**An introduction to Dart and Flutter**

**A bit of background**

It all began in 2011: Xamarin, now a Microsoft-owned company, came up with a solution for hybrid mobile apps through its signature product, Xamarin SDK with C#. And thus began the revolution of hybrid mobile applications, the ease in writing one code base for many platforms.

Ionic sprung up in 2013 with its first release by Drifty Co. Ionic helped web developers use their existing skills in the growing mobile app industry. In 2015, Facebook used React.js to reinvent it for mobile app developers. They gave us React Native, a completely JavaScript code base relying on native SDK’s.

**What is Dart?**

Google had its first ever release of Flutter 1.0 last December, after having it in beta mode for over 18 months. Dart is the programming language used to code Flutter apps. Dart is another product by Google and released version 2.1, before Flutter, in November. As it is starting out, the Flutter community is not as extensive as ReactNative, Ionic, or Xamarin.

Dart looks a bit like C and is an object-oriented programming language. So, if you prefer the C languages or Java, Dart is the one for you, and you’ll likely be proficient in it.

Dart is not only used for mobile app development but is a programming language. Approved as a standard by Ecma (ECMA-408), it’s used to build just about anything on the web, servers, desktop and of course, mobile applications (Yes, the same people who standardized our favourites ES5 and ES6.)

Dart, when used in web applications, is transpired to JavaScript so it runs on all web browsers. The Dart installation comes with a VM as well to run the .dart files from a command-line interface. The Dart files used in Flutter apps are compiled and packaged into a binary file (.apk or .ipa) and uploaded to app stores.

**What does code in Dart look like?**

Like most ALGOL languages (like C# or Java):

1. The entry point of a Dart class is the main() method. This method acts as a starting point for Flutter apps as well.
2. The default value of most data types is null.
3. Dart classes only support single inheritance. There can be only one superclass for a particular class, but it can have many implementations of Interfaces.
4. The flow control of certain statements, like if conditions, loops (for, while and do-while), switch-case, break and continue statements are the same.
5. Abstraction works in a similar manner, allowing abstract classes and interfaces.

**Unlike them (and sometimes a bit like JavaScript):**

1. Dart has type inference. The data type of a variable need not be explicitly declared, as Dart will “infer ”what it is. In Java, a variable need to have its type explicitly given during declaration. For example, String something; But in Dart, the keyword is used instead like so, var something; The code treats the variable according to whatever it contains, be it a number, string, bool or object.
2. All data types are objects, including numbers. So, if left uninitialized, their default value is not a 0 but is instead null.
3. A return type of a method is not required in the method signature.
4. The type num declares any numeric element, both real and integer.
5. The super() method call is only at the end of a subclass’s constructor.
6. The keyword new used before the constructor for object creation is optional.
7. Method signatures can include a default value to the parameters passed. So, if one is not included in the method call, the method uses the default values instead.
8. It has a new inbuilt data type called Runes, that deal with UTF-32 code points in a string. For a simple example, see emojis and similar icons.

**Dart also has inbuilt libraries installed in the Dart SDK, the most commonly used being:**

1. dart:core for core functionality; it is imported in all dart files.
2. dart:async for asynchronous programming.
3. dart:math for mathematical functions and constants.
4. dart:convert for converting between different data representations, like JSON to UTF-8.

You can find more information on Dart libraries [here](https://www.dartlang.org/guides/libraries/library-tour).

**Using Dart in Flutter**

Flutter has more app-specific libraries, more often on user interface elements like:

1. Widget: common app elements, like the Text or ListView.
2. Material: containing elements following Material design, like FloatingActionButton.
3. Cupertino: containing elements following current iOS designs, like CupertinoButton.

**Setting up Flutter**

So, to get this thing into gear, follow the [Flutter docs.](https://flutter.io/docs/get-started/install) It gives details on installing the Flutter SDK and setting up your preferred IDE; mine would be VS code. Setting up VS code with the Flutter extension is helpful. It comes with inbuilt commands, as opposed to using the terminal.

Follow the docs again to create your first app. In my case, run the extension command Flutter: New Project. Afterward, type the project name and pick the destination folder.

If you prefer using the terminal, move to the destination folder of the app. Then use the command flutter create <app\_name> to create the app folder. This generates the entire app folder, including the Android and iOS project folder. To open these folders, use Android Studio and XCode, for building the app.

In the root of the project, you find pubspec.yaml. This file contains the app's dependencies. This includes both external libraries/modules and assets like images and config files. It works like a package.json, containing all external modules of the app. To install these packages, enter the package name and version under the dependencies: section of the pubspec.yaml. Run the command flutter packages get. Include the assets of the app inside the flutter: section of the same file.

The entry point of the app is main.dart, found inside the lib folder. This folder also contains all Dart classes (app pages or reusable components). On creation of the app, the main.dart file comes with a simple pre-written code. Before running this code, a device is either connected to the PC, with USB debugging enabled. Afterward, run the command flutter run on the terminal.

iPhone 8 & iPhone X with and without bezels

**Project Structure**

Let’s first see what’s in the project generated by the Flutter framework:

* **lib/** - just as [pub](https://www.dartlang.org/guides/libraries/create-library-packages) (Dart’s package manager), all the code will be here
* **pubspec.yml** - stores a list of packages that are required to run the application, just like **package.json** does it. You should remember that in Flutter projects you cannot use pub directly, but instead, you will use the Flutter command: flutter pub get <package\_name>
* **test/** - I’m sure you know what this is about. Right? You can run them via flutter test
* **ios/** & **android/** - the code specific for each platform, including app icons and settings where you set what permissions you’ll need for your application (like access to location, Bluetooth).

We don’t need to know more about the files in the folder for now. Let’s open the **lib/** folder where **main.dart** is waiting for us. As you can guess this one is the entry point of our application. Just like in the C language (or tons of others) the app will be executed by calling the main() function.

**About widgets (Hello World is here)**

In Flutter everything is built on Widgets. UI elements, styles, themes, and even state is managed in specific Widgets. Let’s start from a small application.

Replace the code from **main.dart** with the one given below, read the comments, and run the application.

**runApp(…)** only has a widget argument. The widget will become the root widget for the whole application. BTW, changing the root widget cannot be handled by Hot-reload so you’ll have to restart your application to see changes.

[**Text(…)**](https://docs.flutter.io/flutter/widgets/Text-class.html) - Flutter cannot render text without knowing what’s the preference for text direction. To render text, we have to set **Text.textDirection**. Don’t confuse it with the CSS [**text-align**](https://developer.mozilla.org/en-US/docs/Web/CSS/text-align) rule. It is the analogy of [**direction**](https://developer.mozilla.org/en-US/docs/Web/CSS/direction) - the part of the internationalization API. However, don’t worry, we won’t need to set it for each **Text** widget - later we’ll see how to set it for the whole app.

A screenshot of a cell phone

Description automatically generated

**Widgets**

* There are broadly two types of widgets in Flutter.
* **State-full Widgets** and **Stateless Widgets**. The names are self-explanatory.
* State-full Widgets are sensitive to what happens within its boundaries and gets rebuilt when a state change is detected. Conversely, Stateless widgets are not state sensitive and remain static throughout its life cycle.

How to Write a State-full Widget?



How to Write a Stateless Widget?

A screenshot of a cell phone

Description automatically generated

**Container Widget**

Containers are widgets that wrap around other child widgets to define their:

* Alignment- aligns the child widgets

A picture containing bird

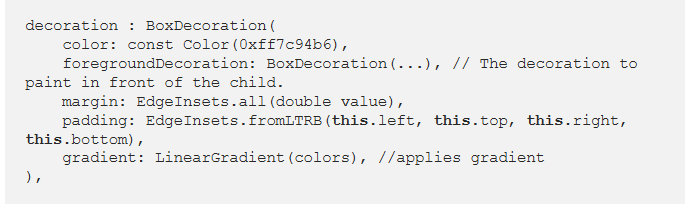
Description automatically generated

[**Constraints**](https://api.flutter.dev/flutter/widgets/Container/constraints.html)

**A screenshot of a social media post

Description automatically generated**

[**Decoration**](https://api.flutter.dev/flutter/widgets/Container/decoration.html)



**Wrap VS Row, Column**

**A picture containing drawing

Description automatically generated**

**ROW**

**A screenshot of a computer

Description automatically generated**

* A Row widget arranges its children horizontally.
* [**mainAxisAlignment**](https://api.flutter.dev/flutter/rendering/RenderFlex/mainAxisAlignment.html) and [**crossAxisAlignment**](https://api.flutter.dev/flutter/rendering/RenderFlex/crossAxisAlignment.html) properties are used to understand the position of the child widgets.
* A Row widget doesn’t scroll. However, to have a scroll-able line of widgets, a [**ListView**](https://api.flutter.dev/flutter/widgets/ListView-class.html) can be used. Or the row widget can be wrapped around a [**SingleChildScrollView**](https://api.flutter.dev/flutter/widgets/SingleChildScrollView-class.html) Widget with scroll direction set to horizontal.
* The [**Expanded**](https://api.flutter.dev/flutter/widgets/Expanded-class.html) **Widget** is wrapped around a child so that it can take all the available space in the row. However, the usage of Expanded Widget is optional.
* It’s a good practice to specify the child widgets with a Flex factor so that the child gets expanded and fills up the available space as per the flex-priority.
* Yellow and black warning stripes may appear if the children of the row are together wider than the row itself.

|  |
| --- |
| A screenshot of a social media post  Description automatically generated |
| A screenshot of a cell phone  Description automatically generated |
| A picture containing knife  Description automatically generated |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| **COLUMN**  **A screenshot of a cell phone  Description automatically generated** |
| * A Column widget arranges its children vertically. * How the children should be placed is determined by the mainAxisAlignment and crossAxisAlignment properties. * A Column widget behaves like a row widget in every aspect except, a column widget’s mainAxisAlignment is always vertical.   A screenshot of a social media post  Description automatically generated |  |

**A screenshot of a cell phone

Description automatically generated**

**A First Look at the Flutter App**

The app currently looks like this now:

A screenshot of a cell phone

Description automatically generated

When we look at the current code for the Home page, we see that it’s a Stateful page. If the counter variable increases, the framework tries to find the least expensive way to re-render the page. In this case, find the minimal difference between the current widget description and the future one. It takes into account the changed state.

The Scaffold class is a material design layout structure and is the main container for the Home page. The AppBar, also a material design element is the title bar found at the top of the page. All other components, like the floating button and two text tags, fall under the body of the page. The Center class is a layout class that centers its child components vertically and horizontally.

The Column class, another layout widget, lists each child element vertically. Each of its child elements is added to an array and put underneath the children: section.

The two texts speak for themselves. The first displays the text ‘You have pushed.’ The second one displays the current value in the \_counter variable.

The FloatingActionButton is part of the Material design widgets. It displays a + icon and triggers the increment of the \_counter variable.

A screenshot of a cell phone

Description automatically generated

**Lets now Create a simple app of your own**

**A screenshot of a cell phone

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**Try and click on the colors!!**

**A picture containing flower

Description automatically generated**