LECTURE NOTES

ON

Mobile Application Development using Flutter

2101CS402

B.Tech. 4th Semester

(Computer Science and Engineering)

Prepared By

Mehul D. Bhundiya

Assistant Professor

mehul.bhundiya@darshan.ac.in

7984991167

School of Computer Science

Darshan UNIVERSITY

योगः कर्मसु कौशलम्

Darshan UNIVERSITY
योगः कर्मसु कौशलम्

Table of Content

Unit-1 3

Introduction to Flutter 3

1.1. Dart Programming - Overview 3

1.2. Write command-line apps 4

1.3. Variables 4

1.4. Sound null safety 6

1.5. Operators 7

1.6. Control flow 9

1.7. Function 11

1.8. Class 15

1.9. Object 16

1.10. Assessing Variable and Function 16

1.11. Flutter Introduction 17

1.12. Flutter Architecture 17

1.13. Pubspec file in flutter 19

1.14. Flutter widgets 21

Unit-2 22

UI Design, State Management, Navigation 22

2.1. Scaffold 22

2.2. Safearea 22

2.3. Text 23

2.4. Row 23

2.5. Column 24

2.6. Stack 24

2.7. Container 25

2.8. Material Components 25

2.9. Flutter App Life Cycle 26

2.10. MaterialPageRoute 30

2.11. Navigation with Named Routes 33

Unit-1
Introduction to Flutter

1.1. Dart Programming - Overview

- Dart is Object-oriented language and is quite similar to that of Java Programming
- Dart is a client-optimized language for developing fast apps on any platform like Stand-alone, Web and Mobile.
- Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks.
- Dart is the open-source programming language originally developed by Google. It is meant for both server side as well as the user side.
- The Dart SDK comes with its compiler – the Dart VM and a utility dart2js.
- Which is meant for generating JavaScript equivalent of a Dart Script so that it can be run on those sites also which don't support Dart.

1.1.1. Why you use Dart

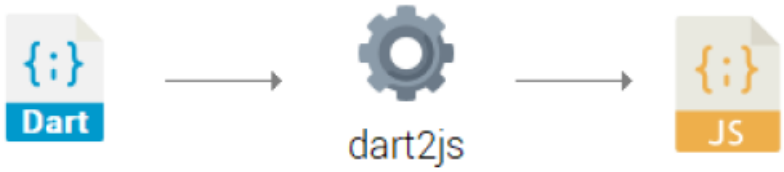
- Google created Dart and uses it internally with some of its big products such as Google AdWords.
- Made available publicly in 2011, Dart is used to build mobile, web, and server applications.
- Dart is productive, fast, portable, approachable, and most of all reactive.

1.1.2. Supported platforms

- Dart is a very flexible language, Once the source code has been written and tested it can be deployed in many different ways:
- ***Stand-alone: -***
    - Dart program is executed with the Dart Virtual Machine (DVM).
    - There's the need to download and install the DVM which to execute Dart in a command-line environment.
- ***AOT Complied: -***
    - The **A**head **O**f **T**ime compilation is the act of translating a high-level programming language, like Dart, into native machine code.
    - Basically, starting from the Dart source code you can obtain a single binary file that can execute natively on a certain operating system.
    - AOT is really what makes Flutter fast and portable.

- ***Web: -***

- Thanks to the dart2js tool, your Dart project can be "transpiled" into fast and compact JavaScript code.
- By consequence Flutter can be run, for example, on Firefox or Chrome and the UI will be identical to the other platforms.



1.1.3. Package system

- Dart's core API offers different packages, such as *dart:io* or *dart:collection*, that expose classes and methods for many purposes.
- In addition, there is an official online repository called pub containing packages created by the Dart team, the Flutter team or community users.
- If you head to https://pub.dev you will find an endless number of packages for any purpose: I/O handling, XML serialization/de-serialization, localization, SQL/NoSQL database utilities and much more.

1.2. Write command-line apps

- In dart main() function is predefined method.
- main() method acts as the entry point to the application.
- A dart script needs the main() method for execution of the code.
- The program code goes like this

```
void main() {
  print("Welcome to Darshan University");
}
```

1.3. Variables

- Variable is used to store the value and refer the memory location in computer memory.
- When we create a variable, the Dart compiler allocates some space in memory.
- The size of the memory block of memory is depended upon the type of variable.

1.3.1. Rule to Create Variable

- In Dart, all variables are declared public by default.
- By starting the variable name with an underscore (_), you can declare it as private.
- By declaring a variable private, you are saying it cannot be accessed from outside classes/functions. it can be used only from within the declaration class/function
- The variable cannot contain special characters such as whitespace, mathematical symbol, Unicode character, and keywords.
- The first character of the variable should be an alphabet (A to Z & a to z).

- Digits are not allowed as the first character.
- The special character such as #, @, ^, &, * are not allowed.
- The variable name should be meaningful.

1.3.2. Number Variable

- Declaring variables as numbers restricts the values to numbers only. Dart allows numbers to be int(integer) or double.
- **int: -** Use the int declaration if your numbers do not require decimal point precision, like 8 or 22.
- **double: -** Use the double declaration if your numbers require decimal point precision, like 8.5 or 88.25.
- Both int and double allow for positive and negative numbers, and you can enter extremely large numbers and decimal precision since they both use 64-bit (computer memory) values.
- e.g.,

```
void main() {
  int age=19;
  double price=25.50;
}
```

1.3.3. Strings Variable

- Declaring variables as String allows values to be entered as a sequence of text characters.
- To add a single line of characters, you can use single or double quotes like 'Darshan University' or "Darshan University".
- e.g.

```
void main() {
  String name = 'Darshan University';
  String name = "Darshan University";
}
```

1.3.4. Booleans Variable

- Declaring variables as bool (Boolean) allows a value of true or false to be entered.
- e.g.

```
void main() {
  bool isDone = false;
  isDone = true;
}
```

1.3.5. Lists

- Declaring variables as List (comparable to arrays) allows multiple values to be entered.
- List is an ordered group of objects.
- The **dart:core** library provides the List class that enables creation and manipulation of lists.

- To access elements, the List uses zero-based indexing, where the first element index is at 0, and the last element is at the List length is minus 1.
- A List can be a **fixed length** or **growable**, depending on your needs.
- By default, a List is created as growable by using List() or [].
- To create a fixed-length List, you add the number of rows required by using this format: List(15).
- e.g.

```
void main() {
  // List Growable
  List days = List();

  // or
  List days = [];
  List days = ['Sunday', 'Monday, 'Tuesday];

  // List fixed-length
  List days = List(7);
}
```

1.3.6. Maps

- The Map data type represents a set of values as key-value pairs.
- Keys and values on a map may be of any data type.
- Mapping allows recalling values by their Key ID.
- Keep in mind that the Key needs to be unique since the Value is retrieved by the Key.
- e.g.

```
void main() {
  // Maps - An object that associates keys and values.
  Map <int, String>  students = {‘101’: ”Ravi”, ‘102’: “Shyam”, ‘103’: “Mehul”};
}
```

1.4. Sound null safety

- Null Safety in simple words means a variable cannot contain a ‘null’ value.
- It prevents errors that result from unintentional access of variables set to null.
- There are two types on variables:
  - Non-nullable types
  - Nullable Types

1.4.1. Nullable & Non-Nullable Types

- ***Non-nullable types***
  - When we use null safety, all types are by default non-nullable.

- For example, an int variable must have an integer value.

```
void main() {
  int number;
  number = 0;
}
```

- If a variable is non-nullable, it must always be set to a non-null value.

- ***Nullable types***
  - Nullable type (?) operators specify if a variable can be null.
  - A nullable variable does not need to be initialized before being used.

```
void main() {
  String? houseLocationName;  // By default, it's set to null.
  int? number = 36;  // By default, it's set to non-null.
  number = null; // It's possible to reassign it to null.
}
```

1.5. Operators

- An operator is a symbol used to perform arithmetic, equality, relational, type test, assignment, logical, conditional, and cascade notation.

1.5.1. Arithmetic operators

- Arithmetic operators are commonly used on int and double to build expressions.

Operator	Description	Sample Code
+	Add two values	5 + 2 = 7
-	Subtract two values	5 – 2 = 3
*	Multiply two values	5 * 2 = 10
/	Divide two values	5 / 2 = 2.5
~/	Integer division of two values	5 ~/ 2 = 2
%	Modulo, Remainder of an int division	5 % 2 = 1

1.5.2. Relational operators

- Equality and relational operators are used in boolean expression, generally inside if statements or as a stop condition of a while loop

Operator	Description	Sample Code
==	Equal	5 == 2 = false
!=	Not equal	5 != 2 = true
>	Greater than	5 > 2 = true
<	Less than	5 < 2 = false
>=	Greater than or equal to	5 >= 2 = true 5 >= 5 true
<=	Less than or equal to	5<= 2 = false

	2 <= 2 = true

1.5.3. Type test operators

- They are used to check the type of an object at runtime.

Operator	Description	Sample Code
as	Cast a type to another	obj as String
is	True if the object has a certain type	obj is double
is!	False if the object has a certain type	obj is! int

1.5.4. Assignment operators

- Operators which are used to perform comparison operation on the operands.

Operator	Description	Sample Code
=	Assigns value	a=5
??=	Assigns value only if variable being assigned to has a value of null	Null ??= 2 (ans: 2) 7 ??= 2 = (ans: 7)
+=	Adds to current value	5 += 2 = 7
-=	Subtracts from current value	5 -= 2 = 3
*=	Multiplies from current value	5 *= 2 = 10
/=	Divides from current value	5 /= 2 = 2.5

1.5.5. Logical operators

- When you have to create complex conditional expressions you can use the logical operators

Operator	Description	Sample Code
!	! is a logical 'not'. Returns the opposite value of the variable/expression.	if (!(7 > 3)) = false
&&	&& is a logical 'and'. Returns true if the values of the variable/expression are all true.	if ((7 > 3) &&(3 > 7)) = false
||	|| is a logical 'or'. Returns true if at least one value of the variable/expression is true.	if ((7 > 3) ||(3 > 7)) = true

1.5.6. Conditional/Ternary Operator

- The conditional operator is a shorthand version of an if-else condition.
- There are two types of conditional operator syntax in Dart.
- One with a null safety check and the other is the same old one we encounter normally.

Operator	Description	Sample Code
condition ? value1 : value2	If the condition evaluates to true, it returns value1. If the condition evaluates to false, it returns value2. The value can also be obtained by calling methods.	(7 > 3) ? true : false = true (7 < 3) ? true : false = false

1.5.7. Null Aware Operator

- It depicts a conditional statement that is similar to a conditional operator statement.

- when you want to evaluate and return an expression if another expression resolves to null.
- It is also called the if-null operator.

Operator	Description	Sample Code
expression1 ?? Value	The null-aware operator is ??, which returns the expression on its left unless that expression's value is null	var b; String a = b ?? 'Hello'; print(a)//output: Hello

1.5.8. Cascade Notation Operators

- This operator allows you to perform a sequence of operation on the same element.

Operator	Description	Sample Code
..	The cascade notation is represented by double dots (..) and allows you to make a sequence of operations on the same object.	Matrix4.identity() ..scale(1.0, 1.0) ..translate(30,30);

1.6. Control flow

1.6.1. if statement

- The if statement compares an expression, and if true, it executes the code logic.
- The if statement also supports multiple optional else statements, which are used to evaluate multiple scenarios.
- There are two types of else statements: else if and else.
- You can use multiple else if statements, but you can have only one else statement, usually used as a catchall scenario.

```
void main() {
  final random = 13;
  if (random % 2 == 0)
        print("Number is even");
  else
        print("Number is odd");
}
```

1.6.2. switch statement

- The switch statement compares integer, string, or compile-time constants using == (equality).
- The switch statement is an alternative to the if and else statements.
- The switch statement evaluates an expression and uses the *case* clause to match a condition and executes code inside the matching case.
- Each *case* clause ends by placing a break statement as the last line.
- Use the *default* clause to execute code that is not matched by any of the case clauses, placed after all the case clauses.

```
void main() {
  String operation = '+';
  switch (operation) {
      case '+':
          print(5+2);
          break;
      case '-':
          print(5-2);
          break;
      case '*':
          print(5*2);
          break;
      default:
          print(5/2);
      }
}
```

1.6.3. for Loop

- The standard for loop allows you to iterate a List of values. Values are obtained by restricting the number of loops by a defined length.

```
void main() {
  //for(initialization; condition; expression)
  for (int i = 0; i < 5; i++) {
      print('Darshan University');
    }
```

1.6.4. for...in Loop

- The for...in loop is used to loop through an object's properties.
- In each iteration, one property from the object is assigned to the variable.
- This loop continues till all the properties of the object are exhausted.

```
void main() {
  var obj = [12,13,14];
  //for(variablename in listOfObject)
    for (var prop in obj) {
      print(prop);
    }
}
```

1.6.5. for each loop

- The for-each loop iterates over all elements in some collection and passes the elements to some specific function.

```
void main() {
  var obj = [12,13,14];
  //collection.forEach(void f(value))
   obj.forEach((var num)=> print(num));
}
```

1.6.6. while and do-while

- Both the while and do-while loops evaluate a condition and continue to loop as long as the condition returns a value of true.
- The while loop evaluates the condition before the loop is executed.
- The do-while loop evaluates the condition after the loop is executed at least once.

```
void main() {
  // while - evaluates the condition before the loop
  while (isClosed){
      askToOpen();
  }
  // do while - evaluates the condition after the loop
  do {
      askToOpen();
  } while (isClosed);
}
```

1.6.7. Break Statement

- Using the break statement allows you to stop looping by evaluating a condition inside the loop.

1.6.8. Continue Statement

- By using the continue statement, you can stop at the current loop location and skip to the start of the next loop iteration.

1.7. Function

- Functions are the building blocks of readable, maintainable, and reusable code.
- A function is a set of statements to perform a specific task.
- Functions organize the program into logical blocks of code.
- Once defined, functions may be called to access code.
- This makes the code reusable, easy to read & maintain the program's code.

```
return_type  function_name ( parameters ) {
    // Body of function return value;
}
```

- A function declaration tells the compiler about a function's name, return type, and parameters.
    - function_name defines the name of the function.
    - return_type defines the data type in which output is going to come.
    - return value defines the value to be returned from the function.
- **Function Call: -** function_name (argument_list);
    - function_name defines the name of the function.
    - argument_list is the list of the parameter that the function requires.

```
void main() {
     // Calling the function
    var output = add(10, 20);
    // Printing output
    print(output);
}
int add(int a, int b)
{
   // Creating function
   int result = a + b;
   // returning value result
   return result;
}
```

1.7.1. Arrow/Lambda Function

- The arrow function allows us to create with single expression.
- We can omit the curly brackets and the return keyword.

```
return_type function_name(parameters...) => expression;
```

- return_type consists of datatypes like void,int,bool, etc..
- function_name defines the name of the function.
- parameters are the list of parameters function requires.

```
void main() {
  perimeterOfRectangle(47, 57);
}
void perimeterOfRectangle(int length, int breadth) => print('The
perimeter of rectangele is ${2 * (length +   breadth)}');
```

1.7.2. Optional Parameter Function

- Function overloading is not supported in Dart at all.
- Optional parameters are those parameters that don't need to be specified when calling the function.
- Optional parameters allow us to pass default values to parameters that we define.
- There are two types of optional parameters
  - Ordered (positional) optional parameters
  - Named optional parameters
  - Anonymous function
- ***Ordered (positional) optional parameter Function:***
  - The square brackets [] are used to specify optional positional parameters.
  - Optional parameters can be used when arguments need not be compulsorily passed for a function's execution.

```
void main() {
    print(pow(2, 2));
    print(pow(2, 3));
    print(pow(3));
}
int pow(int x, [int y = 2]) {
    int r = 1;
    for (int i = 0; i < 2; i++) {
        r *= x;
    }
    return r;
}
```

- ***Named optional parameters Function:***
  - Optional named parameters are specified inside curly {} brackets.
  - When passing the optional named parameter, we have to specify both the parameter name and value, separated with colon.
  - It is necessary for you to use the name of the parameter if you want to pass your argument.

```
void main() {
    var name = "John Doe";
     var occupation = "carpenter";
     info(name, occupation: occupation);
  }
void info(String name, {String occupation}) {
    print("$name is a $occupation");
}
```

- ***Anonymous function***
  - Anonymous functions do not have a name.
  - An anonymous function can have zero or more parameters with optional type annotations.

```
(parameter_list) {
  statement(s)
}
```

  - Example:

```
void main() {
  var list = ["James","Patrick","Mathew","Tom"];
  print("Example of anonymous function");
  list.forEach((item) {
      print('${list.indexOf(item)}: $item');
});
}
```

1.7.3. Nested function

- A nested function, also called an inner function.
- It is a function defined inside another function.
- We have a helper buildMessage function which is defined inside the main function.

```
void main() {
    String buildMessage(String name, String occupation) {
        return "$name is a $occupation";
    }
    var name = "John Doe";
    var occupation = "gardener";
    var msg = buildMessage(name, occupation);
    print(msg);
}
```

1.7.4. Function as parameter

- A function can be passed to other functions as a parameter.
- This function is called a higher-order function.

```
void main() {
    exec(10, plusOne);
    exec(10, double);
}
void exec(int i, Function f) {
    print(f(i));
}
int plusOne(int i) => i + 1;
int double(int i) => i * i;
```

1.8. Class

- Dart is an object-oriented programming language.
- It supports the concept of class, object … etc.
- We use the class keyword to define class.

```
class ClassName {
    // Body of class
}
```

- Classes are the blueprint of the object.
- A class can contain fields, functions, constructors, etc.
- It is a wrapper that binds/encapsulates the data and functions together; that can be accessed by creating an object.

```
void main() {
  // Defining class
 class Student {
   var stdName;
   var stdAge;
   var stdRoll_nu;

   // Class Function
    showStdInfo() {
       print("Student Name is : ${stdName}");
       print("Student Age is : ${stdAge}");
       print("Student Roll Number is : ${stdRoll_nu}")
}
```

1.9. Object

- Dart is object-oriented programming, and everything is treated as an object.
- An object is a variable or instance of the class used to access the class's properties.
- Objects have two features - state and behaviour.
- Syntax (with new Keyword)

```
var object_name  = new ClassName(<constructor_arguments>);
```

- Syntax (without new Keyword)

```
var object_name  =  ClassName(<constructor_arguments>);
```

- Example

```
void main() {
    var std = Student();
 }

 // Defining class
 class Student {
   var stdName;
   var stdAge;
   var stdRoll_nu;

   // Class Function
    showStdInfo() {
        print("Student Name is : ${stdName}");
        print("Student Age is : ${stdAge}");
        print("Student Roll Number is : ${stdRoll_nu}")
}
```

1.10. Assessing Variable and Function

- We can access the fields and methods of the class using (.) operator with the object name.
- like objectName.propName or objectName.methoName()

1.10.1. Access Modifiers

- In Dart we don't have keywords like public, protected, and private to control the access scope for a property or method.
- It uses _ (underscore) at the start of the variable name to make a data member private.
- In Dart, the privacy is at library level rather than class level.
- It means other classes and functions in the same library still have the access.
- So, a data member is either public (if not preceded by _) or private (if preceded by _).

- Example

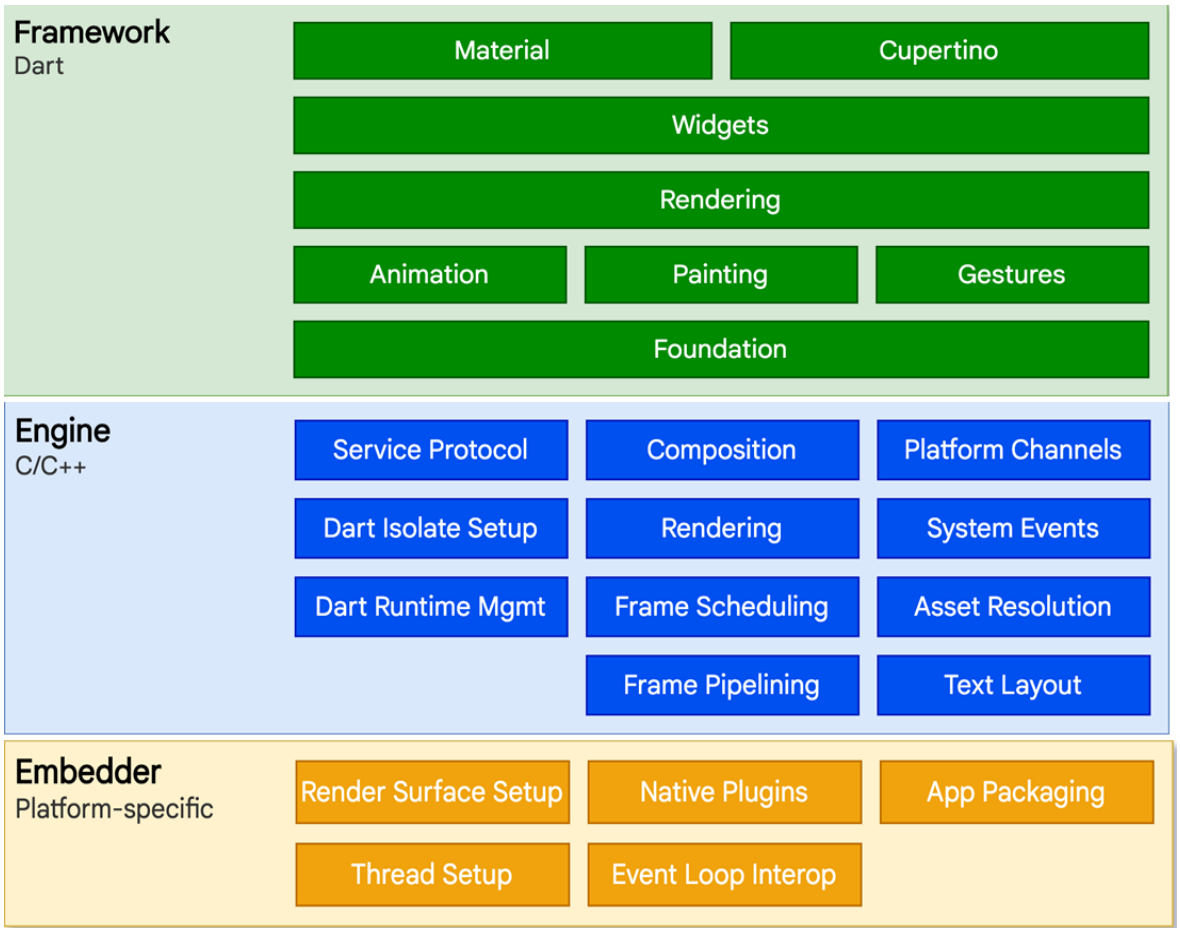
```
class A
{
   String first=‘ ’;
   String _second=‘ ’;
}
void main()
{
    A a = new A();
    a.first = 'New first';
    a._second = 'New second';
    print('${a.first}: ${a._second}');
}
```

1.11. Flutter Introduction

- Flutter is an open-source framework developed by Google
- It used for building beautiful, natively compiled, multi-platform applications from a single codebase.
- It uses dart as a programming language for fast apps on any platform.
- The first version of Flutter was known as Sky and ran on the Android operating system.
- Flutter 1.0 was release on 11 December, 2019.
- Flutter 3.3 was announced On August 30, 2022.
- It Support two themes
    - Material Design widgets implement Google's design language of the same name.
    - Cupertino widgets implement Apple's iOS Human interface guidelines
- Flutter Supports IDES and Editors via plugin.
    - IntelliJ IDEA
    - Android Studio
    - Visual Studio Code
    - Emacs

1.12. Flutter Architecture

- Flutter is a cross-platform UI toolkit.
- It is designed to allow code reuse across operating systems such as iOS and Android.
- During development apps run in a VM that offers stateful hot reload of changes without needing a full recompile.
- For release apps are compiled directly to machine code or to JavaScript if targeting the web.
- It is designed as an extensible, layered system.
- Every part of the framework level is designed to be optional and replaceable.



- Flutter is designed as an extensible, layered system.
- It exists as a series of independent libraries that each depend on the underlying layer.
- No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.
- Flutter applications are packaged in the same way as any other native application.
    - Embedder (lowest layer)
    - Engine
    - Framework (highest layer)

1.12.1. Embedder layer

- An entry point is provided by a platform-specific embedder.
- It coordinates with the underlying operating system to access services such as
    - accessibility,
    - rendering surfaces,
    - input.
- The embedder is written in a platform-specific language like.
    - Java and C++ for Android
    - Objective-C/Objective-C++ for iOS and macOS
    - C++ for Windows and Linux

- Flutter code can be embedded into an existing application as a module or as the complete application's content using the embedder.

1.12.2. Engine layer

- The engine layer is written in C/C++.
- It takes care of the input, output, network requests.
- It handles the difficult translation of rendering whenever a frame needs to be painted.
- Flutter uses Skia as its rendering engine.
- It is revealed to the Flutter framework through the dart : ui.
- It wraps the principal C++ code in Dart classes.

1.12.3. Framework layer

- The framework layer is the part where users can interact with Flutter.
- The Flutter framework provides a reactive framework that is written in Dart.
- It comprises of
    - Rendering
    - Widgets
    - Material and Cupertino
- It also has foundational classes and building block services like animation, drawing, and gestures, which are required for writing a Flutter application.

1.13. Pubspec file in flutter

- Every Flutter project includes a pubspec.yaml file, often referred to as the pubspec.
- A basic pubspec is generated when you create a new Flutter project.
- It's located at the top of the project tree and contains metadata about the project that the Dart and Flutter tooling needs to know.
- The pubspec is written in YAML, which is human readable, but be aware that white space (tabs v spaces) matters.
- The pubspec file specifies dependencies that the project requires, such as particular packages (and their versions), fonts, or image files.
- It also specifies other requirements, such as dependencies on developer packages or particular constraints on the version of the Flutter SDK.

```
name: <project name>
description: A new Flutter project.

publish_to: 'none'

version: 1.0.0+1

environment:
  sdk: ">=2.7.0 <3.0.0"

dependencies:
  flutter:       # Required for every Flutter project
    sdk: flutter # Required for every Flutter project

  cupertino_icons: ^1.0.2 # Only required if you use Cupertino (iOS style) icons

dev_dependencies:
  flutter_test:
    sdk: flutter # Required for a Flutter project that includes tests

flutter:

  uses-material-design: true # Required if you use the Material icon font

  assets:  # Lists assets, such as image files
    - images/a_dot_burr.jpeg
    - images/a_dot_ham.jpeg

  fonts:              # Required if your app uses custom fonts
    - family: Schyler
      fonts:
        - asset: fonts/Schyler-Regular.ttf
        - asset: fonts/Schyler-Italic.ttf
          style: italic
    - family: Trajan Pro
      fonts:
        - asset: fonts/TrajanPro.ttf
        - asset: fonts/TrajanPro_Bold.ttf
```

1.14. Flutter widgets

- The first thing to note is that in Flutter, everything is a widget.
- A widget is simply an instruction that you place within your code and they are the basic building blocks of a Flutter application's UI.
- Widgets indicate how its configuration and status should appear in their display.
- When a widget's state changes, it rebuilds its description.
- The framework compares to the previous description to see what changes in the underlying render tree to transition from one state to the next.
- A widget can be in the form of a button, an image, an icon, or a layout, and placing the widgets together creates a widget tree.

1.14.1. Widget States

- State is the behavior of an app at any given moment.
- It is a widget's information when it is first created and how it defines the properties of that widget.
- This information might change during the lifetime of the widget.
- To build the UI in Flutter there is two types of widgets:
    - Stateless widgets
    - Stateful widgets
- ***Stateless Widget:***
    - A stateless widget is a widget that describes part of the user interface by building a design that describe the user interface.
    - Stateless widget are useful when user interface does not depend other than the configuration information.
    - The build method of a stateless widget is typically only called in three situations:
        - The first time the widget is inserted in the tree.
        - Widget's parent changes its configuration.
        - When an InheritedWidget it depends on changes.

```
class GreenFrog extends StatelessWidget
{
    const GreenFrog({ super.key });

    @override
    Widget build(BuildContext context)
    {
        return Container(
            color: const Color(0xFF2DBD3A),
                );
    }
}
```

Unit-2
UI Design, State Management, Navigation

2.1. Scaffold

- The Scaffold widget implements the basic Material Design visual layout, allowing you to easily add various widgets such as AppBar, BottomAppBar, FloatingActionButton, Drawer, SnackBar, BottomSheet, and more.
- AppBar: The AppBar widget usually contains the standard title, toolbar, leading, and actions properties, as well as many customization options.
- title: The title property is typically implemented with a Text widget. You can customize it with other widgets such as a DropdownButton widget.
- leading: The leading property is displayed before the title property. Usually this is an IconButton or BackButton.
- actions: The actions property is displayed to the right of the title property. It's a list of widgets aligned to the upper right of an AppBar widget usually with an IconButton or PopupMenuButton.
- flexibleSpace: The flexibleSpace property is stacked behind the Toolbar or TabBar widget. The height is usually the same as the AppBar widget's height. A background image is commonly applied to the flexibleSpace property, but any widget, such as an Icon, could be used.

```
Scaffold(
  appBar: AppBar(
          actions:[ IconButton(
                          icon: constIcon(Icons.info),
  onPressed:() {...} ),
                    ]
  )
)
```

2.2. Safearea

- The SafeArea widget automatically adds sufficient padding to the child widget to avoid intrusions by the operating system.
- You can optionally pass minimum padding or a Boolean value to not enforce pad- ding on the top, bottom, left, or right.

```
body: Padding(
    padding: EdgeInsets.all(16.0),
    child: SafeArea(
        child: SingleChildScrollView(
         child: Column( children: <Widget>[], ),
        ),
    ),
),
```

2.3. Text

- The Text widget displays a string of text with a single style.
- The string might break across multiple lines or might all be displayed on the same line depending on the layout constraints.

```
Text(
  'Hello, $_name! How are you?',
  textAlign: TextAlign.center,
  overflow: TextOverflow.ellipsis,
  style: const TextStyle(fontWeight: FontWeight.bold),
)
```

2.4. Row

- A widget that displays its children in a horizontal array.
- To cause a child to expand to fill the available horizontal space, wrap the child in an Expanded widget.
- The Row widget does not scroll.
- If you have a line of widgets and want them to be able to scroll if there is insufficient room, consider using a ListView.

```
Row(
  children: const <Widget>[
    Expanded(
      child: Text('Deliver features faster', textAlign: TextAlign.center),
    ),
    Expanded(
      child: Text('Craft beautiful UIs', textAlign: TextAlign.center),
    ),
    Expanded(
      child: FittedBox(
        child: FlutterLogo(),
      ),
    ),
  ],
)
```

2.5. Column

- To cause a child to expand to fill the available vertical space, wrap the child in an Expanded widget.
- The Column widget does not scroll.
- If you have a line of widgets and want them to be able to scroll if there is insufficient room, consider using a ListView.

```
Column(
  children: const <Widget>[
    Text('Deliver features faster'),
    Text('Craft beautiful UIs'),
    Expanded(
      child: FittedBox(
        child: FlutterLogo(),
      ),
    ),
  ],
)
```

2.6. Stack

- This class is useful if you want to overlap several children in a simple way, e.g., having some text and an image, overlaid with a gradient and a button attached to the bottom.
- Each child of a Stack widget is either positioned or non-positioned.

- Positioned children are those wrapped in a Positioned widget that has at least one non-null property.
- The stack sizes itself to contain all the non-positioned children, which are positioned according to alignment.

```
Stack(
  children: <Widget>[
    Container(
      width: 100,
      height: 100,
      color: Colors.red,
    ),
    Container(
      width: 90,
      height: 90,
      color: Colors.green,
    ),
    Container(
      width: 80,
      height: 80,
      color: Colors.blue,
    ),
  ],
)
```

2.7. Container

- A convenience widget that combines common painting, positioning, and sizing widgets.
- A container first surrounds the child with padding and then applies additional constraints to the padded extent.
- The container is then surrounded by additional empty space described from the margin.

```
Center(
  child: Container(
    margin: const EdgeInsets.all(10.0),
    color: Colors.amber[600],
    width: 48.0,
    height: 48.0,
  )
```

2.8. Material Components

- Material Components for Flutter unite design and engineering with a library of components that create a consistent user experience across apps and platforms.

- These components are updated to ensure consistent pixel-perfect implementation, adhering to Google's front-end development standards.
- It provides visual, behavioural, and motion-rich widgets implementing the Material Design guidelines for the below types of widgets.
    - App structure and navigation
    - Buttons
    - Input and selections
    - Dialogs, alerts, and panels
    - Information displays
    - Layout

2.9. Flutter App Life Cycle

- The lifecycle of the Flutter App is the show of how the application will change its State.
- It helps in understanding the idea driving the smooth progression of our applications.
- Everything in Flutter is a Widget, so before thinking about Lifecycle, we should think about Widgets in Flutter.
- Flutter has majorly two types of widgets:
    - Stateless Widgets
    - Stateful Widgets

2.9.1. Stateless Widgets

- Stateless Widgets are those widgets that don't need to deal with the State as they don't change powerfully at runtime.
- It becomes permanent like on variables, buttons, symbols, and so forth, or any express that can't be changed on the application to recover information.
- Returns a widget by overwriting the build method. We use it when the UI depends on the data inside the actual item.

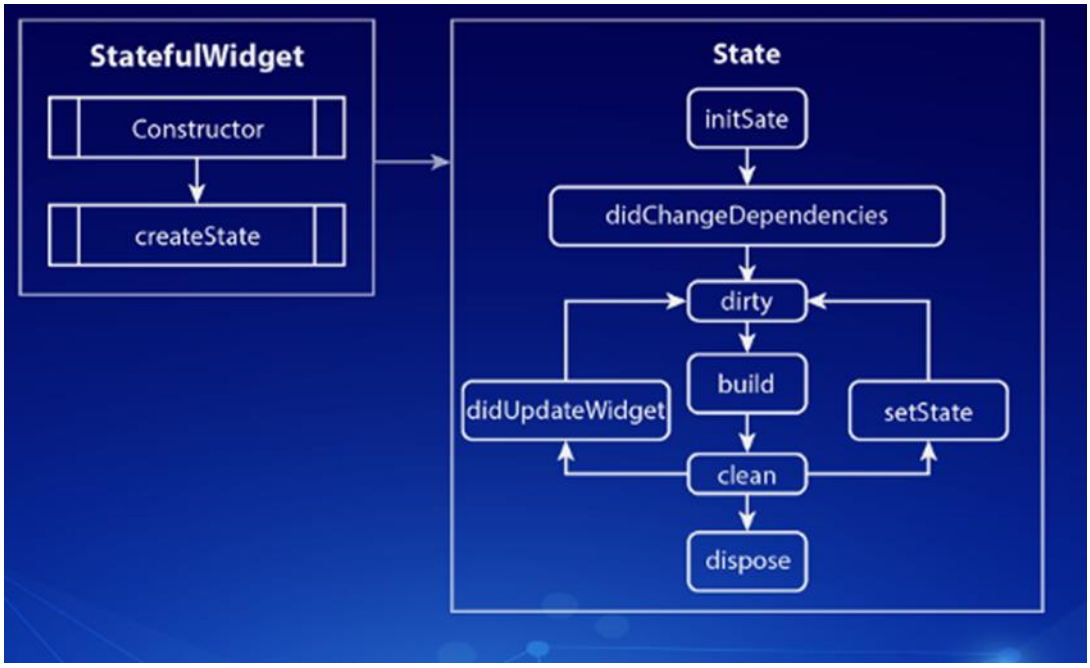
2.9.2. Stateful Widgets

- Stateful Widgets are those widgets that hold the State and the UI being portrayed can change progressively at runtime.
- It is a changeable widget, so it is attracted numerous times during its lifetime.
- We utilize this when the user progressively updates the application screen.
- This is the most significant of the multitude of widgets, as it has state widgets, everybody realizes that something has been updated on our screen.
- Thus, as we are examining the LifeCycle of Flutter, we will focus on 'Stateful Widgets' as they need to deal with the State.

2.9.3. Stages Of App Lifecycle

- The life cycle depends on the state and how it changes.
- A stateful widget has a state so we can clarify the life cycle of flutter dependent on it.
- Stage of the life cycle:
    - createState()

- initState()
- didChangeDependencies()
- build()
- didUpdateWidget()
- setState()
- deactivate()
- dispose()



- ***createState(): -*** This method is called when we create another Stateful Widget. It is an obligatory strategy. The createState() returns a case of a State-related with it.

```
class HomeScreen extends StatefulWidget {
HomeScreen({Key key}) : super(key: key);

@override
HomeScreenState<StatefulWidget> createState() => HomeScreen();
}
```

- ***initState(): -***
  - This is the strategy that is considered when the Widget is made interestingly and it is called precisely once for each State object.
  - If we characterize or add some code in the initState() method this code will execute first even before the widgets are being built.
  - This method needs to call super.initState() which essentially calls the initState of the parent widget (Stateful widget).

- Here you can initialize your variables, objects, streams, AnimationController, and so on.

```
@override
void initState(){
super.initState();
}
```

- ***didChangeDependencies(): -***
  - This method is called following the initState() method whenever the widget initially is constructed.
  - You can incorporate not many functionalities like API calls dependent on parent data changes, variable re-initializations, and so forth.

```
@override
void didChangeDependencies() {

}
```

- ***build (): -***
  - This strategy is the main method as the rendering of all the widgets relies upon it.
  - It is called each time when we need to render the UI Widgets on the screen.
  - At whatever point you need to update your UI or on the other hand on the off chance that you click hot-reload, the Flutter structure modifies the build() strategy!.
  - Assuming you need to expressly revamp the UI if any information is transformed, you can utilize setState() which teaches the framework to again run the form method

```
@override
Widget build(BuildContext context) {
  return Scaffold()
}
```

- ***didUpdateWidget(): -***
  - This strategy is utilized when there is some adjustment of the configuration by the Parent widget.
  - It is essentially called each time we hot reload the application for survey the updates made to the widget.
  - If the parent widget changes its properties or designs, and the parent needs to modify the child widget, with a similar Runtime Type, then, at that point, didUpdateWidget is triggered.
  - This withdraws to the old widget and buys into the arrangement changes of the new widget.

```
@protected
void didUpdateWidget(Home oldWidget) {
  super.didUpdateWidget(oldWidget);
}
```

- ***setState(): -***
  - The setState() method illuminates the framework that the internal state of this item has changed in a manner that may affect the UI which makes the structure plan a build for this State of the object.
  - It is an error to call this method after the system calls dispose().
  - This inside state could conceivably influence the UI apparent to the user and subsequently, it becomes important to rebuild the UI.

```
void function(){
  setState(() {});
}
```

- ***deactivate(): -***
  - This method is considered when the State is removed out from the tree, however, this strategy can be additionally be re-embedded into the tree in another part.
  - This strategy is considered when the widget is as of now not joined to the Widget Tree yet it very well may be appended in a later stage.
  - The best illustration of this is the point at which you use Navigator.
  - push to move to the following screen, deactivate is called because the client can move back to the past screen and the widget will again be added to the tree.

```
@override
void deactivate(){
  super.deactivate();
}
```

- ***dispose(): -***
  - This strategy is essentially something contrary to the initState() method and is likewise important.
  - It is considered when the object and its State should be eliminated from the Widget Tree forever and won't ever assemble again.
  - Here you can unsubscribe streams, cancel timers, dispose animations controllers, close documents, and so on.
  - At the end of the day, you can deliver every one of the assets in this strategy.
  - Presently, later on, if the Widget is again added to Widget Tree, the whole lifecycle will again be followed

```
@override
void dispose(){
  super.dispose();
}
```

2.9.4. Stateless vs Stateful Widget

Stateless widget	Stateful widget
They are static widgets; they are updated only when initialized.	They are dynamic in nature.
It is not dependent on data changes or behavior changes.	It is dependent and changes when the user interacts.
Examples are Text, Icons, or a RaisedButton.	Examples are Checkbox, RadioButton, or Slider.
Doesn't include a setState().	It has an internal setState().
Cannot be updated during the application's runtime. An external event is necessary for the trigger.	It can be updated during the runtime.
It doesn't have life cycle	It contains lifecycle

2.10. MaterialPageRoute

- A modal route that replaces the entire screen with a platform-adaptive transition.
- For Android, the entrance transition for the page zooms in and fades in while the existing page zooms out and fades out.
- The exit transition is similar, but in reverse.
- For iOS, the page slides in from the right and exits in reverse.
- The page also shifts to the left in parallax when another page enters to cover it.
- By default, when a modal route is replaced by another, the previous route remains in memory.
- To free all the resources when this is not necessary, set *maintainState* to false.
- The *fullscreenDialog* property specifies whether the incoming route is a *fullscreen* modal dialog.
- On iOS, those routes animate from the bottom to the top rather than horizontally.
- The type T specifies the return type of the route which can be supplied as the route is popped from the stack via *Navigator.pop* by providing the optional result argument.

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

void main() {
  runApp(const MaterialApp(
    title: 'Navigation Basics',
    home: FirstRoute(),
  ));
}

class FirstRoute extends StatelessWidget {
  const FirstRoute({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Route'),
      ),
      body: Center(
        child: ElevatedButton(
          child: const Text('Open route'),
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => const SecondRoute()),
            );
          },
        ),
      ),
    );
  }
}
```

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

class SecondRoute extends StatelessWidget {
  const SecondRoute({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Route'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Go back!'),
        ),
      ),
    );
  }
}
```

- To switch to a new route, use the Navigator.push() method.
- The push() method adds a Route to the stack of routes managed by the Navigator.

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

- By using the Navigator.pop() method.
- The pop() method removes the current Route from the stack of routes managed by the Navigator.

```
// Within the SecondRoute widget

onPressed: () {
  Navigator.pop(context);
}
```

2.11. Navigation with Named Routes

- If you need to navigate to the same screen in many parts of your app, this approach can result in code duplication. The solution is to define a named route, and use the named route for navigation.
- To work with named routes, use the Navigator.pushNamed() function. This example replicates the functionality from the original recipe, demonstrating how to use named routes using the following steps:
    - Create two screens
    - Define the routes
    - Navigate to the second screen using Navigator.pushNamed()
    - Return to the first screen using Navigator.pop()

```
import 'package:flutter/material.dart';
void main() {
  runApp(
    MaterialApp(
      title: 'Named Routes Demo',
      initialRoute: '/',
      routes: {
        '/': (context) => const FirstScreen(),
        '/second': (context) => const SecondScreen(),
      },
    ),
  );
}

class FirstScreen extends StatelessWidget {
  const FirstScreen({super.key});
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(context, '/second');
          },
          child: const Text('Launch screen'),
```

```
          ),
        ),
      );
    }
  }
  class SecondScreen extends StatelessWidget {
    const SecondScreen({super.key});

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        appBar: AppBar(
          title: const Text('Second Screen'),
        ),
        body: Center(
          child: ElevatedButton(
            onPressed: () {
              Navigator.pop(context);
            },
            child: const Text('Go back!'),
          ),
        ),
      );
    }
  }
```