pretty much Java, but with slight changes here and there. In order to familiarize ourselves with DART, a small primer course can be found [here](https://youtu.be/FLQ-Vhw1NYQ).

In order to test out DART, we can use DartPad at <https://dartpad.dev/> .

In order to highlight certain points in DART refer this handy list:

* DART is statically typed. This means that we have to specify the datatype of the variable while declaring it. It has all the standard datatypes like int, float, double, String, etc.
* Although being statically typed, it however has a datatype called as ‘dynamic’. This datatype can hold any type of value like int, float, char, String, etc. This is therefore like ‘var’ in JavaScript. However, it is not recommended to use the dynamic datatype as a single change in the variable can crash our entire program.
* It has a function called ‘print’ to print stuff out on the screen.
* Like C or C++ it needs to have a main() function which serves as the starting point of the application.
* The syntax of declaration of functions is the same as that of C/C++ or Java:

return\_type func\_name (parameters)

{

//code

}

* A slight change is observed for shorthand notations of functions. We can use the ‘arrow’ (=>) operator to quickly define functions. However this shorthand notation ONLY works for those functions which DO NOT HAVE any computational logic, and only return values.

The function: int myfunc()

{

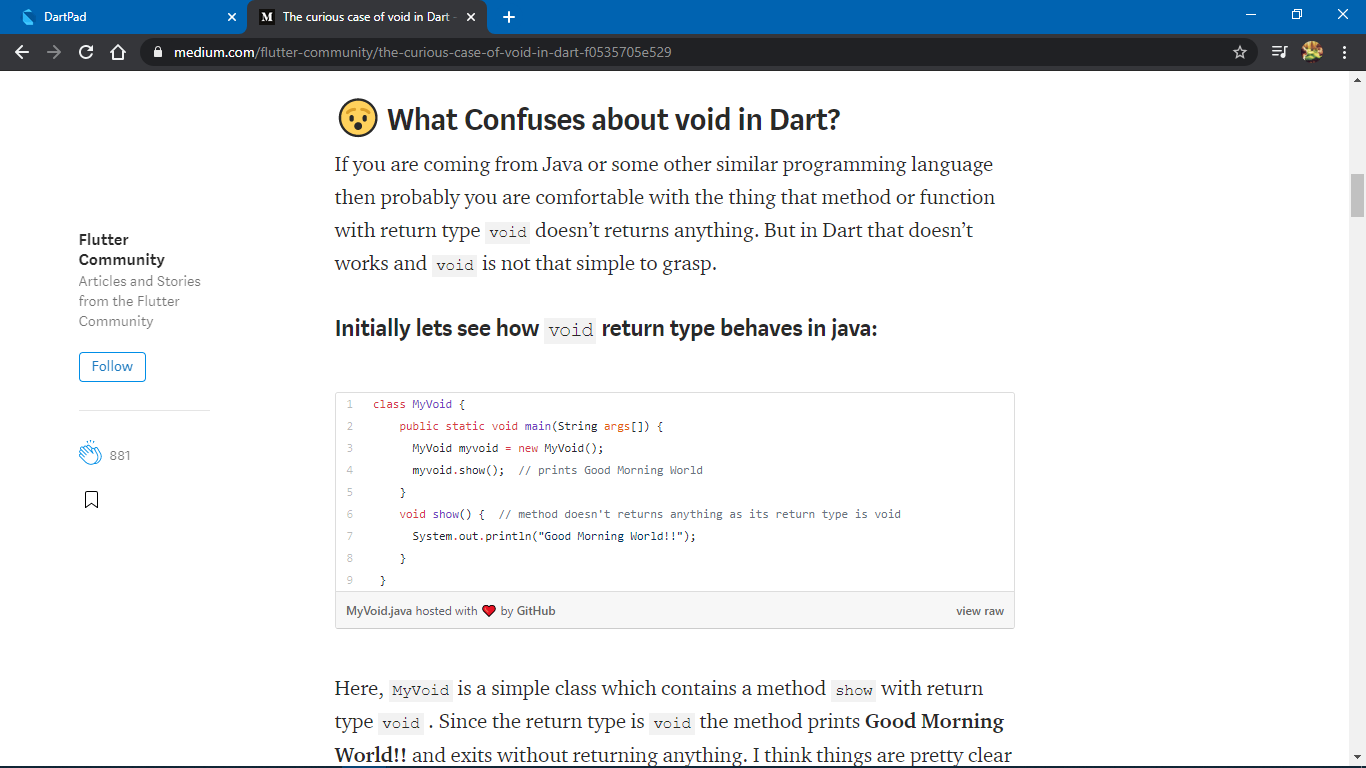
return 20;

}

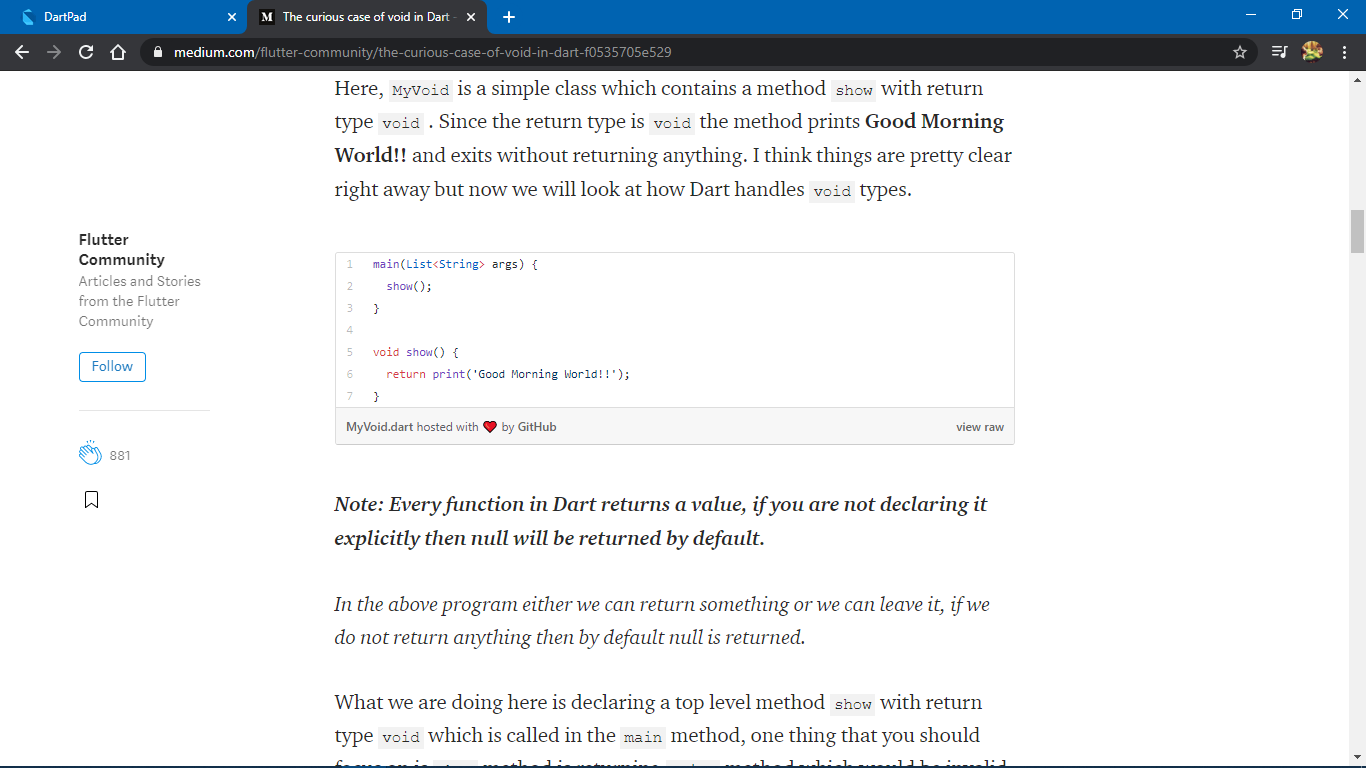
Can be written as: int myfunc => 20;

* Functions in Dart behave like first-class objects meaning they can be stored in a variable, passed as an argument or returned like a normal return value of a function. In Dart => xxx is just a syntactic sugar to avoid { return xxx; }.
* If we are coming from Java or some other similar programming language then we probably are comfortable with the concept that method or function with return type void doesn’t returns anything. But in Dart that doesn’t work and void is not that simple to grasp. Refer [here](https://medium.com/flutter-community/the-curious-case-of-void-in-dart-f0535705e529).

Initially let’s see how void return type behaves in java:



Here, MyVoid is a simple class which contains a method show with return type void . Since the return type is void the method prints Good Morning World!! and exits without returning anything. I think things are pretty clear right away but now we will look at how Dart handles void types:



***Note: Every function in Dart returns a value, if you are not declaring it explicitly then null will be returned by default.***

*In the above program either we can return something or we can leave it, if we do not return anything then by default null is returned.*

What we are doing here is declaring a top level method show with return type void which is called in the main method, one thing that you should focus on is show method is returning print method which would be invalid in languages like Java. Since in Dart almost everything is an object including functions/methods which are object of class Function and we are allowed to do some more stuff with Function like:

* Declare it as a parameter
* Pass it as an argument
* Assign it to a variable
* Return a function from a function
* Return a result of a function from a function (This is what we are doing above)

So, now you must understand that Dart functions are no ordinary methods, in fact they are functions on steroid. Now returning to our code above, inside show method we are first executing the method print and after that, we are returning the result of print method.

***Note: The return type only specifies what the DartAnalyzer warns you about. There are type checks now in Dart2 that also check that at runtime.***

If the return type of show function is void then only three things can be returned, returning any other result will cause a type warning.

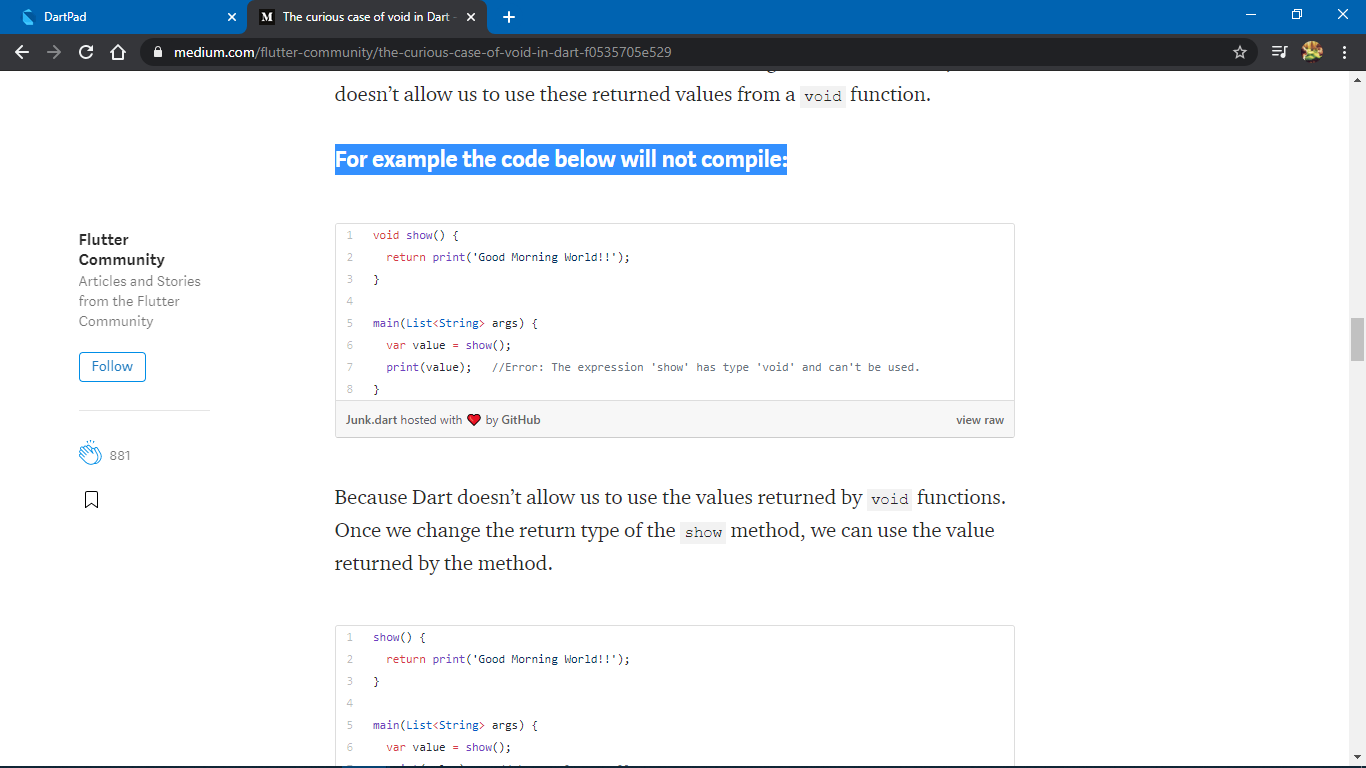
1) Return null

2) Return dynamic variable

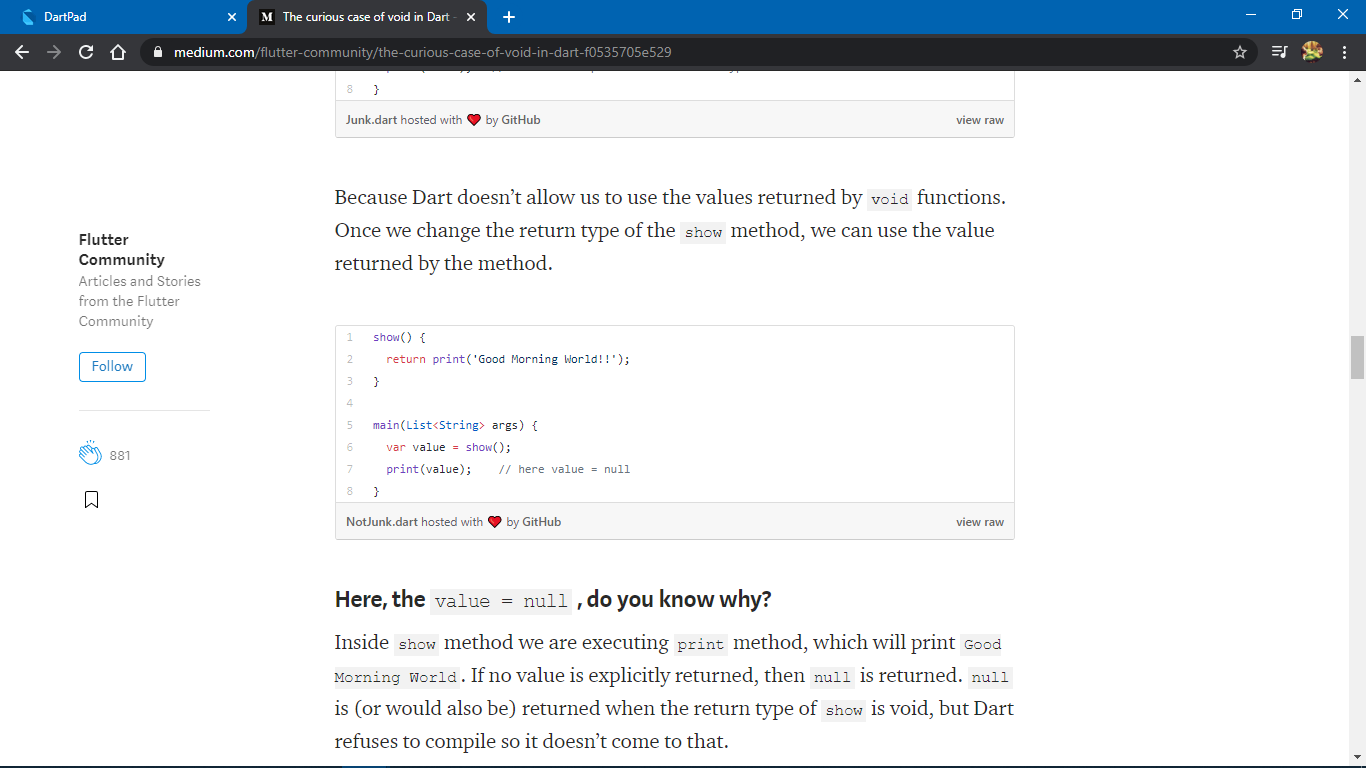
3) Return void function

But even if Dart allows to return these three things in a void function, it doesn’t allow us to use these returned values from a void function.

**For example the code below will not compile:**



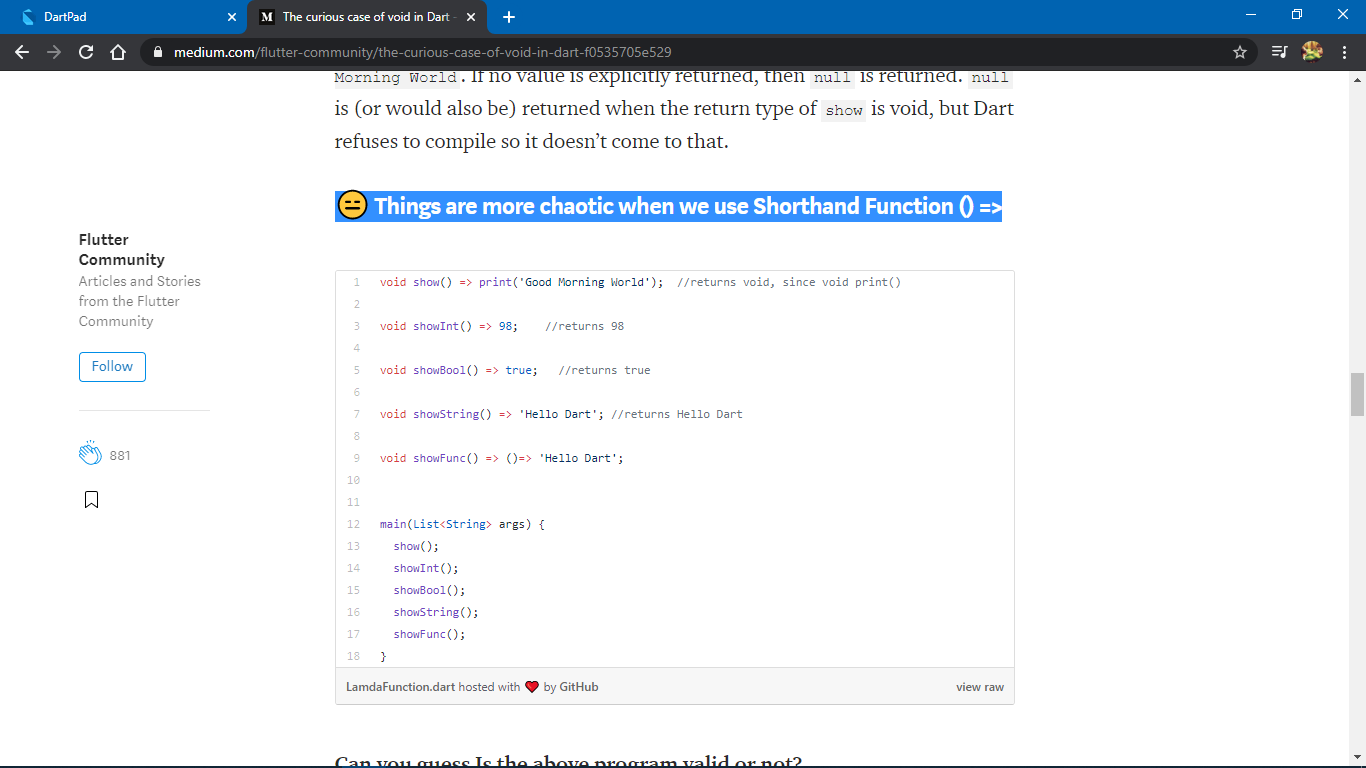
Because Dart doesn’t allow us to use the values returned by void functions. Once we change the return type of the show method, we can use the value returned by the method.



**Here, the**value = null**, do you know why?**

Inside show method we are executing print method, which will print Good Morning World. If no value is explicitly returned, then null is returned. null is (or would also be) returned when the return type of show is void, but Dart refuses to compile so it doesn’t come to that.

**😑 Things are more chaotic when we use Shorthand Function () =>**



The program will compile without any error and it is 100% valid. Yes you can ask that the return types are void then how can we return int , String , bool or a function ? The answer is you can return anything from a shorthand function with return type void. But guess what? You can’t communicate with or use those returned values in any way.

**So why is returning everything allowed through a shorthand function[()=>] even if the return type is void?**

*As we have discussed above in normal functions with return type void we can only return three types of values(i.e. void , null and dynamic) but in shorthand function things are little different because shorthand functions always returns a value and if we limit the values that could be returned then functions with void return types will become invalid. Let’s take an example of setter function as the default return type of every setter is void and they are often declared as a shorthand function.*

* Arrays in DART are called Lists. So in DART, List is actually a datatype. A list can store elements of a wide variety of datatypes. So, we can have a list like:

List chungus=[‘Worship’, ‘the’,’cat’,’god’,20,true];

* We, however can make a list to be limited to a specific datatype by mentioning the datatype inside angular brackets (<>). So, we can have a list like:

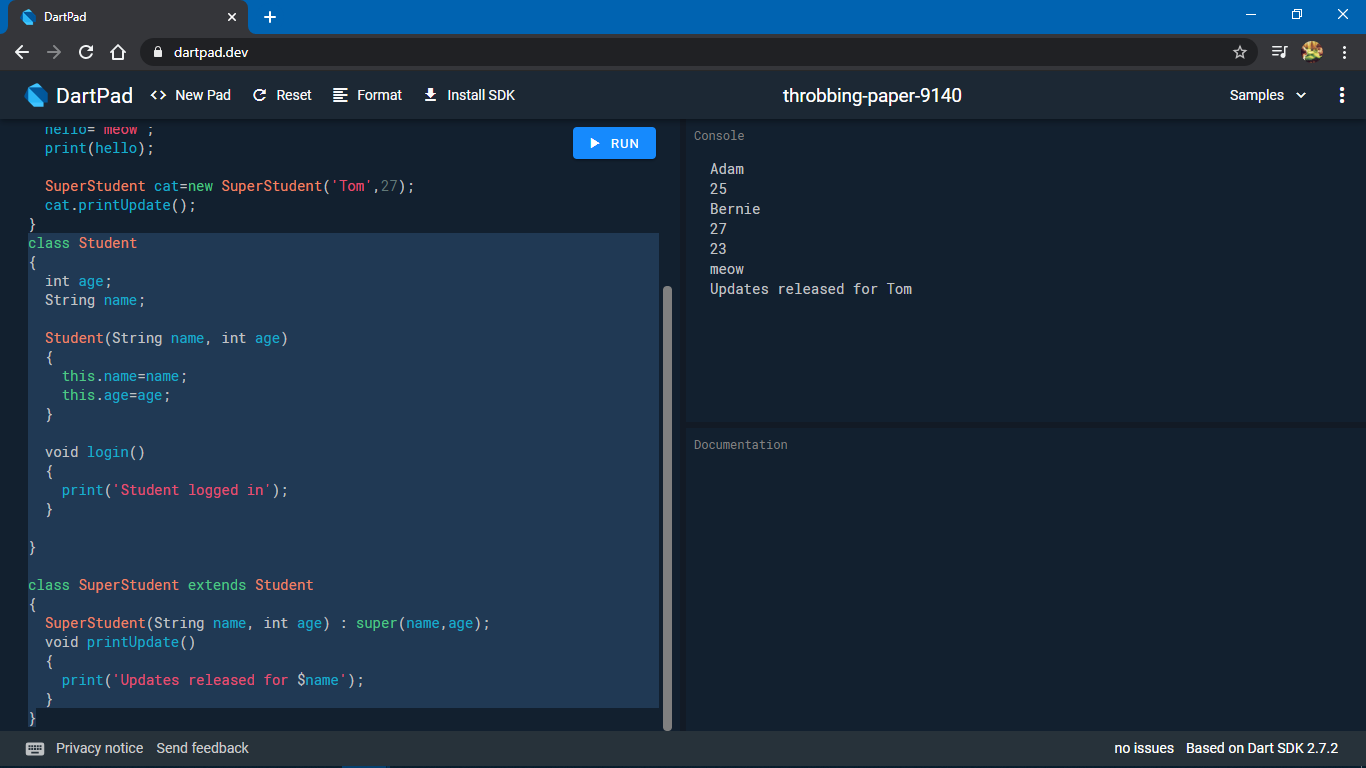
List numericChungus=[10,20,30,40,50];

Inserting a String inside this list will give an error.

* Lists also have built in functions like add() which appends elements to the end of the list, and remove() which removes the specified element from the list.
* The concept of classes and objects is EXACTLY the same as in Java. We have instance variables as well as methods for classes. To declare object of a class we have:

Class\_Name obj\_name=Class\_Name();

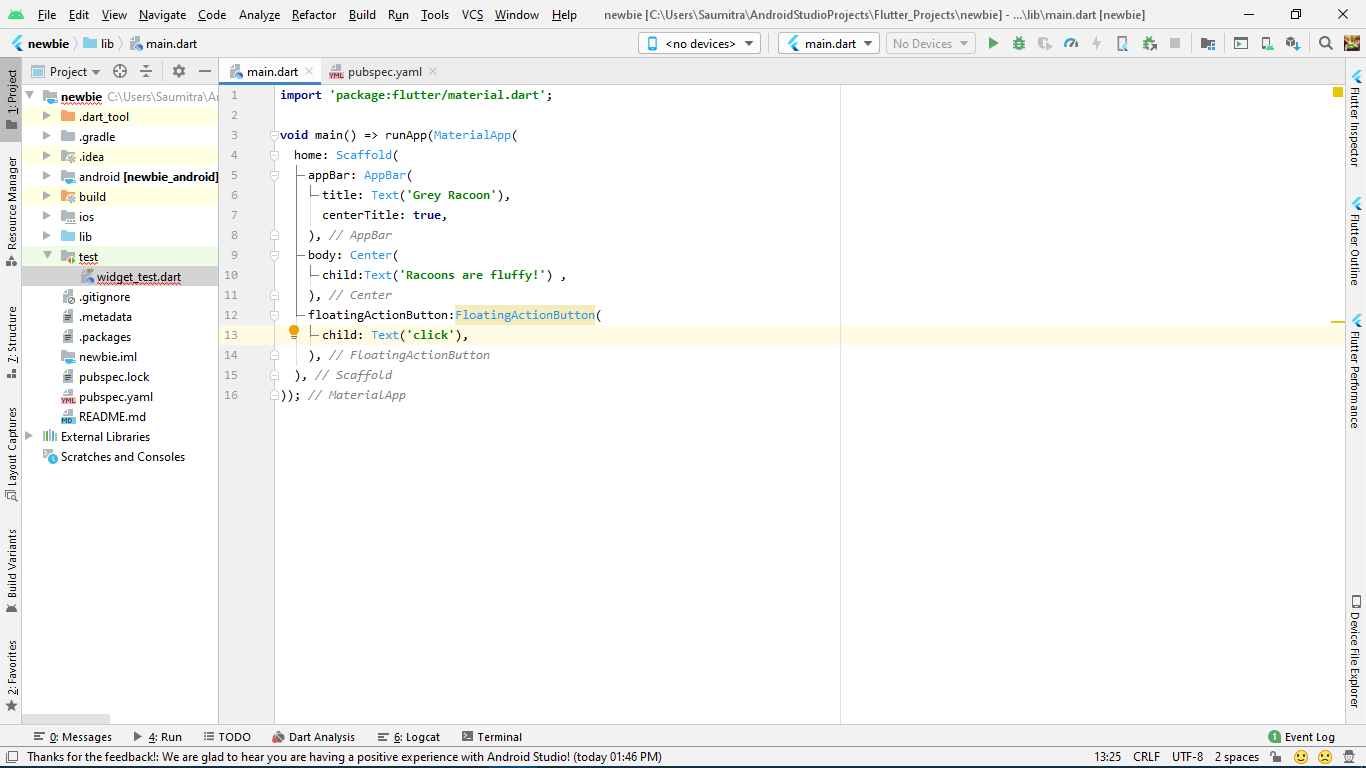
* The classes can also be inherited. So, a new class can ‘extend’ an existing class, implying that it has all the functionality of the parent class ALONG with certain special functionality of its own.
* The extended class however should have its own constructor, we can also piggyback the superclass’s (the parent class’s) constructor using the ‘super’ keyword. Refer this code:



* Oh, and yes, in order to print a variable inside a print() statement, we precede it with a dollar ($) sign.

Coding In Flutter (Refer App: newbie)

Let’s revisit the layout of Android Studio in Flutter mode for a brief overview.



Your app has 2 very basic and very important files:

1>main.dart

2>pubspec.yaml

The main.dart file is where you write the entire code of the app is written in the DART language. It doesn’t separate the UI from the logic which was the case with Java and XML. We write the UI elements taking help of the out-of-the-box material design features instead of the ordinary XML.

The pubspec.yaml file is somewhat similar to a combination of the android\_manifest.xml and the build.gradle files. It contains all information about the including its dependencies, and the different packages that the app uses. It also contains the version number of dependencies as well as the SDK.

The first statement in our dart file is an import statement. It imports the material design features for flutter in our DART environment.

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

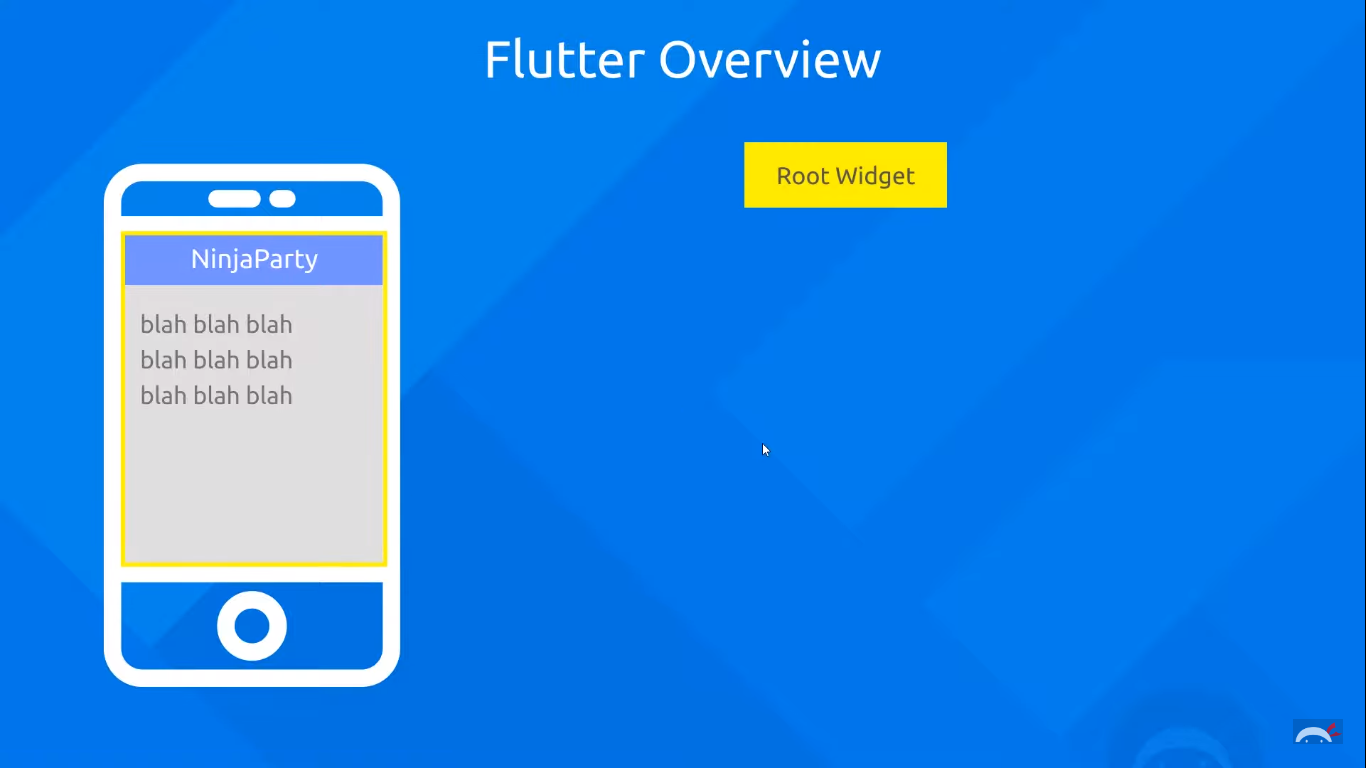
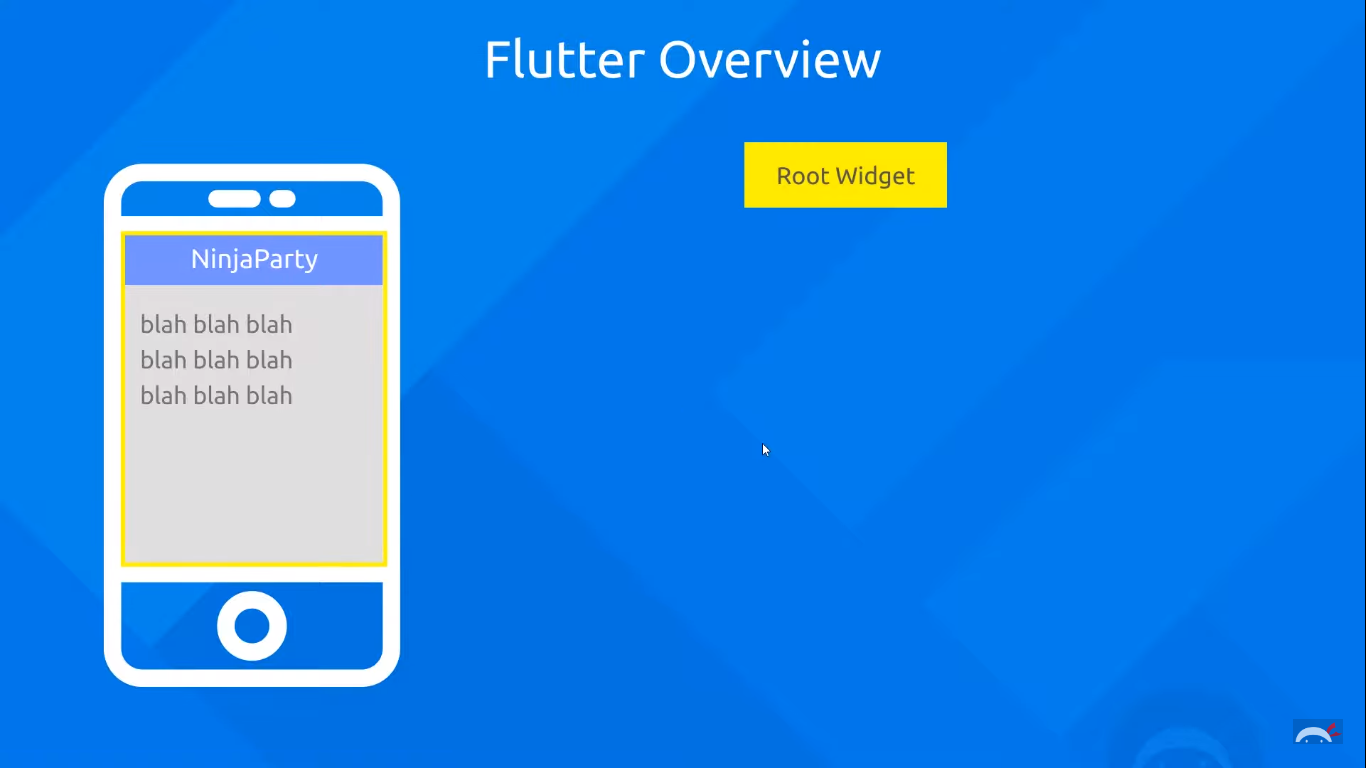
Then comes our most important function—the main() function. With the help of the arrow (=>) notation, we return the value returned by the runApp() function. The runApp() function returns void (see dart primer above).

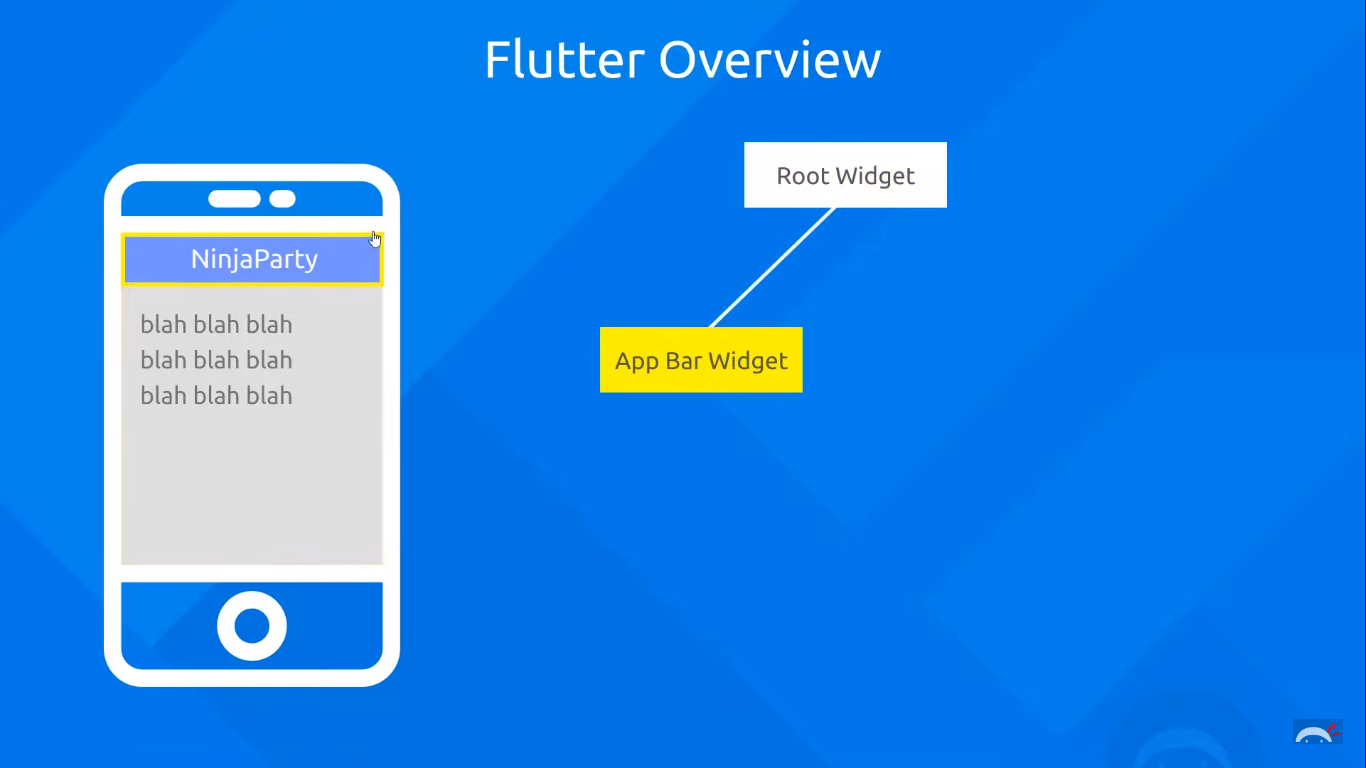
**void** main() => runApp();

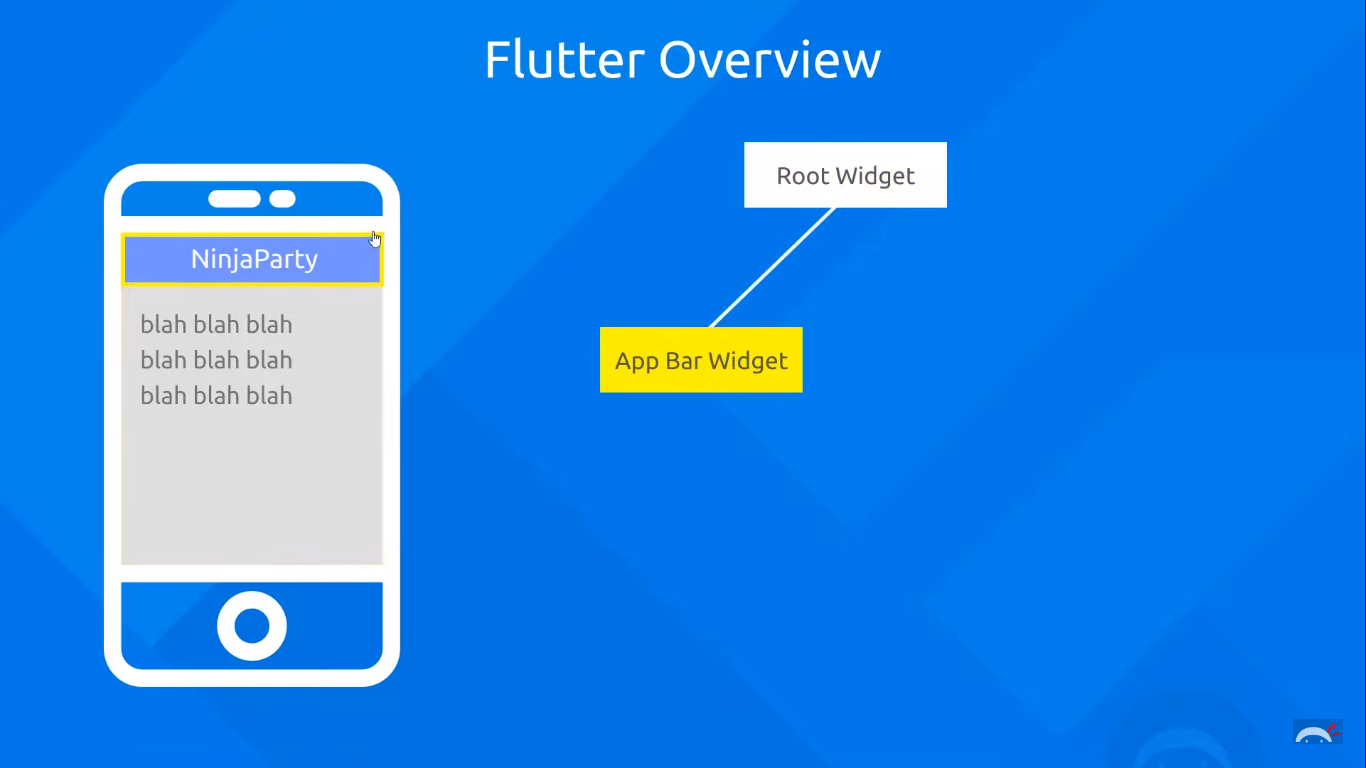
Okay, now we need to understand a key concept regarding flutter. Flutter uses elements called as ‘widgets’. Each and every element that you see on the screen is called as a ‘widget’. Like the ‘AppBar’, the ‘FAB—Floating Action Button’, the text on the screen, the side drawer, etc. All these widgets are provided to us by Flutter out-of-the-box.

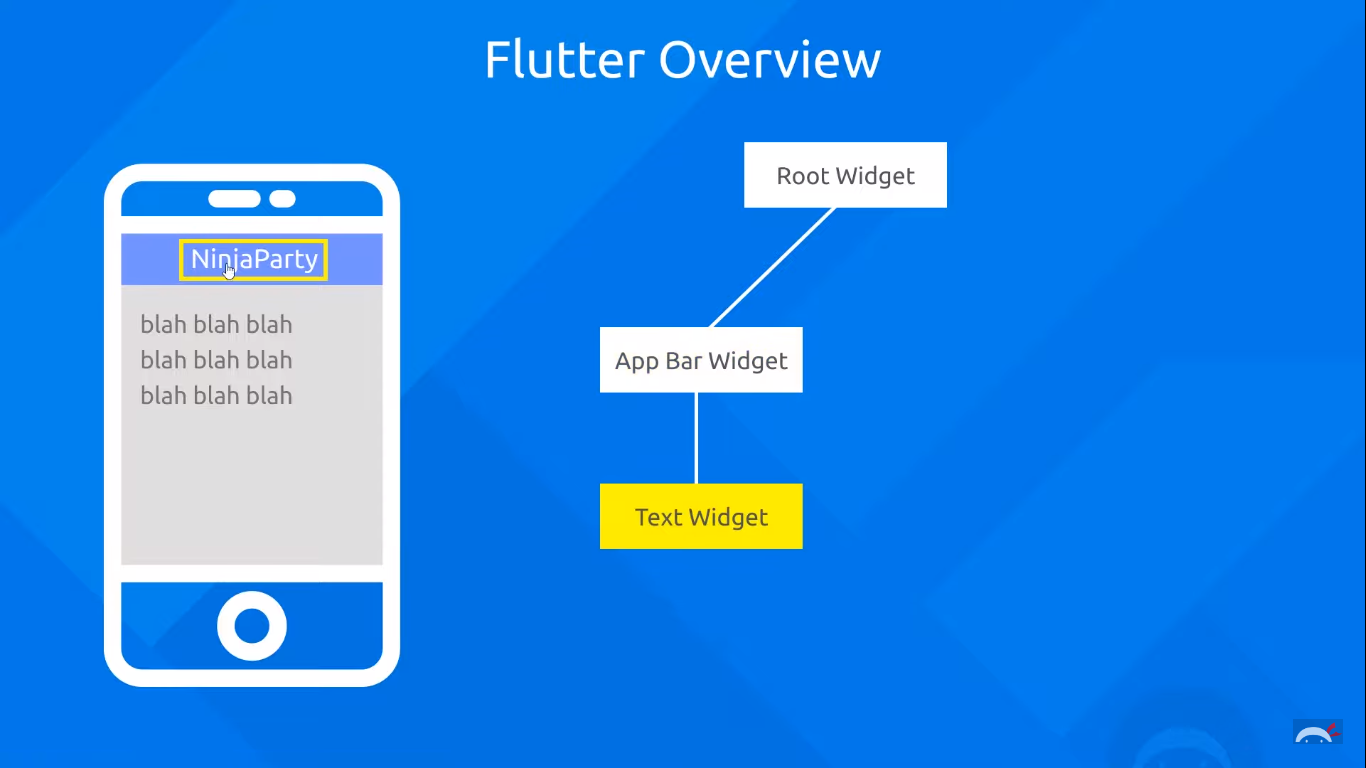
These widgets follow a tree structure. Like all trees, this ‘widget-tree’ also has a root element. All of the elements on the screen are child nodes of this root widget. Let’s say that we have an app that looks like:

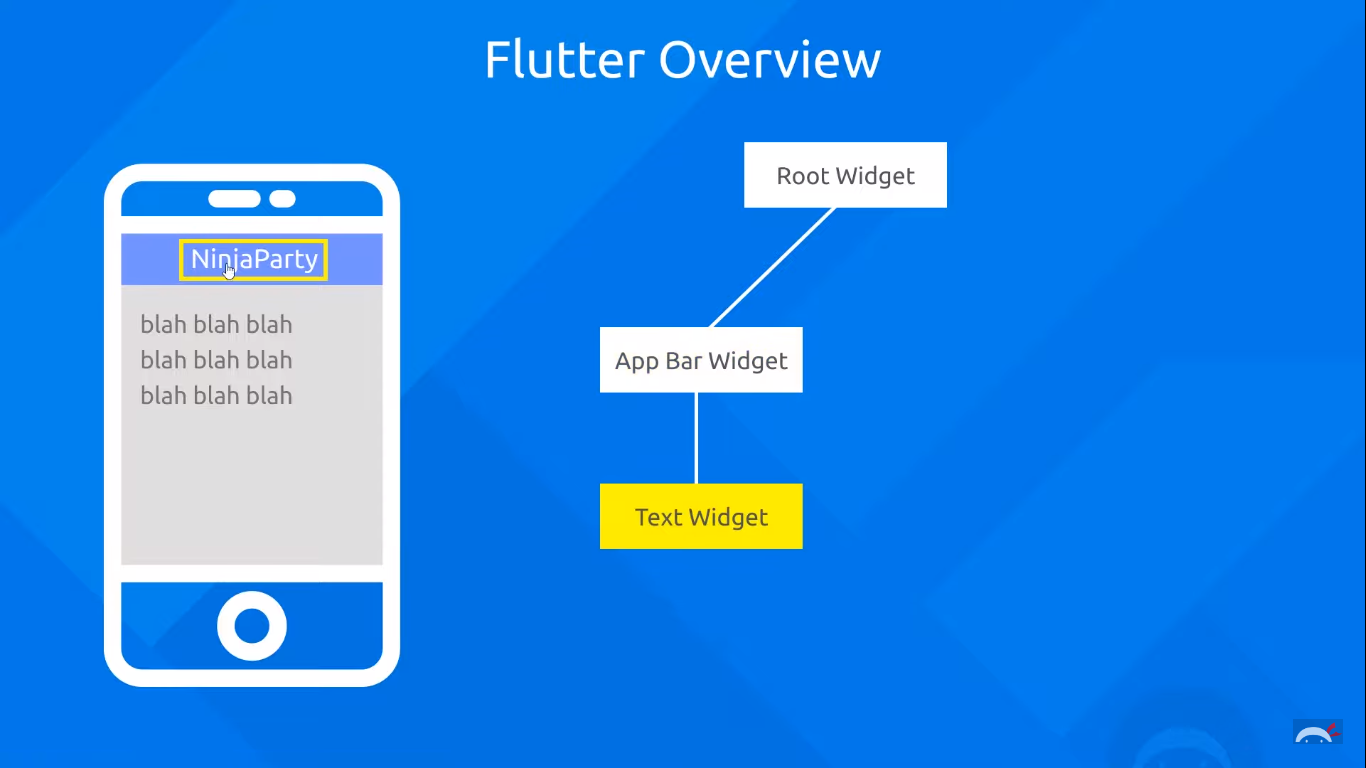
 This pretty basic app has a lot of different widgets floating around on the screen. We have the ‘root widget’ which forms the wrapper, or the container of all the other widgets.



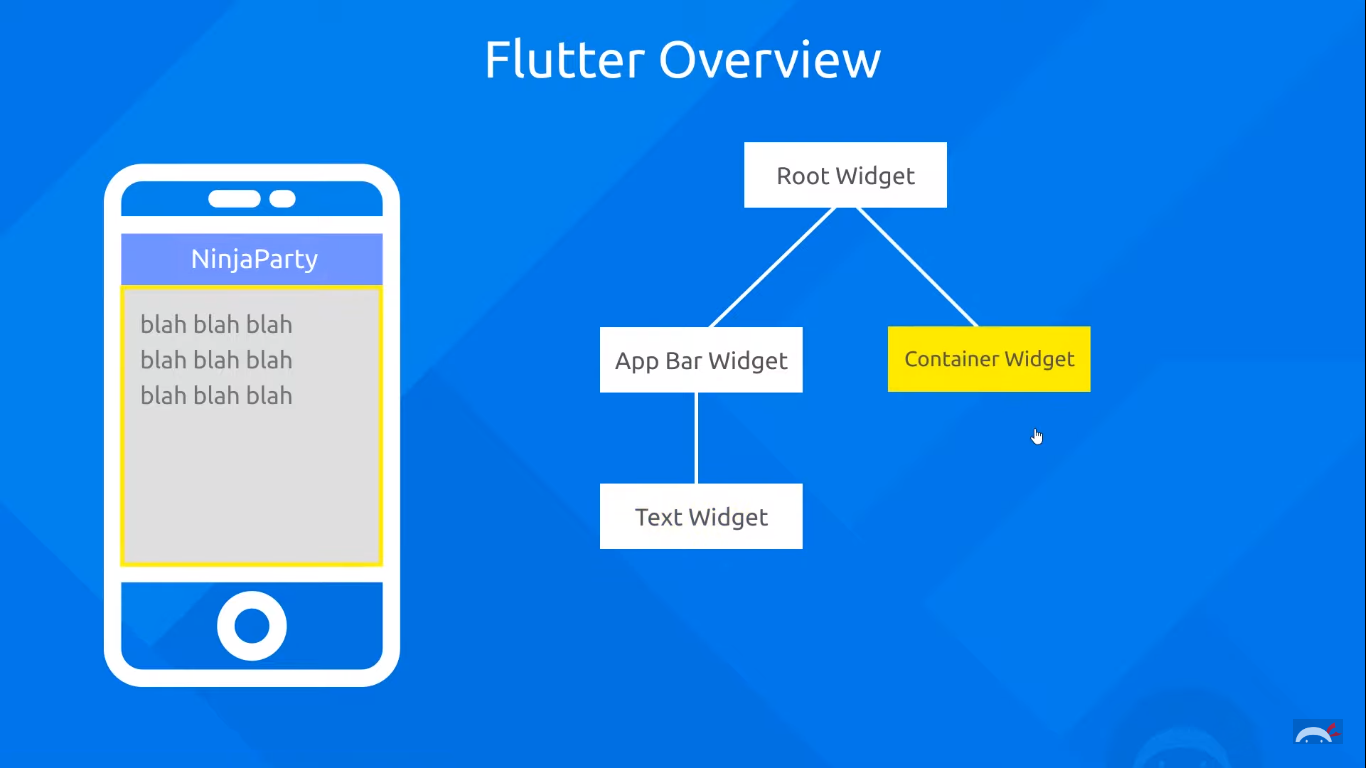


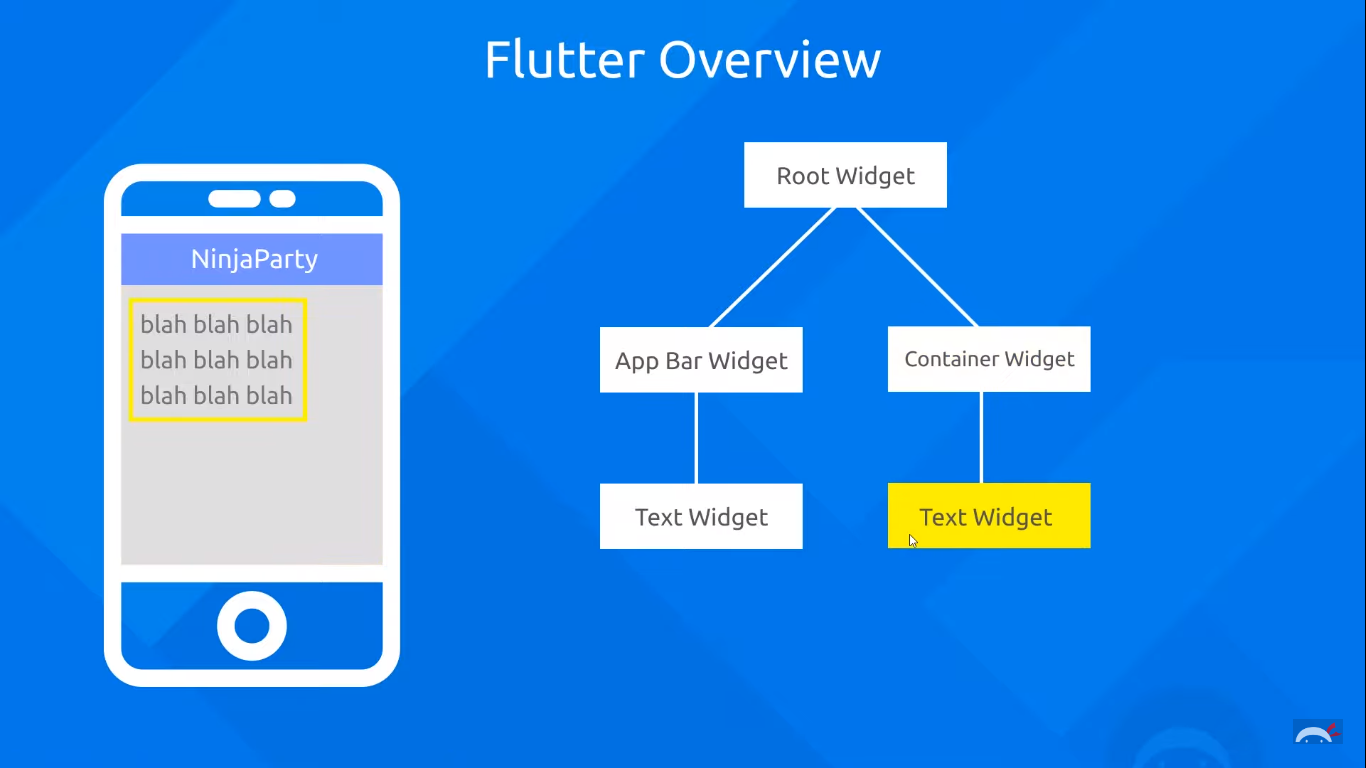


Then we have AppBar widget INSIDE this root widget. The AppBar forms the top portion of the application screen which usually displays the app name. The AppBar in turn has a text widget nested inside it.

****

Similarly, we have:





Widgets are nothing but classes in DART. Each widget has its own programmatic behaviour which defines how it looks on the screen. Each widget has a host of different properties which can be altered in order to change the way the widget appears on the screen. Like for example:



Nesting widgets inside one another is the main feature of flutter.

Coming back to our runApp function, this function runs our app, after accepting a root widget for our app. We can create our own root widget by creating a DART class (we’ll look at that later on), but for simplicity sake, let’s use a root widget which has already been created by Google’s Material Design. The name of this root widget is: MaterialApp().

**void** main() => runApp(MaterialApp());

Currently our app can execute, but will just have a black screen.

As discussed earlier, inside our widgets, we can specify properties. So, for our root widget ‘MaterialApp’, we shall specify a property called as ‘home’. This property specifies what will be displayed on the very first screen when the app launches (kind of like the launcher activity). Now each of the properties has values which are specified by us. These values CAN THEMSELVES BE WIDGETS. This is what gives rise to the nesting of the widgets.

The general syntax of specifying properties of the widgets is:

Widget\_name(

Comma separated

property\_name\_1 : value1,

The values themselves can be widgets

property\_name\_2 : value2,

property\_name\_3 : value3,

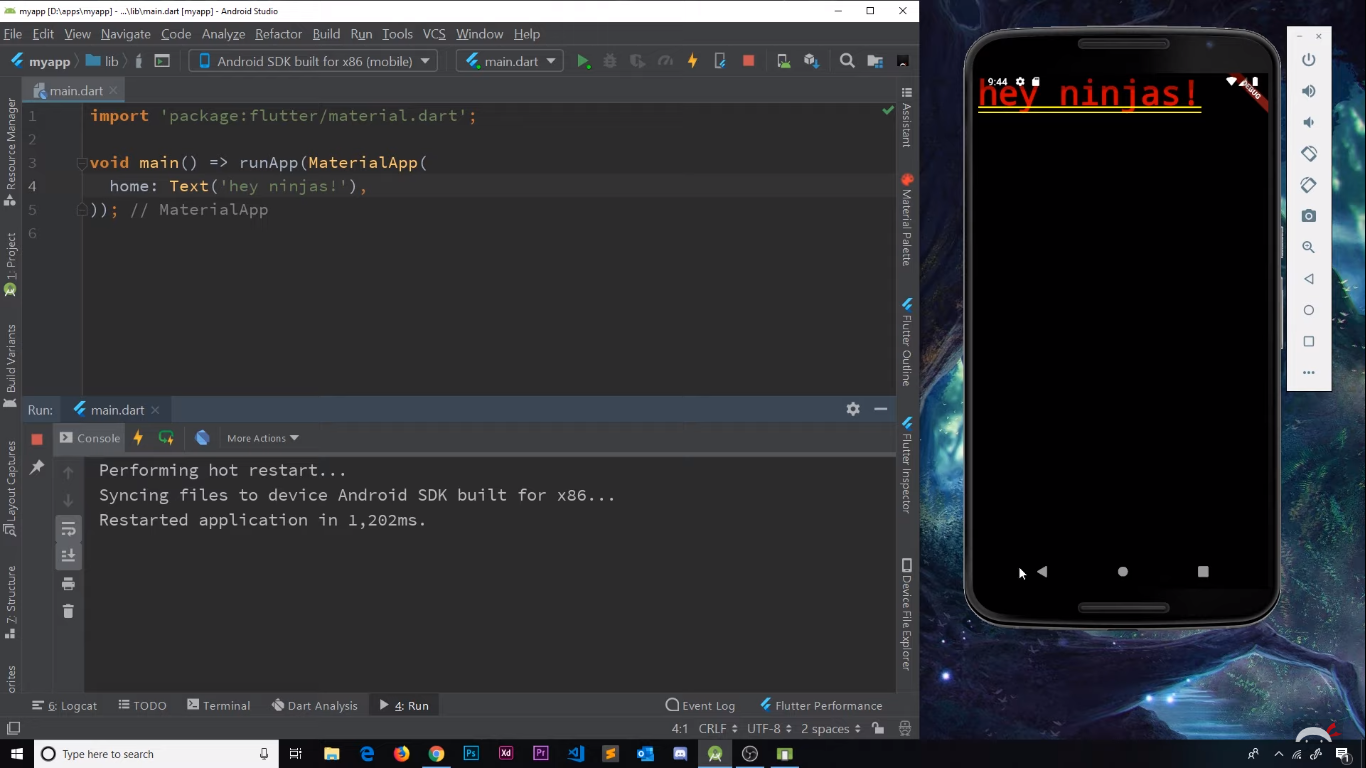
)

So, let us insert a Text widget AS THE VALUE into the home property. The text widget accepts just the text as the property.

**void** main() => runApp(MaterialApp(  
 home: Text(‘hey ninjas!’),

));

The output looks like:



Pretty ugly, innit?

Don’t worry, we’ll improve it in a moment!

However, let’s take a moment to appreciate that how little code was required to make a working app. This app is cross platform (for Android as well as iOS), and employs Google’s Material Design toolkit.

Scaffold and AppBar Widgets:

Okay, let’s give some shape to our app. We do so by using what is called the ‘Scaffold’ widget. The scaffold widget allows us to give a basic layout for our app. We can add an AppBar on the top, some body elements, FABs, and much more. Similar to how the ‘Scaffold’ of a building is used to support the structure during construction, this widget is used to support all the other widgets of the app during development.

We add the Scaffold widget to the home property, implying that the first thing that the app displays after we launch it, is the scaffold.

**void** main() => runApp(MaterialApp(

home: Scaffold();

));

We now add an appBar to this scaffold. We simply do this by altering the appBar property of the scaffold (Yes, there is an appBar property of the Scaffold). We add an appBar widget to this property.

**void** main() => runApp(MaterialApp(  
 home: Scaffold(  
 appBar: AppBar(),

),

),

);

Thus, we successfully NESTED a widget INSIDE another. We can give a title to this appBar, by changing its title property. But we cannot directly assign text to it. If we want to output some text, we need to use the Text widget. So, here we go again, nesting the Text widget inside the AppBar widget.

**void** main() => runApp(MaterialApp(  
 home: Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),

In order to center the title text, we can also set the value of the property called ‘Center Title’ to ‘true’.

**void** main() => runApp(MaterialApp(  
 home: Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 ),

We can also add a body to our Scaffolding, and add text to our body.

**void** main() => runApp(MaterialApp(  
 home: Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 ),  
 body: Text (**'Racoons are fluffy'**),

),

),

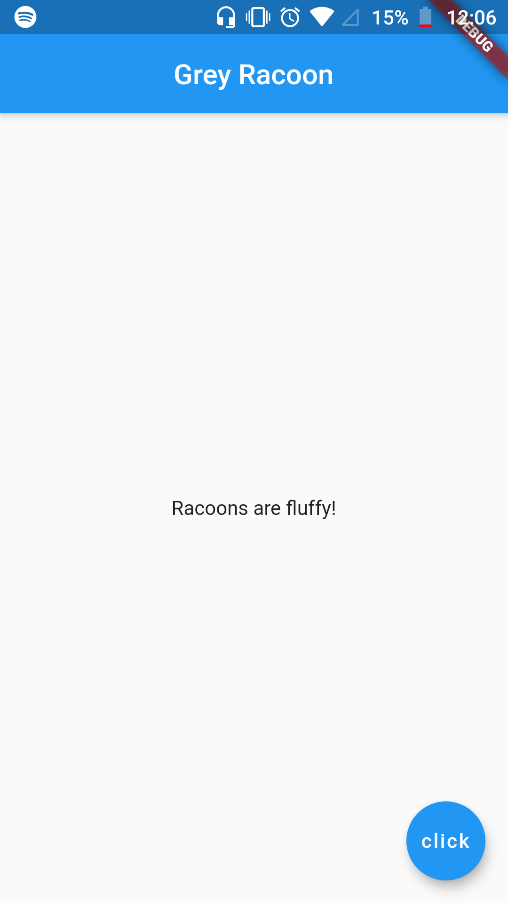
);

To bring the body text to the centre of the screen, we use the Center() widget and insert our Text() widget inside it. But we cannot insert a widget directly. We need to explicitly tell Flutter that, ONE OF THE CHILDS in the Widget tree is centered. So, we use the child property of the center widget.

**void** main() => runApp(MaterialApp(  
 home: Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 ),  
 body: Center(  
 child:Text(**'Racoons are fluffy!'**) ,  
 ),

Last, but not the least, we can add a floating action button to the scaffold. With the child property, we can also set the text of the FAB. The entire code looks like:

**import 'package:flutter/material.dart'**;  
  
**void** main() => runApp(MaterialApp(  
 home: Scaffold(  
 appBar: AppBar(  
 title: Text(**'Grey Racoon'**),  
 centerTitle: **true**,  
 ),  
 body: Center(  
 child:Text(**'Racoons are fluffy!'**) ,  
 ),  
 floatingActionButton:FloatingActionButton(  
 child: Text(**'click'**),  
 ),  
 ),  
));

Et Voila!

A few lines of code, and our app looks much better!