SILVER OAK UNIVERSITY

EDUCATION TO INNOVATION

Subject: **Flutter Framework**

Chapter 3: Flutter Basics

Unit-name : Flutter Basics

Topics : Concept Of State
State management in Flutter
Type of Layout
Layout a Widget
Flutter Navigation and Routing

Concept of State

What is the State:

A state is information that can be **read** when the widget is built and might **change or modified** over a lifetime of the app. If you want to change your widget, you need to update the state object, which can be done by using the setState() function available for Stateful widgets. The **setState()** function allows us to set the properties of the state **object** that triggers a redraw of the UI.

We know that in Flutter, everything is a widget. The widget can be classified into two categories, one is a **Stateless widget,** and another is a **Stateful widget.** The Stateless widget does not have any internal state. It means once it is built, we cannot change or modify it until they are initialized again. On the other hand, a Stateful widget is dynamic and has a state. It means we can modify it easily throughout its lifecycle without reinitialized it again.

State management in Flutter

State management: The state management is one of the most popular and necessary processes in the lifecycle of an application. According to official documentation, Flutter is declarative. It means Flutter builds its UI by reflecting the current state of your app. The following figure explains it more clearly where you can build a UI from the application state.

Example: Suppose you have created a list of customers or products in your app. Now, assume you have added a new customer or product dynamically in that list. Then, there is a need to refresh the list to view the newly added item into the record. Thus, whenever you add a new item, you need to refresh the list. This type of programming requires state management to handle such a situation to improve performance. It is because every time you make a change or update the same, the state gets refreshed.

In Flutter, the state management categorizes into two conceptual types, which are given below:

1. Ephemeral State
2. App State

Ephemeral State

This state is also known as UI State or local state. It is a type of state which is related to the **specific widget,** or you can say that it is a state that contains in a single widget. In this kind of state, you do not need to use state management techniques. The common example of this state is **Text Field**.

```
class MyHomepage extends StatefulWidget {
  @override
  MyHomepageState createState() => MyHomepageState();
}

class MyHomepageState extends State<MyHomepage> {
  String _name = "Peter";
  @override
  Widget build(BuildContext context) {
    return RaisedButton(
      child: Text(_name),
      onPressed: () {
        setState(() {
          _name = _name == "Peter" ? "John" : "Peter";
        });
      },
    );
  }
}
```

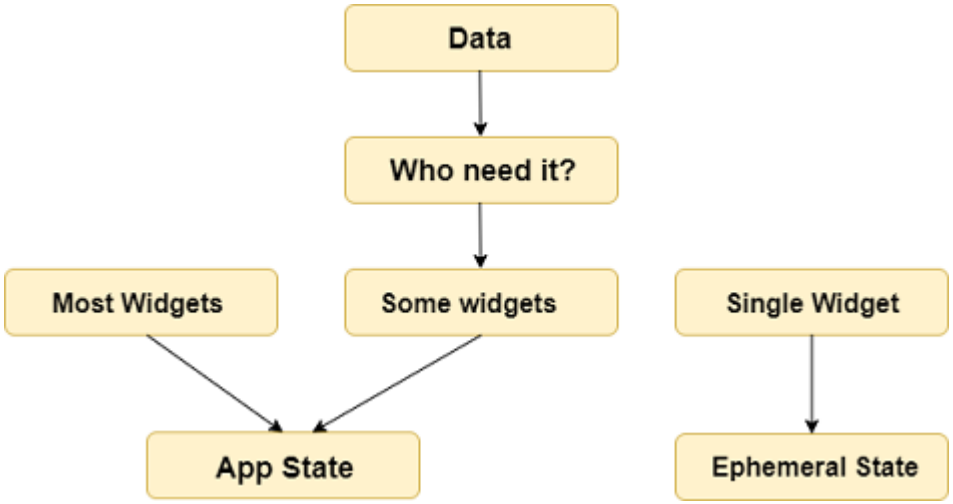
In the above example, the **_name** is an ephemeral state. Here, only the setState() function inside the StatefulWidget's class can access the _name. The build method calls a setState() function, which does the modification in the state variables. When this method is executed, the widget object is replaced with the new one, which gives the modified variable value.

App State

It is different from the ephemeral state. It is a type of state that we want to **share** across various parts of our app and want to keep between user sessions. Thus, this type of state can be used globally. Sometimes it is also known as application state or shared state.

Some of the examples of this state are User preferences, Login info, notifications in a social networking app, the shopping cart in an e-commerce app, read/unread state of articles in a news app, etc.

The following diagram explains the difference between the ephemeral state and the app state more appropriately.



The simplest example of app state management can be learned by using the **provider package.** The state management with the provider is easy to understand and requires less coding. A provider is a **third-party** library. Here, we need to understand three main concepts to use this library.

1. ChangeNotifier
2. ChangeNotifierProvider
3. Consumer

ChangeNotifier

ChangeNotifier is a simple class, which provides change notification to its listeners. It is easy to understand, implement, and optimized for a small number of listeners. It is used for the listener to observe a model for changes. In this, we only use the **notifyListener()** method to inform the listeners.

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

class Counter with ChangeNotifier {
  int _counter;

  Counter(this._counter);

  getCounter() => _counter;
  setCounter(int counter) => _counter = counter;

  void increment() {
    _counter++;
    notifyListeners();
  }

  void decrement() {
    _counter--;
    notifyListeners();
  }
}
```

ChangeNotifierProvider

ChangeNotifierProvider is the widget that provides an **instance** of a ChangeNotifier to its descendants. It comes from the provider package. The following code snippets help to understand the concept of ChangeNotifierProvider.

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(
        primarySwatch: Colors.indigo,
      ),
      home: ChangeNotifierProvider<CounterModel>(
        builder: (_) => CounterModel(),
        child: CounterView(),
      ),
    );
  }
}
```

Consumer

It is a type of provider that does not do any fancy work. It just calls the provider in a new widget and delegates its build implementation to the builder. The following code explains it more clearly./p>

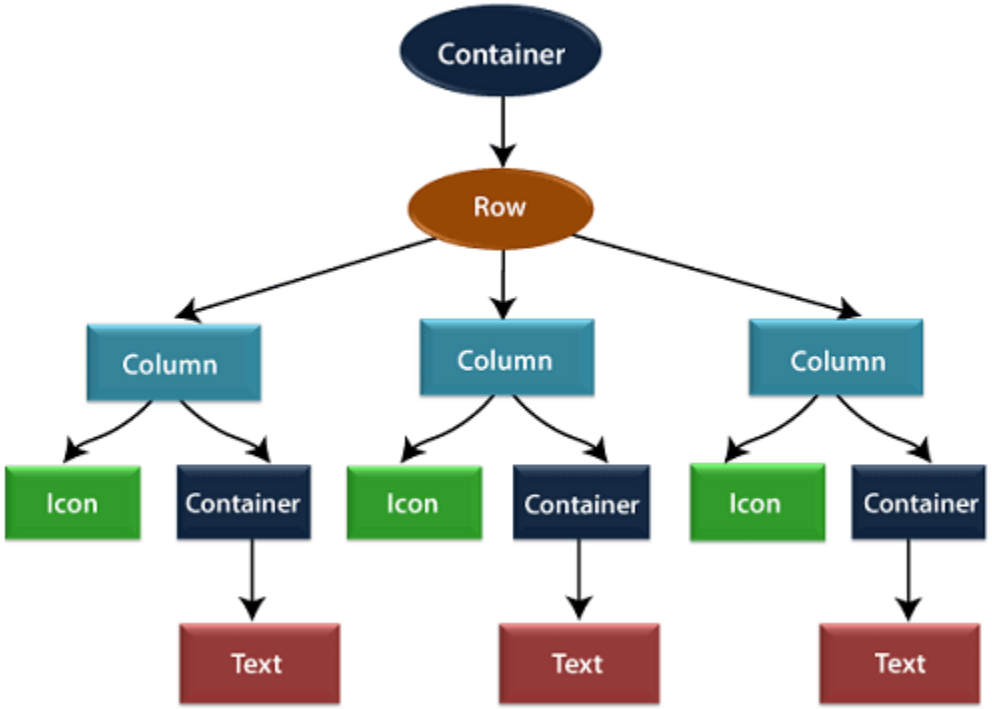
```
return Consumer<Counter>(
  builder: (context, count, child) {
    return Text("Total price: ${count.total}");
  },
);
```

Flutter Layouts

The main concept of the layout mechanism is the widget. We know that flutter assume everything as a widget. So the image, icon, text, and even the layout of your app are all widgets. Here, some of the things you do not see on your app UI, such as rows, columns, and grids that arrange, constrain, and align the visible widgets are also the widgets.

Flutter allows us to create a layout by composing multiple widgets to build more complex widgets. **For example**, we can see the below image that shows three icons with a label under each one.

In the second image, we can see the visual layout of the above image. This image shows a row of three columns, and these columns contain an icon and label.



In the above image, the **container** is a widget class that allows us to customize the child widget. It is mainly used to add borders, padding, margins, background color, and many more. Here, the text widget comes under the container for adding margins. The entire row is also placed in a container for adding margin and padding around the row. Also, the rest of the UI is controlled by properties such as color, text.style, etc.

Layout a widget

Let us learn how we can create and display a simple widget. The following steps show how to layout a widget:
Step 1: First, you need to select a Layout widget.
Step 2: Next, create a visible widget.
Step 3: Then, add the visible widget to the layout widget.
Step 4: Finally, add the layout widget to the page where you want to display.

Types of Layout Widgets

We can categories the layout widget into two types:

1. Single Child Widget
2. Multiple Child Widget

Single Child Widgets

The single child layout widget is a type of widget, which can have only **one child widget** inside the parent layout widget. These widgets can also contain special layout functionality. Flutter provides us many single child widgets to make the app UI attractive. If we use these widgets appropriately, it can save our time and makes the app code more readable. The list of different types of single child widgets are:

Container: It is the most popular layout widget that provides customizable options for painting, positioning, and sizing of widgets.

```
 Container(
   margin: const EdgeInsets.all(15.0),
   color: Colors.blue,
   width: 42.0,
   height: 42.0,
 ),
```

Padding: It is a widget that is used to arrange its child widget by the given padding. It contains **EdgeInsets**.

```
Padding(
    padding: EdgeInsets.all(14.0),
    child: Text('Hello JavaTpoint!'),
  ),
```

Center: This widget allows you to center the child widget within itself.

Align: It is a widget, which aligns its child widget within itself and sizes it based on the child's size. It provides more control to place the child widget in the exact position where you need it

```
Center(
  child: Container(
    height: 110.0,
    width: 110.0,
    color: Colors.blue,
    child: Align(
      alignment: Alignment.topLeft,
      child: FlutterLogo(
        size: 50,
      ),
    ),
  ),
)
```

SizedBox: This widget allows you to give the specified size to the child widget through all screens.

```
SizedBox(
  width: 300,
  height: 450,
  child:  Card(child: Text('Hello JavaTpoint!')),
)
```

AspectRatio: This widget allows you to keep the size of the child widget to a specified aspect ratio.

```
AspectRatio(
  aspectRatio: 5/3,
  child: Container(
    color: Colors.bluel,
  ),
),
```

Baseline: This widget shifts the child widget according to the child's baseline.

```
  child: Baseline(
         baseline: 30.0,
         baselineType: TextBaseline.alphabetic,
         child: Container(
              height: 60,
              width: 50,
              color: Colors.blue,),)
```

ConstrainedBox: It is a widget that allows you to force the additional constraints on its child widget. It means you can force the child widget to have a specific constraint without changing the properties of the child widget.

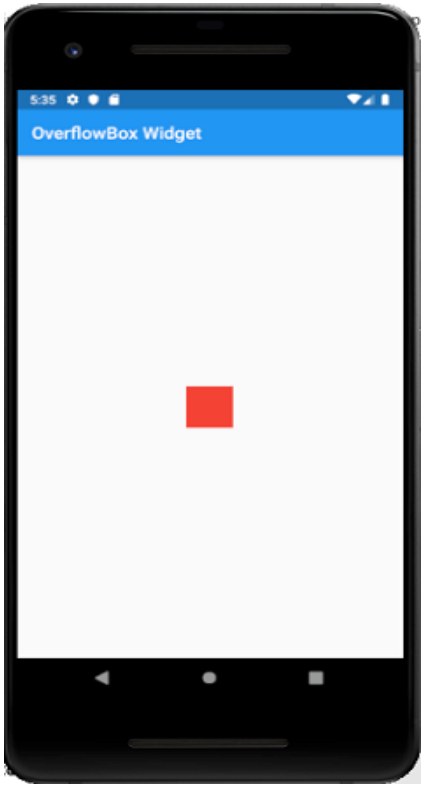
```
ConstrainedBox(
  constraints: new BoxConstraints(
    minHeight: 150,
    minWidth: 150,
    maxHeight: 300,
    maxWidth: 300,
  ),
  child: new DecoratedBox(
    decoration: new BoxDecoration(color: Colors.red),
  ),),
```

- **CustomSingleChildLayout:** It is a widget, which defers from the layout of the single child to a delegate. The delegate decides to position the child widget and also used to determine the size of the parent widget.
- **FittedBox:** It scales and positions the child widget according to the specified **fit**.
- **FractionallySizedBox:** It is a widget that allows to sizes of its child widget according to the fraction of the available space.
- **IntrinsicHeight and IntrinsicWidth:** They are a widget that allows us to sizes its child widget to the child's intrinsic height and width.
- **LimitedBox:** This widget allows us to limits its size only when it is unconstrained.
- **Offstage:** It is used to measure the dimensions of a widget without bringing it on to the screen.
- **OverflowBox:** It is a widget, which allows for imposing different constraints on its child widget than it gets from a parent. In other words, it allows the child to overflow the parent widget.

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

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

class MyApp extends StatelessWidget {
  // It is the root widget of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Single Layout Widget',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
```

```dart
class MyHomePage extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("OverflowBox Widget"),
      ),
      body: Center(
      child: Container(
        height: 50.0,
        width: 50.0,
        color: Colors.red,
        child: OverflowBox(
          minHeight: 70.0,
          minWidth: 70.0,
          child: Container(
            height: 50.0,
            width: 50.0,
            color: Colors.blue,
            ),
          ),
        ),
      ),
    );
  }
}
```

Output:



Multiple Child widgets

The multiple child widgets are a type of widget, which contains **more than one child widget**, and the layout of these widgets are **unique**. For example, Row widget laying out of its child widget in a horizontal direction, and Column widget laying out of its child widget in a vertical direction. If we combine the Row and Column widget, then it can build any level of the complex widget.

Here, we are going to learn different types of multiple child widgets:

Row: It allows to arrange its child widgets in a horizontal direction.

```
Row(
      children: [
        Expanded(
          child: Text('Peter', textAlign: TextAlign.center),
        ),
        Expanded(
          child: Text('Parker', textAlign: TextAlign.center ),

        ),
        Expanded(
          child: FittedBox(
            fit: BoxFit.contain, // otherwise the logo will be tiny
            child:  FlutterLogo(),
          ),
        ),
      ],
    ),
```

Column: It allows to arrange its child widgets in a vertical direction.

ListView: It is the most popular scrolling widget that allows us to arrange its child widgets one after another in scroll direction.

GridView: It allows us to arrange its child widgets as a scrollable, 2D array of widgets. It consists of a repeated pattern of cells arrayed in a horizontal and vertical layout.

Expanded: It allows to make the children of a Row and Column widget to occupy the maximum possible area.

Table: It is a widget that allows us to arrange its children in a table based widget.

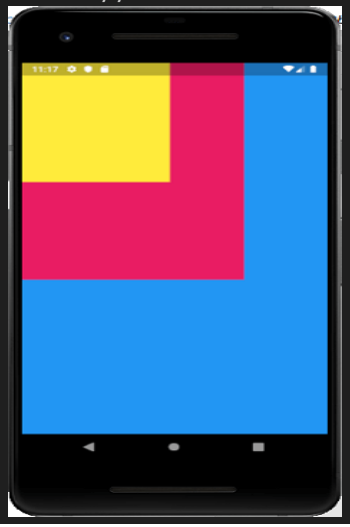
Flow: It allows us to implements the flow-based widget.

Stack: It is an essential widget, which is mainly used for overlapping several children widgets. It allows you to put up the multiple layers onto the screen. The following example helps to understand it.

```
import 'package:flutter/material.dart';
  void main() => runApp(MyApp());
  class MyApp extends StatelessWidget {
  // It is the root widget of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Multiple Layout Widget',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        // This is the theme of your application.
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(),
    );
  }
}
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        alignment: Alignment.center,
        color: Colors.white,
```

```
        child: Stack(
          children: <Widget>[
            // Max Size
            Container(
              color: Colors.blue,
            ),
            Container(
              color: Colors.pink,
              height: 400.0,
              width: 300.0,
            ),
            Container(
              color: Colors.yellow,
              height: 220.0,
              width: 200.0,
            )
],),),);}}
```

//OUTPUT



Flutter Navigation and Routing

Navigation and routing are some of the core concepts of all mobile application, which allows the user to move between different pages. We know that every mobile application contains several screens for displaying different types of information. **For example,** an app can have a screen that contains various products. When the user taps on that product, immediately it will display detailed information about that product.

In Flutter, the screens and pages are known as **routes,** and these routes are just a widget. In Android, a route is similar to an **Activity,** whereas, in iOS, it is equivalent to a **ViewController.**

In any mobile app, navigating to different pages defines the workflow of the application, and the way to handle the navigation is known as **routing.** Flutter provides a basic routing class **MaterialPageRoute** and two methods **Navigator.push()** and **Navigator.pop()** that shows how to navigate between two routes. The following steps are required to start navigation in your application.

Step 1: First, you need to create two routes.
Step 2: Then, navigate to one route from another route by using the Navigator.push() method.
Step 3: Finally, navigate to the first route by using the Navigator.pop() method.
Let us take a simple example to understand the navigation between two routes:

Create two routes

Here, we are going to create two routes for navigation. In both routes, we have created only a **single button.** When we tap the button on the first page, it will navigate to the second page. Again, when we tap the button on the second page, it will return to the first page. The below code snippet creates two routes in the Flutter application.

```
class FirstRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Route'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Open route'),
          onPressed: () {
            // Navigate to second route when tapped.
          },
        ),
      ),
    );}}
class SecondRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Route"),
      ),
      body: Center(
        child: RaisedButton(
          onPressed: () {
            // Navigate back to first route when tapped.
          },
          child: Text('Go back!'),
        ),
      ),);}}
```

```
// Within the `FirstRoute` widget
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SecondRoute()),
  );
}
```

Return to the first route using Navigator.pop() method

Now, we need to use Navigator.pop() method to close the second route and return to the first route. The **pop()** method allows us to remove the current route from the stack, which is managed by the Navigator.

To implement a return to the original route, we need to update the **onPressed()** callback method in the SecondRoute widget as below code snippet:
Now, let us see the full code to implement the navigation between two routes. First, create a Flutter project and insert the following code in the **main.dart** file.

```
  // Within the SecondRoute widget
  onPressed: () {
    Navigator.pop(context);
  }
```

Now, let us see the full code to implement the navigation between two routes. First, create a Flutter project and insert the following code in the **main.dart** file.

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

void main() {
  runApp(MaterialApp(
    title: 'Flutter Navigation',
    theme: ThemeData(
```

```
      // This is the theme of your application.
      primarySwatch: Colors.green,
    ),
    home: FirstRoute(),
  ));
}


class FirstRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First Screen'),
      ),
      body: Center(
        child: RaisedButton(
          child: Text('Click Here'),
          color: Colors.orangeAccent,
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondRoute()),
            );
          },
        ),
      ),
    );
  }
}
```

```
class SecondRoute extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Second Screen"),
      ),
      body: Center(
        child: RaisedButton(
          color: Colors.blueGrey,
          onPressed: () {
            Navigator.pop(context);
          },
          child: Text('Go back'),
        ),
      ),
    );
  }
}
```

Output

When you run the project in the **Android Studio,** you will get the following screen in your emulator. It is the first screen that contains only a single button.

Click the button **Click Here,** and you will navigate to a second screen as below image. Next, when you click on the button **Go Back,** you will return to the first page.