# Flutter and Dart

## Up and Running

Build native apps for both iOS and Android using a single codebase

Dr. Deepti Chopra
Roopal Khurana

bpb

# Flutter and Dart

## Up and Running

Build native apps for both iOS and Android using a single codebase

Dr. Deepti Chopra
Roopal Khurana

# Flutter and Dart: Up and Running

*Build native apps for both iOS and Android using a single codebase*

**Dr. Deepti Chopra**
**Roopal Khurana**

www.bpbonline.com

**First published:** 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-5551-377-9

**www.bpbonline.com**

# Dedicated to

*Our Parents, friends*

*&*

*Our Son* ***Pramudit Khurana***

# About the Authors

**Dr. Deepti Chopra** has done her Ph.D in the area of Natural Language Processing from Banasthali Vidyapith. She is currently working as Associate Professor at JIMS Rohini, Sector 5.

Dr. Chopra is an author of five books and two MOOCs. Two of her books have been translated into Chinese and one has been translated into Korean. She has 2 Australian Patent and 1 Indian Patent in her name.

Dr. Chopra has several publications in various International Conferences and journals of repute. Her areas of interest include Artificial Intelligence, Natural Language Processing and Computational Linguistics. Her primary research works involve machine translation, information retrieval, and cognitive computing

**Roopal Khurana** is working as Assistant General Manager at Railtel Corporation of India Ltd., IT Park, Shastri Park, Delhi. He is currently working in the field of Data Networking, MPLS Technology.

Mr. Khurana has done BTech in Computer Science and Engineering from GLA University, Mathura, India. He is a technology enthusiast. Previously, he has worked with companies such as Orange and Bharti Airtel.

# Acknowledgement

We want to thank God most of all, because without God we would not be able to do any of this. We acknowledge with gratitude and sincerely thank all my family and friends for the blessings and good wishes conveyed to us to achieve the goal to publish this Flutter and Dart based book.

There are a few people we would like to thank for the continued and ongoing support they have given us during the writing of this book. First and foremost, we would like to thank our parents for continuously encouraging us for writing the book — we could have never completed this book without their support.

Our gratitude also goes to the team at BPB Publications for being supportive enough to provide me quite a long time to finish the book and being supportive in each and every phase of book completion. Our gratitude goes to the editorial and publishing professionals of BPB Publications for their keen interest and support in bringing out this book. Finally, we would like to thank Mr. Manish Jain at BPB Publications for giving us this opportunity to write our second book for them.

# Preface

Welcome to the world of Flutter and Dart! In this book, you will learn how to use these powerful tools to build beautiful, high-performance mobile applications for iOS and Android.

Flutter is a mobile app development framework created by Google. It allows developers to build natively compiled applications for mobile, web, and desktop from a single codebase. Flutter is known for its fast development cycle, expressive and flexible UI, and impressive performance.

Dart is the programming language that Flutter is built with. It is a modern, object-oriented language that is easy to learn and powerful enough to build complex applications. Dart is also used for web development and has strong support for asynchronous programming, making it a great choice for building responsive apps.

In this book, you will learn the fundamentals of Flutter and Dart, as well as how to use them to build real-world applications. We will start with the basics, such as setting up your development environment and creating your first Flutter app. From there, we will delve into more advanced topics such as state management, network programming, and integration with native code.

So, whether you are a seasoned mobile developer looking to learn Flutter and Dart, or a beginner just getting started in the world of app development, this book has something for you. Let us dive in and start building amazing apps with Flutter and Dart!

# Code Bundle and Coloured Images

Please follow the link to download the
***Code Bundle*** and the ***Coloured Images*** of the book:

## https://rebrand.ly/5as0ipn

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Flutter-and-Dart-Up-and-Running**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

# Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**errata@bpbonline.com**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

## Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

## If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

# Table of Contents

# CHAPTER 1

# Introduction to Flutter

## Introduction

Flutter Framework is used to create simple and efficient mobile applications. Only a single codebase is required to develop a mobile application across different platforms such as Android and iOS. This chapter will give an introduction to Flutter. It will discuss in detail why Flutter is required to develop sophisticated and elegant mobile applications. It would give insight into the architecture of Flutter Framework, its pros and cons and why Flutter proves to be better option for designing mobile applications compared to the other mobile application frameworks.

## Structure

In this chapter, we will cover the following topics:

- Introduction to Flutter
- Features of Flutter
- Architecture of Flutter Framework
- Advantages of Flutter
- Disadvantages of Flutter
- Capabilities that make Flutter more preferred compared to other frameworks

## Objectives

This chapter aims to give a basic introduction to Flutter Framework that may be used to develop exciting mobile applications. From this chapter, readers will be able to know about the architecture of Flutter Framework in detail, the characteristics of Flutter, its positive and negative aspects and also know reasons what makes it one of the most preferable mobile application frameworks for developing mobile applications compared to the other competitive frameworks. We shall conclude the chapter with some examples that can be implemented with Flutter Framework in later chapters.

# Introduction to Flutter

A Flutter may be referred to as a high performance and simple framework that is used along with Dart language to develop mobile application. Flutter framework supports both Android and iOS. In order to create modern applications, Flutter provides many useful and simple widgets that can be used for designing mobile applications easily. Gestures and Animations are supported by these widgets. Building Mobile Applications using Flutter is based on Reactive programming. Widget may be represented using state. If there are some changes made in Widget, then as a part of Reactive Programming, there is a comparison between the old and the new state. Then, instead of re-rendering the whole mobile application, only the changes that have been made in new state as compared to the old state are rendered in the mobile application. The logo of Flutter is given in *Figure 1.1*:

***Figure 1.1:*** *Logo of Flutter*

*Figure 1.2* shows logo of Flutter and Dart. Dart is a programming language that helps to build applications using Flutter framework. An example of Dart programming for building mobile application is shown in *Figure 1.2*:

***Figure 1.2:*** *Logo of Dart and Flutter*

In 2019, Google launched Dart 2.5 and Flutter 1.9. Flutter 1.9 helps in building web, desktop and mobile applications using single codebase. Please refer to the following *Figure 1.3*:

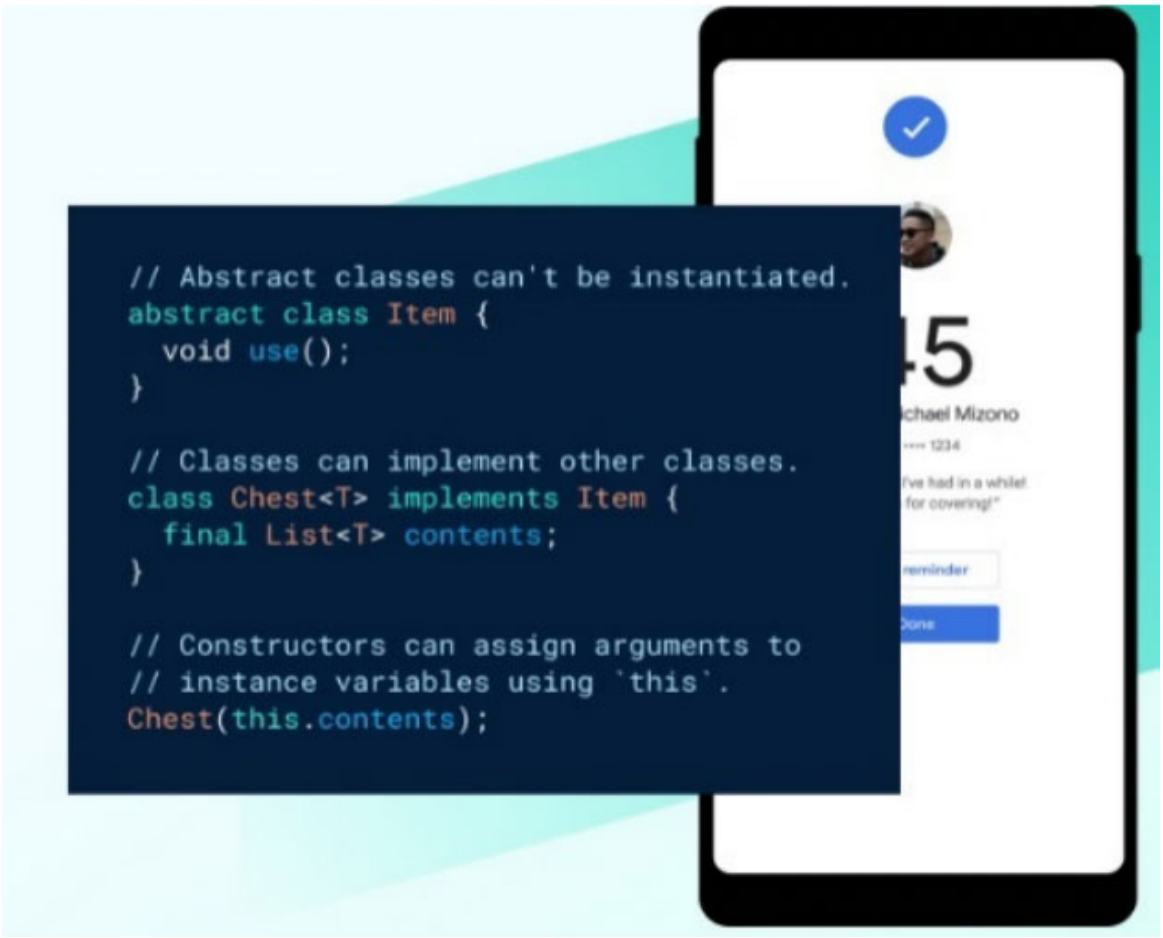


***Figure 1.3:*** *Dart Programming used for developing Mobile App*

# Features of Flutter

Flutter is a mobile application framework that offers many features to the developers. These include the following:

- **Reactive and Modern Framework**

  During development of Mobile application, the Flutter Mobile Application run in VM environment that provides hot reload of stateful changes without the need to recompile the entire code. Developers are able to interact with Flutter with the help of Flutter framework and they provide a proper mapping between a given application state to the interface state. When there are some changes in the application state, Flutter performs the task of modification of Interface. Flutter is a reactive and modern framework written in Dart language.

- **Simple and High-Performance Application**

  Flutter based applications are simple to write and they deliver high level performance when common pitfalls are avoided. The following pitfalls may be avoided while building Flutter application:

  1. While working on building Animation based application, we should avoid using Opacity widget.
  2. Clipping during building Animation based application should be avoided.

- **Supports easy to understand Dart Language**

  Dart is a programming language whose constructs are similar to C and is used for making mobile applications using Flutter. Dart can also be used for developing web applications that will run on all the web browsers. Dart has predefined libraries which are present in Dart SDK. Some of the commonly used Dart libraries include the following:

  1. dart:core-

     This library is imported in all the files. It defines the core functionality.
  2. dart:async-

     This library is used for performing asynchronous programming.
  3. dart:math-

     This library defines mathematical constants and functions.

4. dart:convert

   This library is used for conversion between different data representations.

- **Fast Development**

  Flutter uses Dart language for creating fast mobile, web and desktop applications on any platform using a single codebase. This is referred to as hot reload feature. Flutter's hot reload feature helps to quickly do updates, remove errors and add new features.

- **Supports large Widget Catalog**

  Flutter is used to create fast and pleasing apps with the help of its large collection of interactive, structural, platform and visual widget catalog.

- **Supports same UI for different platforms**
- **Generates beautiful UI**

## Architecture of Flutter

Flutter framework is designed in such a way that single code may be used to develop mobile application in both Android and iOS. Flutter architecture is displayed in *Figure 1.4*. It may be explained in the following sections:

- **The layer model**: It comprises of the components from which flutter is created.
- **Reactive user interfaces**: It involves user interface development in Flutter.
- **Widgets**: Widgets are the building blocks of user interfaces in Flutter.
- **Rendering steps**: This process involves conversion of UI process into pixels.
- **Platform embedders**: It is the code that allows execution of flutter apps on operating system of desktop and mobile.
- **Combining other code and Flutter**: Other techniques may be added in a Flutter app.
- **Web Support**: It involves behavior of Flutter in browser environment.

Flutter comprises of a layered system in which independent and individual libraries depend on the underlying layer. No layer can have privileged access to the other layers present under it. Please refer to the following figure:

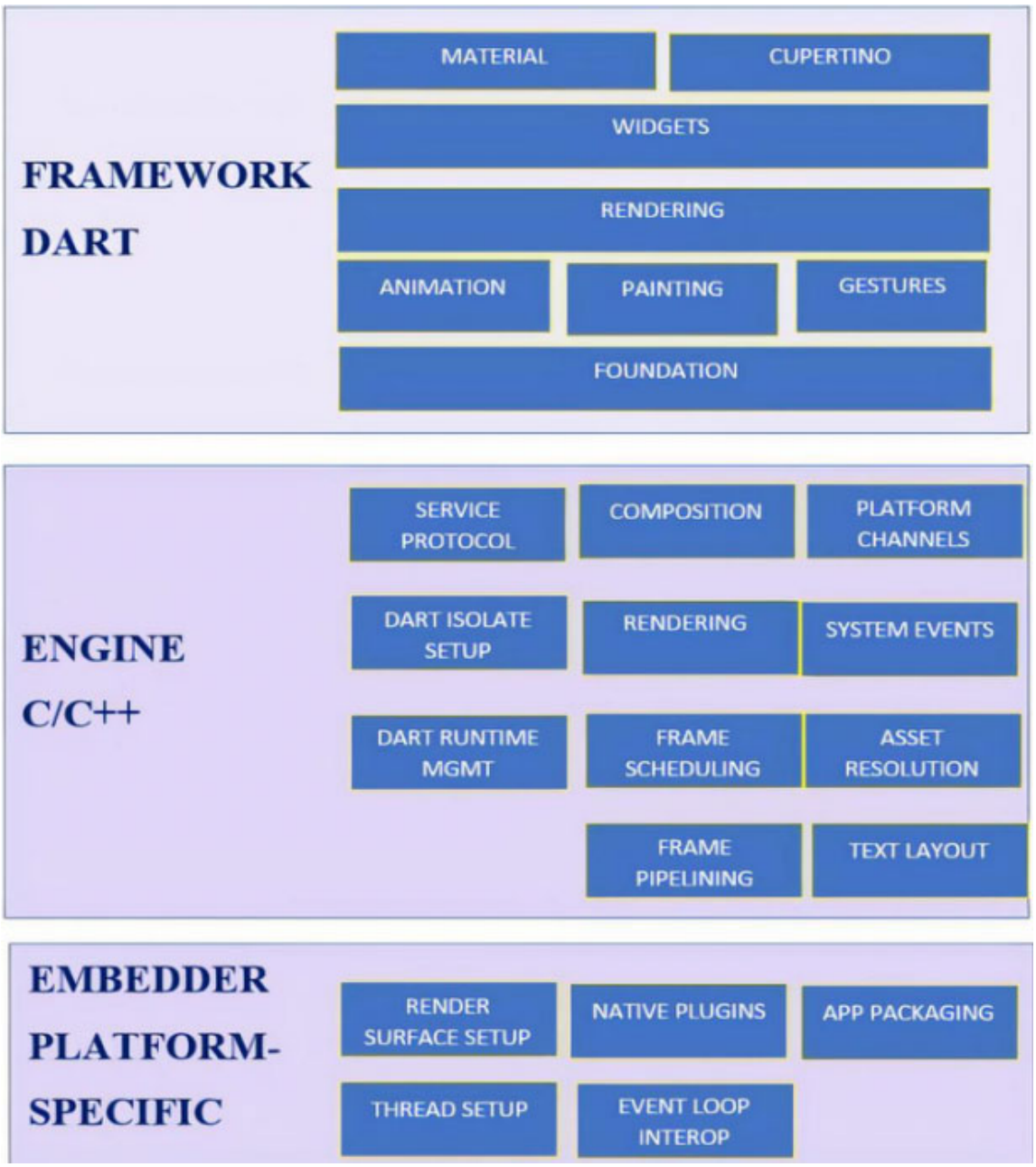



***Figure 1.4:*** *Architecture of Flutter Framework*

In embedder, language is used in accordance with the platform. C++ and Java is used for Android. For macOS and iOS, Objective C and Objective C++ is used. Windows and Linux makes use of C++.

Flutter engine lies at the core of Flutter. It is usually written in C/C++ and it contains all the primitives required to run a Flutter application. Whenever a new frame is to be designed, Flutter engine is responsible for rasterization of different scenes. The flutter engine comprises of C++ code which is present in Dart classes. With the help of dart:ui, Flutter framework comes in contact with Flutter Engine.

Developers are able to communicate with Flutter using Flutter Framework which is written mainly in Dart language. Flutter Framework comprises of collection of Layout, Platform and Foundational libraries. It includes the following functionalities:

- General Foundational classes and basic services such as gestures, painting and animation.
- The rendering layer may be used to construct a tree comprising of renderable objects which may be modified dynamically and also may be used for updating layout automatically.
- The Widget layer introduces the concept of reactive programming model, in which for corresponding to every render object present in the rendering layer, there is a class present in the widget layer. The classes present in the widget layer may be later reused.
- The Material and Cupertino libraries are used to implement the iOS and material languages with the help of widget layer primitives.

Flutter uses an architecture known as **Business Logic Component** (**BLOC**) for developing mobile applications. It has a simple architecture that can be applied for building mobile applications and apply changes in its state easily. Flutter uses an event-state based approach, in which events may be triggered and states are also handled accordingly. The key feature of architecture of Flutter is simplicity. Compilation of ARM Binary Code can be done using Flutter Architecture. The architecture of Flutter does not depend on OEM Widgets. There is no requirement of bridges in architecture of Flutter. Moreover, there is an absence of markup language in architecture of Flutter, as only Dart is used.

Flutter Bloc pattern is very popular these days for building maintainable and reactive mobile applications. Bloc refers to business logic component and it is used for managing state within any mobile application. Significance of Bloc is that a single component is capable of handling all the manipulations within mobile application rather than handling it by different components. With the help of Bloc, all interactions between UI and business logic may be established. The input to the Bloc will be events. After receiving input, it runs necessary algorithm and generates states as output. This is shown in *Figure 1.5*:

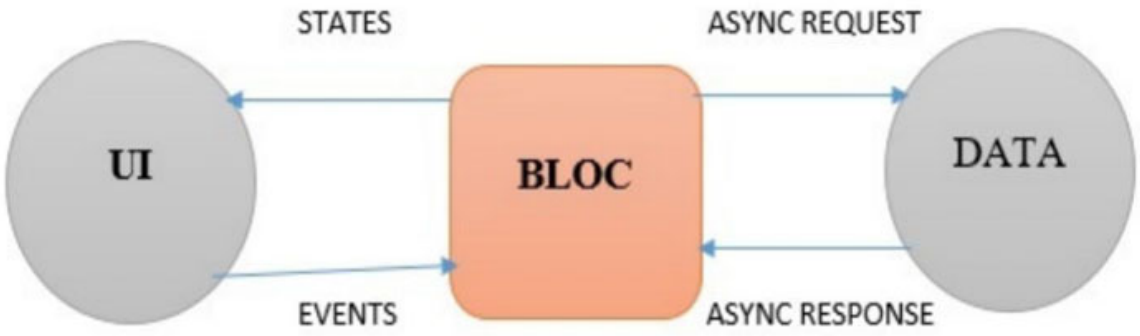



***Figure 1.5:*** *Bloc in Flutter*

Events in Bloc refer to the actions that may be triggered by the actions performed on the software. For example: Clicking of a button. After receiving the event as an input, Bloc performs some logic. After the logic or algorithm runs, a state is returned as an output by the Bloc. State represent some changes that may occur either in the UI of mobile application or in the background. Whenever there is modification in state, child widgets are built up again according to the changes in the state being made. BlocProvider is used when we use Bloc multiple times or inherited widgets are used.

## Advantages of Flutter

Flutter offers lot of advantages compared to its competitive frameworks. Following are some of the advantages of using Flutter framework:

- **Common UI code across all platforms**: Flutter allows sharing of UI and UI code across different platforms. So, there is no need to re-write the full code in case there is change in platform used.
- **Faster Code Development**: Flutter offers many ready to use widgets that makes code development faster. Also, the available widgets can be easily customized according to users requirements. If there are some modifications applied on current layout, Flutter makes these updations instantly on the present application state itself. Flutter uses hot-reload function according to which if there are any modifications, they can be viewed in real time without restarting the mobile app.
- **Increased Marketing speed**: Flutter comes with declarative API that enhances the performance. Also, a single code may be used across multiple platforms. A two dimensional user interface may be implemented in Flutter very efficiently without making any modifications in the native applications.
- **Flutter Desktop Embedding and Flutter Web**: Apart from Flutter being used for mobile application development; it is also used for web

development. We can run the Flutter applications on our desktop as well without changes in the source code.

- **Open-Source Platform**: Flutter is freely available. It has rich documentation and large community that assist developers in case any issue arises.

## Disadvantages of Flutter

Nothing comes with perfection. Same goes with Flutter. Following are some of the cons of Flutter:

- **Flutter - A comparatively new platform:** It is a new platform and may have some of the issues as well. It is still in its experimentation phase. For developing some rare applications, there might not be a solution in Flutter. Flutter is still an evolving platform.
- **Not suitable for building sophisticated apps**: Flutter lacks in advanced features. It might not be suitable for building pixel perfect apps. Moreover, Flutter may not be used for building mobile applications for tvOS, CarPlay, Android Auto and watchOS.
- **Very Novice Platform**: Flutter is a very new field. Hence, there is a lack of experienced Flutter professionals. So, it is difficult to build a team of best professionals who can work on Flutter solutions as compared to the React Native.
- **Absence of Native Widgets**: Flutter does not have native widgets. This may be disadvantageous too. In case there is upgrade of Android and iOS, then mobile app will not be upgraded with new UI changes.
- **Lacks in rich library content**: The library content in Flutter is very limited. The content needed by the developers is sometimes not found in the library. So, content needs to be generated by application developers.
- **Files developed are of large size**: One of the loop holes in developing mobile application using Flutter is that they produce file of very large size.

## Capabilities that make Flutter more powerful over other tools

Flutter can be used for developing both mobile application and web development. It helps in creating promising cross platform mobile application development. Flutter is also used for developing simple, cost effective and quality driven web

applications. Some of the capabilities of Flutter that make it more efficient compared to other counterparts include the following:

- **Exceptional Performance:** Flutter is used to deliver high performance and magnificent mobile, web and desktop applications. It is excellent for developing sleek UIs and takes very less time in testing our applications. Flutter is especially useful for developing user-friendly applications.
- **Rapid Development:** The number of users using Smart Phones has increased drastically over the years. This is shown in *Figure 1.6*.

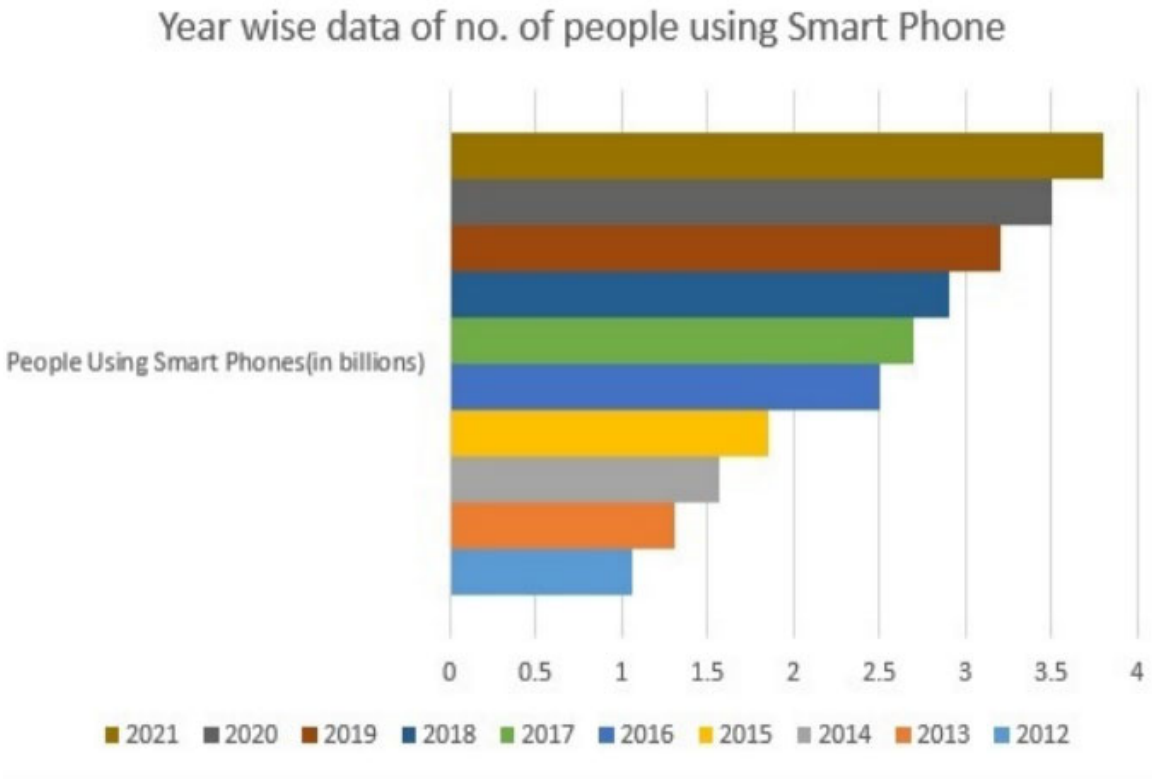



***Figure 1.6:*** *No. of People using Smart Phones*

Since, Smart Phones have become part and parcel of our lives, the development of mobile applications is increasing drastically day by day. It is seen that the number of Android based mobile applications launched in a month has crossed 100 thousand limit, whereas number of iOS based mobile application is more than 30,000 per month. Today, Flutter technology is used by both small and big brand companies for mobile application development, since we can develop very sophisticated and pleasing mobile applications in a very short span using Flutter.

- **Easily Compatible:** Flutter is used for developing mobile applications across multiple platforms using a single code base.

    - Short Learning Curve
    - Strong Community Support
    - Hot Reloading Function

  Flutter makes use of hot reload technology, in which any updates made to the application are reflected instantly.

    - Prebuilt Customizable widgets
    - Cross-Platform-

  Instead of creating two separate mobile applications for Android and iOS, Flutter allows us to create only one mobile application that will run on both iOS and Android.

- **Cost Effective:** In Flutter, we can use existing widgets and combine existing widgets to generate complex layouts. This makes it cost effective as well due to its minimized complexity, lesser developers and reduced development cycle.
- **Superior Productivity:** It helps in developing high performance mobile applications. Even if there are updates in Operating System, Flutter applications remain the same. In Flutter app development, the already developed widgets may be reused, and more creativity may be added to the existing widget to produce high quality applications.
- **Extensive documentation support:** There exists huge community of Flutter developers that keeps on adding contents to its documentation making it more rich day by day.
- **Decreased Code development time:** Any modifications made in the mobile application are immediately reflected using Flutter, using its hot reload technology. Any changes such as addition of new features, fixing of errors and building of UIs is done instantly using Flutter technology.

## Conclusion

Flutter is a newly found framework, suitable for development of mobile applications. In fact, it is also used for developing web applications efficiently. It is a cross platform framework that allows the same code to be used in different operating systems such as iOS and Android. In this chapter, we have discussed about Flutter, its underlying features and architecture, pros and cons of using this new framework and the capabilities that makes it more desirable over other frameworks. In the next chapter, we would discuss about the setting up the

environment of Flutter and related tools assuming we are building a full-stack app or a client-side app.

## Questions

1. Why should we prefer Flutter over other frameworks available for developing mobile applications?
2. What are the flaws of using Flutter framework?
3. Explain architecture of Flutter in detail.
4. List the various characteristics of Flutter.

# CHAPTER 2

# Installing the Flutter SDK

**“I think the biggest change, and the one we’re already starting to see take shape, is that globally the majority of internet usage will be done via a mobile device and for most people the mobile web will be their primary – if not their only – way of experiencing the internet.”**

*– Peter Rojas, Co-founder of Engadget and Gizmodo*

## Introduction

Flutter Framework is an effective method of generating mobile application with the same codebase across multiple platforms. It offers numerous other functionalities that makes it an ideal choice for app development. In the previous chapter, we had discussed the introduction of Flutter, architecture of Flutter Framework, pros and cons of using Flutter for mobile app development and why Flutter is considered a better option for mobile app development compared to other application frameworks. This chapter will discuss the environmental setup of Flutter Frameworks and tools required for building Flutter based applications. It will also discuss how Android Studio can be installed. Some of the tools for Flutter based application include the following: Airship, Square, WonderPush, Panache, Supernova, RevenueCat, Countly, Codemagic, Amplitude and so on.

## Structure

In this chapter, we will cover the following topics:

- Setting up the Environment for Flutter
- Tools for building Flutter based application

## Objectives

This chapter aims to provide information on the installation of Flutter Framework. After studying this chapter, readers will get an idea about the editors used in Flutter Framework and tools used for developing and manipulating mobile

applications using Flutter. This chapter will give basic knowledge about Flutter environment setup. In later chapters, we will discuss mobile applications created using Flutter.

## Setting up environment for Flutter

For installing Flutter in Windows environment, the system must meet the following requirements:

- Operating System should be Windows 7 SP1 or later (64-bit), x86-64 based.
- Disk Space around 1.64 GB
- Following Flutter tools are required :
    - Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
        - Git with Windows 2.x. We should be able to run git commands on Power shell or Command Prompt

## Installation of Flutter SDK on Windows

We can obtain the latest release of Flutter SDK for Windows from the following link:

**https://docs.flutter.dev/get-started/install/windows**

After extracting the zip file, place the extracted files in the desired location. Do not place it in C:\Program Files\

### Update the Path

In order to add Flutter to the environment PATH, follow the following steps:

1. From the Start menu, select env and click on Edit Environment Variables.
2. Under User Name, if path does not exist, add full path till flutter/bin under the entry called Path.

### Run Flutter Doctor

In order to check the status of Flutter installation, we type run Flutter Doctor. It gives details of software that need to be installed and future steps that need to be performed for completing the installation process.

For example, Flutter and Dart may display the following output:

[-] Android toolchain: develop for Android devices

- Android SDK at D:\Android\sdk
- Android SDK is missing command line tools; download from https://goo.gl/XxQghQ
- Try re-installing or updating your Android SDK,

  visit **https://docs.flutter.dev/setup/#android-setup** for detailed instructions.

Flutter doctor makes sure that the installation process is done correctly and there is no missing dependency left.

We have to install Android Studio in order to accomplish complete Android Platform dependencies.

Steps for installation of Android Studio include the following:

1. First download the Android Studio and then perform the installation.
2. Click on Android Studio; Android Studio Setup Wizard will open. Android SDK, Android SDK Build tools and Android SDK Command-line tools will be installed, which is required for using Flutter for Android.
3. Run Flutter doctor that will ensure that Flutter has detected the location of Android Studio. Otherwise, run the following command:

   ```
   flutter config --android-studio-dir <directory>
   ```

   The previous command confirms that Flutter is aware of the directory where Android Studio is installed.

## Setting up Android Device

In order to run Flutter application on an Android device, the device should run Android 4.1 (API level 16) or higher. Then follow the given steps:

1. Enable Developer options as well as USB debugging on Android device.

   The Developer options are available by default on Android 4.1 and lower. We will have to enable Developer options on Android 4.2 and higher. To enable Developer option, we need to tap Build Number option 7 times. The location of Build Number option may vary according to the Android version used.

   For Android 9 (API level 28) and higher, Build Number option may be found at: **Settings** | **About Phone** | **Build Number**

For Android 8.0.0 (API level 26) and Android 8.1.0 (API level 26), Build Number option may be found at: **Settings** | **System** | **About Phone** | **Build Number**

For Android 7.1 (API level 25) and lower, Build Number option may be found at: **Settings** | **About Phone** | **Build Number**

In the Developer options screen, we can toggle the options on and off. We need to keep it on always.

Moreover, we need to enable the USB debugging option, which helps Android Studio as well as other sdk tools to detect our device, when connected using USB. The location of the USB debugging option may vary with the android version used.

- In Android 9 (API level 28) and higher, USB debugging options can be found at: **Settings** | **System** | **Advanced** | **Developer Options** | **USB debugging**
- In Android 8.0.0 (API level 26) and Android 8.1.0 (API level 26), USB debugging options can be found at: Settings | System | Developer Options | USB debugging
- In Android 7.1 (API level 25) and lower, USB debugging options can be found at: Settings | Developer Options | USB debugging

In Android 8.0 and higher, we can use Quick settings developer tiles and select general options required for developing mobile applications. We can drag these selected options from the Quick settings developer tiles into the Quick settings panel. Other General options that may be selected include the following:

- On Android 8.0 and higher, memory options may be used to display statistics of memory used, total available memory and so on.
- We can share log files of the current device as well as bug report, with others.
- We can setup password protection, setup backup password and adb commands for backup.
- We can set our screen to stay on every time we plug in.

## Set up Android Emulator-

In order to run and test Flutter app on Android emulator, the following steps are followed:

1. Enable VM acceleration.

2. Start Android Studio. Click on AVD Manager icon and select or create a suitable Virtual Device.
3. Choose a suitable device definition and select Next.
4. Select suitable image for Android Emulator.
5. In order to enable hardware acceleration, select Hardware - GLES 2.0.
6. Check if AVD configuration is correct, and select Finish. In Android Virtual Device Manager, click Run in the toolbar. The emulator starts up and displays the default canvas for your selected OS version and device.

Click on Run in the toolbar of Android Virtual Device Manager. The emulator starts and displays a canvas related to the OS and device selected.

## Licenses of Android SDK Platform-

Before beginning with Flutter, we must agree with the licenses of Android SDK Platform. Licenses should be agreed when all the tools discussed previously have been installed.

1. Java 8 must be installed and set **`JAVA_HOME`** environment variable to JDK's folder.
2. For signing licenses, run the following command on console:

```
$ flutter doctor -android-licenses
```

3. Before agreeing with the license, review the license very carefully.
4. After agreeing licenses, run flutter doctor to ensure that you can use Flutter.

## Setting up an Editor

We can build mobile applications in Flutter with the help of any text editor, in combination with command-line tools. But, for better experience, it is recommended to use editor plugins. These editor plugins assist in running and debugging, editing widget, highlighting syntax and in completion of code.

Follow the given steps to set up editor plugin of Android Studio, IntelliJ, VS Code, or Emacs on your system:

Android Studio provides a complete and integrated IDE for Flutter. Android Studio, version 2020.3.1 (Arctic Fox) or later may be used. This is shown in *Figure 2.1*:

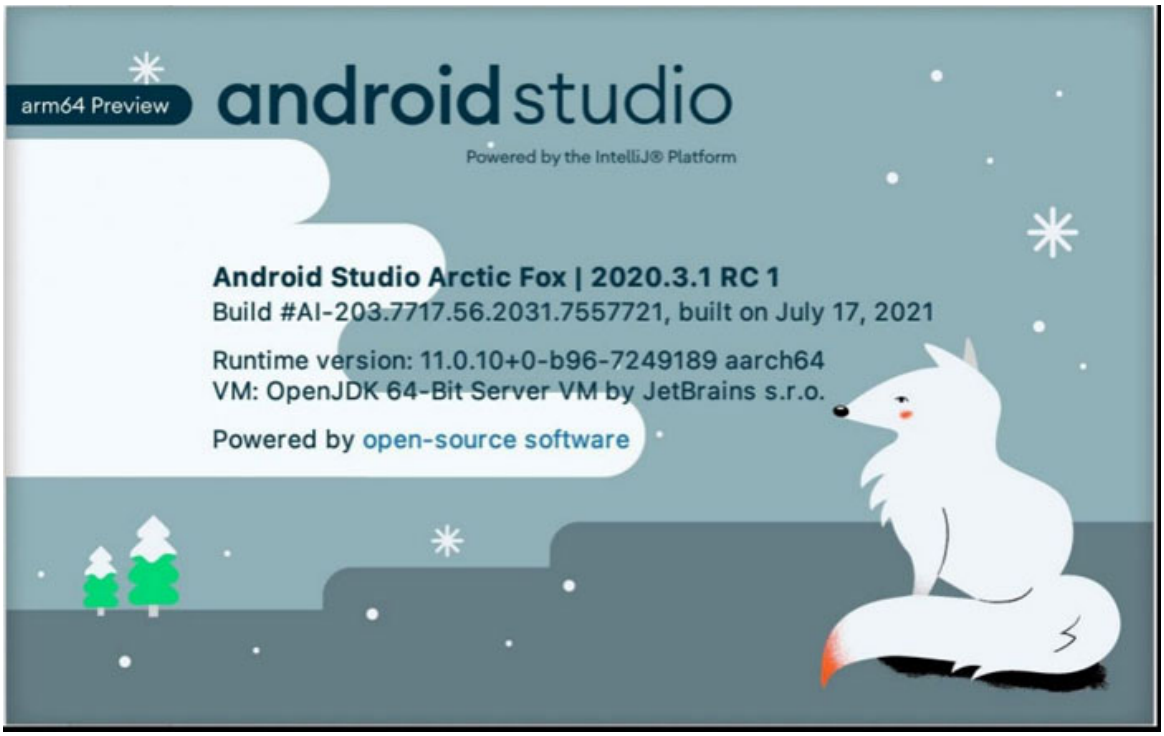



***Figure 2.1:*** *Android Studio, version 2020.3.1*

Moreover, we can use IntelliJ. Following are the versions of IntelliJ:

- IntelliJ IDEA Community, version 2021.2 or later.
- IntelliJ IDEA Ultimate, version 2021.2 or later.

## Installing Flutter and Dart Plugins

The steps to install Flutter and Dart differ according to the platform used.

The following steps are followed to install Flutter and Dart on Mac OS:

1. Start Android Studio.
2. Open plugin preferences as follows:
   Preferences | Plugins as of v3.6.3.0 or later.
3. Select the Flutter plugin and click on Install.
4. Click Yes, so as to install the Dart plugin.
5. Finally, Click Restart.

Following steps are followed to install Flutter and Dart on Windows/Linux:

1. Open plugin preferences (File | Settings | Plugins).
2. Select Marketplace, select the Flutter plugin and click Install.

### Linux or Windows

Use the following instructions for Linux or Windows:

1. Open plugin preferences (File | Settings | Plugins).
2. Select Marketplace, select the Flutter plugin and click Install.

VS Code: It is a lightweight editor which may be used for creating Flutter app. Latest stable version of VS Code may be installed.

Following are the steps to install Flutter and Dart plugins using VS Code:

1. Start VS Code.
2. Invoke View | Command Palette….
3. Type ‘install’ and select Extensions: Install Extensions.
4. From the extensions search field, select Flutter and click Install. This launches the Dart plugin.

## Validating the setup with the Flutter Doctor

Follow the given steps:

1. Select View | Command Palette.
2. Select Flutter: Run Flutter Doctor from the list.
3. Check the presence of issues if there are any in the output. In options, select Flutter.

Emacs: It is also one of the light-weight editors that supports Flutter and Dart. It is shown in *Figure 2.2*.

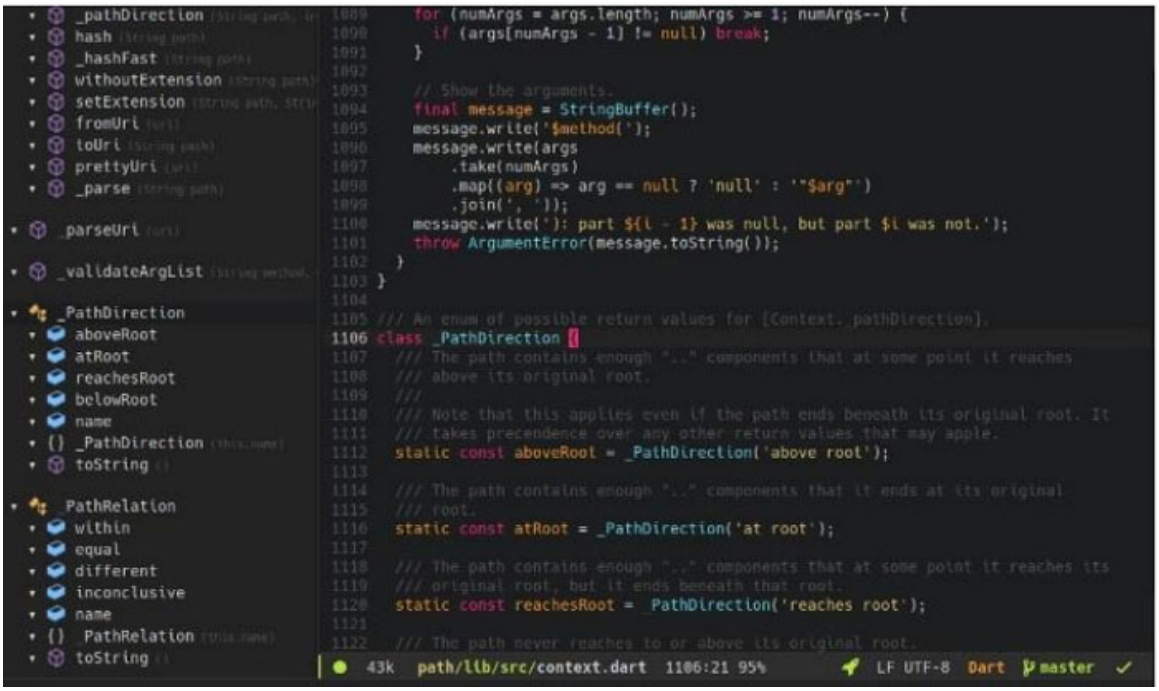



***Figure 2.2:*** *Emacs for Flutter and Dart*

# Tools for Building Flutter Based Application

Mobile Application development using Flutter and Dart has become very popular these days especially among cross-platform mobile developers. Some of the few essential Flutter and Dart based app development tools are listed as follows:

- **Panache**

  Panache is an open-source editor for creating Flutter based mobile applications. It is browser based and freely available that helps in creating fascinating and customizable themes for the Flutter app. Using this tool, we can update the colors of components as well as widgets and export the resultant theme as **`theme.dart`** file. It is shown in *Figure 2.3*.

***Figure 2.3:*** *Panache*

- **Supernova**

  Supernova is a tool that performs transformation of mobile designs (Sketch or Adobe XD) into code for iOS, Android, Flutter and React Native. It also performs asset creation and editing of code in real time. Currently, Supernova is available in the form of Mac app. This is shown in *Figure 2.4*:


***Figure 2.4:*** *Supernova*

- **Codemagic**

  It is a first CI/CD based Flutter tool created by Nevercode. We can integrate it with other tools as well. We can create, test and release apps. Its logo is shown in *Figure 2.5*.


***Figure 2.5:*** *Codemagic*

- **Sylph**

  Sylph is an open-source tool invented by Maurice McCabe. It can be used for running and testing Flutter apps in iOS and Android devices. Its logo is shown in *Figure 2.6*.


***Figure 2.6:*** *Sylph*

- **Amplitude**

  It is an analytics tool that helps to analyze how users are interacting with the app. It is a Flutter developer tool that helps in tracking and monitoring large amounts of data and performs integration with other mobile app development. Its logo is shown in *Figure 2.7*:

***Figure 2.7:*** *Amplitude*

- **Count.ly**

  It is an open-source tool for performing analysis on data. It is considered as one of the best tools for performing analysis in Flutter app development. It focuses on privacy and security of data. This tool helps in tracking KPIs and basic metrics to depict the performance of app.

  The paid version of Count.ly offers additional features, push notification and A/B testing. It is shown in *Figure 2.8*:


***Figure 2.8:*** *Count.ly*

- **AppsFlyer**

  This tool is useful in performing marketing campaigns and understanding the background of users using the app. It performs integration of more than 6000 media partners and analyses how campaigns and marketing channels are performing across apps and devices. Its logo is shown in *Figure 2.9*:


***Figure 2.9:*** *AppsFlyer*

- **Instabug**

  This tool is used to give real time insight of feature requests, crash reports, surveys, bugs reports and so on. Users can report bugs they encounter in the app in real time. This tool maintains a copy of screenshot and log that can be updated and in case an error occurs, it sends error report along with the log information. Its logo is shown in *Figure 2.10*:


***Figure 2.10:*** *Instabug*

- **Airship**

This is a customer engagement tool that creates a push notification and in-app inbox, where users can send and receive messages and share their experiences. Its logo is shown in *Figure 2.11*.


***Figure 2.11:*** *Airship*

- **WonderPush**

  It is also a customer engagement tool that comprises push notification and in-app inbox that promotes effective communication with the user. Other features offered by WonderPush include targeting, segmentation, A/B testing, geo-targeting and so on. Its logo is shown in *Figure 2.12*:

***Figure 2.12:*** *WonderPush*

- **RevenueCat**

  This tool is supported by iOS, Android and Stripe. With the help of this tool, we can manage subscriptions and in-app purchases. It keeps track of the

performance of the app by measuring churn, LTV, MRR and so on. Its logo is shown in *Figure 2.13*:


***Figure 2.13:*** *Revenue Cat*

- **Square**

  This tool is used along with Flutter SDK for payment management. In order to support in-app payment, this app supports stored cards and digital wallets. includes digital wallets and online wallets. Its logo along with Flutter logo is shown in *Figure 2.14*:

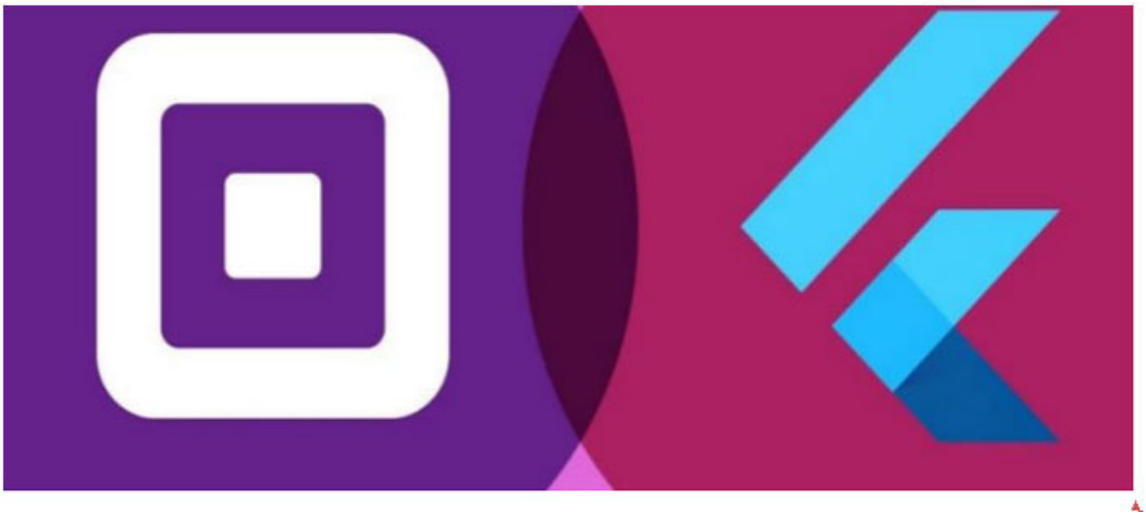


***Figure 2.14:*** *Square*

## Conclusion

Flutter is a newly developed technology for developing interactive mobile applications. In this chapter we have learnt about the installation of Flutter Framework, various editors used in Flutter and tools used in creation and updation of fascinating mobile applications. In the next chapter, we would discuss why Dart is needed for coding flutter and what programming aspects of Dart the reader should know to make the best use of flutter. We would cover all the essential skills of dart coding in the next chapter.

## Questions

1. What are the steps to install Flutter Framework?
2. Which editors may be used for building mobile applications using Flutter?
3. Explain tools required for building mobile applications using Flutter
4. Explain Supernova tool used for developing mobile applications using Flutter.

# CHAPTER 3

# Introduction to Dart

**"Mobile will ultimately be the way you provision most of your services. The way I like to put it is, the answer should always be mobile first. You should always put your best team and your best app on your mobile app."**

*– Eric Schmidt, Executive Chairman of Alphabet Inc.*

## Introduction

Dart was introduced by Google. It is an open-source programming language. Dart is a programming language that can be performed at the user side as well as the server side. The Dart SDK comprises of utility dart2js and Dart VM. Dart VM is the compiler in Dart and utility dart2js is used for generation of JavaScript equivalent for Dart code, since Dart may not be supportable on all the sites. Dart is similar to Java Programming. It is used for the creation of web applications and single page websites. An example of an application built using Dart, is Gmail.

## Structure

In this chapter, we will cover the following topics:

- Introduction to Dart
- Need for Dart coding for Flutter
- Declaring and Referencing Variables
- Operators in Dart
- Control Statements in Dart

## Objectives

Dart is an open source, object-oriented general-purpose programming language. Dart programming has a C like syntax. Dart programming supports concepts such as classes, interfaces and so in. However, Dart does not comprise of Arrays. However, it may perform replication of different data structures such as generics, arrays, optional typing. In this chapter, we will discuss in detail about Dart, the need of dart programming for flutter, variables, operators and control statements used in Dart.

## Introduction to Dart

Dart programming has a C like syntax. It was developed in 2011 by Google. Dart programming can create user interface for different web applications and mobile apps. Dart is a strongly typed language. It is similar to languages such as Java, JavaScript and C#. Like other programming languages, Dart supports concepts such as classes, functions, interfaces and so in. Dart has many similarities with other programming languages such as Kotlin, C#, Java and Swift. Some of the characteristics of Dart Programming are as follows:

- Type inference
- Statically typed
- String Expressions
- Features of functional and object-oriented programming
- Null Safe
- Dart has the ability for developing fast and variety of mobile applications across different platforms.

## Need for Dart Coding for Flutter

The reason why Flutter is popular among programmers is because of the programming language Dart. Dart is referred to as client optimized programming language, whose syntax is similar to C, C++ and Java. Using Dart, we can build web, mobile, server and desktop application. Dart makes use of **Ahead of Time** (**AOT**) and quickly compiles the code into native. It is easier for programmers to do customization using Dart. Animations and transitions are developed easily using Dart. Dart can perform garbage collection as well as object allocation. It does not make use of the concepts

of preemptive scheduling, locks and shared memory. The startup time of Dart codes is very fast. Flutter prefers using Dart as a language since Dart prevents the need to have separate declarative layout such as languages XML and JSX.

Dart has programmatic and declarative layout. Dart comprises of Just in Time compilation that decreases the development time and the response time drastically. Dart is simple and easy to understand since most of its features are very similar to the well-known dynamic and static languages. Dart is a flexible programming language in terms of execution and compilation. Dart is extraordinarily good at compilation at JIT and AOT.

At the development time, Dart does JIT compilation with the help of fast compiler. When the app is deployed and released, Dart performs AOT compilation. Dart has rapid and superior development cycles. It has faster startup and execution times. Compilation of Dart code can be done into JavaScript and the same dart code may be used in Flutter for development of both mobile application and web application. Dart is a very fast language to use. We can compile Dart into other languages. Dart is simple to read and understand. Dart is compiled and executed into native code without conversion into intermediate code. This makes Dart a faster language to execute, especially in case of animation where frames are displayed at 60fps.

Flutter and Dart are both developed by Google and do excellent job together by developing extraordinary impressive mobile and web applications.

## Declaring and referencing variables

Variable is a placeholder in computer's memory that holds some value. Each variable has a name. To declare name of a variable in Dart, we use var keyword.

We can add new variable to main in Dart as:

```
var name='Ryan';
```

A variable in Dart holds the references instead of directly holding values. We can declare a variable using a '**var**' keyword. Variables are also referred to as identifiers.

Rules for variable declaration in Dart are as follows:

- A variable in Dart cannot comprise of a special character such as whitespace, unicode character, keywords and mathematical symbols.
- The first character of a variable cannot be digit. It can be uppercase or lowercase alphabet *([A-Z]/{a-z])*.
- Variable names are case sensitive.
- Variable names may contain underscore (_) and dollar ($) sign but it does not contain special character such as @, #, &, ^ or *.

Variable may be specified along with its data type. Dart supports type checking that checks that the variable holds the value as it is specified in its data type.

For example, **`String name='Ryan';`**

If we do not want to change the variable, we can use final or const. A final variable is set just once. A const variable is a compile time constant.

```
final name='Ryan';
or, final String name='Ryan';
```

The value of const variables may be used in an arithmetic expression.

For example, **`const r=5..5;`**

The date types used in Dart include the following:

- **`int`**: Integers
- **`double`**: Floating-point numbers
- **`bool`**: Booleans
- **`String`**: Sequences of characters

The bool type in Dart is used to hold boolean type values.

```
bool variablename=true;
```

# Operators in Dart

Operators are a way to carry out various operations on the given operands. Similar to other languages such as Kotlin, C and Swift, Dart involves use of following operators:

- Arithmetic Operators
- Relational Operators
- Type Test Operators
- Bitwise Operators
- Assignment Operator
- Logical Operator
- Conditional Operator
- Cascade Operator

## Arithmetic operators

These operators are used to perform arithmetic operations on the operands. These are binary operators as they make use of two operands.

Following are the different types of Arithmetic operators:

- **Addition** (+): It is used to add two operands.
- **Subtraction** (-): It is used to perform subtraction between two operands.
- **Unary minus** (-expr): It reverses the sign of expression.
- **Multiplication** (*): It is used to multiply two operands.
- **Floating point Division** (/): It returns output of division in float.
- **Integer point Division** (~/): It returns output of division in integer.
- **Modulus** (%): It is used to return remainder obtained by division of two operands.

Example of Arithmetic Operator used in Dart Programming:

```
void main()
{
 int p= 6;
 int q = 5;

 // Addition of p and q
 var r = p+q;
 print("Addition of p and q is $r");
```

```
  // Subtracting p and q
  var s = p - q;
  print("The difference between p and q is $s");

  // Using unary minus
  var t = -s;
  print("The negation of difference between p and q is $t");

  // Multiplication of p and q
  var u = p * q;
  print("The product of p and q is $u");

  // Floating point Division of p and q
  var v = p /q;
  print("The quotient of p and q is $v");

  // Integer point Division of p and q
  var w = p ~ / q;
  print("The quotient of p and q is $w");

  // Remainder of p and q
  var x = p % q;
  print("The remainder of p and q is $x");
}
```

**Output:**

```
Addition of p and q is 11
The difference between p and q is 1
The negation of difference between p and q is -1
The product of p and q is 30
The quotient of p and q is 1.2
The quotient of p and q is 1
The remainder of p and q is 1
```

## Relational Operators

These operators are used to perform different relational operations on the operands. It is a binary operator since the relational operator is applied on two operands.

Following are different types of Relational Operators:

- **Greater than** (>): It checks whether the first operand is greater than the second operand or not, and gives the outcome as a boolean expression.
- **Lesser than** (<): It checks whether the first operand is smaller than the second operand or not, and gives the outcome as a boolean expression.
- **Greater than or equal to** (>=): It checks whether first operand is greater than or equal to the second operand and gives the outcome as a boolean expression.
- **Less than or equal to** (<=): It checks whether the first operand is smaller than or equal to the second operand and gives the outcome as a boolean expression.
- **Equal to** (==): It checks whether the two operands are equal or not and gives the outcome as a boolean expression.
- **Not Equal to** (!=): It checks whether the two operands are not equal to each other and gives the outcome as a boolean expression.

Consider the following Dart Program that makes use of relational operators:

```
void main()
{
 int p= 6;
 int q = 5;

 // Greater between p and q
 var r = p > q;
 print("p is greater than q is $r");

 // Smaller between p and q
 var s = p < q;
 print("p is smaller than q is $s");

 // Greater than or equal to
 var t= p >= q;
 print("p is greater than or equal to q is $t");

 // Less than or equal to
 var u = p <= q;
```

```
  print("p is smaller than or equal to q is $u");

  // Equality between p and q
  var v = p == q;
  print("p and q are equal is $v");

  // Unequality between p and q
  var w = p != q;
  print("p and q are not equal is $w");
}
```

**Output:**

```
p is greater than q is true
p is smaller than q is false
p is greater than or equal to q is true
p is smaller than or equal to q is false
p and q are equal is false
p and q are not equal is false
```

## Type test operators

These operators are used for performing comparison on operands. Type Test operators are of following types:

- **Is**: This operator returns true if the object has specific type.
- **is!**: This operator returns false if the object has specific type.

Consider the following Dart Program on Type Test Operator:

```
void main()
{
  String p = 'Ryan';
  double q = 2.7;

  // Comparison using is
  print(p is String);

  // Comparison using is not
  print(q is! int);
}
```

**Output:**

```
true
True
```

## Bitwise operators

These operators are used to perform bitwise operation on the operands. Bitwise operators are of the following types:

- Bitwise AND(&): It performs bitwise AND operation on two operands.
- Bitwise OR(|): It performs bitwise OR operation on two operands.
- Bitwise XOR(^): It performs bitwise XOR operation on two operands.
- Bitwise NOT(~): It performs bitwise NOT operation on two operands.
- Left Shift(<<): It shifts binary representation of p to q bits to left and inserts 0 to the right.
- Right Shift(>>): It shifts binary representation of p to q bits to right and inserts 0 from left.

Consider the following Dart code that makes use of Bitwise Operator:

```
void main()
{
  int p = 5;
  int q = 7;

  // Bitwise AND on p and q
  var r = p & q;
  print(r);

  // Performing Bitwise OR on p and q
  var s = p | q;
  print(s);

  // Performing Bitwise XOR on p and q
  var t = p ^ q;
  print(t);

  // Performing Bitwise NOT on p
```

```
  var u= ~p;
  print(u);

  // Performing left shift on p
  var v= a << b;
  print(v);

  // Performing right shift on a
  var w = a >> b;
  print(w);
}
```

**Output:**

```
5
7
2
4294967290
640
0
```

## Assignment Operators

These operators are used to assign values to the operands. Assignment Operators are of the following types:

- Equal to (=): It is used to assign value to a variable or an expression.
- Assignment Operator(??=)- It is used to assign value only if it is null.

Consider the following Dart program depicting the use of Assignment Operator:

```
void main()
{
  int p = 4;
  int q = 9;

  // Assigning value to r
  var r = p * q;
  print(r);

  // Assigning value to s
```

```
  var s;
  s ? ? = q - p; // Value is assigned as s is null
  print(s);

  // Again trying to assign value to s
  s ? ? = p+q; // Value is not assigned to s as s is not null
  print(s);
}
```

**Output:**

```
36
5
5
```

## Logical operators

These operators are used to logically combine two or more conditions. These operators are of the following types:

- **And Operator (&&):** It returns true if both the conditions are true.
- **OR Operator(||):** It returns true if either one or both the conditions are true.
- **Not Operator(!):** This operator is used to reverse the result.

The following Dart Program is used to depict the use of Logical Operators:

```
void main()
{
  int p = 8;
  int q = 13;

  // And Operator
  bool r = p > 10 && q < 15;
  print(r);

  // Using Or Operator
  bool s = p > 10 || q < 15;
  print(s);

  // Using Not Operator
  bool t = !(p > 10);
```

```
  print(t);
}
```

**Output**

```
false
true
true
```

## Conditional operators

These operators are used to perform comparison on operands. Following are the different types of Conditional Operators:

- **`(Condition)?Expression1:Expression2`**: If condition is true, then it returns expression 1. Otherwise, it returns expression 2.
- **`(Expression1)??Expression2`**: If expression 1 is not null, then it returns its value. Otherwise, it returns value of expression 2.

Consider the following Dart Program depicting use of Conditional Operator:

```
void main()
{
  int p = 9;
  int q = 8;

  // Conditional Statement
  var r = (p < 10) ? "Correct" : "Wrong";
  print(r);

  // Conditional statement
  int s;
  var t = s ? ? "Null value";
  print(t);

  // After assigning value to n
  s = 10;
  t = s ? ? " Null value";
  print(s);
}
```

**Output:**

```
Correct
Null value
10
```

## Cascade Notation Operator

This operator is used to perform multiple operations on the same element. It is denoted by dot(..)

The following Dart code is used to depict the function of Cascading Operator:

```
class abc {
  var p;
  var q;
  void set(x, y)
  {
    this.p = x;
    this.q = y;
  }
  void add()
  {
    var r = this.p + this.q;
    print(r);
  }
}
void main()
{

  // Creating objects of class abc
  abc o1 = new abc();
  abc o2 = new abc();

  // Without using Cascade Notation Operator
  o1.set(3, 6);
  o1.add();

  // Using Cascade Notation Operator
  o2..set(5, 6)..add();
```

```
}
```

**Output:**

```
9
11
```

# Control statements in Dart

Dart has two types of loop control statements:

- Break Statement
- Continue Statement

# Break statement

It is used to perform termination of loop.

**Syntax:**

```
break;
```

Consider the following code depicting the use of break:

```
void main()
{
 int c= 1;
 while (c <= 10) {
  print("The value is $count");
  c++;
  if (c == 5) {
   break;
  }
 }
 print("You are out of while loop");
}
```

**Output:**

```
The value is 1
The value is 2
The value is 3
```

```
The value is 4
You are out of while loop
```

In the preceding code, when c=5, it comes out of while loop and prints ‘You are out of while loop’.

## Continue statement

It is used to continue the flow of control. When a continue statement occurs, it jumps to the next iteration.

### Syntax:

```
continue;
```

The following Dart code shows use of continue statement:

```
void main()
{
 int c = 0;
 while (c <= 10) {
  c++;
  if (c== 5) {
   print("Number 5 is skipped");
   continue;
  }
  print("Value=$count");
 }
 print("You are out of while loop");
}
```

### Output:

```
Value=1
Value=2
Value=3
Value=4
Number 5 is skipped
Value=6
Value=7
Value=8
```

```
Value=9
Value=10
Value=11
You are out of while loop
```

Here, continue causes jump to the next iteration.

# Decision making statements

The decision-making statements allow only those statements to run that follow some condition. Decision making statements are discussed as follows.

## if Statement

The **`if`** statement checks the condition. If the condition is true, then statement within **`if`** is executed. Otherwise, the statement is ignored.

**Syntax:**

```
if ( condition ){
  // body of if
}
  void main()
{
  int value= 10;
  if (value> 3) {
   print("Condition is true");
  }
```

## if…else Statement

This type of statement checks if condition is true. The statements within are executed if the condition is true. If not, then **`else`** statements are executed.

**Syntax:**

```
if ( condition ){
  // body of if
}
else {
```

```
  // body of else
}
```

## else…if… Statement

If the **if** condition is true, then the statement under it is executed; otherwise, the next **if** condition is checked. If that is true, then the statement is executed. Otherwise, the next **if** condition is checked. If none of the **if** conditions are correct, then **else** statement is executed.

**Syntax:**

```
if ( condition1 ){
  // body of if
}
else if ( condition2 ){
  // body of if
}
.else {
  // statement
}
void main()
{
  int value= 10;
  if (value < 9) {
    print("1 is true");
    value++;
  }
  else if (value < 10) {
    print(" 2 is true");
  }
  else if (value >= 10) {
    print(" 3 is true");
  }
  else if (++value > 11) {
    print("4 is true");
  }
  else {
    print("All the conditions are false");
```

```
  }
}
```

**Output:**

```
3 is True
```

## Nested if Statement

In **`Nested if`**, the condition in **`if`** statement is checked; if the condition is true, then the condition which is inside **`if`** is checked. If the condition is true then the statement is executed. Otherwise, the statement under **`else`** is executed.

**Syntax:**

```
if ( condition1 ){
  if ( condition2 ){
  // Body of if
  }
  else {
  // Body of else
  }
}
void main()
{
  int value= 10;
  if (value > 9) {
    value++;
    if (value < 10) {
      print("It is true");
    }
    else {
      print("It is false");
    }
  }
}
```

**Output:**

```
It is false
```

## Switch

The switch statement performs evaluation of an expression, and then of the case which is matched with the expression value; its corresponding statements are then executed.

The switch statement evaluates an expression, matches the expression's value to a case clause and executes the statements associated with that case.

Following is the syntax.

```
switch(variable_expression) { case constant_expr1: { //
statements; } break; case constant_expr2: { //statements; }
break; default: { //statements; } break; }
```

Consider the following code in Dart that makes use of Switch statement:

```
void main() {
  var grade = "B";
  switch(grade) {
    case "A": { print("Excellent"); }
    break;
    case "B": { print("Good"); }
    break;
case "C": { print("Fair"); }
    break;
    case "D": { print("Poor"); }
    break;
  default: { print("Invalid choice"); }
    break;
}
}
```

**Output:**

```
Good
```

## For Loop

For Loop represents a definite loop. It executes a code for a fixed number of times. For loop comprises of initialization, condition and increment or

decrement.

**Syntax of For Loop:**

```
for (initial_count_value; termination-condition; step) {
//statements }
```

Following Dart code makes use of for loop:

```
void main() {
  var num = 4;
  var factorial = 1;
  for( var i = num ; i >= 1; i-- ) {
    factorial *= i ;
  }
  print(factorial); }
```

## While Loop

It performs evaluation of condition first; if condition is true, then block of code is executed.

The following is the syntax for the while loop.

```
while (expression) { Statement(s) to be executed if expression
is true }
```

Consider following Dart code that depicts use of while loop:

```
void main() {
  var num = 4
  var factorial = 1;
  while(num >=1) {
    factorial = factorial * num;
    num--;
  }
  print("The factorial is ${factorial}"); }
```

**Output:**

```
24
```

## Do-while loop

This loop is similar to **`while`**, with the only difference being that even if the condition is false, it executes the block of code at least once.

Following is the syntax for the do-while loop.

```
do { Statement(s) to be executed; } while (expression);
```

Consider the following dart code that depicts the use of do-while loop:

```
void main() {
  var n = 5;
  do {
    print(n);
    n--;
  }
  while(n>=0); }
```

**Output:**

```
5 4 3 2 1
```

# Conclusion

Dart and Flutter are products of Google. In this chapter, we have discussed in detail characteristics of Dart Programming, data types in Dart and operators in Dart Programming. We have also discussed about what makes dart programming most suitable to be used with Flutter Framework. In the next chapter, we will discuss about Classes and Functions in Dart, Import Packages and Implementing Asynchronous Programming.

# Questions

1. What makes Dart the most suitable language to be used with Flutter Framework?
2. Describe Operators in Dart.
3. Describe Control Flow in Dart.
4. Describe different data types in Dart.

# CHAPTER 4

# Classes and Functions in Dart

**“Our lives will be facilitated by a myriad of adaptive applications running on different devices, with different sensors, all of them collecting tidbits about everything we do, and feeding big digital brains that can adapt applications to our needs simply because they get to know us.”**

*– Márcio Cyrillo, Executive Director at CI&T*

## Introduction

Dart is an object-oriented programming language and thus, it comprises of classes, interfaces and so on. Dart also provides built-in support for the concept known as class.

## Structure

In this chapter, we will cover the following topics:

- Classes and Functions in Dart
- Import Packages
- Implementing Asynchronous Programming

## Objectives

Dart is an open source, object-oriented general-purpose programming language. It comprises of concepts such as classes, interfaces and so on. Dart helps in designing complex and innovative mobile and web applications with the help of classes and functions. In the following chapter, we will discuss in detail about classes and functions, import packages and asynchronous programming.

# Classes and Functions in Dart

Dart is an object-oriented programming language. So, it consists of interfaces, classes and so on. A class is used to represent similar objects that have properties and behavior that are common among objects. Data is encapsulated from object in a class.

# Declaration of class

A class keyword is used for declaring class in Dart. For defining a class, a class name is used after class keyword. The class body is placed within a pair of curly braces. The syntax for defining the class is as follows:

```
class classname {
  <fields>
  <getters/setters>
  <constructors>
  <functions>
}
```

After the class keyword, class name is mentioned. For naming a class, the rules for identifiers are followed.

A class definition can comprise of the following components:

- **Fields**: A field is used to represent the data related to objects and is declared within a class.
- **Setters and Getters**: Setters and Getters is used for initializing and retrieving the values from the fields of class. Every class comprises of a default getter/setter. The default getter/setter may be overridden with the help of explicitly defined setter/getter.
- **Constructors**: It is used for allocation of memory of objects present in a class.
- **Functions**: They are also referred to as methods that can be used to represent actions taken by the object.

All these previously-mentioned components when placed together in a class are referred to as data members.

```
Example: class declaration
class Student {
```

```
// field
String name = "Mickel";
// function
void show() {
 print(name);
} }
```

The previously-mentioned example declares a class Student. The class comprises of field called name. The **`show()`** is a function that is used to print the name of the student.

## Creating an instance of the class

In order to generate an instance of a class, we need to use the 'new' keyword followed by name of the class name. The syntax for the same is as follows:

**Syntax:**

```
var objectname = new classname([ arguments ])
```

The **`new`** keyword is used for performing instantiation.

The preceding expression is used for invoking a constructor. The constructor may be used for passing the values in case of a parameterized constructor.

Consider the following example of instantiation of a class:

```
var obj = new Student("name 1")
Accessing Attributes and Functions of class
The attributes and functions can be accessed through the
object. The data members of a class can be accessed using dot
notation.
//Accessing an attribute
obj.fieldname
//Accessing a function
obj.functionname()
```

Consider the following example that depicts how to access the attributes and functions in Dart.

```
void main() {
```

```
 Student s= new Student();
 s.show(); }
class Student {
 // field
 String name = "Mickel";
 // function
 void show() {
  print(name);
 } }
```

The output of the preceding code is as follows:

```
Mickel
```

# Dart Constructors

A constructor is used for initialization of a variable in a class. In Dart, the name of constructor is the same as that of the class. A constructor may be parameterized as it is a function. The difference between a function and a constructor is that a constructor cannot have a return type whereas a function can have a return type.

**Syntax of Constructor in Dart:**

```
Classname(parameterlist) {
 //body of constructor
}
```

For example:

```
The following example depicts the use of constructors in Dart
–
void main()
{
 Student s = new Student('Mickel');
}
class Student
{
 Student(String name)
{
  print(name);
 }
```

```
}
```

The preceding code should give the following output:

```
Mickel
```

## Named Constructors

Dart also includes features of named constructors that are used for defining multiple constructors within a class.

The syntax of named constructors is as follows:

```
Class_name.constructor_name(parameter list)
```

The following example depicts the use of named constructors in Dart:

```
void main()
{
 Student s1 = new Student.namedConst('Mickel');
 Student s2 = new Student();
}
class Student
{
 Student() {
  print("A non parameterized constructor is invoked");
 }
 Student.namedConst(String name)
{
  print("The name is : ${name}");
} }
```

The following output is produced:

```
The name is: Mickel
```

Non-parameterized constructor invoked.

## The ‘this’ Keyword

The keyword ’this’ means the current instance of a class. Here, the class field and parameter name are same. In order to avoid ambiguity and distinguish between a class field and the parameter field, the field of class is

prefixed by ’this’ keyword. The following example provides the use of ‘this’ keyword:

```
void main()
{
 Studenr s1 = new Student('Mickel');
}
class Student
{
 String name;
 Student(String name)
{
   this.name = name;
   print("The name is : ${name}");
 }
}
```

The following output is produced from the preceding code:

```
The name is : Mickel
```

## Getters and Setters in Dart class

Getters and Setters are also referred to as accessors and mutators. They help in initializing and retrieving the class fields values. Getters or accessors are denoted by the get keyword. Setters or mutators are denoted by the set keyword.

Every class comprises of a getter/setter. A getter comprises of no parameters and it returns a value, and the setter comprises of one parameter and it does not return any value.

The syntax for defining a getter is given as follows:

```
Return_type get identifier
{
}
A setter has a following syntax:
set identifier
{
}
```

The following example depicts the use of getters and setters in a Dart class:

```
class Student {
  String name;
  int age;
  String get studentname {
    return name;
  }
  void set studentname(String name) {
    this.name = name;
  }
  void set studentage(int age) {
    if(age<= 2) {
    //print("Age must be greater than 3");
    } else {
    // this.age = age;
    }
  }
  int get studentage {
    return age;
  } }
void main() {
  Student s1 = new Student();
  s1.studentname = 'Mickel';
  s1.studentage = 1;
  print(s1.studentname);
  print(s1.studentage); }
```

This program code should produce the following output:

```
Mickel
Age should be greater than 3
```

# Class Inheritance

The Dart Programming exhibits the concept of inheritance in which a child class or sub class can be derived from the parent class or super class.

A class can inherit from the parent class using an ‘extends’ keyword. A Child class is able to inherit the properties and methods from its respective

parent class, except the constructors. However, Dart does not support multiple inheritance.

Syntax of class inheritance is given as follows:

```
class childclassname extends parentclassname
```

In the following example, we have declared a class Vehicle. The class is extended by the Car class. There exists inheritance relation between parent class or super class and child class or derived class or sub class. The derived class or child class attains implicit access to data members of the parent class.

```
void main() {
  var obj = new Car();
  obj.printvehiclename(); } class Vehicle {
  void printvehiclename() {
    print("Calling printvehiclename defined in the Vehicle
    class");
  } } class Car extends Vehicle {}
```

The following output is generated from the preceding code:

```
Calling printvehiclename defined in the Vehicle class
```

## Types of Inheritance

Following are the types of inheritance:

- **Single level inheritance**: Every child class or derived class in Single level inheritance is able to extend from at most one parent class or super class.
- **Multiple Inheritance**: In this type of inheritance, the child class or the derived class can inherit from multiple parent classes. Dart Programming does not support multiple inheritance.
- **Multi-level inheritance**: In multi-level inheritance, a child class or a derived class may be inherited from another child class.

The following example depicts multi-level inheritance:

```
void main() {
  var obj = new Child();
  obj.str = "hello";
```

```
 print(obj.str); }
class Grandparent
{
 String str;
}
class Parent extends Grandparent{}
class Child extends Parent {} //The Child class indirectly
inherits from Grandparent class
```

The output of preceding code is as follows:

```
Hello
```

## Class Inheritance and Method Overriding

Method Overriding is a process in which a child class or a derived class is able to redefine the method present in the parent class.

The following example depicts Method Overriding in Dart Programming:

```
void main()
{
 Student s = new Student();
 s.m(27);
}
class Teacher {
 void m(int x)
{ print("value of x is ${x}");}
}
class Student extends Teacher {
 //Method Overridding
 void m(int y) {
  print("value of y is ${y}");
 } }
```

The output of preceding code is:

```
value of y is 27
```

In case of Method Overriding, the type and number of function parameters must match; otherwise, Dart Programming will give compile type error.

# The static keyword

The static keyword may be used along with data members of a class. A static variable can retain its values till the program finishes execution. Static members may be referenced by class name.

The following example depicts the use of static keyword:

```
class A
{
 static int num;
 static display()
{
   print("The value of num is ${A.num}") ;
 }
}
void main() {
 A.num = 27;
 // initializing the static variable }
 A.display();
 // invoking the static method }
```

The following output is obtained:

```
The value of num is 27
```

# The Super Keyword

The super keyword may be used for referring to the immediate parent or superclass of a child class. The super keyword can be used to refer to variable, property, or method of the immediate super class. The following example depicts the use of super keyword:

```
void main()
{
 Student s = new Student();
 s.m1(27);
}
class Teacher
{
   void m1(int y)
```

```
{
print("value of y is ${y}");
}
}
class Student extends Teacher {
 @Method override
 void m1(int x) {
  print("value of x is ${x}");
  super.m1(10);
  } }
```

It should produce the following output:

```
value of x is 27
value of y is 10
```

## Import Packages

Dart package is a huge collection of reusable code unit which is well organized. Mobile Applications or Web Applications developed using Dart programming may require packages or third-party libraries. A package comprises of functions, classes, code, compiled program and data sample. Dart comprises of variety of default packages that are loaded when Dart console begins. We need to install and load packages explicitly which are required and are not default packages.

## Dart Package Manager

Every programming language is provided with the functionality for dealing with the external packages such as Gradle or Maven is used for Java, Nuget is used for .NET, npm is used for Node.js, and so on. Dart has its own inbuilt package manager referred to as pub. Pub is used for organizing and managing third-party libraries, tools, dependencies, and is also used for installing the packages in the repository. Every Dart application comprises of **`pubspec.yaml`** file which comprises of metadata of file. The metadata of the package consists of author, version, application name, and description. The full form of the **Yaml** is a **Yet Another Markup Language**. The **`pubspec.yaml`** is used for downloading all the libraries that the application needs during programming. The **`pubspec.yaml`** file looks like the following:

```
name: 'vector_victor'
version: 0.0.1
description: An absolute bare-bones web app.
…
dependencies: browser: '>=0.10.0 <0.11.0'
```

Pub command line is used for creating, downloading, publishing and updating packages. Some of the pub commands is listed as follows:

- **`pub get`**: It is used for getting packages of application it is dependent on.
- **`pub upgrade`**: This command is used for upgrading application dependencies to its modern version.
- **`pub build`**: It is used for construction of web application. It creates build folder and comprises of all necessary scripts in it.
- **`pub help`**: It is used for getting help related to pub commands

### Installing a Package:

Following are the steps involved in the installation of package in a project:

1. The package name is written in the dependencies section given in **`pubspec.yaml`**
2. Run the following command to check whether a package is installed in project or not: **`pub get`**

For example, we can add xml package to our project by writing in the dependencies section. This is shown in the following example:

```
name: NewApp
version: 0.0.1
description: A dart application
dependencies:
xml:
```

We have successfully added xml to our project dependencies. We can now import Dart XML package in the project. It can be imported as follows.

```
import 'package:xml/xml.dart' as xml;
```

The Dart XML makes use of a **`parse()`** method to read the input string. Following is the syntax to read XML string:

```
xml. parse(String input)
```

# Dart Libraries

In Dart, the library comprises of a set of programming instructions or collection of routine. Dart has many built-in libraries that consists of routines such as functions, set of classes, and so on. A Dart library consists of functions, constants, exceptions, properties, typedefs, and classes.

## Importing a library

In Dart, we can use the library in a file by using an 'import' keyword. Using the import keyword, the library can be made available in the current working file. In a single file, multiple libraries can be used. The package manager pub provides the list of libraries and packages that can be used in the current program.

Some of the commonly known libraries are listed as follows:

- **`dart:io`**: This library comprises of HTTP, File, socket and I/O support that can be used for server applications. This library cannot be used for browser-based applications. We do not require to import this library since it is already imported by default.
- **`dart:core`**: This library comprises of various built-in types as well as various core functionalities. We need not import this library as it is imported by default.
- **`dart:math`**: This library comprises of mathematical constant, functions, random number generator and so on.
- **`dart:convert`**: It is used as encoder or decoder for conversion of various data representations such as JSON, UTF and so on.
- **`dart:typed_data:`** It is used to represent the list that can store fixed size data efficiently.

Consider the following example of importing a library function:

```
import 'dart:math'; // Importing built-in library
void main() {
 print("Square root of 49 is: ${sqrt(49)}");
}
```

**Output:**

```
Square root of 49 is: 7.0
```

**Explanation:**

In the preceding code, the built-in library imported is: **`'dart:math'`**. Using this library, we can use many mathematical functions. In this program, we have used **`sqrt()`** function with number. **`sqrt()`** function is used to find square root of a number.

## Name alias of Library

In Dart, if multiple libraries are imported together in program and if these libraries use the same function name, then it may result in a conflict. Dart helps in importing multiple libraries into the current working file. In case we use the same function name within different libraries, then it would lead to conflict while accessing these functions. To overcome this scenario, Dart uses a keyword acting as prefix. This is shown by the following syntax:

```
import 'libraryname' as prefix
Consider the following example -
// We are defining a library: happy.dart
library happy;
void hi(msg){
 print("Say hi ${msg}");
}
//Next, we are defining library: joy.dart
library joy;
void hi(msg){
 print("Say hello ${msg}");
}
// We can import the library with the as prefix.
import 'happy.dart';
import 'joy.dart' as jolly;
// as prefix prevents function name clashes
void main(){
 hi("Mickel");
 jolly.hi("James"); // To eliminate the name conflict
}
```

**Output:**

```
Say hi Mickel
Say hello James
```

# Implementing Asynchronous Programming

An asynchronous operation performs execution in a thread separately from the main thread. When the main application calls an asynchronous program to start functioning, then the application remains in execution until the asynchronous task is completing its task. In a Synchronous programming, the execution of the main application is blocked until the synchronous program completes its task.

Consider the following example that is based on synchronous programming:

```
import 'dart:io';
void main()
{
  print("Please enter your name :");
  // Name is read by the input given by user
  String name = stdin.readLineSync();
  // User's name is read
  print("Hello ${name}");
  print("End void main");
}
```

In the preceding code, **`readLineSync()`** is a synchronous method. This function **`readLineSync()`** is used to block the execution of the main application until it accepts the input from the user.

The output of the preceding code is as follows:

```
Enter your name :
Mickel // Accepts input from user
Hello Mickel
End void main
```

The disadvantage of Synchronous Programming is that if a synchronous method takes a long time to execute, then the remaining methods will also be at halt for a long time and will degrade the performance of the overall

application. Alternatively, an asynchronous programming involves no waiting or non-blocking of main application program. The **`dart:async`** package facilitates implementing asynchronous programming blocks in a Dart script.

Consider the following program which is based on Asynchronous Programming:

**Step 1**: Create a **`asyndata.txt`** file and save it in the data folder in the current project. This file comprises of the following contents:

1. Mickel
2. John
3. Jack
4. Sue

**Step 2**: Write an application program that reads the file without blocking other parts of the application.

```
import "dart:async"; import "dart:io";
void main(){
 File file = new File(
 Directory.current.path+"\\data\\asyndata.txt");
 Future<String> f = file.readAsString();
  f.then((data)=>print(data));
//Reading from the file doesn't block execution of main
application
 print("End of void main");
```

The output of the preceding code is as follows:

```
End of void main
Mickel
2. John
3. Jack
4. Sue
```

In the preceding code, "End of void main" is displayed before the text is read from the file. Here, the execution of main program is not blocked until the text is read from the file. Asynchronous program helps in running a program without getting blocked. Dart makes use of Future objects that

represents the asynchronous operations, in which a value may be displayed at a later amount of time.

Asynchronous operations make the program run without being blocked. Dart uses Future objects to represent asynchronous operations.

## Conclusion

Dart is an effective programming language that can be used to create web and mobile application using Flutter framework. In this chapter, we have discussed in detail about classes and functions in Dart, import packages and Asynchronous Programming in Dart.

## Questions

1. Explain with example how classes are defined in Dart.
2. Explain with example how functions are defined in Dart.
3. Explain the difference between Synchronous and Asynchronous Programming in Dart.

# CHAPTER 5

# Introduction to Widgets

**“The rich and interactive experiences we have come to expect on mobile apps have created new standards and expectations for all digital media including the web. The result is websites are evolving to become more app-like in their rich functionality.”**

*– Raj Aggarwal, CEO of Localytics*

## Introduction

Dart is an object-oriented programming language. So, it supports various object-oriented programming features such as classes, interfaces, and so on. Dart provides built-in support for the concept known as class.

## Structure

In this chapter, we will cover the following topics:

- Creating Files and Folders using Widgets
- Installation of File Manager
- Structuring Widgets
- Understanding Widget Tree

## Objectives

Flutter Widget is used to describe the user interface of a Flutter application. In the given current state and configuration, Flutter widgets describe how the user interface of Flutter application would look like. In the following chapter, we will discuss about creation of files and folders using widgets, structuring widgets and understanding widget tree.

## Creating files and folders using widgets

Whenever a Flutter application is built, a code is run inside the Widget. From the given widget, we can build a Flutter application.

Widget describes the look and feel of the Flutter application. When we perform modifications in the code, the modifications in the Flutter app are applied by computing the difference between the current state of the app and the state of the app prior to the modifications.

Flutter app comprises of the nested widgets, with one widget present inside the other. The root of the widget comprises of other sub components which are also widgets. The File Manager is one of the widgets that can be used to manage files and folders in Flutter. The File Manager Widget is compatible with Android and Linux. Developers are working to make it compatible with Windows as well.

*Figure 5.1* shows the representation of a widget tree:

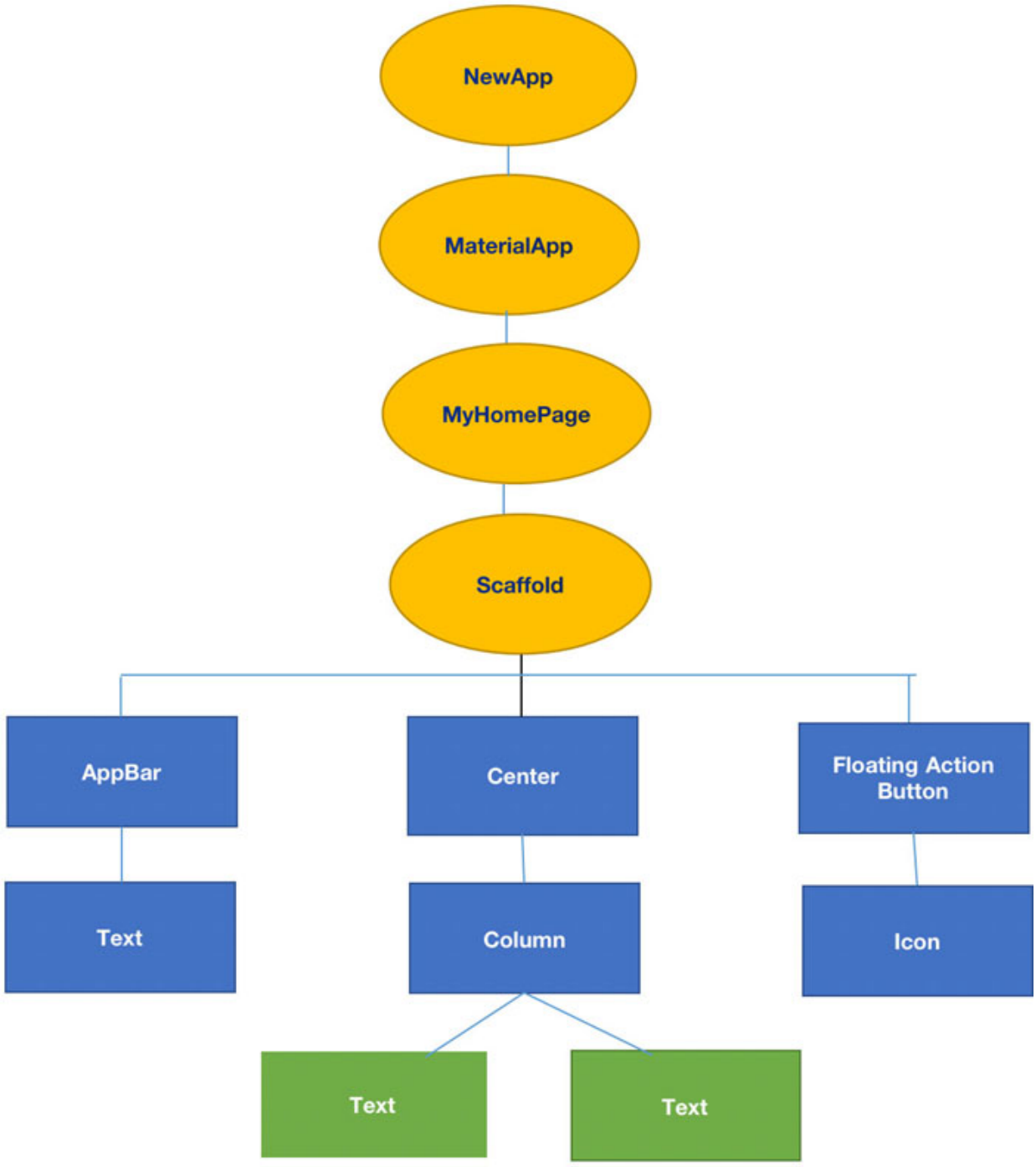
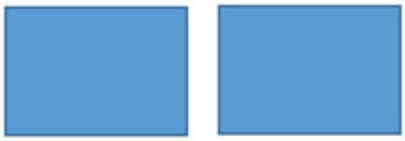
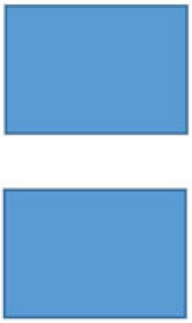



***Figure 5.1:*** *Representation of a Widget Tree*

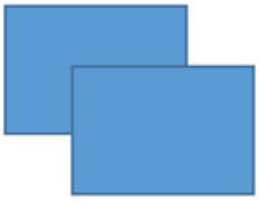
On the top of the tree, that is, root, there is a widget which is NewApp. Other components of widgets in a widget tree is also a widget. The different types of widgets in a Flutter application are represented in a widget tree. This is shown in the following *Figure 5.2*:



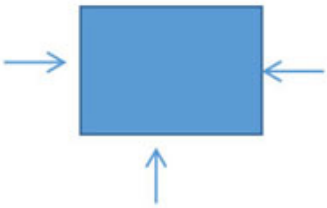
Row Layout is a collection of child widgets that are arranged in a horizontal way.



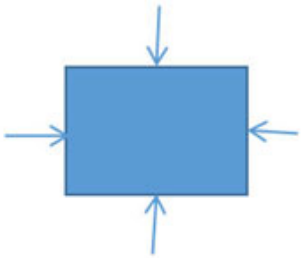
Column Layout is a collection of child widgets that are arranged in a vertical way.



Stack Layout is a collection of child widgets that are arranged in a overlapping way.



Padding Layout places child widget at a given padding location.



Center Layout places child widget at the centre location

***Figure 5.2:*** *Different representations of Widget Layouts*

# Installation of File Manager

In order to install **`pubspec.yaml`**, add the following line in **`pubspec.yaml`** file **`dependencies:`**

```
file_manager:^1.0.0
```

Provide storage permission to application. This is done by following the given ways.

For Android-based application, add **`write_external_storage`** and **`read_external_storage`** to **`android/app/src/main/AndroidManifest.xml`**. This is shown as follows:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
package="com.xxx.yyy">
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
…</manifest>
```

For Android 10, add the following:

```
<application
  android:requestLegacyExternalStorage="true"
  …
```

Storage Permission can be performed from app setting manually. Alternatively, a package, such as a **`permission_handler`** may be used, that provides storage permission from the app. The parameters used in File Manager are controller and builder. The controller may be used for updating the value and notifying the listeners. **`FileManager`** performs its self updating whenever the user performs modification of sort-type or path.

```
Final FileManagerController controller=FileManagerController();
```

Builder is a function that performs creation of custom widgets and also retrieves a list of entities represented by **`List <FileSystemEntity>`**.

Consider the following code that makes use of controller and builder function:

```
FileManager(
 controller: controller,
 builder: (context, snapshot) {
 final List<FileSystemEntity> entities = snapshot;
  return ListView.builder(
  itemCount: entities.length,
  itemBuilder: (context, index) {
   return Card(
    child: ListTile(leading:
    FileManager.isFile(entities[index])
      ? Icon(Icons.feed_outlined)
      : Icon(Icons.folder),
     title: Text(FileManager.basename(entities[index])),
     onTap: () {
      if (FileManager.isDirectory(entities[index])) {
       controller.openDirectory(entities[index]); // Opening
       a directory
      } else {
       // Perform tasks related to file
      }
     },
    ),
   );
  },
  );
 },
)
```

Following are the properties of a File Manager:

- **`loadingScreen`**: Using this property, a custom widget is created. By default, a circular progress indicator is provided.
- **`emptyFolder`**: A custom widget is created for making an empty screen.
- **`HideHiddenEntity`**: This property is used for hiding the files and folders that are hidden.

Some of the properties of **`FileManagerController`** are listed as follows:

- **`getSortedBy`**: This property returns the sorting type that is presently in use.
- **`setSortedBy`**: This property is used for setting the sorting type, that is, **`SortBy`** {name, type, date, size}
- **`getCurrentDirectory`**: This property is used for getting the current directory.
- **`getCurrentPath`**: This property is used for getting the current path.
- **`setCurrentPath`**: This property is used for setting the current path by providing a string value.
- **`isRootDirectory`**: This property returns true if current directory is root, and returns false if current directory is not root.
- **`goToParentDirectory`**: This property is used for jumping to the parent of directory, which is presently opened.
- **`OpenDirectory`**: This property is used for opening the directory specified.

Some of the other properties related to file handling are as follows:

- **`isFile`**: This property makes sure whether **`FileSystemEntity`** is a file or not.
- **`isDirectory`**: This property makes sure whether **`FileSystemEntity`** is a directory or not.
- **`setCurrentPath`**: This property sets the path of the current directory to the path specified in string.
- **`getStorageList`**: This property returns the list of available storage in the device. It returns empty if there is no storage in the device.
- **`createFolder`**: This property is used to create a directory having a particular name at a given path, if it does not exist.

# Structuring Widgets

In Flutter, widgets are the basic building blocks of the app's user interface. They are used to define the structure and appearance of the UI, as well as how it behaves.

Widgets can be simple, such as a button or a label, or they can be more complex, just as a list or a grid. You can use widgets to build the entire user

interface of your app, or you can use them to build smaller, reusable components that can be composed together to create more complex layouts.

Widgets in Flutter are immutable, which means that once they are created, their properties cannot be changed. This makes it easy to reason about the app's user interface and helps to ensure that the app is performant.

To update the UI, you simply create a new widget with the desired properties and replace the old widget with the new one. Flutter will automatically update the UI to reflect the changes.

There are many different types of widgets available in Flutter, and you can use them to build user interfaces that are simple or complex, static or dynamic. Some common types of widgets include:

- **`Container`**: A box that can contain other widgets and apply padding, margins, and other layout constraints.
- **`Text`**: A widget for displaying text.
- **`Image`**: A widget for displaying images.
- **`Button`**: A widget for creating buttons.
- **`TextField`**: A widget for entering text.
- **`ListView`**: A scrollable list of widgets.

Overall, widgets are a key part of building user interfaces in Flutter, and they provide a powerful and flexible way to create beautiful and interactive apps.

Structuring widgets is used for providing a proper structure to the page and arranging other widgets properly inside the page. The structuring widgets are also referred to as the layout widgets. They are of the following types:

- Columns and Sidebars
- Container
- Group Box
- Tab Container

Column and Sidebar widgets are the widgets that represent a fixed number of columns. All widgets in this category are arranged in rows and columns. The layout grid displays the page after widgets are arranged in rows and columns. The layout grid gives the views of page as displayed on different

devices. In the top bar, the switch device mode is used to find different views of the page on tablet, phone and desktop.

A Layout grid comprises of rows and columns. A row comprises of different elements that are placed one after the other. A column represents a cell in the row. There may be multiple items placed in one column.

The layout grid properties comprise of the following:

- Expand
- General
- Design

Select a row. In the Expand section, click on **Add row**; this adds a new row exactly to the one which was selected.

The General section in the layout grid comprises of a property called Full Width. When the property, Full Width is enabled, the layout grid occupies the complete width of the container they are placed in. When Full Width property is disabled, the layout grid is placed in the middle of the page and is adjusted according to the device used.

There are two sections in row and column properties:

- Expand
- Row Layout

The properties and functions in the Expand Section of Column and Row is the same as the layout grid. In row layout, the number of columns and the way they are arranged can be modified. Some of the properties in Row Layout include the following:

- **Desktop**: Using this property, we can modify the number and the width of columns in a desktop.
- **Tablet**: This property modifies the number and the width of column in a tablet.
- **Phone**: This property modifies the number and the width of column in a phone.
- **Container**: It represents a layout element that comprises of a widget or a group of widgets that we can style, delete or drag on page.

- **Group box**: It is used for grouping together widgets. The group box may be dynamically modified and can be expanded or collapsed according to the need. The group box comprises of a General section and the Design section. The properties in a General section are as follows:
    - **Show header**: This property is set to enable by default. It defines whether a header is shown above or not.
    - **Caption**: This property is enabled when the Show Header property is enabled. It is used to define the caption that would be displayed on the header.
    - **Collapsible**: This property is enabled when the Show Header property is enabled. It is used to define whether group box comprising of widgets will be collapsed or expanded.
- **Tab container**: It is a container that is used for organizing various information in the form of tabs. Organizing tab is helpful when the amount of available space on the screen is very less, as compared to the amount of space that needs to be displayed.
    - Tab container general properties comprise of the Tabs property. Radio buttons may be used for switching from one tab to next. The order of the tabs may be modified by clicking and dragging it and dropping it on the desired position. The Add New Tab option adds a new tab to the screen.

Every Tab has the following properties:

- **Caption**: This property is used to define the name of the tab. The caption may be modified by double clicking the tab on the page.
- **Default tab**: This property defines which tab will be visible or active when the page is displayed. By default, no tab is set to Default Tab property. If no tab is set to Default Tab property, then the first will be active or visible when the page is displayed.

## Understanding Widget Tree

Widget Tree describes simple and complex layouts built by positioning different widgets on a page. The widget tree must be as shallow as possible.

An app comprises of the collection of widgets. The type of app depends on the widgets contained in it. The page comprises of several widgets. These widgets, when combined together, form a widget tree. An element is associated with every widget. The collection of these elements forms an element tree. A render tree is defined as a low level layout structure that is formed through the combination of render objects.

The widget tree may be categorized into two types:

- Stateless widget tree
- Stateful widget tree

The Stateless Widget is shown in the following *Figure 5.3*:

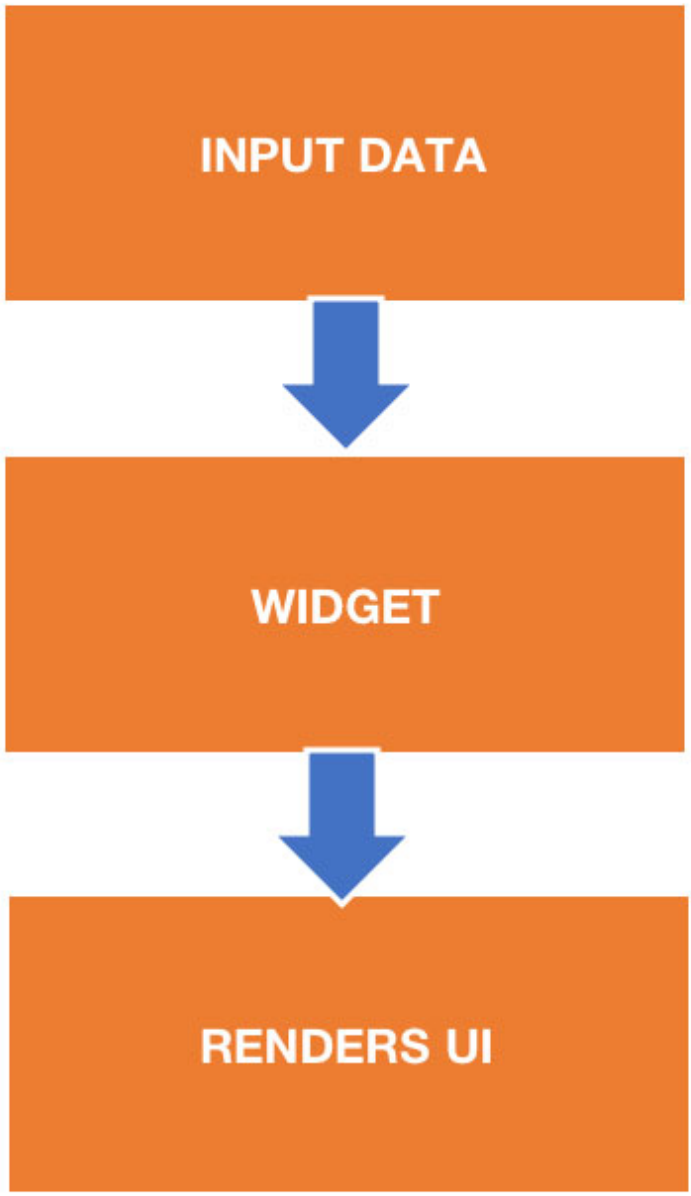



***Figure 5.3:*** *Stateless Widget*

Stateless widget tree is formed by the combination of stateless widgets. It is built when a non-dynamic application has to be constructed. Each stateless widget is also associated with an element. The combination of these elements form an element tree. To create an element in Flutter,

**`createElement`** method is used. The flutter backend assigns a unique id to each element that is created.

Following is the dart code that defines the Stateless Widget tree.

```
class Sampletest extends StatelessWidget {
  @override
  Widget build(BuildContext context){
  return Row(
    children: <Widget>[
      Icon(),
      Text(),
    ],
  );
  }
}
```

In the preceding code, we have created a class called **`Sampletest`** and it extends the class **`StatelessWidget`**. The **`StatelessWidget`** comprises of icon, row and text, as shown in the following *Figure 5.4*:

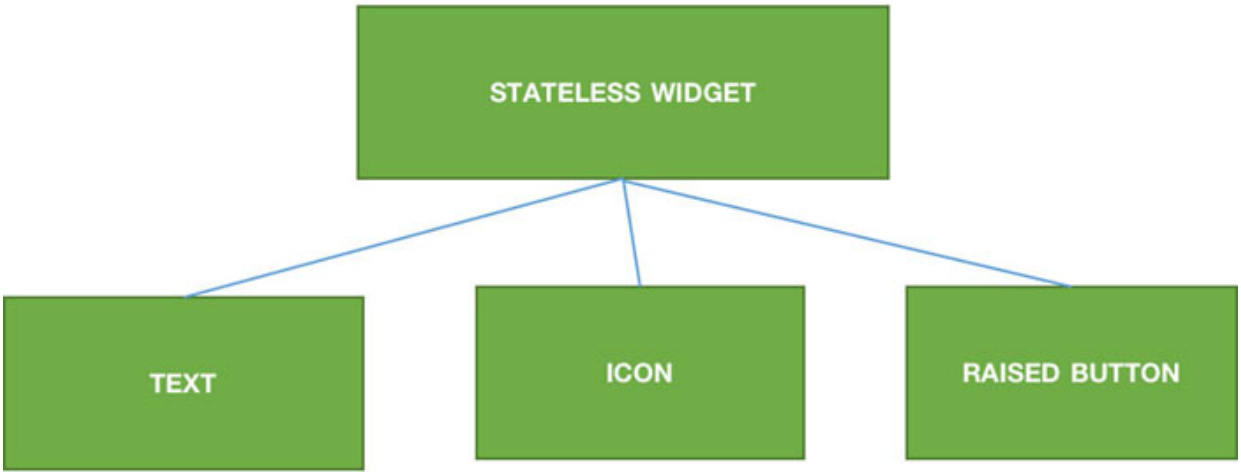



***Figure 5.4:*** *Examples of Stateless Widgets*

Statefull widget tree is formed by the combination of stateful widgets. It is built when a dynamic application has to be constructed. In a Statefull widget tree, when **`createElement`** method is called, a state object is also created. This is because the state is modified by maintaining the previous state unchanged and at the same time, elevating the performance as well. To update state objects or to set new data values, **`setState`** method is used.

Stateless Widget is one that does not store any value. Examples of Stateless Widget is shown in the following *Figure 5.5*:

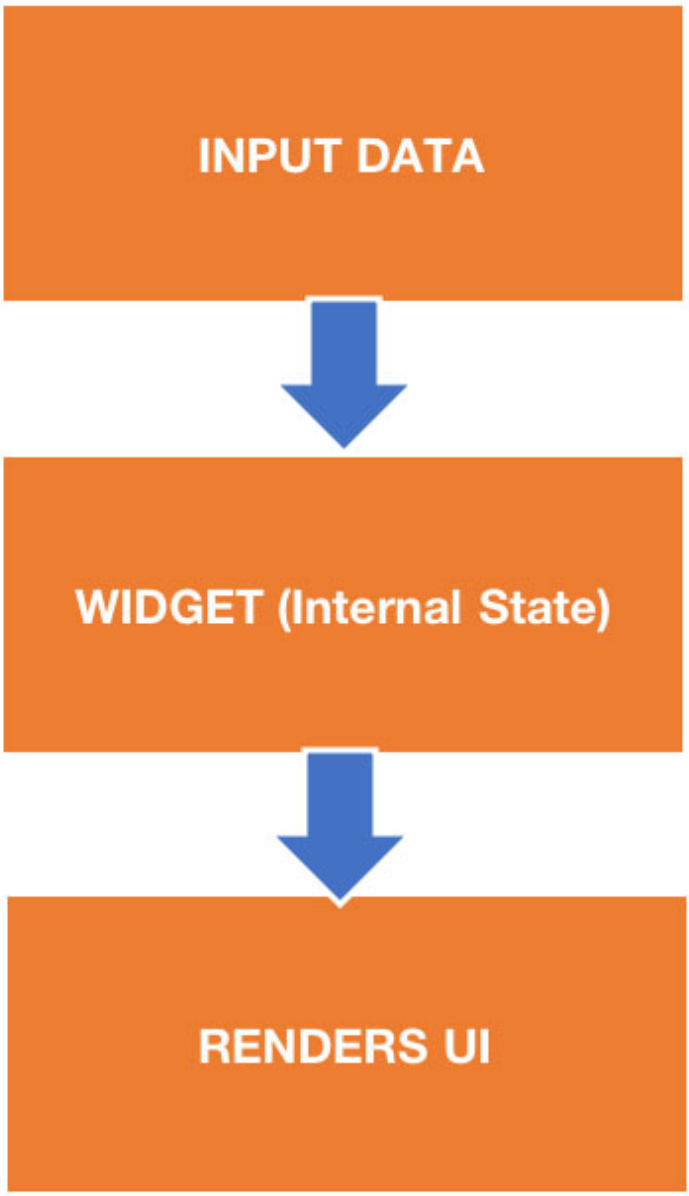



***Figure 5.5:*** *Stateful Widget*

The following dart code shows the use of **`setState()`**:

```
class Sampletest extend StateTextList {
  String letter = 'X';
```

```
  void _onPressed() {
   setState( () {
    letter= 'Y';
   });
  }
  @override
  Widget build (BuildContext xontext){
   return Column(
    children : <Widget> [
     Icon();
     Text('$letter),
     FlatButton(
      onPressed : _onPressed,
     ),
    ],
   );
  }
}
```

In the preceding code, **`Sampletest`** class extends the **`StateTestList`**. Here, **`setState()`** method is used to keep the initial value of state even after changing the state value.

## Inherited Widget

In Flutter, an **`InheritedWidget`** is a widget that holds a given state that may be inherited by the descendant widgets. This allows the descendant widgets to access given state without having to explicitly pass it down through the widget tree.

Here is an example of how we may use an **`InheritedWidget`** to provide a theme to all the widgets in a subtree:

```
class ThemeColor{
 final Color primaryColor;
 final Color secondaryColor;
}
class ThemeExtend extends InheritedWidget {
 final ThemeColor data;
```

```
  ThemeExtend({Key key, @required this.data, @required Widget
  child}) : super(key: key, child: child);

  static ThemeColor of(BuildContext context) {
  return
  (context.dependOnInheritedWidgetOfExactType<ThemeExtend>
  ()).data;
  }

  @override
  bool updateShouldNotify(ThemeExtend oldWidget) {
   return data != oldWidget.data;
  }
}
```

To use the **`ThemeExtend`**, one should wrap the app's root widget in a **`ThemeExtend`** and pass it a **`ThemeColor`** object. Then, any descendant widget that needs to access the theme can call **`ThemeExtend.of(context)`** to get the current theme.

```
void main() {
  runApp(
   ThemeExtend(
    data: ThemeColor(primaryColor: Colors.blue, secondaryColor:
    Colors.red),
    child: MyApp(),
   ),
  );
}
class NewButton extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
   final theme = ThemeExtend.of(context);
   return RaisedButton(
    color: theme.primaryColor,
    onPressed: () { /*…*/ },
    child: Text('Button'),
   );
  }
}
```

By using an **`InheritedWidget`**, you can avoid having to pass the theme down through multiple levels of the widget tree, which can make it easier to manage and change the theme at runtime.

In Flutter, widgets act as the building blocks in a app's user interface. They are used for defining the app looks and behaves. There are many different types of widgets available in Flutter, and we use them to create complex and interactive user interfaces.

Some uses for widgets in Flutter include the following:

- **For setting the user interface**: Widgets such as Container, Row, and Column are used for arranging other widgets on the screen.
- **Displaying text and images**: Widgets such as Text and Image are used for displaying text and images on the screen.
- **Handling user input**: Widgets such as **`TextField`** and Button allow users to fetch input data and interact with the app.
- **Building custom widgets**: We can create your own widgets by composing existing widgets together or by extending one of the built-in widget classes. This allows you to create reusable components that can be easily incorporated into different parts of your app.
- **Adding animations and transitions**: Widgets such as **`AnimatedContainer`** and **`FadeTransition`** allow you to add animations and transitions to your app to make it more visually appealing.

Overall, the use of widgets in Flutter allows you to quickly and easily build powerful and interactive user interfaces for your app.

One disadvantage of using widgets in Flutter is that they can make your codebase larger and harder to manage, especially if you use a large number of nested widgets. This can make it more difficult to understand and maintain the app's user interface, and can also affect the app's performance if not used carefully.

Another potential disadvantage is that widgets can be inflexible and may not always provide the precise level of control that you need over the app's user interface. In some cases, you may need to write custom widgets or use platform-specific APIs to achieve the desired effect.

Finally, because widgets are declarative, it can be harder to debug problems in your app's user interface compared to imperative approaches, where you

have more control over the execution of your code.

Overall, while widgets are a powerful and convenient tool for building user interfaces in Flutter, it is important to use them wisely and consider their tradeoffs when deciding how to implement your app's user interface.

## Conclusion

The kind of mobile or desktop application built is dependent on the type of widgets used. The arrangement of these widgets on the page defines the layout of the application. According to the type and number of widgets used, simple or complex layout-based, applications may be developed. In this chapter, we have discussed about how to create files and folders using widgets, various properties that may modify the layout of application, statefull widget tree and stateless widget tree.

## Questions

1. Explain the differences between Stateful widget tree and Stateless widget tree.
2. What is an element tree?
3. Explain how can you create file and folder using widget.

# CHAPTER 6

# Using Common Widgets

**“Mobile is becoming not only the new digital hub, but also the bridge to the physical world. That’s why mobile will affect more than just your digital operations — it will transform your entire business.”**

*– Thomas Husson,*
*Vice President and Principal Analyst at Forrester Research*

## Introduction

Widgets are the fundamental elements in a Flutter application. Widgets fall into different categories, on the basis of the task performed by them. A good Flutter app is dependent on the good choice of widgets used in building it.

## Structure

In this chapter, we will cover the following topics:

- Using Common Widgets
- Adding Animation to App
- Creating an App’s navigation

## Objectives

Widgets form the part and parcel of Flutter app. Widgets are very important since the decision to use the kind of widget decides the quality and the type of resultant Flutter app obtained. Widgets may belong to different categories ranging from general or basic widget, to widgets performing specialized tasks. In this chapter, we will discuss about basic widgets, different categories of widgets and widgets used in Flutter application for animation and navigation purpose.

# Using common widgets

Flutter is a User Interface toolkit developed by Google, and is used for the creation of apps based on iOS and Android, with the help of single code base. Widgets are the building blocks of entire mobile applications. Widgets describe the layout of a mobile application. There are various commonly used widgets, such as row widget, text widget, column widget, container widget and so on. Every element present in a Flutter application represents a widget. The entire view or layout of the mobile app depends on the kind of widgets in building a mobile application. The code structure of a mobile app is represented by the widget tree. Flutter widgets may be divided into 14 different categories on the basis of functionalities offered by them. These categories include the following:

- **Accessibility**: There are widgets that increase the accessibility of a Flutter application. These kinds of widgets come under this category.
- **Motion and animation**: The widgets that fall in this category help in adding motion and animation to other widgets.
- **Images, assets and icons**: These widgets help in displaying images and showing icons.
- **Async**: In a given Flutter Application, these categories of widgets help in providing async functionality.
- **Basics**: The widgets in this category provide basic or general functionality for development of Flutter application.
- **Cupertino**: These widgets are used for designing iOS-based applications.
- **Input**: In a given Flutter Application, these widgets are used for providing input facility.
- **Interaction models**: In a given Flutter Application, the widgets in this category are used for managing the touch events and provide different views to the users using the Flutter application.
- **Layout**: These widgets are used for placing other widgets on the page in an organized manner.
- **Material components**: These widgets follow the material design given by Google.

- **Painting and effects**: These widgets help in modifying the visual effect of the child widgets without affecting their shape and layout.
- **Scrolling**: These widgets provide scrolling feature to the other widgets that do not have scrolling feature by default.
- **Styling**: These widgets deal with modification of size, response and theme of Flutter app.
- **Text**: The widget in this category is used for displaying text.

Some of the basic widgets used in Flutter application are as follows:

- **Scaffold**: It is used for implementing basic design structure of app.
- **AppBar**: It is used for creation of the bar on the top of screen.
- **Text**: This widget is used to write text on the application.
- **Container**: This widget is used to contain other widgets within it.
- **Center**: It is used for aligning widgets at the center.
- **Image**: This widget is used for displaying images on the mobile application.
- **Placeholder**: This widget draws a box, as to where other widgets will be on it.
- **Row**: This widget places the child widgets in a horizontal manner.
- **FlutterLogo**: This widget is used to represent the IconTheme of Flutter.
- **ElevatedButton**: It is referred to as Material Designfilled Elevated button, as it gets elevated when it is pressed.
- **Appbar**: This widget comprises of toolbar and widgets such as FlexibleSpaceBar and TabBar.
- **Column**: This widget displays the child widgets in a vertical direction.
- **Stack**: This widget is used for placing widgets on top of each other. It is related to absolute positioning layout model of web.
- **GestureDetector**: This widget is used to recognize the gesture made by the user.

*Figure 6.1* provides a simple example of how some common widgets might be used in a Flutter app:

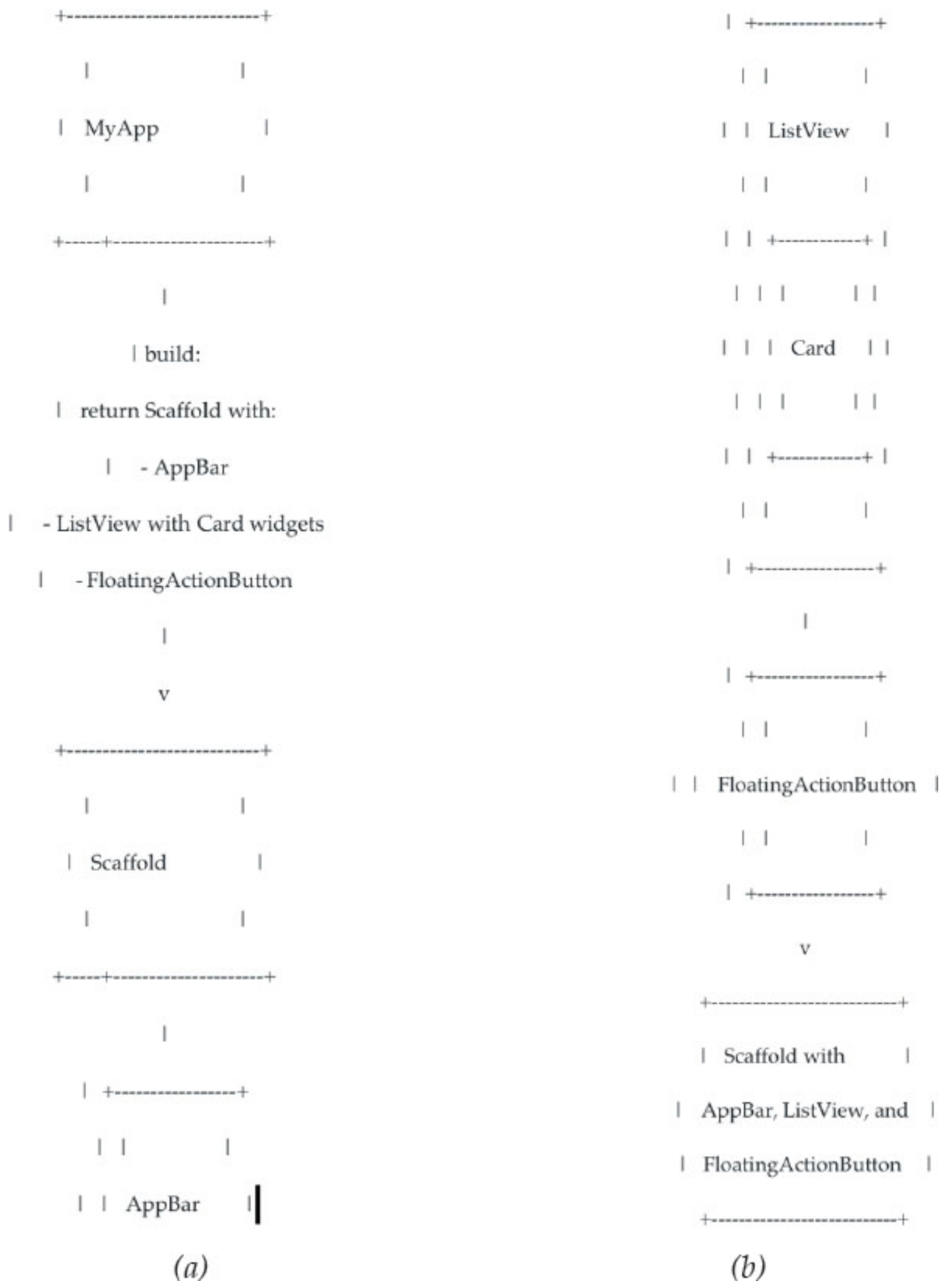



***Figure 6.1:*** *Example of how common widgets are used in Flutter app*

In this example, the MyApp widget is the root of the app, and it returns a Scaffold widget in its build method. The Scaffold widget contains an AppBar, a ListView with a number of Card widgets, and a FloatingAction Button.

## Adding Animation to App

There are numerous widgets that contribute in performing animation in mobile application. Some of these widgets include the following:

## AnimatedAlign Widget

This widget is responsible for changing the child's widget position. This widget makes use of curve and duration, for changing the alignment of the child widget.

The following code makes use of **`AnimatedAlign`** widget:

```
import 'package:flutter/material.dart';
void main() => runApp(const NewApp());

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

 static const String _title = 'Welcome to Flutter';
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   title: _title,
   home: Scaffold(
    appBar: AppBar(title: const Text(_title)),
    body: const NewStatefulWidget(),
   ),); }}
class NewStatefulWidget extends StatefulWidget {
 const NewStatefulWidget({super.key});
 @override
 State<NewStatefulWidget> createState() =>
 _NewStatefulWidgetState();
}
```

```
class _NewStatefulWidgetState extends State<NewStatefulWidget>
{
 bool selected = false;
 @override
 Widget build(BuildContext context) {
  return GestureDetector(
   onTap: () {
    setState(() {
     selected = !selected;
    });},
   child: Center(
    child: Container(width: 325.0,height: 325.0,color:
    Colors.blue,
     child: AnimatedAlign(alignment: selected ?
     Alignment.topLeft : Alignment.bottomRight,
      duration: const Duration(seconds: 3),curve:
      Curves.fastOutSlowIn,child: const FlutterLogo(size:
      75.0),
     ),),),
 );
}}
```

# AnimatedBuilder Widget

A general-purpose widget for building animations.

**`AnimatedBuilder`** is useful for more complex widgets that wish to include an animation as part of a larger build function. To use **`AnimatedBuilder`**, simply construct the widget and pass it a builder function.

For simple cases without additional state, consider using **AnimatedWidget**.

## Performance optimizations

If your builder function contains a subtree that does not depend on the animation, it is more efficient to build that subtree once instead of rebuilding it on every animation tick.

If you pass the pre-built subtree as the child parameter, the **AnimatedBuilder** will pass it back to your builder function so that you can

incorporate it into your build.

Using this pre-built child is entirely optional, but it can improve performance significantly in some cases and is therefore a good practice.

```
import 'dart:math' as math;
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
 const MyApp({super.key});

 static const String _title = 'Welcome to Flutter';
@override
 Widget build(BuildContext context) {
  return const MaterialApp(
   title: _title,
   home: MyStatefulWidget(),
  );
 }
}
class MyStatefulWidget extends StatefulWidget {
 const MyStatefulWidget({super.key});
 @override
 State<MyStatefulWidget> createState() =>
 _MyStatefulWidgetState();
}
class _MyStatefulWidgetState extends State<MyStatefulWidget>
  with TickerProviderStateMixin {
 late final AnimationController _controller =
 AnimationController(
  duration: const Duration(seconds: 15),
  vsync: this,
 )..repeat();
 @override
 void dispose() {
  _controller.dispose();
  super.dispose();
 }
 @override
```

```
 Widget build(BuildContext context) {
  return AnimatedBuilder(
   animation: _controller,
   child: Container(
    width: 300.0,
    height: 300.0,
    color: Colors.orange,
    child: const Center(
     child: Text('Congrats!'),
   ),
   ),
   builder: (BuildContext context, Widget? child) {
    return Transform.rotate(angle: _controller.value * 2.0 *
    math.pi,child: child,);
   },); }
}
```

## AnimatedBuilder

It is a type of widget in which Listenable notifies its listeners. It is applied to only specific widgets, leaving the rest as it is. The following code is based on **`AnimatedBuilder`** widget:

```
import 'package:flutter/material.dart';
class NewApp extends StatelessWidget {
 const NewApp({super.key, required
 this.counterValueNotifier});

 final ValueNotifier<int> counterValueNotifier;
 @override
 Widget build(BuildContext context) {return Center(child:
 Column(mainAxisAlignment: MainAxisAlignment.center,
  children: <Widget>[
   const Text(‘Value of Counter:'),
   AnimatedBuilder(animation: counterValueNotifier,builder:
   (BuildContext context, Widget? child) {
     return Text('${counterValueNotifier.value}');
    },),],),);}}
```

```
class ThisApp extends StatefulWidget {
 const ThisApp({super.key});
 @override
 State<ThisApp> createState() => _ThisAppState();
}
class _ThisAppState extends State<ThisApp> {
 final ValueNotifier<int> _counter = ValueNotifier<int>(0);
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
   home: Scaffold(appBar: AppBar(title: const
   Text('AnimatedBuilder example')),
    body: CounterBody(counterValueNotifier: _counter),
    floatingActionButton: FloatingActionButton(
     onPressed: () => _counter.value++,
     child: const Icon(Icons.add),
    ),),);
 }}
void main() {
 runApp(const ThisApp());
}
```

# AnimatedContainer Widget

The **`AnimatedContainer`** widget performs animation to the child widget whose property values changes with time duration. Properties that represent null are not animated. Consider the following code based on **`AnimatedContainer`** Widget:

```
import 'package:flutter/material.dart';
void main() => runApp(const NewApp());
class NewApp extends StatelessWidget {
 const NewApp({super.key});
 static const String _title = 'Flutter Code Sample';
 @override
 Widget build(BuildContext context) {
 return MaterialApp(
  title: _title,
```

```
   home: Scaffold(
    appBar: AppBar(title: const Text(_title)),
    body: const NewStatefulWidget(),
   ),); }}
class NewStatefulWidget extends StatefulWidget {
 const NewStatefulWidget({super.key});
 @override
 State<NewStatefulWidget> createState() =>
 _NewStatefulWidgetState();
}
class _NewStatefulWidgetState extends State<NewStatefulWidget>
{
 bool selected = false;
 @override
 Widget build(BuildContext context) {
 return GestureDetector(
  onTap: () {
   setState(() {
    selected = !selected;
   });
  },
  child: Center(
   child: AnimatedContainer(width: selected ? 300.0 :
   200.0,height: selected ? 300.0 : 200.0,
    color: selected ? Colors.red :
    Colors.blue,alignment:selected ? Alignment.center :
    AlignmentDirectional.topCenter, duration: const
    Duration(seconds: 2), curve: Curves.fastOutSlowIn, child:
    const FlutterLogo(size: 75),
  ), ), );
 }
}
```

## AnimatedCrossFade Widget

This widget is used to perform cross-fading between two given child widgets. The widgets having same width are faded.

## AnimatedDefaultTextStyle Widget

This widget performs transitions of default text style.

The properties such as **`textWidthBasis`**, **`softWrap`**, **`textAlign`**, **`overflow`**, **`maxLines`** and **`textHeightBehavior`** are not animated.

## AnimatedListState class Null safety

This widget performs animation when the child widgets are added or removed. When a child widget is added, animation starts running and is made through **`AnimatedList.itemBuilder`**. When a widget is removed, the animation is stopped and it is made through **`removeItem`** builder parameter.

## AnimatedModalBarrier class Null safety

It is a widget that obstructs the user from interaction with other widgets behind it and can be represented with an animated color value.

## AnimatedPhysicalModel Widget

This widget performs animation of elevation and **`borderRadius`**. If the **`animateColor`** property is kept to set, then the color is animated. Shape is not animated using this widget.

## AnimatedPositioned Widget

This widget is used for performing transition on child's widget position.

There are basically two types of animation used in Flutter mainly:

- Code-based animations, and
- Drawing-based animations

### Code-based animations

The main aim of code-based animations is to perform animation on the widgets such as row, container, column, stack and so on. It helps in modifying the position, size, scale, among others, of the widget. Using explicit and implicit animations, code-based animations may be created.

## Drawing-based animations

The main aim of Drawing-based animations is to animate drawings.

The **`AnimationController`** is used for controlling the duration and timing of the animation. The Animation is a value that changes over time, and the **`AnimatedBox`** widget is built using the **`AnimatedWidget`** class, which listens to changes in the animation and rebuilds itself whenever the animation changes.

We can also use the **`AnimationBuilder`** widget for building more complex animations with the help of Animation and **`AnimationController`** classes.

The following *Figure 6.2* illustrates the animation of a widget's size in a Flutter app using the **`AnimationController`** and **`AnimatedWidget`** classes.

The arrow labeled **`AnimationController ticks`** represents the action of the **`AnimationController`**, generating tick events at regular intervals. These tick events cause the Animation to update its value, which in turn causes the **`AnimatedBox`** widget to rebuild itself with the new value. This results in the appearance of the widget animating its size over time.

The **`AnimatedBox`** widget is built using the **`AnimatedWidget`** class, which listens to changes in the Animation and rebuilds itself whenever the Animation changes. The Animation is controlled by the **`AnimationController`**, which is responsible for changing the value of the Animation over time. The **`AnimatedBox`** widget uses the value of the Animation to set the size of the contained Container widget.

Refer to the following *Figure 6.2*:

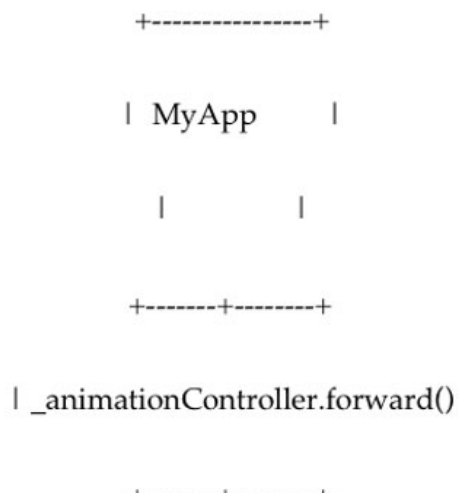



***Figure 6.2:*** *Adding animation in Flutter app*

# Creating an App's Navigation

In Flutter, the pages and screens are referred to as routes.

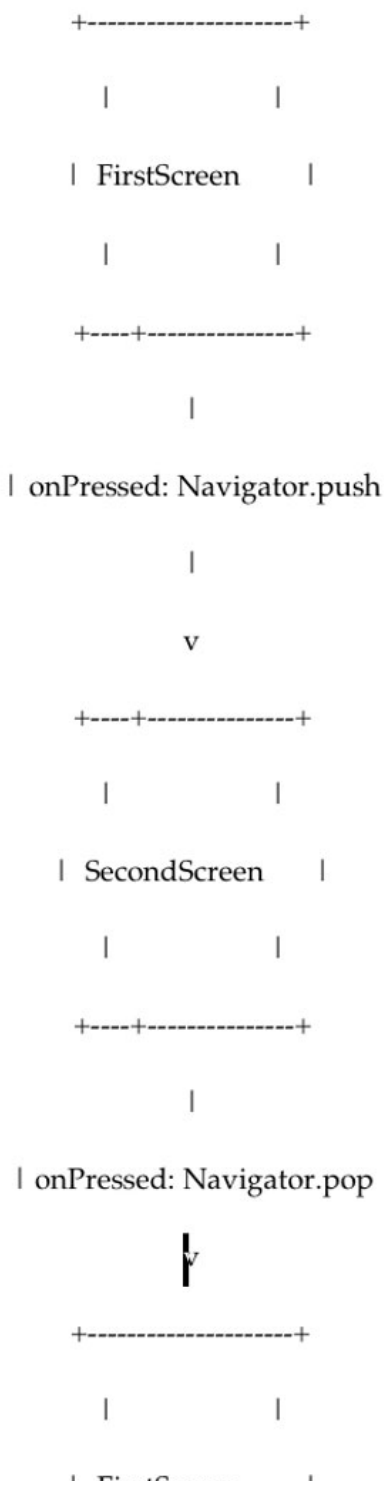
In Android, a route is referred to as an Activity. In iOS, a route is like a `ViewController`. In Flutter, a route is termed as widget.

In Flutter, we can perform navigation between two routes by carrying out the following steps:

1. Build two routes.

2. Use **`Navigator.push()`** for navigating to the second route.
3. Use **`Navigator.pop()`** for navigating to the first route.

App Navigation is shown in the following *Figure 6.3*:



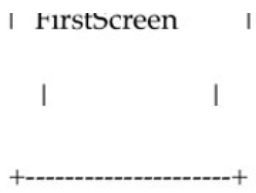



***Figure 6.3:*** *App Navigation in Flutter*

## Build two routes

Let us create two routes. On each route, there is a button. When we tap button on the first route, we are able to jump to the second route and if we tap button on the second route, we are able to jump to the first route.

```
Class Routeone extends StatelessWidget {
 const Routeone({super.key});

 @override
 Widget build(BuildContext context) {
 return Scaffold(
  appBar: AppBar(
   title: const Text('Route One'),
  ),
  body: Center(
   child: ElevatedButton(
    child: const Text('Open route'),
    onPressed: () {
      },
    ),
   ),
  );
 }
}
class Routetwo extends StatelessWidget {
 const Routetwo({super.key});

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
```

```
      title: const Text('Route Two'),
    ),
    body: Center(
      child: ElevatedButton(
      onPressed: () {
          },
        child: const Text('Go back!'),
      ),
    ),
   );
  }
}
```

## Use Navigator.push() to navigate to second route

We can use **`Navigator.push()`** method in order to switch to new route. The Navigator uses **`push()`** method to add a Route to the stack of routes. In the **`build()`** method of the **`Routeone`** widget, we can update the **`onPressed()`** method as follows:

```
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => const Routetwo()),
  );
}
```

## Use Navigator.pop() method to return to the first route

The **`pop()`** method is able to eliminate the current Route from the stack of all routes. To implement it, we can update the **`onPressed()`** callback in the Routetwo widget as:

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

**Example:**

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

```
void main() {
 runApp(const MaterialApp(
   title: 'Navigation Basics',
  home:Oneroute(),
 ));
}
class Oneroute extends StatelessWidget {
 const Oneroute({super.key});

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: const Text('Route One'),
   ),
   body: Center(
    child: ElevatedButton(
     child: const Text('Open the route'),
     onPressed: () {
      Navigator.push(
       context,
       MaterialPageRoute(builder: (context) => const
       Tworoute()),
      );
     },
    ),
   ),
  );
 }
}
class Tworoute extends StatelessWidget {
 const Tworoute({super.key});

 @override
  Widget build(BuildContext context) {
   return Scaffold(
    appBar: AppBar(
     title: const Text('Route Two'),
```

```
      ),
      body: Center(
       child: ElevatedButton(
        onPressed: () {
         Navigator.pop(context);
        },
        child: const Text('Please return back..'),
       ),
      ),
     );
   }
}
```

The preceding code explains how we can perform navigation from one route to another in Flutter on pressing button.

## Conclusion

Widgets are the building blocks of a Flutter application. The type of widgets used, decides the type of flutter application built. Animation in Flutter is the process of changing the visual properties of a widget over a period of time. It can be used to add visual interest and polish to an app, and can also be used to help convey information to the user.

To create an animation in Flutter, we can use the **`AnimationController`** class and the **`AnimatedWidget`** class. The **`AnimationController`** is responsible for controlling the duration and timing of the animation, while the **`AnimatedWidget`** listens to changes in the animation and rebuilds itself whenever the animation changes.

In this chapter, we have discussed some basic widgets, various categories of flutter, widgets responsible for creating animation and navigation in Flutter application.

## Questions

1. Explain some basic widgets used in Flutter.
2. What are the different categories of widgets used in Flutter?
3. How can we create animation using different widgets in Flutter?

4. How can we perform navigation in Flutter?

# CHAPTER 7

# Building Flutter Application

**“I think the biggest change, and the one we’re already starting to see take shape, is that globally the majority of internet usage will be done via a mobile device and for most people the mobile web will be their primary – if not their only – way of experiencing the internet.”**

*– Peter Rojas, Co-founder of Engadget and Gizmodo*

## Introduction

Widgets are the fundamental elements in a Flutter application. Widgets fall into different categories, on the basis of the task performed by them. A good Flutter app is dependent on the good choice of widgets used in building it. Flutter application may be built using Android specific code or iOS specific code.

## Structure

In this chapter, we will cover the following topics:

- Building Flutter Application using Android specific code
- Building Flutter Application using iOS specific code

## Objectives

Widgets form the part and parcel of Flutter app. Widgets are very important since the decision to use the kind of widget decides the quality and type of resultant Flutter app obtained. Widgets may belong to different categories, ranging from general or basic widgets, to widgets performing specialized tasks. In this chapter, we will discuss about building flutter application using Android specific code and iOS specific code.

# Building Flutter application using Android specific code

Flutter allows code to be written for a given platform in a language that is supported by that platform. The languages supported by the platforms include the following:

- On iOS, Objective C or Swift is used.
- On Windows, C++ is used.
- On Linux, C is used.
- On MacOS, Objective-C is used.
- On Android, Kotlin or Java is used.

The message passing of structured messages along with code generation, is performed by Flutter's built-in platform-specific API support, that makes use of the Pigeon package. This is performed in the following steps:

1. In an app, the Flutter part is capable of sending structured typedef messages to the host, which is non dart part of the application on a given platform channel.
2. The host is able to receive message from a platform channel. With the help of native programming languages, numerous platform-specific APIs are called. The back response is sent by the Flutter part of app.

At the client side, method calls are performed by **`MethodChannel`** that is capable of sending messages. At the platform side, message is received by the **`MethodChannel`**. **`MethodChannel`** is referred to as **`MethodChannelAndroid`** in case of Android; and **`FlutterMethodChannel`** in case of iOS, is referred to as **`MethodChanneliOS`**.

The platform channel makes use of message codec that performs binary serialization of JSON-like values, such as Strings, booleans, byte buffers, numbers and Lists. When we send as well as receive these messages, the process of deserialization and serialization takes place.

The mapping of Dart values into Java is shown in the following *Table 7.1*:

| Dart | Java |
|---|---|
| int | `Int` |
| null | `null` |

| | |
|---|---|
| Int64List | `long[]` |
| bool | `java.lang.Boolean` |
| Float64List | `double[]` |
| int, if 32 bits not enough | `java.lang.Long` |
| Int32List | `int[]` |
| Float32List | `float[]` |
| String | `java.lang.String` |
| double | `java.lang.Double` |
| Uint8List | `byte[]` |
| Map | `java.util.HashMap` |
| List | `java.util.ArrayList` |

***Table 7.1:*** *Mapping of Dart Values into Java*

The mapping of Dart values into Kotlin is given in the following *Table 7.2*:

| Dart | Kotlin |
|---|---|
| int | `java.lang.Integer` |
| null | `null` |
| Int64List | `LongArray` |
| bool | `Boolean` |
| Float64List | `DoubleArray` |
| int, if 32 bits not enough | `Long` |
| Int32List | `IntArray` |
| Float32List | `FloatArray` |
| String | `String` |
| double | `Double` |
| Uint8List | `ByteArray` |
| Map | `HashMap` |
| List | `List` |

***Table 7.2:*** *Mapping of Dart Values into Kotlin*

The mapping of Dart values into Objective-C is shown in the following *Table 7.3*:

| Dart | Objective-C |
|---|---|
| | |

| | |
|---|---|
| int | **NSNumber numberWithInt:** |
| null | **nil (NSNull when nested)** |
| Int64List | **FlutterStandardTypedData typedDataWithInt64:** |
| bool | **NSNumber numberWithBool:** |
| Float64List | **FlutterStandardTypedData typedDataWithFloat64** |
| int, if 32 bits not enough | **NSNumber numberWithLong:** |
| Int32List | **FlutterStandardTypedData typedDataWithInt32:** |
| Float32List | **FlutterStandardTypedData typedDataWithFloat32:** |
| String | **NSString** |
| double | **NSNumber numberWithDouble:** |
| Uint8List | **FlutterStandardTypedData typedDataWithBytes:** |
| Map | **NSDictionary** |
| List | **NSArray** |

***Table 7.3:*** *Mapping of Dart Values into Objective-C*

The mapping of Dart values into Swift is shown in the following *Table 7.4*:

| Dart | Swift |
|---|---|
| int | `NSNumber(value: Int32)` |
| null | `nil` |
| Int64List | `FlutterStandardTypedData(int64: Data)` |
| bool | `NSNumber(value: Bool)` |
| Float64List | `FlutterStandardTypedData(float64: Data)` |
| int, if 32 bits not enough | `NSNumber(value: Int)` |
| Int32List | `FlutterStandardTypedData(int32: Data)` |
| Float32List | `FlutterStandardTypedData(float32: Data)` |
| String | `String` |
| double | `NSNumber(value: Double)` |
| Uint8List | `FlutterStandardTypedData(bytes: Data)` |

| | |
|---|---|
| Map | Dictionary |
| List | Array |

***Table 7.4:*** *Mapping of Dart Values into Swift*

The mapping of Dart values into C++ is shown in the following *Table 7.5*:

| Dart | C++ |
|---|---|
| int | `EncodableValue(int32_t)` |
| null | `EncodableValue()` |
| Int64List | `EncodableValue(std::vector)` |
| bool | `EncodableValue(bool)` |
| Float64List | `EncodableValue(std::vector)` |
| int, if 32 bits not enough | `EncodableValue(int64_t)` |
| Int32List | `EncodableValue(std::vector)` |
| Float32List | `EncodableValue(std::vector)` |
| String | `EncodableValue(std::string)` |
| double | `EncodableValue(double)` |
| Uint8List | `EncodableValue(std::vector)` |
| Map | `EncodableValue(std::map<EncodableValue, EncodableValue>)` |
| List | `EncodableValue(std::vector)` |

***Table 7.5:*** *Mapping of Dart Values into C++*

The mapping of Dart values into C is shown in the following *Table 7.6*:

| Dart | C |
|---|---|
| int | `FlValue(int64_t)` |
| null | `FlValue()` |
| Int64List | `FlValue(int64_t*)` |
| bool | `FlValue(bool)` |
| Float64List | `FlValue(double*)` |
| Int32List | `FlValue(int32_t*)` |
| Float32List | `FlValue(float*)` |
| String | `FlValue(gchar*)` |

| | |
|---|---|
| double | `FlValue(double)` |
| Uint8List | `FlValue(uint8_t*)` |
| Map | `FlValue(FlValue, FlValue)` |
| List | `FlValue(FlValue)` |

***Table 7.6:*** *Mapping of Dart Values into C*

Consider the following example in which platform specific API is called, that can be used for showing the battery level status. For this purpose, an iOS **`device.batteryLevel API`**, **`getBatteryLevel()`**, **`GetSystemPowerStatus API`** and Android **`BatteryManager API`** is used. Following are the steps for creating Flutter app.

# Install Flutter and set up an editor

Install Flutter. Follow the instructions at **https://flutter.dev/docs/get-started/install** to install Flutter on your machine.

Set up an editor: install an editor of your choice, such as Visual Studio Code (**https://code.visualstudio.com/**), Android Studio (**https://developer.android.com/studio**), or IntelliJ IDEA (**https://www.jetbrains.com/idea/**).

# Create a new Flutter project

Create a new Flutter project. Open your editor, and use its terminal to create a new Flutter project by running the flutter create command.

To create a new Flutter project in your editor, follow these steps:

1. Open your editor and access the terminal.
2. Run the following command to create a new Flutter project:

   ```
   flutter create <project_name>
   ```

3. Replace **`<project_name>`** with the desired name for your project. This will create a new Flutter project with the specified name in the current directory.
4. After the project has been created, navigate to the project directory by running the following command:

   ```
   cd <project_name>
   ```

5. You can then open the project in your editor by running the following command:
   ```
   open -a <editor_name> .
   ```
6. Replace **`<editor_name>`** with the name of your editor (for example, Visual Studio Code, Android Studio, and so on). This will open the project in the specified editor.
7. You should now be able to see the files and directories of the Flutter project in your editor. You can start building your Flutter app from here.

## Designing the app's UI

Use the Flutter framework's widgets to design the app's UI. You can use the built-in widgets, or create your own custom widgets.

To design the app's UI using Flutter, you can use a combination of built-in widgets and custom widgets.

To start, you can create a new Flutter project in your preferred development environment and navigate to the **`lib/main.dart`** file, which will contain the main function that runs when the app launches.

In the main function, you can use the **`runApp`** function to launch the app and pass it a **`MaterialApp`** widget, which provides a number of features and styles that conform to the Material Design guidelines.

Within the **`MaterialApp`** widget, you can use the home parameter to specify the default route of your app and pass it a Scaffold widget, which provides a basic structure to your app and allows you to add an app bar, body, and **Floating Action Button** (**FAB**).

To design the layout of the app, you can use a combination of built-in widgets such as Container, Row, and Column, as well as custom widgets that you can create yourself.

For example, you might use a Container widget to hold a logo or image, a Row widget to lay out a series of buttons horizontally, and a Column widget to arrange content in a vertical list.

To customize the appearance of your widgets, you can use various parameters such as color, padding, and margin, as well as the Theme data provided by the **`MaterialApp`** widget.

You can also use the Text widget to add text to your app, and the Icon widget to add icons.

Overall, designing the UI of a Flutter app involves choosing and arranging the appropriate widgets to create the desired layout and appearance.

## Implement app logic

Use Dart programming language to implement the app's logic. This includes handling user interactions, making network requests, reading and writing to a database, and so on.

To implement the app's logic in Flutter using the Dart programming language, you can use a variety of techniques and approaches.

One common approach is to use the **`StatefulWidget`** and State classes to manage the app's state and handle user interactions. The **`StatefulWidget`** class allows you to create a widget that can change its state over time, while the State class provides a way to manage that state and rebuild the widget when the state changes.

For example, you might create a **`StatefulWidget`** for a login form that has fields for a username and password. The State class for the widget could then manage the state of the form, including the values of the fields and whether the form is currently being submitted. When the user submits the form, the State class could make a network request to authenticate the user and update the state of the widget accordingly.

To make network requests, you can use the http package provided by Dart, which allows you to send HTTP requests and receive responses. You can also use packages such as **`dio`** or chopper for more advanced networking needs.

To read and write to a database, you can use a package such as **`sqflite`** to interact with a SQLite database, or you can use a package such as **`firebase_database`** to interact with a Cloud Firestore database.

Overall, implementing the app's logic in Flutter involves using a combination of Dart language features and packages to manage state, make network requests, and interact with databases as needed.

## Test the app

Test the app on an emulator or a physical device to ensure that it works as expected.

There are several ways to test a Flutter app to ensure that it works as expected. Here are a few approaches you can use:

- **Unit testing:** You can use the built-in `flutter_test` package to create unit tests for your app. Unit tests allow you to test specific functions or methods in your app's code to ensure that they are working correctly.
- **Widget testing:** You can use the `flutter_test` package to create widget tests, which allow you to test the behavior and appearance of individual widgets. This can be useful for ensuring that a widget is rendering correctly and responding to user interactions as expected.
- **Integration testing:** You can use the `flutter_driver` package to create integration tests that test the app as a whole by simulating user interactions and checking the app's behavior. This can be useful for testing the app's overall flow and functionality.

To run your tests, you can use the flutter test command from the command line or run the tests from within your development environment.

You can also test the app on an emulator or a physical device, to ensure that it works as expected in a real-world environment. To run the app on an emulator, you can use a tool such as the Android Emulator or the iOS Simulator. To run the app on a physical device, you can connect the device to your computer and run the app from your development environment.

Overall, testing is an important step in the development process and helps ensure that your Flutter app is working correctly and behaving as expected.

## Deploy the app

Build the app for release and deploy it to the App store or Google Play store.

It is possible to use Android-specific code in a Flutter app by using platform channels. Platform channels allow you to communicate between the Flutter code running on the Dart virtual machine and native code running on the platform.

Here are the steps to use Android-specific code in a Flutter app:

- Identify the functionality that you want to access from Android. This could be an API, a library, or some other functionality that is specific to Android.
- Create a platform channel. In your Flutter code, define a platform channel by calling **`MethodChannel`** and specifying a unique channel name.

In Flutter, you can define a platform channel by calling the **`MethodChannel`** constructor and passing in a unique channel name. The channel name is used to identify the channel and must be the same on both the Flutter and Android sides.

Here is an example of how to define a platform channel in Flutter:

```
import 'package:flutter/services.dart';
final MethodChannel _channel = MethodChannel('my_channel');
```

This creates a **`MethodChannel`** object named **`_channel`** that can be used to communicate with Android using the channel named **`'my_channel'`**.

Once the channel is defined, you can use the **`invokeMethod`** method to call a method on the Android side, and the **`setMethodCallHandler`** method to handle method calls from Android.

For example, to call a method on Android, you can do the following:

```
final String result = await _channel.invokeMethod('myMethod',
{'arg1': 123});
```

This will call the **`myMethod`** method on Android and pass it the argument **`'arg1'`** with a value of 123. The result of the method call will be returned as a String.

## Implement the platform channel on Android

In your Android code, implement the platform channel by calling **`MethodChannel`** and using the same channel name.

In order to implement the platform channel on Android, you need to call the **`MethodChannel`** constructor and use the same channel name as on the Flutter side.

Here's an example of how to implement a platform channel in Android:

```
import io.flutter.plugin.common.MethodChannel;
```

```
MethodChannel channel = new MethodChannel(getFlutterView(),
"my_channel");
```

This creates a **`MethodChannel`** object named channel that can be used to communicate with Flutter using the channel named **`'my_channel'`**.

Once the channel is created, you can use the **`setMethodCallHandler`** method to handle method calls from Flutter.

For example, to handle a method call from Flutter, you can do the following:

```
channel.setMethodCallHandler(new
MethodChannel.MethodCallHandler() {
  @Override
  public void onMethodCall(MethodCall call,
  MethodChannel.Result result) {
    if (call.method.equals("myMethod")) {
      int arg1 = call.argument("arg1");
      Do something with the argument…
      result.success("Done!");
    } else {
      result.notImplemented();
    }
  }
});
```

This sets up a method call handler that will be called whenever a method is called on the **`'my_channel'`** channel from Flutter. The handler checks the method name and, if it is **`'myMethod'`**, it retrieves the **`'arg1'`** argument and does something with it. Finally, it returns a result to the Flutter side using the success method of the Result object.

## Invoke the platform channel from Flutter

From your Flutter code, call the **`invokeMethod`** method on the **`MethodChannel`** object and pass in the arguments for the method you want to call on Android.

Once you have defined a **`MethodChannel`** object in your Flutter code and implemented the channel on the Android side, you can use the **`invokeMethod`** method to call a method on the Android side.

The **`invokeMethod`** method takes two arguments: the name of the method to call, and a map of arguments to pass to the method. It returns a Future that completes with the result of the method call.

Here is an example of how to call a method on Android from Flutter:

```
final String result = await _channel.invokeMethod('myMethod',
{'arg1': 123});
```

This calls the **`'myMethod'`** method on Android and passes it the argument **`'arg1'`** with a value of 123. The result of the method call will be returned as a String.

You can pass any number of arguments to the method by adding more key-value pairs to the map. The arguments and return value can be any of the supported types: Boolean, int, long, double, String, and byte arrays.

Keep in mind that the **`invokeMethod`** method is asynchronous and will return a Future, so you need to use **`await`** when calling it or use the then method to handle the result.

## Handle the method call on Android

In your Android code, override the **`onMethodCall`** method and use the method name and arguments passed from Flutter to execute the desired functionality.

To execute functionality in your Android code from Flutter, you can use the **`MethodChannel`** class provided by the Flutter framework.

To start, you will need to define a unique channel name for your app and create a **`MethodChannel`** object in your Android code using that channel name. You can then override the **`onMethodCall`** method of the **`MethodChannel`** object to receive method calls from Flutter.

In your Flutter code, you can use the **`MethodChannel`** class to invoke methods on the Android side. To do this, you will need to pass the channel name that you defined in your Android code, as well as the name of the method you want to call and any arguments that the method requires.

When a method call is received in the **`onMethodCall`** method on the Android side, you can use the method parameter to determine which method was called, and the arguments parameter to access any arguments

that were passed. You can then use this information to execute the desired functionality.

For example, you might define a method in your Android code called **`showToast`** that displays a toast message on the screen. In your Flutter code, you can invoke this method by calling **`MethodChannel.invokeMethod`** and passing the name of the method and any required arguments. When the **`showToast`** method is called on the Android side, it can use the arguments passed from Flutter to determine the message to display in the toast.

Overall, using the **`MethodChannel`** class allows you to invoke methods on the Android side from Flutter and execute the desired functionality.

## Return a result from Android

Once the functionality has been executed on Android, use the result argument of the **`onMethodCall`** method to return a result to the Flutter code.

By using platform channels, you can access any Android-specific functionality from your Flutter app. However, keep in mind that using platform channels can add complexity to your app, so it is important to use them judiciously.

It is possible to include Android-specific code in a Flutter app by using platform-specific code in your Flutter project. This allows you to write code that is specific to the Android platform and reuse it in your Flutter app.

To include Android-specific code in your Flutter project, you can create a new module within your project by selecting **`File`** | **`New`** | **`New Flutter Module`** in your development environment. This will create a new folder within your project that contains a standalone Flutter module.

You can then create Android-specific code within this module by creating files with the .java or .kotlin extension (for Java or Kotlin code, respectively) and placing them in the Android directory of the module.

To access this code from your Flutter app, you can use the **`MethodChannel`** class provided by the Flutter framework, to invoke methods on the Android side. You can then use these methods to access the Android-specific code and execute the desired functionality.

It is also possible to create custom Android widgets using the View class, and use them in your Flutter app by creating a **`PlatformView`** widget in your

Flutter code and passing it the custom View as a parameter.

Overall, using platform-specific code and custom Android widgets, allows you to include Android-specific functionality in your Flutter app and reuse it as needed.

Here is an example of how you can use Android-specific code in your Flutter app:

1. First, you need to include the **`android_alarm_manager`** package in your **`pubspec.yaml`** file:

```
dependencies:
  android_alarm_manager: ^0.4.5+1
```

2. Import the package in your Dart code:

```
import
'package:android_alarm_manager/android_alarm_manager.dart'
;
```

3. You can then use the **`AndroidAlarmManager`** class to schedule an alarm:

```
void scheduleAlarm() {
  final int alarmId = 0;
  final DateTime now = DateTime.now();
  final DateTime scheduledTime = now.add(Duration(seconds:
  5));
  AndroidAlarmManager.oneShotAt(
    scheduledTime,
    alarmId,
    callback,
    wakeup: true,
  );
}
void callback() {
  print('Alarm fired!');
}
```

This will schedule an alarm that will fire in 5 seconds and call the callback function.

**Note: This code will only work on Android devices, as it uses the Android Alarm Manager API. If you want to use platform-specific code in your Flutter app, you can use the Platform class to check the current platform and execute code accordingly.**

```
if (Platform.isAndroid) {
  // Android-specific code
} else if (Platform.isIOS) {
  // iOS-specific code
}
```

## Building Flutter application using iOS specific code

It is possible to include iOS-specific code in a Flutter app by using platform-specific code in your Flutter project. This allows you to write code that is specific to the iOS platform and reuse it in your Flutter app.

To include iOS-specific code in your Flutter project, you can create a new module within your project by selecting **`File`** | **`New`** | **`New Flutter Module`** in your development environment. This will create a new folder within your project that contains a standalone Flutter module.

To include iOS-specific code in your Flutter project, you can follow these steps:

1. Create a new module within your project by selecting **`File`** | **`New`** | **`New Flutter Module`** in your development environment. This will create a new folder within your project that contains a standalone Flutter module.
2. Within the new module, create a new file with the .m or .mm extension (for Objective-C or Objective-C++ code, respectively) and place it in the iOS directory of the module.
3. In your Flutter code, use the **`MethodChannel`** class to create a channel and specify the name of the channel. You can then use this channel to invoke methods on the iOS side from your Flutter code.
4. In your iOS code, override the **`onMethodCall`** method of the **`MethodChannel`** object and use the method and arguments parameters

to determine which method was called and any arguments that were passed. You can then use this information to execute the desired functionality.

5. If you want to use custom iOS widgets in your Flutter app, you can create a **`UIView`** subclass and override its **`drawRect`** method to customize its appearance. You can then create a **`UiKitView`** widget in your Flutter code and pass it the custom UIView as a parameter.

Overall, using platform-specific code and custom iOS widgets allows you to include iOS-specific functionality in your Flutter app and reuse it as needed.

You can then create iOS-specific code within this module by creating files with the .m or .mm extension (for Objective-C or Objective-C++ code, respectively) and placing them in the iOS directory of the module.

To access this code from your Flutter app, you can use the **`MethodChannel`** class provided by the Flutter framework to invoke methods on the iOS side. You can then use these methods to access the iOS-specific code and execute the desired functionality.

It is also possible to create custom iOS widgets using the **`UIView`** class, and use them in your Flutter app by creating a **`UiKitView`** widget in your Flutter code and passing it the custom **`UIView`** as a parameter.

Here is an example of how you can use iOS-specific code in your Flutter app:

1. First, you need to include the **`local_auth`** package in your **`pubspec.yaml`** file:

```
dependencies:
  local_auth: ^0.6.1
```

2. Import the package in your Dart code:

```
import 'package:local_auth/local_auth.dart';
```

3. You can then use the LocalAuthentication class to perform biometric authentication:

```
Future<void> authenticate() async {
  final LocalAuthentication auth = LocalAuthentication();
  bool authenticated = false;
```

```
    try {
     authenticated = await auth.authenticateWithBiometrics(
      localizedReason: 'Please authenticate to access app
      features',
      useErrorDialogs: true,
      stickyAuth: true,
     );
    } on PlatformException catch (e) {
     print(e);
    }
    if (authenticated) {
     print('User is authenticated!');
    } else {
     print('User is not authenticated.');
    }
   }
```

This will show a biometric authentication prompt to the user (for example, Touch ID on iOS).

**Note: This code will only work on iOS devices, as it uses the iOS Local Authentication framework. If you want to use platform-specific code in your Flutter app, you can use the Platform class to check the current platform and execute code accordingly.**

```
if (Platform.isAndroid) {
  // Android-specific code
} else if (Platform.isIOS) {
  // iOS-specific code
}
```

For building a Flutter app, you can use Android-specific code or iOS-specific code to access native functionality that is not available in the Flutter framework.

Using Android-specific code in your Flutter app allows you to access Android-specific APIs and libraries, such as the Android Alarm Manager or the Android Location Manager. This can be useful if you want to use a

feature that is only available on Android, or if you want to optimize your app for Android devices.

Using iOS-specific code in your Flutter app allows you to access iOS-specific APIs and libraries, such as the iOS Local Authentication framework or the iOS Map Kit. This can be useful if you want to use a feature that is only available on iOS, or if you want to optimize your app for iOS devices.

It is important to note that using platform-specific code in your Flutter app means that your app will only work on the respective platform. For example, if you use Android-specific code, your app will only work on Android devices. If you want your app to work on both Android and iOS, you should use cross-platform code that is available in the Flutter framework.

Overall, whether you choose to use Android-specific code, iOS-specific code, or cross-platform code in your Flutter app, it will all depend on your specific needs and requirements.

There are a few key differences between building a Flutter app using Android-specific code and building an app using iOS-specific code:

- **APIs:** Android and iOS have different APIs for accessing native features. For example, Android has the Android Alarm Manager API for scheduling alarms, while iOS has the Local Authentication framework for biometric authentication. When building a Flutter app, you need to use the appropriate API for the platform you are targeting.
- **Code structure:** Android and iOS use different code structures and conventions. For example, Android uses Java or Kotlin, while iOS uses Swift or Objective-C. When writing platform-specific code for a Flutter app, you need to follow the conventions of the target platform.
- **Platform checks:** In a Flutter app, you can use the Platform class to check the current platform and execute code accordingly. This is useful when you want to use platform-specific code, but you also need to make sure your code works on both Android and iOS.

Overall, building a Flutter app using Android-specific code or iOS-specific code is not very different from building a native Android or iOS app. The

main difference is that you are using Flutter as a bridge between the platform-specific code and the rest of your app, which is written in Dart.

## Conclusion

To conclude, using Android-specific code or iOS-specific code in a Flutter app involves finding a package that provides the desired functionality. There are many packages available on the Flutter package repository that offer platform-specific functionality. Add the package to your `pubspec.yaml` file and run flutter pub get to install it.

Package is imported in the Dart code and it is used to access the desired functionality. Platform class is used to ensure that the code is executed on the appropriate platform.

By following these steps, we can easily incorporate platform-specific code into your Flutter app and take advantage of native features and functionality.

## Questions

1. Explain the steps to build Flutter application on Android Specific code.
2. Explain the steps to build Flutter application on iOS Specific code.

# CHAPTER 8

# Introduction to Packages

**“Tech will transform from something we actively use to a more seamless integrated experience that is ‘on’ all the time.”**

*– Daniel Bæk, Co-founder of Nodes*

## Introduction

Packages are the means through which functionalities may be organized and depicted. Packages are used for sharing useful modules and libraries, and for building real life mobile applications.

## Structure

In this chapter, we will cover the following topics:

- Type of packages using Dart packages
- Flutter plugin package

## Objectives

Packages are used both in web and mobile applications, to make the complex code simpler. With the help of package, the modules and libraries may be shared. In this chapter, we will discuss types of packages using dart packages and flutter plugin package.

## Type of packages using Dart Packages

In Dart, set of functionalities may be organized and shared with the help of packages. A Dart package comprises of collection of modules and libraries. The difference between Dart Application and Dart Package is that a Dart Application comprises of main as an entry point, whereas Dart Package does not have main.

The general structure of a dart package is as follows:

**`lib/src/*-`** This represents private dart code file.

Consider an example of **`myapp.dart`** present in **`myapp`** package. The **`myapp.dart`** may be imported as follows:

```
import ‘package:myapp/myapp.dart’
```

A dart file arranged in XYZ folder may be accessed as follows:

```
import 'package:myapp/XYZ/newfile.dart'
```

**`pubspec.yaml`** gives the project specification.

The dart files present in a package comprises of the Dart classes.

# Types of Packages

Package is defined as the dart program that comprises of independent and reusable units. It comprises of classes, functions, sampled data and compiled data. Dart comprises of set of default packages that is loaded when a Dart console gets started.

There are several types of packages that can be created using the Dart programming language, and they are elaborated as follows.

## Library packages

These are packages that contain reusable libraries of Dart code. They can be used by other packages or by applications that depend on them.

Library packages are packages that contain reusable libraries of Dart code. They are designed to be used by other packages or by applications that depend on them. Library packages can contain a wide range of functionality, from simple utility functions to complex data structures and algorithms. They are a useful way to share code between different projects, and can help to reduce the amount of code that needs to be written and maintained.

To use a library package in your own code, you will need to add a dependency on the package in your **`pubspec.yaml`** file and then import the package's libraries into your code using an import statement. Once the package is imported, you can use the functions and classes that it provides, as if they were part of your own codebase.

For example, if you want to use the http package in your code, you would add the following dependency to your **`pubspec.yaml`** file:

```
dependencies:
  http: ^0.12.0
```

Then, in your code, you can import the package and use its functions like this:

```
import 'package:http/http.dart' as http;
// Use the http.get function to make an HTTP GET request
http.get('https://example.com/api/v1/users').then((response) {
  // Do something with the response
});
```

## Application packages

These are packages that contain a standalone application written in Dart. They can be run directly from the command line or from within an application that depends on them.

Application packages are packages that contain standalone applications written in Dart. These applications can be run directly from the command line or from within another application that depends on them. Application packages are used to create command-line applications or standalone desktop or mobile apps. To create an application package in Dart, you will need to create a bin directory at the root of your package and add a file with a main function, that will be the entry point for your application. Then, you can run your application by using the dart command, followed by the path to the file containing the main function. For example, if your application is contained in a file called **`main.dart`**, you can run it with the following command: **`dart bin/main.dart`**.

## Command-line packages

These are packages that contain command-line tools written in Dart. They can be run from the command line and are often used for tasks such as testing, building, or deploying applications.

Command-line packages are packages that contain command-line tools written in Dart. These packages are designed to be run from the command line and are often used for tasks such as testing, building, or deploying

applications. Command-line packages can be created by creating a **`bin`** directory at the root of your package and adding one or more command-line tools to it. Each command-line tool should be contained in its own file and have a main function that serves as the entry point for the tool. To run a command-line tool from the command line, you will need to use the **`dart`** command followed by the path to the file containing the main function for the tool. For example, if your command-line tool is contained in a file called **`tool.dart`**, you can run it with the following command: **`dart bin/tool.dart`**.

## Plugin packages

These are packages that provide integration with platform-specific APIs or functionality. They can be used to extend the capabilities of an application or to provide platform-specific functionality.

Plugin packages are packages that provide integration with platform-specific APIs or functionality. They can be used to extend the capabilities of an application or to provide platform-specific functionality. Plugin packages are used to create plugins for Flutter, which is a mobile application development framework for building cross-platform apps for Android and iOS using the Dart programming language. To create a plugin package in Dart, you will need to create a lib directory at the root of your package and add a file with a top-level import and export statement. This file will be the entry point for your plugin. Then, you will need to create one or more classes that define the plugin's functionality and annotate them with the **`@plugin`** annotation. The plugin's API should be defined in a separate file in the lib directory, and any platform-specific code should be placed in separate directories under the lib directory. For example, you might have a **`lib/ios`** directory for iOS-specific code and a **`lib/android`** directory for Android-specific code.

## Experimental packages

These are packages that are still in development or are not yet ready for production use. They may contain experimental or unfinished code and should be used with caution.

Experimental packages are packages that are still in development or are not yet ready for production use. They may contain experimental or unfinished

code and should be used with caution. Experimental packages can be used to test new ideas or features, or to get feedback from the community before releasing a package as a stable version. It is important to note that experimental packages may not be fully tested or documented, and they may contain bugs or unfinished functionality. As such, they should not be used in production environments or relied upon for critical tasks. Instead, they should be used for experimentation and testing purposes only.

In addition to these types of packages, there are also several package categories that are used to classify packages based on their intended use or functionality. Some common categories include the following:

## Web

Web packages are packages that are used for web development, such as packages for working with HTTP, HTML, or CSS. Web packages can be used to build web applications, web servers, or other types of web-based software using the Dart programming language. Some examples of web packages in Dart might include packages for working with HTTP requests and responses, parsing HTML or XML, generating and manipulating CSS stylesheets, or interacting with web browsers. To use a web package in a Dart application, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement.

## Mobile

Mobile packages are packages that are used for mobile development, such as packages for working with Android or iOS APIs. Mobile packages can be used to build mobile applications for Android or iOS, using the Dart programming language and the Flutter framework. Some examples of mobile packages in Dart might include packages for working with device sensors, accessing device storage, or interacting with device hardware such as the camera or Bluetooth. To use a mobile package in a Dart application, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement.

## Desktop

Packages that are used for desktop development, such as packages for working with Electron or Flutter for desktop.

Desktop packages are packages that are used for desktop development, such as packages for working with Electron or Flutter for desktop. Desktop packages can be used to build desktop applications for Windows, macOS, or Linux using the Dart programming language and frameworks such as Electron or Flutter. Some examples of desktop packages in Dart might include packages for working with the filesystem, interacting with the operating system, or creating native-like user interfaces. To use a desktop package in a Dart application, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement.

## Testing

Testing packages are packages that are used for testing, such as packages for running unit tests or integration tests. Testing packages can be used to create and run tests for Dart applications to ensure that they are working correctly and to identify and fix any problems that may arise. Some examples of testing packages in Dart might include packages for running unit tests, integration tests, or performance tests. To use a testing package in a Dart application, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement. Then, you can use the package's APIs to define and run your tests.

## Utilities

Utility packages are packages that provide utility functions or libraries, such as packages for working with dates, strings, or collections. Utility packages can be used to simplify common tasks or to provide additional functionality that is not included in the Dart standard library. Some examples of utility packages in Dart might include packages for parsing and formatting dates, manipulating strings, or working with lists or maps. To use a utility package in a Dart application, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement. Then, you can use the package's APIs to perform the desired tasks.

Packages required in building web or mobile application may be categorized into different types on the basis of their functionality, and they

are as follows:

## Dart Package

This package may be utilized for web and mobile app development. For example, there is a package called **`english_words`** that comprises of 5000 words and consists of functions such as syllables, nouns and so on.

In the Dart programming language, a package is a collection of Dart source code files and other resources that are organized and published together, typically as a library or application. Packages in Dart are used to share and reuse code and to modularize larger applications.

To create a package in Dart, you will need to create a directory for your package and add a **`pubspec.yaml`** file to it. The **`pubspec.yaml`** file is used to define the package's metadata, such as its name, version, and dependencies. You can then add your source code files and other resources to the package directory.

Packages in Dart can be published and shared with the Dart community via the **`pub.dev`** package repository, which is the official package repository for the Dart ecosystem. To publish a package, you will need to create a package account on **`pub.dev`** and follow the steps to upload your package.

To use a package in a Dart application, you will need to add it as a dependency in your **`pubspec.yaml`** file and run the **`pub get`** command to download the package. Then, you can import the package into your code using an import statement and use its APIs to access its functionality.

Packages in Dart can be of different types, such as library packages, application packages, command-line packages, plugin packages, and so on. Each type of package serves a different purpose and may have its own specific requirements or conventions.

Overall, packages in Dart are a powerful and essential tool for sharing and reusing code, and they play a central role in the Dart ecosystem.

## Generic dart code

It is useful in making mobile applications and makes use of Flutter framework. Flutter is a mobile application development framework that uses the Dart programming language and allows you to build cross-platform

apps for Android and iOS. Flutter includes a variety of built-in components, such as widgets and animation libraries, as well as a fast and powerful rendering engine.

Fluro is a popular package for Flutter that provides a custom router. It allows you to define custom routes for your Flutter app and navigate between them easily. Fluro makes use of the Flutter framework and can be used to build more complex and feature-rich apps.

In general, generic Dart code can be used to build mobile applications using the Flutter framework. However, Dart can also be used to build other types of applications, such as web or desktop applications, or to create standalone libraries or command-line tools. The possibilities are vast, and the Dart ecosystem includes a wide range of packages and tools to help you build whatever you can imagine.

## Flutter Plugin

It represents a dart code that is based on the platform used and the flutter framework. For example, camera plugin may be used for using device camera.

A Flutter plugin is a package that provides integration with platform-specific APIs or functionality. It is written in Dart and is designed to be used with the Flutter framework, which is a mobile application development framework for building cross-platform apps for Android and iOS using the Dart programming language.

A Flutter plugin is typically structured as a library package with a lib directory at the root of the package. The plugin's API is defined in a Dart source file in the lib directory and annotated with the @plugin annotation. The plugin may also include platform-specific code in separate directories under the lib directory, such as a **`lib/ios`** directory for iOS-specific code or a **`lib/android`** directory for Android-specific code.

A Flutter plugin is typically used to provide access to platform-specific functionality or APIs that are not available in the Flutter framework. For example, the camera plugin you mentioned is a Flutter plugin that provides access to the device's camera. Other examples of Flutter plugins might include plugins for working with device sensors, accessing device storage, or interacting with device hardware such as the Bluetooth or NFC.

To use a Flutter plugin in a Flutter app, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement. Then, you can use the plugin's API to access its functionality.

For building a web or mobile application, a file may be imported from a package. The file from a given package may be imported and utilized in building a mobile or web application. The **`english_words.dart`** file may be imported from a **`english_words`** package. This is shown as follows:

```
import 'package:english_words/english_words.dart';
```

From **`english_words.dart`**, we can retrieve first 100 noun and print them as follows:

```
nouns.take(100).forEach(print);
```

# Develop a Flutter Plugin Package

Development of a Flutter Plugin is the same as developing a Dart application or a Dart Package. The only difference is that the plugin makes use System API (Android or iOS); which is used for achieving required platform dependent property.

To develop a Flutter plugin in Dart, you will need to follow these steps:

## Create a new package

Create a new directory for your plugin and add a **`pubspec.yaml`** file to it. The **`pubspec.yaml`** file is used to define the package's metadata, such as its name, version, and dependencies.

To create a new package in Dart, you will need to create a new directory for your package and add a **`pubspec.yaml`** file to it. The **`pubspec.yaml`** file is used to define the package's metadata, such as its name, version, and dependencies.

Here is an example of a **`pubspec.yaml`** file for a package called **`my_package`**:

```
name: my_package
version: 1.0.0
description: A package for doing something cool
dependencies:
```

```
  some_dependency: ^1.0.0
dev_dependencies:
  some_dev_dependency: ^1.0.0
```

The name field specifies the name of the package. The version field specifies the package's version number. The description field provides a brief description of the package. The dependencies field lists any packages that the package depends on, and the **`dev_dependencies`** field lists any packages that are only needed for development purposes.

Once you have created your **`pubspec.yaml`** file, you can add your source code files and other resources to the package directory. Then, you can use the pub tool to publish your package to the **`pub.dev`** package repository or to add it as a dependency in other projects.

## Define the plugin's API

Create a Dart source file in the lib directory and define the plugin's API. The plugin's API should consist of classes and functions that define the plugin's functionality. Annotate the main class with the **`@plugin`** annotation to indicate that it is a Flutter plugin.

To define the plugin's API in a Flutter plugin, you will need to create a Dart source file in the **`lib`** directory and define the plugin's API as a set of classes and functions. The plugin's API should consist of the functionality that you want to expose to Flutter apps that use the plugin.

For example, let us say you are creating a plugin that provides access to the device's camera. Your plugin's API might consist of a Camera class with methods for taking pictures and recording video, as well as a **`CameraPreview`** widget for displaying the camera's preview in a Flutter app.

To indicate that your plugin is a Flutter plugin, you will need to annotate the main class with the **`@plugin`** annotation. For example:

```
import 'package:flutter/widgets.dart';
import 'package:flutter/services.dart';
import
'package:flutter_camera_plugin/flutter_camera_plugin.dart';
class FlutterCameraPlugin {
  static const MethodChannel _channel =
    const MethodChannel('flutter_camera_plugin');
```

```
  static Future<String> get platformVersion async {
  final String version = await
  _channel.invokeMethod('getPlatformVersion');
  return version;
  }
}
class Camera {
  final MethodChannel _channel;
  Camera(this._channel);
  Future<void> takePicture() async {
  await _channel.invokeMethod('takePicture');
  }
  Future<void> startRecording
```

## Add platform-specific code

If your plugin requires platform-specific code, you can add it to separate directories under the **`lib`** directory. For example, you might create a **`lib/ios`** directory for iOS-specific code and a **`lib/android`** directory for Android-specific code.

Flutter plugin requires platform-specific code, you can add it to separate directories under the lib directory. For example, you might create a **`lib/ios`** directory for iOS-specific code and a **`lib/android`** directory for Android-specific code.

For example, let us say you are creating a plugin that provides access to the device's camera. You might have platform-specific code for each platform that handles the details of accessing the camera and returning the captured image or video to your plugin's API.

To add platform-specific code to your plugin, you will need to create a directory for each platform under the lib directory and add your platform-specific code to it. For example:

```
lib
├── android
│   └── android.dart
└── ios
  └── ios.dart
```

Your platform-specific code should be written in the appropriate language for the platform (for example, Java for Android or Swift for iOS). You can then use the **`MethodChannel`** class in your plugin's API to communicate with your platform-specific code and invoke its methods.

Overall, adding platform-specific code to a Flutter plugin allows you to access platform-specific functionality or APIs that are not available.

## Test the plugin

Test the plugin by creating a new Flutter app and adding the plugin as a dependency in the app's **`pubspec.yaml`** file. Then, import the plugin into your code and use its API to test its functionality.

Testing is an important step in the process of developing a Flutter plugin. It allows you to ensure that your plugin is working correctly and to identify and fix any problems that may arise.

To test a Flutter plugin, you will need to create a new Flutter app and add the plugin as a dependency in the app's **`pubspec.yaml`** file. Then, import the plugin into your code and use its API to test its functionality.

Here is an example of how you might test a Flutter plugin in a Flutter app:

```
import 'package:flutter/material.dart';
import
'package:flutter_camera_plugin/flutter_camera_plugin.dart';
void main() {
 runApp(MyApp());
}
class MyApp extends StatefulWidget {
 @override
 _MyAppState createState() => _MyAppState();
}
class _MyAppState extends State<MyApp> {
 String _platformVersion = 'Unknown';
 @override
 void initState() {
  super.initState();
  initPlatformState();
 }
```

```
// Initialize the plugin's platform state
Future<void> initPlatformState() async {
 String platformVersion;
 try {
  platformVersion = await
  FlutterCameraPlugin.platformVersion;
 } on PlatformException {
  platformVersion = 'Failed to get platform version.';
 }

// If the widget was removed from
```

## Publish the plugin

When your plugin is ready for publication, you can publish it to the **`pub.dev`** package repository by creating a package account and following the steps to upload your package.

In general, developing a Flutter plugin requires a good understanding of the Flutter framework and the Dart programming language, as well as familiarity with platform-specific APIs and functionality. It may also require some knowledge of native development for Android and iOS. However, the Flutter documentation and community are a great resource for learning more and getting started with plugin development.

# Dart Package Manager

Dart comprises of inbuilt Package Manager referred to as pub package manager. It is used for installing, organizing and managing tools, libraries as well as dependencies. Dart application comprises of **`pubspec.yaml`** file as well as third-party libraries and the application dependencies. Most of the Dart IDEs provide support for building, updation and publishing the packages.

The Dart Package Manager, also known as pub, is a tool for managing packages in the Dart programming language. It is used to download, install, and manage packages from the **`pub.dev`** package repository, which is the official package repository for the Dart ecosystem.

To use the Dart Package Manager, you will need to have the Dart SDK installed on your machine. You can then use the pub command-line tool to perform various package management tasks, such as adding a package as a dependency in your project, downloading and installing packages, or updating an existing package.

Here are some common pub commands:

- **`pub get`**: Downloads and installs the dependencies of a package.
- **`pub upgrade`**: Upgrades the dependencies of a package to the latest compatible version.
- **`pub publish`**: Publishes a package to the **`pub.dev`** package repository.
- **`pub run`**: Runs a command from a package's bin directory.
- **`pub global activate`**: Activates a global package and adds it to the PATH.

The Dart Package Manager is an essential tool for developers who use the Dart programming language. It helps to manage dependencies, share and reuse code, and build more scalable and maintainable applications.

## Installing a Package

The steps for installing a Package are as follows:

1. In **`pubspec.yaml`** file, add the name of package in the dependencies section of a project. To get the information about the package installed, the following command can be used in the application directory:

   ```
   pub get
   ```

   This command would download all the packages which are present in the packages folder in a given application directory.

2. The **`dart create`** command is used for the creation of initial directory and the structure of the package as follows:

   ```
   $ dart create -t package <Name_of_Package>
   ```

For creation of individual libraries referred to as mini libraries, library packages are used that can be extended, maintained and tested easily. The

main library file is created in a lib folder as **`lib/name_of_package.dart`**. The lib directory also comprises of non src, importable and libraries.

When a library file is imported from another package, package directive is used to provide the URL of the file as follows:

```
import 'package:utilities/utilities.dart';
```

When library file has to be imported from the package, relative path is used when both the files are present inside **`lib`** or both the files are present outside **`lib`**. We make use of '**`Use package:`**' when the file to be imported is in **`lib`** and importer is placed outside.

It is simple to test a library package which is well designed. Tests should be written with the help of test package, where test code is placed in a given test directory present at the top in the package.

While publishing of a package in **`pub.dev`** site, files required are **`CHANGELOG.md`** and **`README.md`**. With the help of dart doc tool, API docs may be generated. **`dart.doc`** performs parsing of the source and seeks for /// (documentation comments).

The dart doc tool should be run before package publishing, in order to be sure that all the documents are successfully generated and look as exactly expected. The Versions tab is checked after package publishing, which makes sure that all the documents are successfully generated. In the Versions tab, click on failed to check dart document output; in case, documents are not generated.

# Flutter Plugin Package

Packages involve development of modular code that is easily shareable. A minimal package comprises of: a **`pubspec.yaml`** file, metafile, lib directory and dart file. A metadata file comprises of a package name, author, version and so on. A library directory comprises of public code for the package.

# Steps for developing Dart packages

The steps for developing Dart packages are as follows:

## Creating a package

For creating a Dart package, we use **`--template=package`** flag along with flutter create as follows:

```
$ flutter create --template=package welcome
```

This leads to creation of package in welcome/folder having following contents:

`lib/welcome.dart:`

## Implementing the package

For pure Dart `packages`, simply add the functionality inside the **`main lib/<package name>.dart`** file, or in several files in the lib directory.

To test the package, add unit tests in a test directory.

For additional details on how to organize the package contents, see the **Dart library package** documentation.

For developing a package that is based on platform specific API, we need to develop a package plugin. A plugin package is similar to a dart package, with the only difference that a plugin package also comprises of platform specific implementation requirements.

By default, Swift is used in plugin project for iOS code and Kotlin is used in plugin project for Android code.

In dart code, the API of a plugin package is defined. Inside welcome/folder, we may locate a file named **`lib/welcome.dart`**.

We can edit Android platform code using Android Studio and we can edit iOS code using Xcode.

Finally, API is connected with the platform specific implementations.

In a package, the following documentation must be added:

- **README.md file**: This helps in giving information about a package. A README.md file is a markdown file that provides information about a package. It is typically located at the root of the package directory and is displayed on the package's page on the **`pub.dev`** package repository. The README.md file is an important resource for developers who are using your package. It should provide a clear and concise description of what your package does, as well as any

instructions or guidelines for using it. Here are some common elements that you might include in a README.md file for a package:

  - **Description**: A brief overview of what the package does and why it is useful.
  - **Installation**: Instructions for adding the package as a dependency in a Flutter app.
  - **Usage**: Examples of how to use the package's API in code.
  - **API Reference**: Documentation for the package's API, including a list of classes, functions, and other symbols.
  - **Contributions**: Guidelines for contributing to the package, such as how to report bugs or request features.
  - **License**: Information about the license under which the package is distributed.
  - In general, a well-written README.md file is an important part of any package, as it helps to provide context and instructions for using the package. It can also help to improve the discoverability and usability of your package for other developers.

- **CHANGELOG.md file**: It provides information about modifications in each version
- **LICENSE file**: It gives information about terms and conditions for a package.

## Publishing packages

When a package is published, API documentation is created and is published under **`dartdocs.org`**.

When a package is implemented, we can be publish it on **`pub.dev`**, so that it can be used by other developers. We need to review the **`pubspec.yaml`**, **`CHANGELOG.md`** and **`README.md`** before publishing is performed, in order to make sure that content is correct and complete.

Prior to publishing, make sure to review the **`pubspec.yaml`**, **`README.md`**, and **`CHANGELOG.md`** files to make sure their content is complete and correct. In order to improve the usability and quality of a package, the following items should be added:

- Examples depicting diverse usage of code.
- Videos, screenshots and animated gifs.
- A corresponding link to the code repository.

A dry-run command is run to check whether it passes the analysis:

```
content_copy
$ flutter pub publish --dry-run
```

Finally, the publish command is run as follows:

```
content_copy
$ flutter pub publish
```

Dart package represent modules and libraries that are shareable. Dart package is similar to Dart Application except it does not have main.

Consider a demo package, **`welcome`** package, as depicted:

- **`lib/src/*`**: It represent private dart files code.
- **`lib/welcome.dart`**: It represents the main dart file code. It is imported as **`import 'package:welcome/welcome.dart'`**

The private code file may be exported into the main code file as follows:

```
export src/my_private_code.dart
```

**`lib/*`**: It stores Dart code files which are arranged in a custom folder structure. This may be accessed as,

```
import 'package:welcome/custom_folder/custom_file.dart'
```

A Flutter plugin package is a package that provides integration with platform-specific APIs or functionality in Flutter, a mobile application development framework for building cross-platform apps for Android and iOS using the Dart programming language.

A Flutter plugin is typically structured as a library package with a lib directory at the root of the package. The plugin's API is defined in a Dart source file in the lib directory and annotated with the **`@plugin`** annotation. The plugin may also include platform-specific code in separate directories under the lib directory, such as a **`lib/ios`** directory for iOS-specific code or a **`lib/android`** directory for Android-specific code.

To use a Flutter plugin in a Flutter app, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an

import statement. Then, you can use the plugin's API to access its functionality.

Examples of Flutter plugin packages might include plugins for working with device sensors, accessing device storage, or interacting with device hardware such as the Bluetooth or NFC. Flutter plugin packages can be published to the **`pub.dev`** package repository and shared with other developers.

# Conclusion

Packages are an important part of the Flutter ecosystem, as they allow developers to share and reuse code and functionality between Flutter apps. There are many different types of packages available for Flutter, including application packages, command-line packages, plugin packages, experimental packages, and packages for web, mobile, desktop, testing, and utilities.

To use a package in a Flutter app, you will need to add it as a dependency in your **`pubspec.yaml`** file and import it into your code using an import statement. Then, you can use the package's API to access its functionality.

Flutter plugin packages are a special type of package that provide integration with platform-specific APIs or functionality. They can be used to extend the capabilities of a Flutter app or to provide platform-specific functionality.

Overall, packages are a valuable resource for developers working with Flutter, as they allow you to leverage the work of others and build more feature-rich and scalable apps.

In this chapter, we have discussed about the different types of packages using dart package and flutter plugin package.

# Questions

1. Explain the different types of packages using dart package.
2. How can Flutter plugin be constructed?

# CHAPTER 9

# Building Layouts

**“I think the biggest change, and the one we’re already starting to see take shape, is that globally the majority of internet usage will be done via a mobile device and for most people the mobile web will be their primary – if not their only – way of experiencing the internet.”**

*– Peter Rojas, Co-founder of Engadget and Gizmodo*

## Introduction

In Flutter, a layout is a widget that arranges other widgets according to specific rules. The layout mechanism in Flutter is based on the box model, which aligns children within a box along both the main and cross axis. The main axis is the direction in which the widgets are laid out, while the cross axis is perpendicular to the main axis. There are several built-in layout widgets in Flutter that you can use to arrange your widgets, such as Container, Row, Column, Expanded, ListView, Stack, and Card. You can also create your own custom layouts by combining and arranging these widgets. The layout widgets in Flutter use a combination of size, position, and other formatting properties to control the position and appearance of their children. You can use layout widgets to create a wide variety of user interface designs, from simple to complex, depending on your needs.

## Structure

In this chapter, we will cover the following topics:

- Introduction to Layout
- Type of Layout Widgets
- Layout Application

# Objectives

In Flutter, a layout is a widget that arranges other widgets according to specific rules. Layout widgets are used to structure the visual design of a user interface. They define the way in which the user interface is arranged, including the position and size of children widgets within the parent widget. Some common layout widgets in Flutter include Container, Row, Column, Expanded, ListView, Stack, and Card. These widgets can be used individually or combined to create custom layouts. For example, you might use a Container widget to wrap a Text widget in a box with a specific background color, or you might use a Row widget to arrange several Text widgets in a horizontal line. By using layout widgets, you can create beautiful and responsive user interfaces for your Flutter applications.

In this chapter, we will discuss an introduction to layouts, types of layout widgets and layout application.

# Introduction to Layout

In Flutter, layouts are responsible for arranging the visual components on the screen. They define the position and size of widgets within the parent widget, and play a crucial role in creating a visually appealing and easy-to-use user interface.

There are several built-in layout widgets in Flutter, such as Container, Row, and Column, which can be used to arrange children widgets in a specific way. For example, the Container widget can be used to wrap a child widget in a box with a specific background color or padding. The Row and Column widgets can be used to arrange children widgets in a horizontal or vertical line, respectively.

In addition to these built-in layout widgets, Flutter also provides several specialized layout widgets, such as ListView, which is used to create scrolling lists, and Stack, which is used to overlap several children widgets.

By using these layout widgets, developers can create complex and visually appealing user interfaces for their Flutter applications.

In Flutter, layouts are responsible for positioning and organizing the visual elements of an app's user interface. They define the way in which user

interface elements are arranged on the screen, including the position, size, and shape of each element.

Flutter provides a variety of built-in layout widgets that you can use to build your app's user interface. Some common layout widgets include the following.

## Container

The container is a box that can contain other widgets. It can be decorated with a background color or image, and can have padding and margins. In Flutter, the Container widget is a box that can contain other widgets. It is often used as a way to add padding, margins, and background color to other widgets.

Here is an example of using a Container widget to add some padding and a blue background color to a Text widget:

```
Container(
  padding: EdgeInsets.all(16.0),
  color: Colors.blue,
  child: Text('Hello World'),
)
```

This will create a blue box with a Text widget inside it, and the text will be indented by 16 pixels on all sides.

You can also use the Container widget to set the width and height of its child widget, add a border, and set other visual properties such as alignment and shape. For example:

```
Container(
  width: 200.0,
  height: 100.0,
  alignment: Alignment.center,
  decoration: BoxDecoration(
  border: Border.all(width: 1.0, color: Colors.black),
  borderRadius: BorderRadius.circular(10.0),
  ),
  child: Text('Hello World'),
)
```

This will create a 200x100 pixel container with a black border and rounded corners, and the text will be centered within the container.

The Container widget is a very flexible layout widget that can be used in a variety of situations to add visual structure to your app's user interface.

## Row and Column

Row and Column arrange their children in a horizontal or vertical line, respectively. You can use these widgets to create simple grids or more complex layouts.

The Row and Column widgets are used to arrange their children widgets in a horizontal or vertical line, respectively. They are often used to create simple grids or more complex layouts.

Here is an example of using a Row widget to create a simple grid of three Text widgets:

```
Row(
  children: [
  Text('Column 1'),
  Text('Column 2'),
  Text('Column 3'),
  ],
)
```

This will create a horizontal row of three Text widgets, with each widget appearing next to the other in a single line.

You can also use the Column widget to create a vertical layout:

```
Column(
  children: [
  Text('Row 1'),
  Text('Row 2'),
  Text('Row 3'),
  ],
)
```

This will create a vertical column of three Text widgets, with each widget appearing below the other in a single column.

You can use the Row and Column widgets to create simple grids, or you can combine them with other layout widgets such as Expanded and Container to create more complex layouts. They are an essential part of building user interfaces in Flutter.

## Expanded

This takes up remaining free space. This widget is often used in combination with Row or Column to give a child widget the remaining space within the parent widget.

The Expanded widget is a layout widget that takes up remaining free space. It is often used in combination with Row or Column to give a child widget the remaining space within the parent widget.

Here is an example of using Expanded to give one of the children in a Row widget all of the remaining free space:

```
Row(
  children: [
  Text('Column 1'),
  Expanded(
   child: Text('Column 2'),
  ),
  Text('Column 3'),
  ],
)
```

In this example, the Text widget in the middle column will take up all of the remaining free space within the Row widget, while the other two Text widgets will only take up the space that they need.

You can also use the Expanded widget to give a child widget a specific proportion of the remaining free space. For example:

```
Row(
  children: [
  Text('Column 1'),
  Expanded(
   flex: 2,
   child: Text('Column 2'),
  ),
```

```
  Expanded(
   flex: 1,
   child: Text('Column 3'),
  ),
  ],
 )
```

In this example, the Text widget in the middle column will take up twice as much space, as the Text widget in the third column.

The Expanded widget is a useful tool for creating flexible layouts that can adjust to different screen sizes and orientations. It can be used to give one or more children in a Row or Column widget, the remaining free space within the parent widget.

## ListView

This is a scrolling list of widgets. You can use this widget to display a large number of items in a vertical or horizontal list.

**`ListView`** is a scrolling widget that displays a scrolling list of widgets. It is one of the most commonly used widgets in Flutter, and it is an essential part of many app's user interfaces. **`ListView`** is a very flexible widget that can be used in a variety of situations, and it is a good choice when you need to display a large number of items in a scrollable list.

One of the key features of **`ListView`** is that it allows you to specify a custom item builder, which is a callback that is called for each item in the list. This allows you to customize the appearance of the items in the list, and it gives you a lot of control over how the list looks and behaves.

In addition to the item builder, you can also specify a custom scroll controller for the **`ListView`**, which allows you to control how the list behaves when the user scrolls through it. This can be useful if you want to implement custom scrolling behaviors, or if you want to synchronize the scrolling of multiple lists.

**`ListView`** also has several other features and options that you can use to customize its behavior, such as the ability to specify a separator widget to be displayed between items in the list, or the ability to specify a "physics" object to control how the list responds to user input.

Overall, **`ListView`** is a very powerful and flexible widget, that is an essential part of many Flutter app's user interfaces. It is a good choice when you need to display a large number of items in a scrollable list, and it is a useful building block for many different types of user interfaces.

## Stack

Stack overlaps several children widgets. This widget allows you to position widgets on top of each other, with the most recently added widget appearing on top.

The Stack widget is a basic layout element in which children with the same parent are positioned on top of one another with the most recently added child appearing on top. You can use the Positioned widget to position children within the Stack and specify how the child should be positioned within the Stack.

Here is an example of using the Stack widget:

```
Stack(
  children: [
  Positioned(
   left: 10,
   top: 10,
   child: Text('I am over here'),
  ),
  Positioned(
   left: 100,
   top: 50,
   child: Text('I am over here too'),
  ),
  ],
 )
```

This will create a Stack with two children, each positioned at a specific location within the Stack. The first child will be positioned 10 pixels from the left and 10 pixels from the top, while the second child will be positioned 100 pixels from the left and 50 pixels from the top. The Stack widget does not provide any layout or sizing behavior, so you will need to use other widgets or manually specify the dimensions of the Stack and its children.

# Card

A material design card widget is used to display information in a visually appealing and easy-to-read format.

The Card widget is a material design card which can be used to display information in a visually appealing and easy-to-read format. It typically contains a title, subtitle, and an image.

Here is an example of using the Card widget:

```
Card(
  child: Column(
  children: [
   ListTile(
   title: Text('Card title'),
   subtitle: Text('Card subtitle'),
   ),
   Image.asset('assets/image.jpg'),
  ],
  ),
)
```

This will create a Card with a title and subtitle displayed in a **`ListTile`** widget, as well as an image. The Card widget also has several customization options, such as the ability to change the background color, add a border, and specify padding.

You can also nest other widgets within the Card to create more complex layouts. For example, you could add a Button widget as a child of the Card to allow users to take an action.

In addition to these built-in layout widgets, you can also create your own custom layouts by combining and arranging these widgets. For example, you might use a Row widget to arrange several Container widgets in a horizontal line, or you might use a Stack widget to overlap a Text widget and an image.

By using layout widgets, you can create beautiful and responsive user interfaces that look great on a variety of devices.

Widgets are the part of Flutter layout. In Flutter, everything is built from widgets. Widgets comprises of icons, images, text, columns, rows and grids

in a Flutter app.

# Creating a text, image or icon

In Flutter, the following steps are involved in creating a text, image or icon on the screen.

## Selection of a layout widget

The layout widget is chosen from the selection, on the basis of how we can align a given visible widget.

When selecting a layout widget in Flutter, you should consider the following factors:

- **The size and position of the children**: Do you need to specify the size of the children, or should the layout widget handle it for you? Do you need to position the children in a specific way, or can the layout widget handle it automatically?
- **The structure of the layout**: Do you need a simple, linear layout or a more complex grid-based layout? Do you need to nest multiple layout widgets to achieve the desired structure?
- **The responsiveness of the layout**: Will the layout need to adjust based on the size of the screen or the orientation of the device?
- **The scrolling behavior:** Will the layout require scrolling, and if so, do you need to support scrolling in a specific direction?

By considering these factors, you can choose the layout widget that best fits the needs of your app. Some common layout widgets in Flutter are as follows:

- **Container**: A box that can contain other widgets.
- **Row and Column**: Lay out children in a horizontal or vertical line.
- **Expanded**: Takes up remaining free space.
- **Flow**: A flow-based layout.
- **Wrap**: A flow-based layout that wraps lines of widgets.
- **Stack**: Overlap several children widgets.
- **Card**: A material design card.

You can also create custom layout widgets by extending the **`SingleChildLayoutDelegate`** class and implementing the **`performLayout`** method.

## Creation of a Visible Widget

We can create the Text widget as follows:

```
content_copy
Text(‘Hello World’),
```

We can also create the image widget as follows:

```
content_copy
image.asset(‘images/sunset.jpg’,fit:BoxFit.cover,),
```

We can create the Icon widget as follows:

```
content_copy
Icon(Icon.star,color: Colors.red[500],),
```

To create a visible widget in Flutter, you can use one of the built-in widgets provided by the framework, such as Text, Image, or Container.

For example, to create a simple Text widget, you can use the following code:

```
Text('Hello World')
```

To customize the appearance of the Text widget, you can pass additional arguments to the constructor. For example, to change the font size and color:

```
Text(
  'Hello World',
  style: TextStyle(
  fontSize: 24,
  color: Colors.red,
  ),
)
```

To create an Image widget, you can use the **`Image.asset`** method to display an image from your assets folder:

```
Image.asset('assets/image.jpg')
```

You can also use the **`NetworkImage`** class to display an image from a URL:

```
Image.network('https://example.com/image.jpg')
```

To create a Container widget, you can use the following code:

```
Container(
  width: 100,
  height: 100,
  color: Colors.red,
  child: Text('I am a container'),
)
```

This will create a red container that is 100 pixels wide and 100 pixels tall, with the text "*I am a container*" displayed inside. The Container widget can be used to hold and style other widgets, such as images or buttons.

All the layout widgets include the following:

A child property comprising of a single child. For example, Container, Center and children property comprising of list of widget. For example, Row, Column, Stack, ListView, or Stack.

## Addition of layout widget to page

A Flutter app is itself a widget, and most widgets includes the **`build()`** method. A widget is displayed by instantiation and returning of a widget to the app's **`build()`** method.

## Material apps

For a Material app, `Scaffold` widget may be used. It includes background color, default banner and API for addition of snack bars, bottom sheets and drawers. Then you can add the Center widget in the body property of home page.

```
lib/main.dart(ProjectApp)
content_copy
class ProjectApp extends StatelessWidget{
  const ProjectApp({super.key});
Widget build(BuildContext context){
  return MaterialApp{
  title: 'Flutter layout demo',
   home: Scaffold(
```

```
   appBar: AppBar(
    title: const Text('My first Flutter app'),
   ),
   body: const Center(
    child: Text('Welcome'),
   ),
   ),
  );
 }}
```

Standard Widgets include the following:

- **Container**: It includes borders, background color, padding, borders, margins and so on.
- **ListView:** Lays widgets out as a scrollable list.
- **Stack:** Widgets are overlaped top of another.
- **GridView:** Widgets are layed on a scrollable grid.

Material Widgets include the following:

- **Card:** It organizes information in a box, having rounded corners and a drop shadow.
- **ListTile:** It organizes, at most, 3 lines of text, along with icons in a row.

### Stack

Stack is used for arranging a widget on another widget. Use `Stack` to arrange widgets on top of a base widget—often an image.

## Type of Layout Widgets

There are several types of layout widgets in Flutter that you can use to arrange and position the children of a widget, and they are as follows:

- **Linear layout**: Arranges children in a single line, either horizontally or vertically. Examples include Row, Column, and **`ListView`**.
- **Flow layout**: Arranges children in a flow-based layout, wrapping lines of children as needed. Examples include Flow and Wrap.

- **Stack layout**: Overlaps children widgets on top of each other, with the most recently added child appearing on top. The Stack widget can be used to position children manually using the Positioned widget.
- **Expand layout**: Takes up remaining free space. The Expanded widget can be used to divide the available space among its children.
- **Card layout**: A material design card layout. The Card widget can be used to display information in a visually appealing and easy-to-read format.
- **Custom layout**: You can create custom layout widgets by extending the `SingleChildLayoutDelegate` class and implementing the `performLayout` method.

By selecting the appropriate layout widget for your needs, you can create complex and responsive layouts for your Flutter app.

Layout widgets may be categorized into two categories on the basis of number of children it possesses:

- Widget having a single child
- Widget having multiple child

## Single Child Widgets

In this layout, each widget comprises of one child widget and every widget has unique functionality.

A single child widget may be used for creation of high quality widget with functionalities such as label, button and so on.

The code to create a simple button using Container widget is as follows:

```
class OkButton extends StatelessWidget {
  OkButton({Key key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
   return Container(
    decoration: const BoxDecoration(
     border: Border(
      top: BorderSide(width: 1.0, color: Color(0xFFFFFFFFF)),
```

```
      left: BorderSide(width: 1.0, color:
      Color(0xFFFFFFFFF)),
      right: BorderSide(width: 1.0, color:
      Color(0xFFFFF00000)),
      bottom: BorderSide(width: 1.0, color:
      Color(0xFFFFF00000)),
     ),
    ),
    child: Container(
     padding: const
     EdgeInsets.symmetric(horizontal: 20.0, vertical: 2.0),
     decoration: const BoxDecoration(
      border: Border(
       top: BorderSide(width: 1.0, color:
       Color(0xFFFFDFDFDF)),
       left: BorderSide(width: 1.0, color:
       Color(0xFFFFDFDFDF)),
       right: BorderSide(width: 1.0, color:
       Color(0xFFFF7F7F7F)),
       bottom: BorderSide(width: 1.0, color:
       Color(0xFFFF7F7F7F)),
      ),
      color: Colors.grey,
     ),
     child: const Text(
      'OK',textAlign: TextAlign.center, style:
      TextStyle(color: Colors.red)
     ),
    ),
   );
  }}
```

The preceding code displays a container widget and a text widget and a submit button (OK Button) is displayed.

# Layout Application

Layout is an important aspect of app development as it determines how the **User Interface** (**UI**) of an app is displayed on the screen. A well-designed layout can improve the usability and appearance of an app, while a poorly designed layout can make an app difficult to use and unattractive.

In Flutter, layout is achieved using widgets and their relationships to each other. There are several types of layout widgets available in Flutter, such as Container, Row, Column, and Stack, that you can use to arrange and position the children of a widget.

Here is an example of using the Container widget to create a simple layout:

```
Container(
  child: Column(
  children: [
   Container(
   width: 100,
   height: 100,
   color: Colors.red,
   ),
   Container(
   width: 50,
   height: 50,
   color: Colors.blue,
   ),
  ],
  ),
)
```

This will create a layout with a red container and a blue container stacked vertically. The Column widget is used to arrange the children in a vertical line, while the Container widgets are used to specify the size and appearance of each child.

You can also use other layout widgets, such as Row and Stack, to create more complex layouts. For example, you can use a Stack to overlap several children widgets on top of each other, or a Row to arrange children in a horizontal line.

By selecting the appropriate layout widgets and nesting them as needed, you can create a wide range of layouts for your Flutter app.

Widgets are the basic building blocks of the Flutter app. Widgets may comprise of layout elements such as padding, rows and so on. Consider the following code for creating a new Flutter project:

```
import 'package:flutter/material.dart';
 void main() => runApp(MyApp());
 class NewProject extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
  return MaterialApp(
   home: Scaffold(
    appBar: AppBar(
     title: Text("Building New layout"),
    ),
    body: myLayoutWidget(),
   ),
  );
  }
 }
  Widget myLayoutWidget() {
  return Text("Welcome!");
 }
```

# Padding in Flutter

ng may also be considered as a widget in Flutter, as it can be used to add an extra space around the widget. Padding may be performed as follows:

```
 Widget NewProjectWidget() {
  return Padding(
  padding: EdgeInsets.all(7.0),
  child: Text("Welcome!"),
  );
 }
```

In the preceding code, the **`EdgeInsets`** parameter is used for providing the amount of padding. Here all the sides (left, top, right, and bottom) have an equal padding.

# Align Widget

It is used for aligning the widgets at different locations. We can either pass an arbitrary x or y value, or use a predefined value. For example:

```
Alignment.topLeft-Alignment(-1.0,-1.0)
Alignment.topRight-Alignment(1.0,-1.0)
Alignment.topCenter-Alignment(0.0, -1.0)
Alignment.centerLeft-Alignment(-1.0, 0.0)
Alignment.center-Alignment(0.0, 0.0)
Alignment.centerRight - Alignment(1.0, 0.0)
Alignment.bottomLeft-Alignment(-1.0, 1.0)
Alignment.bottomRight-Alignment(1.0, 1.0)
Alignment.bottomCenter-Alignment(0.0, 1.0)
```

# Container

Container Widget in Flutter is a combination of several simpler widgets. Container widget comprises of padding, align, **`DecoratedBox`** and **`ConstrainedBox`**. **`DecoratedBox`** widget is used for defining background color, border color and so on. **`ConstrainedBox`** widget is used for size constraints. Consider the following code on Container:

```
Widget myProjectWidget() {
   return Container(
   margin: EdgeInsets.all(40.0),
   padding: EdgeInsets.all(9.0),
   alignment: Alignment.topRight,
   width: 300,
   height: 200,
   decoration: BoxDecoration(
    color: Colors.red,
    border: Border.all(),
   ),
   child: Text("Welcome!", style: TextStyle(fontSize: 40)),
   );
  }
```

Margin refers to spacing present outside the border.

Rows and columns are also widgets. The widgets in rows are passed in children widget. Consider the following code that depicts the row widget:

```
  Widget newProjectWidget() {
    return Row(
    children: [
      Icon(Icons.home),
      Icon(Icons.home),
      Icon(Icons.home),
      Icon(Icons.home),
    ],
    );
  }
```

In the preceding code, if we replace row with column, we get column widget.

To make contents present in row and column evenly placed, we use Expanded widget.

```
 Widget newProjectWidget() {
    return Row(
    children: [
      Expanded(child: Icon(Icons.home)),
      Expanded(child: Icon(Icons.home)),
      Expanded(child: Icon(Icons.home)),
      Expanded(child: Icon(Icons.home)),
    ],
    );
  }
```

The Expanded widget comprises of a flex parameter. This is helping for providing size weights to the children. For example, if we are having two Containers in a row, the first container takes 60% of size weights of the row and the second takes 40% of size weights.

```
 Widget myLayoutWidget() {
    return Row(
    children: [
      Expanded(
        flex: 6,
        child: Container(
```

```
      color: Colors.red,
     ),
    ),
    Expanded(
     flex: 4,
     child: Container(
      color: Colors.green,
     ),
    ),
   ],
   );
 }
```

## Stacks

The Stack widget places its children in a form of a stack or pile of pancakes. We can place it in form of Row and Column widgets. The child that comes first is placed at the bottom.

Stack widget may be represented as follows:

```
Widget myLayoutWidget() {
 return Stack(
  children: [
   Icon(Icons.home),
   Icon(Icons.home),
   Icon(Icons.home),
   Icon(Icons.home),
  ],
 );
}
```

## GridView

This widget places content in the form of grid of rows and columns.

## Scaffold

Scaffold widget is given by the Material package. In Scaffold, we can add Drawer, BottomNavigationBar, an AppBar, FloatingActionButton, SnackBar and so on.

## Conclusion

Layout is an important aspect of app development in Flutter as it determines how the UI of an app is displayed on the screen. There are several types of layout widgets available in Flutter, such as Container, Row, Column, and Stack, that you can use to arrange and position the children of a widget.

To create a layout in Flutter, you can nest layout widgets and their children as needed, to achieve the desired structure and appearance. You can also customize the appearance and behavior of the layout widgets using various arguments and properties.

By using the appropriate layout widgets and designing your layouts carefully, you can create intuitive, visually appealing, and responsive UI for your Flutter app.

Layouts in Flutter may be build using Widgets. Complex widgets are a combination of different widgets that are also used for building layouts. In this chapter, we have discussed about Flutter layout, type of layout widgets and layout application.

## Questions

1. What are Flutter Layouts? How can they be built?
2. Explain the different types of layout widgets.
3. Explain Layout application in detail.

# CHAPTER 10

# Flutter Database Concepts

**“Mobile will ultimately be the way you provision most of your services. The way I like to put it is, the answer should always be mobile first. You should always put your best team and your best app on your mobile app.”**

*– Eric Schmidt, Executive Chairman of Alphabet Inc.*

## Introduction

Using Flutter, we are able to work with databases. Databases may be accessed in Flutter with the help of certain packages such as **`sqflite`**, **`firebase_database`** and so on. A database is a structured collection of data stored and accessed electronically. In a mobile app, a database can be used to store and retrieve data locally on the device or remotely on a server.

In Flutter, you can use a variety of approaches to access and work with a database, depending on your needs. Some options include:

- **SQLite**: SQLite is a popular open-source SQL database engine that provides a lightweight disk-based database. You can use the **`sqflite`** package to access a SQLite database in Flutter.
- **Firebase**: Firebase is a popular cloud-based mobile development platform that provides a variety of services, including a NoSQL document database called Cloud Firestore. You can use the **`cloud_firestore`** package to access Cloud Firestore in Flutter.
- **Hive**: Hive is a lightweight, fast, and efficient NoSQL database for Flutter. It uses a key-value store and is easy to set up and use. You can use the hive package to access a Hive database in Flutter.
- **Moor**: Moor is a library that provides a convenient API to work with SQLite databases in Flutter. It uses code generation to simplify

database access and includes support for Streams and Futures. You can use the moor package to access a SQLite database in Flutter.

By using a database in your Flutter app, you can store and retrieve data locally or remotely, allowing you to create more powerful and interactive apps.

## Structure

In this chapter, we will cover the following topics:

- SQLite
- Adding Firebase
- Adding Firestore Backend
- JSON, XML, HTML
- Hive
- Moor

## Objectives

We can develop mobile apps and web applications using Flutter. While developing web applications and mobile apps, we need to make use of databases as well. The packages used for databases are: **`sqflite`**, **`firebase_database`** and so on. The package **`sqflite`** is used for accessing and manipulation of SQLite databases. The package **`firebase_database`** is used for accessing and manipulation of NoSql databases given by Google. In this chapter, we will discuss about SQLite, Adding Firebase, Adding Firestore Backend, JSON, XML, HTML and so on.

## SQLite

SQLite is said to be SQL based database engine. It is a small database engine. In Flutter, if we wish to use SQLite database, we make use of **`sqflite`** package that provides numerous functionalities, to work efficiently with SQLite database. SQLite is an open-source SQL database engine that provides a lightweight disk-based database. It is widely used in mobile apps due to its portability and small footprint.

In Flutter, you can use the **sqflite** package to access an SQLite database. To use **sqflite**, you will need to add it to your **pubspec.yaml** file and install it:

```
dependencies:
  sqflite: ^1.3.0
```

Then, you can import the **sqflite** package in your code and use it to perform database operations.

Here is an example of how to open a database and create a table:

```
import 'package:sqflite/sqflite.dart';

// Open the database
final database = openDatabase(
  'my_database.db',
  version: 1,
  onCreate: (db, version) {
    // Run the CREATE TABLE statement on the database
    return db.execute(
      'CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT, age
      INTEGER)',
    );
  },
);
```

This will open the **my_database.db** database (create it if it does not exist) and run the **CREATE TABLE** statement to create a user's table with an id, name, and age column.

You can then use the database object to perform various operations on the database, such as inserting, updating, or deleting rows, or querying the data.

Here is an example of how to insert a row into the users table:

```
Copy code
// Insert a row into the table
await database.insert(
  'users',
  {'name': 'Alice', 'age': 25},
  conflictAlgorithm: ConflictAlgorithm.replace,
);
```

This will insert a new row into the users table with the name Alice and age 25.

You can use similar code to update or delete rows, or to query the data using SELECT statements.

By using **`sqflite`** and SQLite, you can easily store and retrieve data in a local database in your Flutter app.

Following are the functionalities provided by **`sqflite`** package:

- Creation or Opening a SQLite database using **`openDatabase`** method.
- Execution of a SQL statement in a SQLite database using **`execute`** method.
- Advanced methods for querying a SQLite database involves using a **`query`** method that optimizes the query code to fetch relevant information from SQLite database.

We can store and extract information stored in SQLite database using **`sqflite`** package.

In database, a primary key is required. Apart from this, methods such as **`fromMap`** and **`toMap`** are used for performing manipulations in a database.

Some other methods include the following:

- **`-getApplicationDocumentsDirectory`**: It returns the application path directory.
- **`-join`**: It is used for creation of system specific path. It is used for creation of database path.
- **`-openDatabase`**: It is used for opening a SQLite database.
- **`-onOpen`**: It is used for writing code for opening a database.
- **`-onCreate`**: It is used for writing a code when a database is generated for the very first time.
- **`db.execute`**: It is used for execution of SQL queries. It accepts the input query. If there lies a placeholder (?), then in the second argument, values are accepted in the list.

# Adding Firebase

To add Firebase to your Flutter app, you will need to create a Firebase project and add the Firebase dependencies to your Flutter project.

Here are the steps to add Firebase to your Flutter app:

1. Go to the Firebase console and create a new project.
2. In the project dashboard, click on the "Add app" button and select "Flutter" as the platform.
3. Follow the instructions to set up Firebase in your Flutter app. This will involve adding the google-services plugin to your **`pubspec.yaml`** file and adding the required dependencies.
4. Initialize the Firebase app in your Flutter code. You can do this by calling the **`Firebase.initializeApp()`** method in the **`main()`** function of your app.
5. Add the desired Firebase features to your app. For example, to use Cloud Firestore, you can add the **`cloud_firestore`** package to your **`pubspec.yaml`** file and import it in your code.

Here is an example of how to initialize Firebase and use Cloud Firestore in a Flutter app:

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_core/firebase_core.dart';

Future<void> main() async {
  // Initialize Firebase
  await Firebase.initializeApp();

  // Use Cloud Firestore
  Firestore.instance.collection('users').add({
    'name': 'Alice',
    'age': 25,
  });
}
```

This will initialize Firebase and add a new document to the users collection in Cloud Firestore with the name Alice and age 25.

By adding Firebase to your Flutter app, you can access a variety of cloud-based services, such as Cloud Firestore, Firebase Auth, and Firebase Storage, to build powerful and interactive apps.

In order to use Firebase in Flutter, we need to install IDE or any editor.

The following needs to be installed along with Flutter:

- Flutter SDK
- Necessary libraries support
- Platform-specific software
- SDKs

After installation, we need to sign into Firebase using our Google account.

We need to create a Flutter App using a preferred editor or IDE, and then follow the given steps:

1. Install the necessary command line tools as well as Firebase CLI.
2. Using Google account, log into Firebase and use the following command:

   ```
   firebase login
   ```

   FlutterFire CLI may be installed by running the following command:

   ```
   dart pub global activate flutterfire_cli
   ```
3. App may be configured to use Firebase. In order to connect Firebase to Flutter, FlutterFire CLI is used. From your Flutter project directory, run the following command to start the app configuration workflow:

   ```
   flutterfire configure
   ```
4. Initializing Firebase in app. From the given Flutter project directory, we can run the following command in order to install the core plugin:

   ```
   flutter pub add firebase_core
   ```

To ensure that the Firebase configuration is up-to-date, run the following command from Flutter project directory:

```
flutterfire configure
```

In **`lib/main.dart`** file, we can import the Firebase core plugin and the configuration file:

```
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';
```

In **`lib/main.dart`** file, we may initialize Firebase using the **`DefaultFirebaseOptions`** object which may be exported by the

configuration file, as follows:

```
await Firebase.initializeApp(
  options: DefaultFirebaseOptions.currentPlatform,
);
```

Flutter application is re-built using the following command:

```
flutter run
```

In Flutter project directory, run the following command:

```
flutter pub add PLUGIN_NAME
```

From the Flutter project directory, the following command is run:

```
flutterfire configure
```

Running the preceding command is to make sure that Flutter app's Firebase configuration is already up-to-date. Rebuilding Flutter project is performed as follows:

```
flutter run
```

## Adding Firestore Backend

To add Cloud Firestore as a backend to your Flutter app, you will need to create a Firebase project and add the Cloud Firestore dependencies to your Flutter project.

Follow the given steps to add Cloud Firestore as a backend to your Flutter app:

1. Go to the Firebase console and create a new project.
2. In the project dashboard, click on the "Add app" button and select "Flutter" as the platform.
3. Follow the instructions to set up Firebase in your Flutter app. This will involve adding the google-services plugin to your **`pubspec.yaml`** file and adding the required dependencies.
4. Initialize the Firebase app in your Flutter code. You can do this by calling the **`Firebase.initializeApp()`** method in the **`main()`** function of your app.
5. Add the **`cloud_firestore`** package to your **`pubspec.yaml`** file and import it in your code.

6. Use the Firestore instance to access Cloud Firestore and perform database operations.

Here is an example of how to use Cloud Firestore as a backend in a Flutter app:

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_core/firebase_core.dart';

Future<void> main() async {
  // Initialize Firebase
  await Firebase.initializeApp();

  // Add a new document to the 'users' collection
  Firestore.instance.collection('users').add({
    'name': 'Alice',
    'age': 25,
  });

  // Query the 'users' collection
  QuerySnapshot snapshot = await
  Firestore.instance.collection('users').getDocuments();
  snapshot.documents.forEach((doc) {
```

In order to use Cloud Firestore package, we can import it in our project file as follows:

```
import 'package:cloud_firestore/cloud_firestore.dart';
```

Before making use of Firestore, we must make sure that **`FlutterFire`** is initialized. In order to generate a new Firestore instance, we would call the instance getter on **FirebaseFirestore** as follows:

```
FirebaseFirestore firestore = FirebaseFirestore.instance;
```

The preceding code helps in interaction with Firestore using the default Firebase. We can also use Firestore in our app using **`instanceFormethod`**:

```
FirebaseApp secondaryApp = Firebase.app('SecondaryApp');
FirebaseFirestore firestore =
FirebaseFirestore.instanceFor(app: secondaryApp);
```

Firestore performs storage of data in "documents" that are present in "collections". Documents may also comprise of nested collections.

**Query.get** method or the **`DocumentReference.get`** method is used for reading documents or collection in Firestore, as follows:

```
Stream collectionStream =
FirebaseFirestore.instance.collection('users').snapshots();
Stream documentStream =
FirebaseFirestore.instance.collection('users').doc('XYZ123').s
napshots();
```

For managing the state of stream and its disposal, **`listen()`** method is used.

A **`QuerySnapshot`** is used to find the number of documents present, provides access to the required document and the modifications performed on the document since last query.

The exists property is used to find out whether a document is present or not, as follows:

```
FirebaseFirestore.instance
.collection('users')
.doc(userId)
.get()
.then((DocumentSnapshot documentSnapshot) {
if (documentSnapshot.exists) {
print('Document is present in database');
}
});
```

If the document exists, you can read the data of it by calling the data method, which returns a **`Map<String, dynamic>`**, or null if it does not exist:

```
FirebaseFirestore.instance
.collection('users')
.doc(userId)
.get()
.then((DocumentSnapshot documentSnapshot) {
if (documentSnapshot.exists) {
print('Document data: ${documentSnapshot.data()}');
} else {
print('Document does not exist on the database');
}
});
```

We can perform ordering in documents by a specific value with the help of **`orderBy`** method:

```
FirebaseFirestore.instance
.collection('users')
.orderBy('age', descending: true)
.get()
.then(…);
Start & End Cursors#
```

If we wish to insert new document to the collection, it can be done as follows:

```
class UserAddition extends StatelessWidget {
final String fullName;
final String company;
final int age;
UserAddition(this.fullName, this.company, this.age);
@override
Widget build(BuildContext context) {
CollectionReference users =
FirebaseFirestore.instance.collection('users');
Future<void> addUser() {
return users
.add({
'full_name': fullName, // Robin
'company': company, // Amazon
'age': age // 42
})
.then((value) => print("Added User"))
.catchError((error) => print("Failure: $error"));
}
return FlatButton(
onPressed: addUser,
child: Text(
"Adding User",
),
);
}
```

```
}
```

In order to delete the document, **`delete`** method is used.

```
CollectionReference users =
FirebaseFirestore.instance.collection('users');
Future<void> deleteUser() {
return users
.doc('XYZ123')
.delete()
.then((value) => print("User Deleted Successfully"))
.catchError((error) => print("Failure: $error"));
}
```

If we wish to remove specific property, then it is deleted as follows:

```
CollectionReference users =
FirebaseFirestore.instance.collection('users');
Future<void> deleteField() {
return users
.doc('XYZ123')
.update({'age': FieldValue.delete()})
.then((value) => print("Property Deleted Successfully"))
.catchError((error) => print("Failure!: $error"));
}
```

# JSON, XML, HTML

There are numerous packages in Dart and Flutter that supports viewing, writing and reading files written in JSON, XML and HTML.

- Dart XML is a library for parsing, building and querying XML based documents.
- **`xml2json`** package is used for conversion of document in XML format into JSON format.
- **`Japx`** package is used for transforming complex JSON into simplified JSON document.
- **`gpx`** is used for manipulating and storing GPS data in GPX format.

In Flutter, you can use the **`dart:convert`** library to parse and generate XSON, XML, and HTML data.

Here is an example of how to parse a JSON string into a Dart object using the jsonDecode function:

```
import 'dart:convert';

String jsonString = '{"name": "Alice", "age": 25}';

Map<String, dynamic> data = jsonDecode(jsonString);

print(data['name']); // Output: "Alice"
print(data['age']); // Output: 25
```

To generate a JSON string from a Dart object, you can use the **`jsonEncode`** function:

```
Map<String, dynamic> data = {'name': 'Alice', 'age': 25};
String jsonString = jsonEncode(data);
print(jsonString); // Output: '{"name": "Alice", "age": 25}'
```

To parse and generate XML data, you can use the **`xml`** package. Here is an example of how to parse an XML string using the parse function:

```
import 'package:xml/xml.dart' as xml;
String xmlString = '<users><user id="1"><name>Alice</name>
<age>25</age></user></users>';
var document = xml.parse(xmlString);
print(document.rootElement.children[0].getAttribute('id')); //
Output: "1"
print(document.rootElement.children[0].children[0].text); //
Output: "Alice"
print(document.rootElement.children[0].children[1].text); //
Output: "25"
```

To generate an XML string from a Dart object, you can use the **`build`** function:

```
xml.XmlBuilder builder = xml.XmlBuilder();
builder.element('users', nest: () {
 builder.element('user', attributes: {'id': '1'}, nest: () {
  builder.element('name', nest: 'Alice');
  builder.element('age', nest: '25');
 });
```

```
});
var xmlString = builder.build();
print(xmlString); // Output: '<users><user id="1">
<name>Alice</name><age>25</age></user></users>'
```

To parse and generate HTML data, you can use the html package. Here is an example of how to parse an HTML string using the parse function:

```
import 'package:html/parser.dart';

String htmlString = '<div><h1>Hello World</h1><p>This is some
text</p></div>';

var document = parse(htmlString);
print(document.firstChild.localName); // Output:
"div"print(document.firstChild.firstChild.localName); //
Output: "h1"print(document.firstChild.firstChild.text); //
Output: "Hello
World"print(document.firstChild.lastChild.localName); //
Output: "p"print(document.firstChild.lastChild.text); //
Output: "This is some text"
```

To generate an HTML string from a Dart object, you can use the **`build`** function:

```
import 'package:html/parser.dart';

String htmlString = '<div><h1>Hello World</h1><p>This is some
text</p></div>';

var document = parse(htmlString);
print(document.firstChild.localName); // Output: "div"
print(document.firstChild.firstChild.localName); // Output:
"h1"
print(document.firstChild.firstChild.text); // Output: "Hello
World"
print(document.firstChild.lastChild.localName); // Output: "p"
print(document.firstChild.lastChild.text); // Output: "This is
some text"
```

# Hive

Hive is a lightweight, fast, and scalable NoSQL database that is designed to be used on mobile devices. It is written in Dart and is compatible with Flutter, and so, it is a good choice for storing data in Flutter apps.

One of the key features of Hive is its simplicity and ease of use. It has a very small API surface, which makes it easy to learn and use. It also has good performance, with fast read and write times and low memory overhead.

In addition to its simplicity and performance, Hive also has several other features that make it a good choice for storing data in Flutter apps. It has support for transactions, which allows you to perform multiple operations on the database in an atomic way. It also has support for encryption, which allows you to secure your data by encrypting it before storing it in the database.

Hive is a good choice for storing small to medium-sized datasets in Flutter apps. It is not suitable for storing large datasets or for use as a standalone database server, but it is a good choice for many other use cases. If you need a fast and simple way to store data in your Flutter app, Hive might be a good option to consider.

## Moor

Moor is a powerful and feature-rich library for accessing databases from Flutter apps. It is built on top of the SQLite database engine, and it provides a simple and intuitive API for working with databases in Flutter.

One of the key features of Moor is its ability to automatically generate code based on your database schema. This allows you to focus on the business logic of your app, rather than having to write a lot of boilerplate code to access the database. Moor also provides support for streaming data, which allows you to efficiently display large datasets in your app.

In addition to its code generation and streaming support, Moor also has a number of other features that make it a good choice for accessing databases from Flutter apps. It has support for transactions, which allow you to perform multiple operations on the database in an atomic way. It also has support for custom functions, which allow you to define your own functions to be used in your database queries.

Overall, Moor is a powerful and feature-rich library for accessing databases from Flutter apps. If you need to store and retrieve data from a database in your Flutter app, Moor is a good option to consider.

# Conclusion

Database is very important in Flutter, for developing a mobile application or a web-based application. In this chapter, we have learnt about SQLite, adding Firebase, adding Firestore Backend and packages used for JSON, XML, HTML and so on. In Flutter, you can use a variety of approaches to access and work with a database, depending on your needs. Some options include using SQLite, Firebase, Hive, or Moor.

SQLite is a popular open-source SQL database engine that provides a lightweight disk-based database. You can use the sqflite package to access a SQLite database in Flutter.

Firebase is a popular cloud-based mobile development platform that provides a variety of services, including a NoSQL document database called Cloud Firestore. You can use the **`cloud_firestore`** package to access Cloud Firestore in Flutter.

Hive is a lightweight, fast, and efficient NoSQL database for Flutter. It uses a key-value store and is easy to set up and use. You can use the hive package to access a Hive database in Flutter.

Moor is a library that provides a convenient API to work with SQLite databases in Flutter. It uses code generation to simplify database access and includes support for Streams and Futures. You can use the moor package to access a SQLite database in Flutter.

By using a database in your Flutter app, you can store and retrieve data locally or remotely.

# Questions

1. Explain the use of Sqlite in Flutter.
2. Explain the use of Firebase in Flutter.
3. Explain the use of Firestore in Flutter.

# Index

## A



## B

# C



# D

## E



## F

## G



## H



## I



## J

# K



# L



# M

## N



## O



## P



## Q



## R

## S



## T



## U



## V



## W

# X



# Y