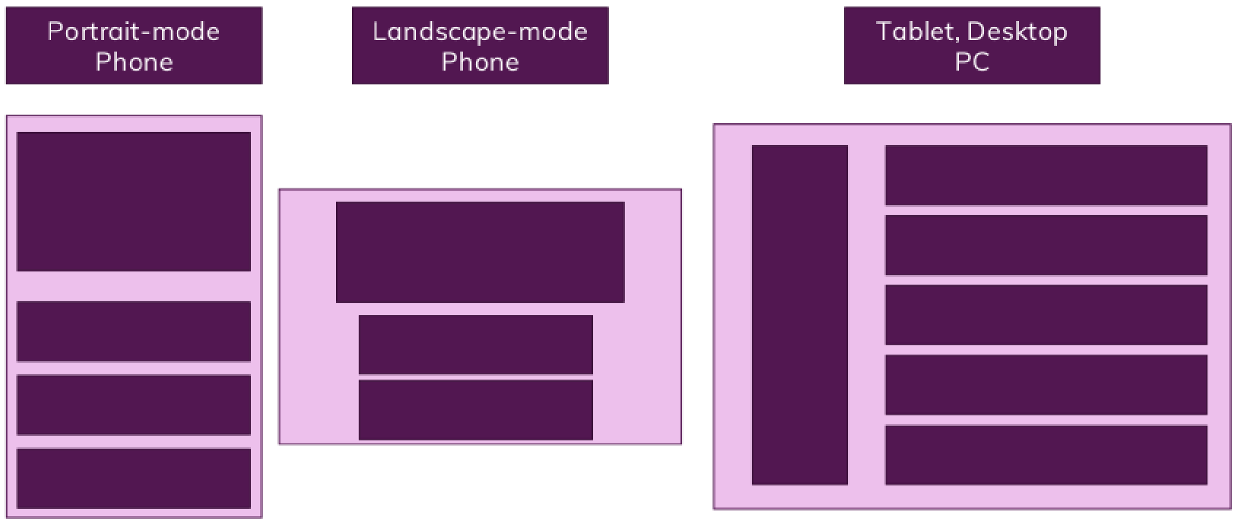
Chapter 4: Responsive and Adaptive UI

4.1. What does responsive and adaptive mean?

What does "Responsive" mean?

Responsive design is a graphic user interface (GUI) design approach used to create content that adjusts smoothly to various screen sizes. Designers size elements in relative units (%) and apply media queries, so their designs can automatically adapt to the mobile screen to ensure content consistency across devices.

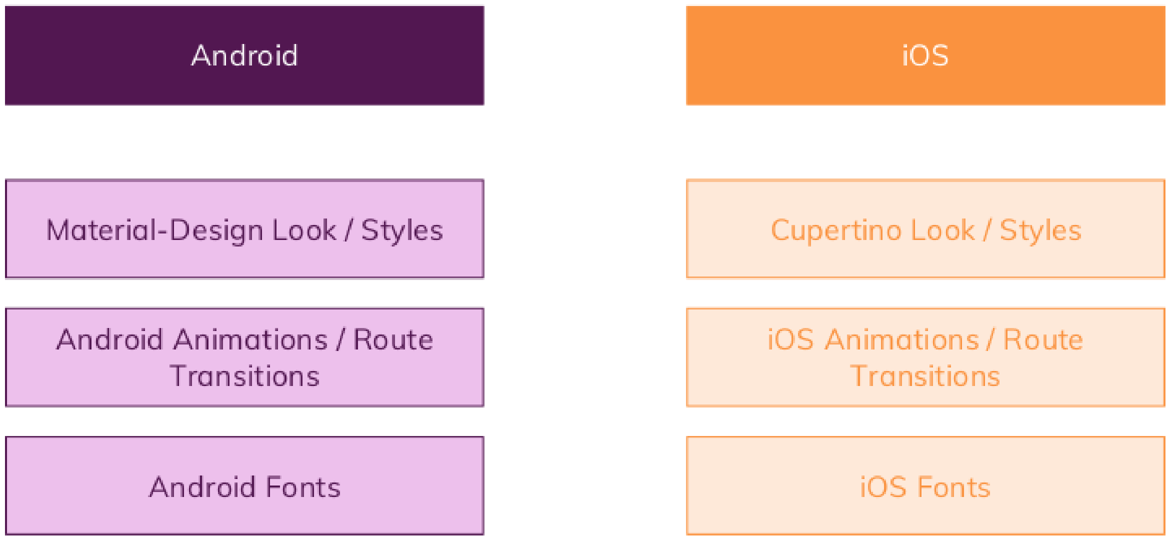


What does "Adaptive" mean?

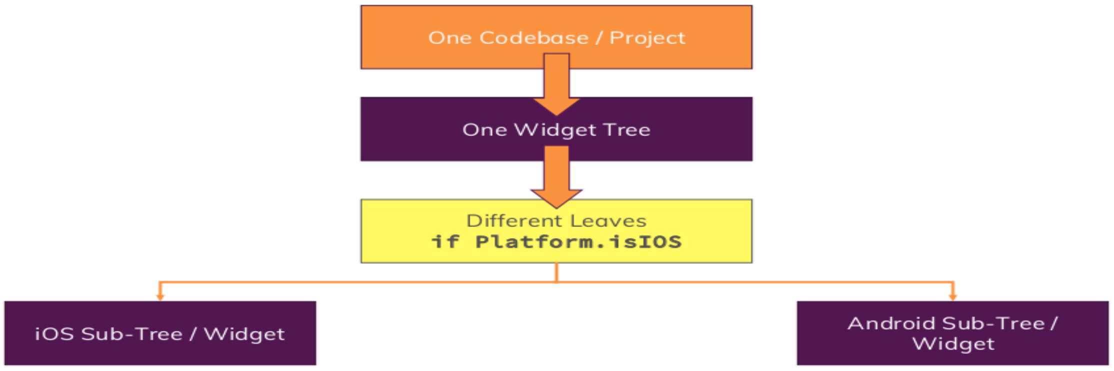
Flutter provides new opportunities to build apps that can run on mobile, desktop, and the web from a single codebase. However, with these opportunities, come new challenges. You want your app to feel familiar to users, adapting to each platform by maximizing usability and ensuring a comfortable and seamless experience. It means that you have a certain look on Android and you want to have a certain probably different look and feel on iOS. Still, of course, you want to use one and the same project and one and the same codebase. So, Flutter gives you tools to adjust the look of your app or the behavior of your app depending on the platform.

For Android, you of course typically want that material design look and the material design styles which you have in Flutter apps out of the box, so Android is the default operating system Flutter targets, probably because Flutter is developed by a Google team. You of course also want to have the typical Android animations when you're switching between different screens. you might also want to use certain Android-specific fonts if you're not bringing your own fonts, as we are actually doing it.

And on iOS, unsurprisingly you want to have that iOS or Cupertino look in your app. Cupertino because Apple sits in Cupertino and therefore this iOS look, the iOS design language is often also referred to as Cupertino design, Cupertino styling. You will obviously also want to have typical iOS animations when you're switching between pages and by the way, it's not just about animations, it's also tiny details, for example on iOS apps, you can swipe with your finger to go back and forth between different pages in your app. On Android, that's typically not supported, you could bring either feeling, either user experience to the other platform, you could add swiping to Android and you can get rid of it on iOS but typically you want to preserve that default behavior users of the platform are used to to make your app feel familiar to them.


Android
iOS
Material-Design Look / Styles
Cupertino Look / Styles
Android Animations / Route Transitions
iOS Animations / Route Transitions
Android Fonts
iOS Fonts

So that is what responsive and adaptive means and of course, our overall goal in Flutter is that we still only use one codebase, one project. We don't want to start building totally different apps just for the different platforms because then, we would be almost at the same point we are if we just use the native programming languages for the platforms.

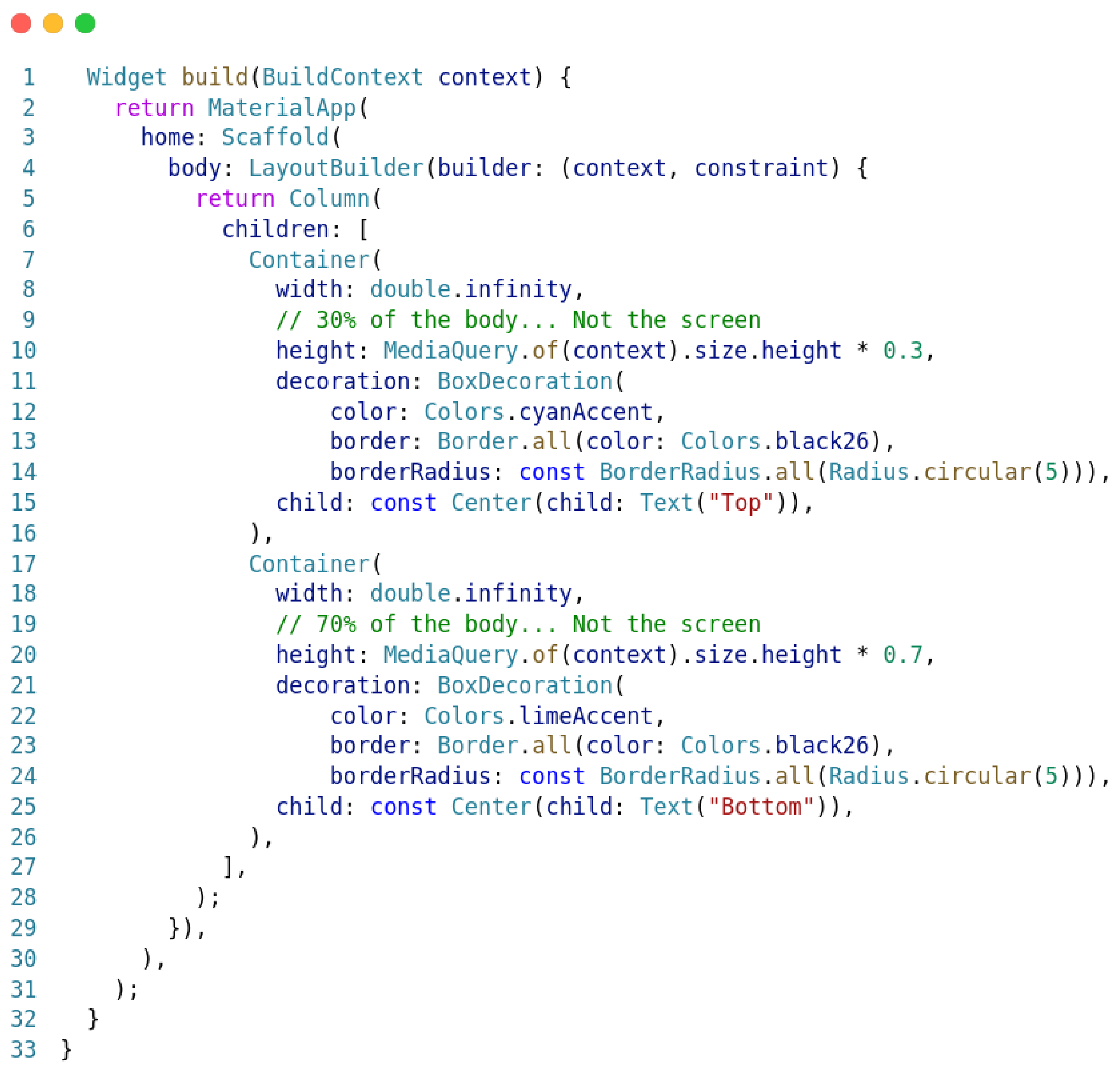


2.2. Calculating Sizes Dynamically using MediaQuery

During the process of developing an app for both phones and tablets, it is standard practice to have different UI layouts for different screen sizes for a better user experience. If the user has a preference set for different font sizes or wants to curtail animations. This is where MediaQuery comes into action, you can get information about the current device size, as well as user preferences, and design your layout accordingly. MediaQuery provides a higher-level view of the current app's screen size and can also give more detailed information about the device and its layout preferences. In practice, MediaQuery is always there. It can simply be accessed by calling ***MediaQuery.of*** in the build method.

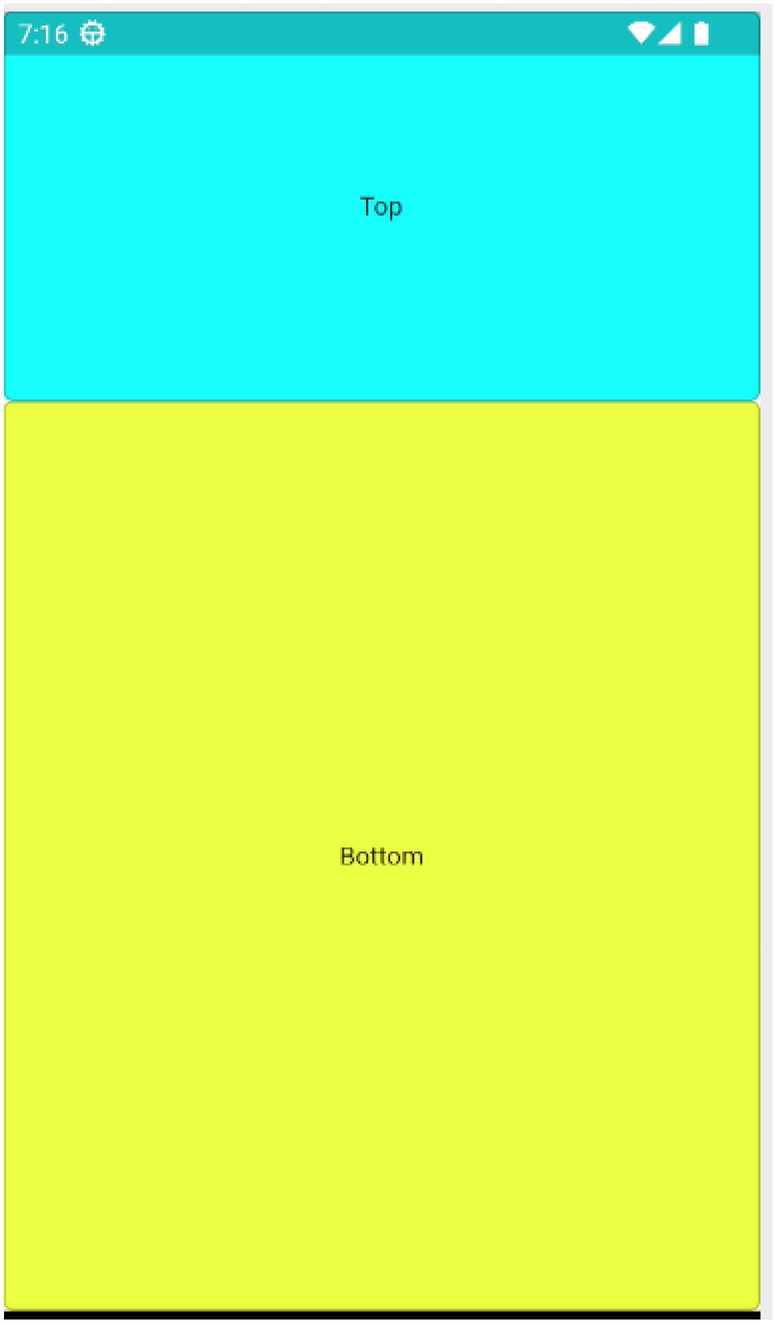
From there you can look up all sorts of interesting information about the device you're running on, like the size of the screen, and build your layout accordingly. MediaQuery can also be used to check the current device's orientation or can be used to check if the user has modified the default font size. It can also be used to determine if parts of the screen are obscured by a system UI, similar to a safe area widget.

Example:



```dart
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: LayoutBuilder(builder: (context, constraint) {
          return Column(
            children: [
              Container(
                width: double.infinity,
                // 30% of the body... Not the screen
                height: MediaQuery.of(context).size.height * 0.3,
                decoration: BoxDecoration(
                    color: Colors.cyanAccent,
                    border: Border.all(color: Colors.black26),
                    borderRadius: const BorderRadius.all(Radius.circular(5))),
                child: const Center(child: Text("Top")),
              ),
              Container(
                width: double.infinity,
                // 70% of the body... Not the screen
                height: MediaQuery.of(context).size.height * 0.7,
                decoration: BoxDecoration(
                    color: Colors.limeAccent,
                    border: Border.all(color: Colors.black26),
                    borderRadius: const BorderRadius.all(Radius.circular(5))),
                child: const Center(child: Text("Bottom")),
              ),
            ],
          );
        }),
      ),
    );
  }
}
```

So the result looks like:



2.3. LayoutBuilder widget

In Flutter, *LayoutBuilder* Widget is similar to the Builder widget except that the framework calls the builder function at layout time and provides the parent widget's **constraints**. This is useful when the parent constrains the child's size and doesn't depend on the child's intrinsic size. The LayoutBuilder's final size will match its child's size.

The builder function is called in the following situations:

- The first time the widget is laid out.
- When the parent widget passes different layout constraints.
- When the parent widget updates this widget.
- When the dependencies that the builder function subscribes to change.

Syntax:

```
LayoutBuilder(
  builder: (BuildContext context, BoxConstraints constraints) {
    return Widget();
  }
)
```

2.4. Controlling Device Orientation

All the applications developers create in flutter run on multiple devices with a variety of screen pixels and sizes. Managing screen orientation in flutter plays an important role in establishing a successful app that runs on multiple devices. In flutter, you can pick the best way to deal with such challenges of device orientation in an easy and handy way. In this article, we will walk you through Screen Orientation in Flutter.

What is the need for orientation?

To learn orientation management, first, you need to understand the need to do so. Screen orientation management improves the user experience to a great extent. When you change the screen orientation, the action currently running over the app changes and re-builds itself according to the new orientation. For example, Changing orientation to landscape mode while playing a car racing game.

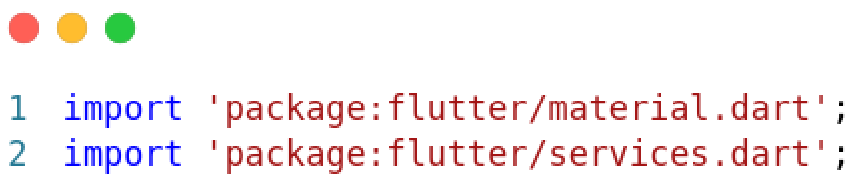
How to set device orientation?

Flutter provides namely 2 ways to orient the device namely:

1. Portrait
2. LandScape

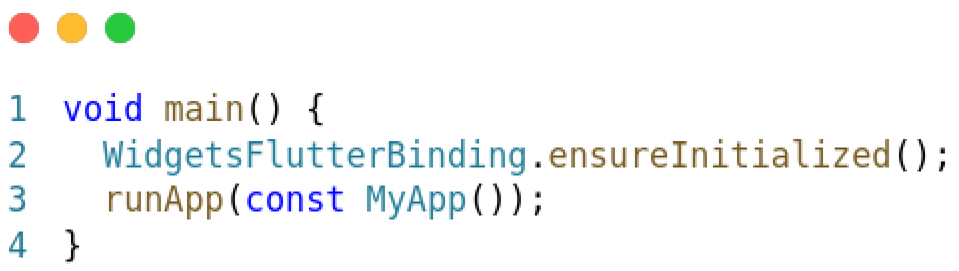
Given below are the steps illustrated to change your mobile application orientation according to your suitability.

Step 1: Include the **services packages** in your main file.



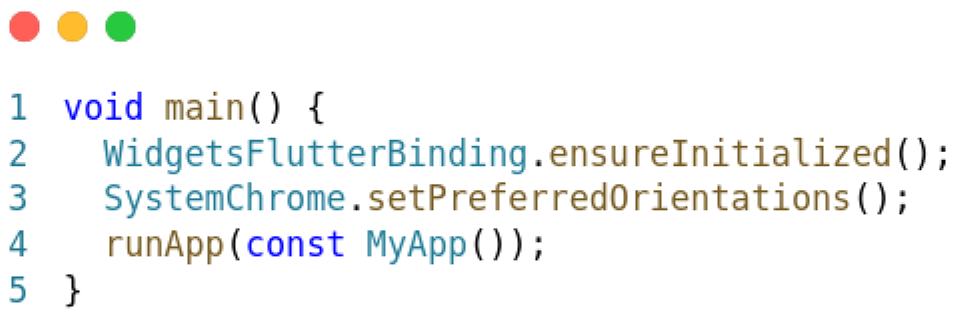
```dart
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
```

Step 2: Call "**WidgetsFlutterBinding.ensureInitialized()**" method after the main function call.



```dart
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  runApp(const MyApp());
}
```

Step 3: Call " **SystemChrome.setPreferredOrientations**" after your "**WidgetsFlutterBinding.ensureInitialized()**" method.

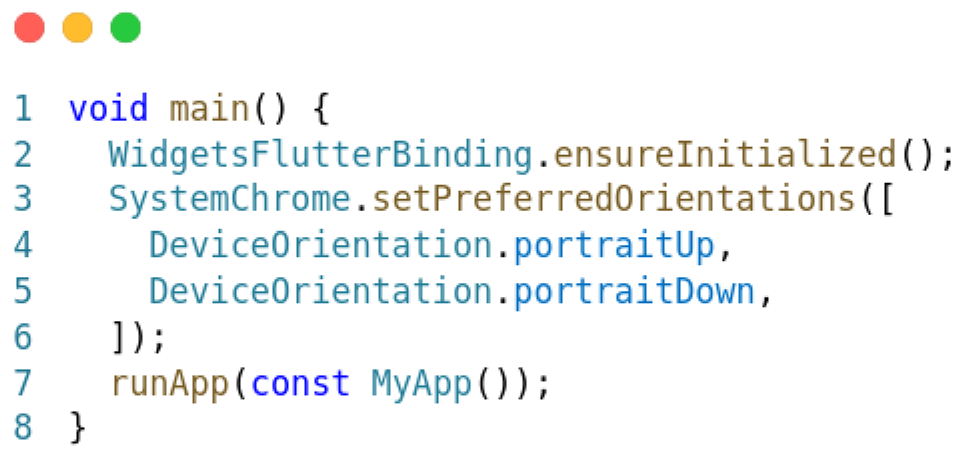


```dart
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  SystemChrome.setPreferredOrientations();
  runApp(const MyApp());
}
```

Now to set a portrait mode, flutter provides two orientation options namely:

- DeviceOrientation.portraitUp,

- DeviceOrientation.portraitDown,



```
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
    DeviceOrientation.portraitDown,
  ]);
  runApp(const MyApp());
}
```

To set landscape mode, flutter provides two orientation options:

- DeviceOrientation.landscapeLeft
- DeviceOrientation.landscapeRight

4.5. SafeArea Widget

SafeArea is an important and useful widget in Flutter which makes UI dynamic and adaptive to a wide variety of devices. While designing the layout of widgets, we consider different types of devices and their pre-occupied constraints of screen like status bar, notches, navigation bar, etc. But new devices are being launched with different designs and in certain scenarios, your app might overlay any of those pre-occupied constraints. So, in order to make our UI adaptive and error-free, we use *SafeArea* widget.

In simple words, *SafeArea* is basically a padding widget, which adds any necessary padding to your app, based on the device it is running on. If your app's widgets are overlapping any of the system's features like notches, status bar, camera holes, or any other such features, then *SafeArea* would add padding around the app, as required.

Internally *SafeArea* uses *MediaQuery* to check the dimensions of the display screen and includes extra padding if needed.

Constructor :

```
const SafeArea({
   Key key,
   bool left: true,
   bool top: true,
   bool right: true,
   bool bottom: true,
   EdgeInsets minimum: EdgeInsets.zero,
   bool maintainBottomViewPadding: false,
   @required Widget child}
)
```

Above shown is the constructor for *SafeArea.* You can decide whether to avoid intrusions in a particular direction by changing the boolean value to true or false.

For instance, if you want to use *SafeArea* in only top and bottom directions, then you can specify in the following way.

```
SafeArea(
    left: false,
    top: true,
    bottom: true,
    right: false,
    child: Text('Your Widget')
)
```

The above code would add padding only at top and bottom while other directions (left and right) would remain unaffected. If you don't specify any directions, then the default would be true for all directions.

If you want to add minimum padding in any of the directions, you can do it in the following way:

```
SafeArea(
   minimum: const EdgeInsets.all(15.0),
   child: Text('Your Widget'),
)
```

In the above code snippet, we are specifying the minimum padding that we need to add in all the directions. If you don't specify this Flutter would automatically calculate the required padding for all the directions.

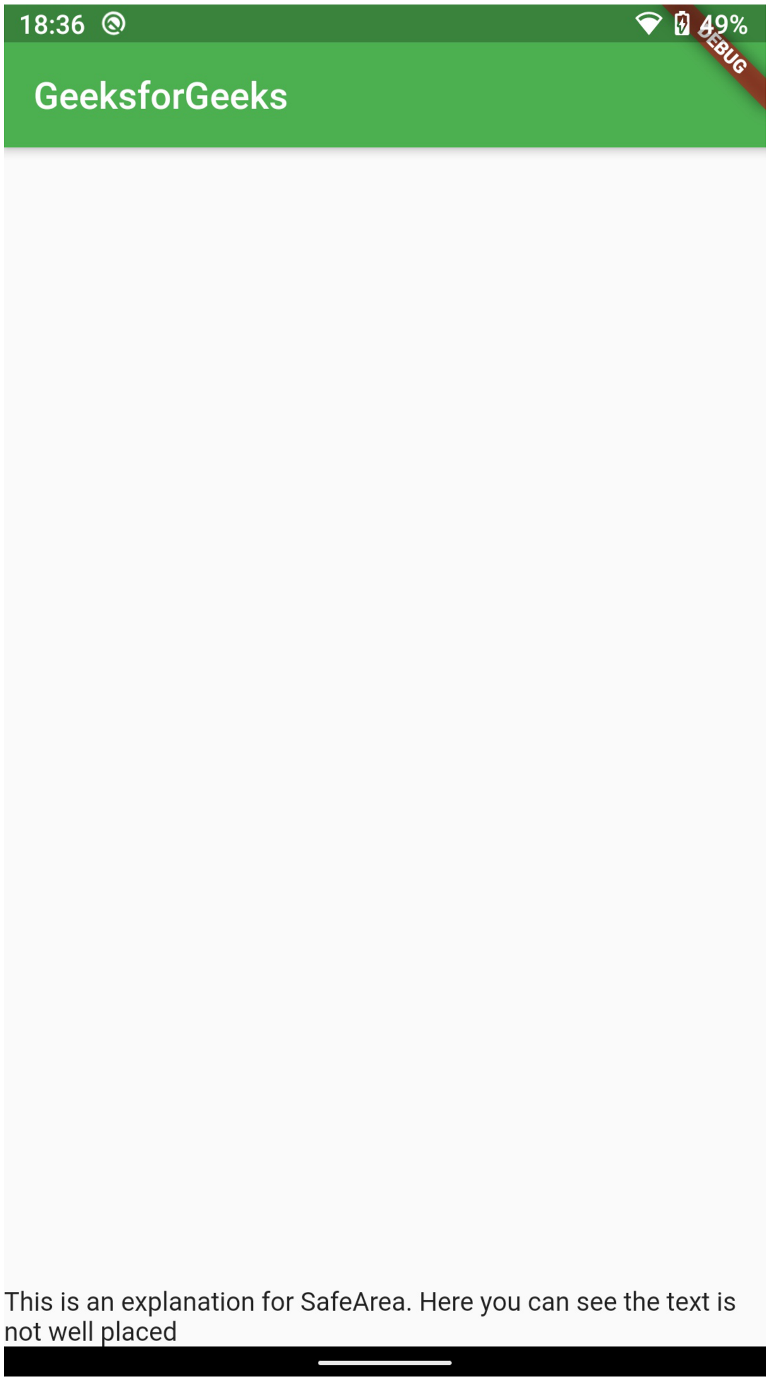
Properties :

- **bottom :** This property is of type bool. It is true by default and setting it to false would disable *SafeArea* from adding padding to the bottom of the screen.
- **top :** This property is also of type bool and setting it to false would avoid padding at top of the screen.
- **left :** This property is of type bool and setting it to false would avoid padding at left side of the screen.
- **right** : This property is of type bool and setting it to false would avoid padding at right side of the screen.
- **minimum :** This property is of type *EdgeInsets*. You can specify the minimum padding to be added using this property.
- **maintainBottomViewPadding :** This property is of type bool and it specifies whether *SafeArea* should maintain the *viewPadding* instead of *padding*. For instance, if you are using an on-screen keyboard with *SafeArea* , the the padding can be maintained below the obstruction rather than being consumed.

Explanation :

Now we'll see how the above-mentioned properties work and when to use them.

Suppose, you are displaying a Text widget or any other widget at the bottom end of the screen, then on your device it might work, but on other devices, it might not be properly formatted.

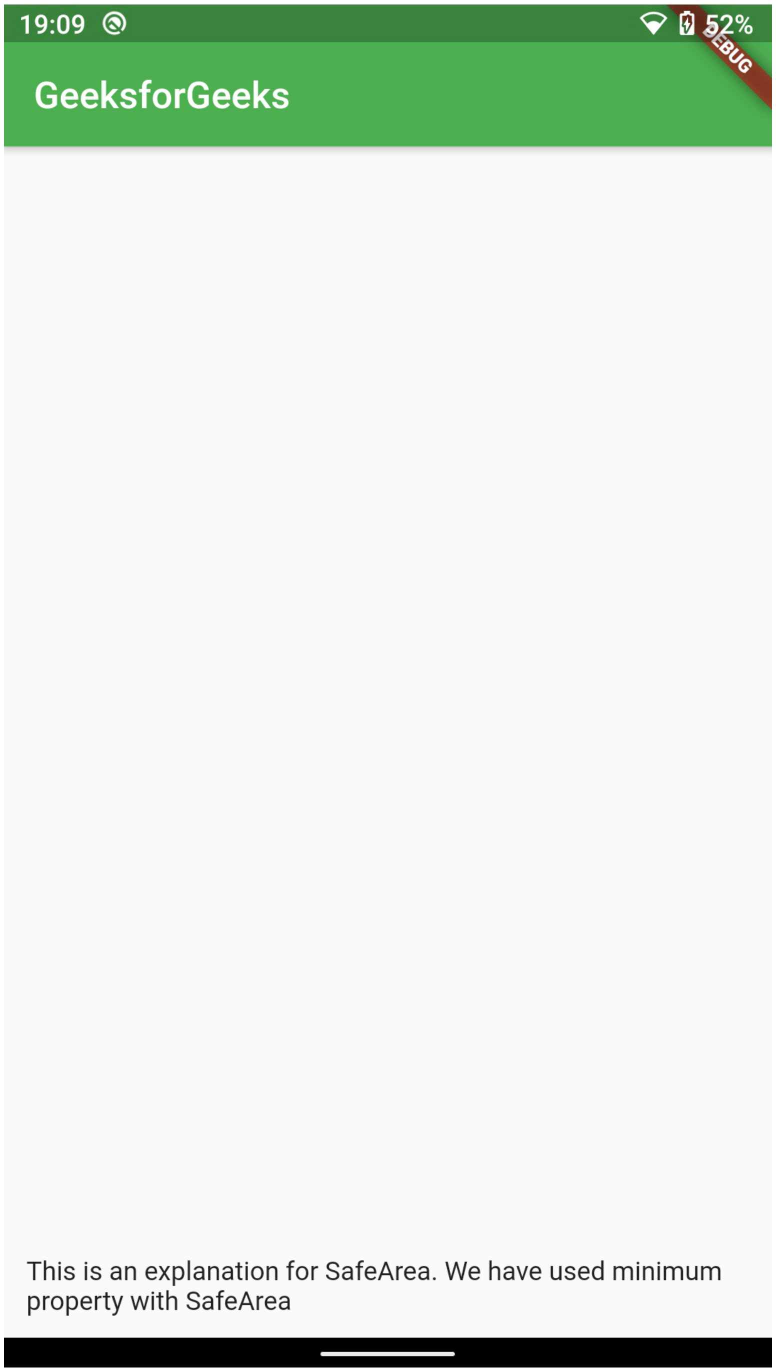


This is an explanation for SafeArea. Here you can see the text is not well placed

You can see that in the above app the text widget is not properly formatted. The above-shown device is rectangular in shape, but if you test the same app on an iOS device, then the text might cover the app drawer. As a developer, your app should be free from all bugs on all devices. In this case, we will use *SafeArea.*

Refer to below example :

```
SafeArea(
   minimum: const EdgeInsets.all(12.0),
   child: Text('Your Widget'),
)
```



This is an explanation for SafeArea. We have used minimum property with SafeArea