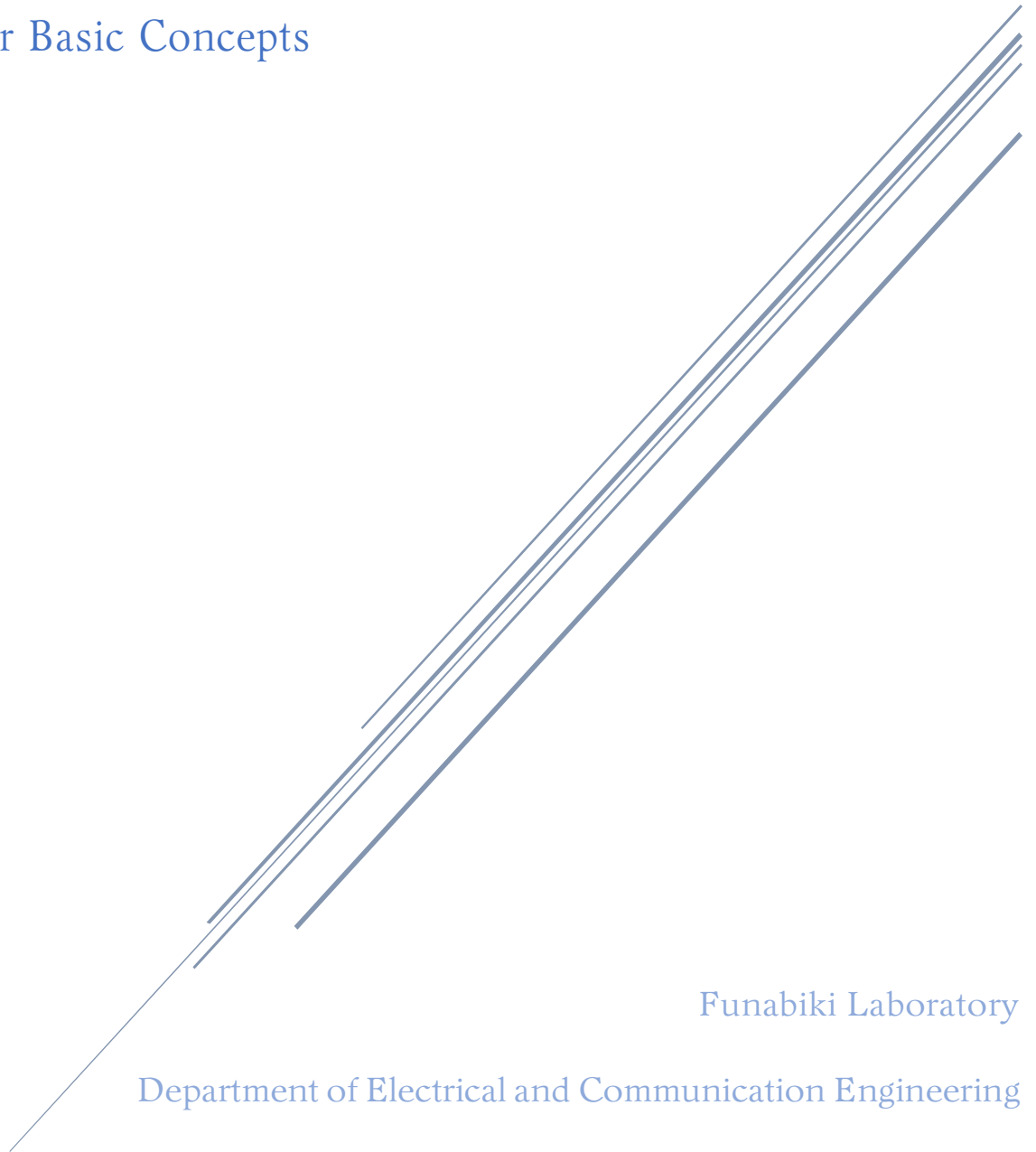


STUDY OF FLUTTER PROGRAMMING LEARNING ASSISTANT SYSTEM

Flutter Basic Concepts



Funabiki Laboratory

Department of Electrical and Communication Engineering

Okayama University

Flutter Programming Learning Assistant System

Correspondence Between Each Topic and Related CMPs	3
What is Dart Language?	4
What is Flutter?	4
Why is Flutter so important?	5
Feature on Flutter	6
What are the differences and similarities between Java and JavaScript?	6
Difference between Flutter and JavaScript	6
Basic Syntax Dart Programming	8
Dart Programming Data Types	9
Dart Programming Operators	10
Dart Programming Decision Making and Loops.....	14
Conditional statement	14
Loops	15
Dart Programming Function	16
Introduction to Flutter Widget	16
MaterialApp	18
Scaffold.....	18
AppBar.....	18
Bottom Navigation Bar	19
Text.....	20
Container	20
Image.....	21
TextField.....	22
Button	22
Icon	23
Card	23
ListTile	24
DropDownButton	25
RadioButton.....	25
Checkbox.....	26
Progress Bar	27
Introduction to Flutter Layouts	27
Single-child layout widgets.....	28
Padding.....	28
Align	28

SizedBox.....	28
Multi-child layout widgets	29
Column.....	29
Row.....	30
ListView.....	30
GridView	31
Stack.....	33
ScrollView	34
Introduction to Flutter Gestures	35
Introduction to Flutter State Management.....	37
Ephemeral state	38
App state	40
Introduction to Flutter Navigation and Routing.....	41
MaterialPageRoute	41
Navigation.push.....	41
Navigation.pop	41
Dialog	41
SnackBar	42

Correspondence Between Each Topic and Related CMPs

※ Click the link to go reference pages directly

Flutter Concepts	Related CMPs Version	Related Problem Numbers
<u>Basic Syntax Dart Programming</u>	F_CMP1	1 to 20
<u>Dart Programming Data Types</u>	F_CMP1	6,7,10,11,12,13,14,15,17,19,20
<u>Dart Programming Operators</u>	F_CMP1	6,7,12,15,16,17,19,20
<u>Conditional statement</u>	F_CMP1	6,7,13,15,17,20
<u>Loops</u>	F_CMP1	14
<u>Dart Programming Function</u>	F_CMP1	12,20
<u>AppBar</u>	F_CMP1	2,3,4,5,6,8,9,10,11,12,15,18,20
<u>Bottom Navigation Bar</u>	F_CMP1	19
<u>Text</u>	F_CMP1	1,3,5
<u>Container</u>	F_CMP1	2,8,16
<u>Image</u>	F_CMP1	3,4,5,6,7,12,18
<u>TextField</u>	F_CMP1	6,8,13
<u>Button</u>	F_CMP1	6,8
<u>Icon</u>	F_CMP1	8,9,11
<u>Card</u>	F_CMP1	9,11
<u>ListTile</u>	F_CMP1	9,19
<u>DropDownButton</u>	F_CMP1	13
<u>Radio Button</u>	F_CMP1	14
<u>Checkbox</u>	F_CMP1	16,17
<u>Progress Bar</u>	F_CMP1	20
<u>Introduction to Flutter Layouts</u>	F_CMP1	5,16
<u>Padding</u>	F_CMP1	3,4,9,10,17
<u>Align</u>	F_CMP1	1,2,10
<u>SizedBox</u>	F_CMP1	4,5,11
<u>Column</u>	F_CMP1	4,8,11,18
<u>Row</u>	F_CMP1	4,18
<u>ListView</u>	F_CMP1	10,15,18
<u>GridView</u>	F_CMP1	11
<u>ScrollView</u>	F_CMP1	12
<u>Introduction to Flutter Gestures</u>	F_CMP1	7,16,18
<u>Introduction to Flutter State Management</u>	F_CMP1	6,13,14,15,16,17,19,20
<u>MaterialPageRoute</u>	F_CMP1	18
<u>Navigation.push</u>	F_CMP1	18
<u>Navigation.pop</u>	F_CMP1	16
<u>Dialog</u>	F_CMP1	16
<u>SnackBar</u>	F_CMP1	17,19

What is Dart Language?

Dart is one of the programming languages developed by Google and the official programming language for Flutter. Dart is called an open source and general purpose programming language. Dart was created to meet the development of fast apps for various platforms in the form of a client-optimized language. Which means that Dart provides productive programming for multi-platform development such as mobile, desktop, and web.

Dart was originally used to build the web on Google. The initial goal was to replace JavaScript which was considered to have many disadvantages. After that, the release of Flutter SDK for multi-platform development became a new spotlight on the Dart language.

Dart uses static type checking to ensure that the value of a variable always matches the static type of the variable. This means that the type checking process of a variable is always performed at compile time. This process is performed by the Dart compiler to ensure that the code written will not generate errors at runtime. The code typing system in Dart is also flexible, allowing it to be used with dynamic types that will be combined with runtime checks. Dart also has built-in sound null safety where values cannot be null unless the developer says they can be. With sound null safety, Dart can protect against null exceptions at runtime through static code analysis.

What is Flutter?

Flutter is an SDK (Software Development Kit) developed by Google to make good applications that can run on various platforms. Mobile flutter is a platform that has been widely used by flutter developers to create mobile applications with attractive designs by only utilizing one type of base coding (codebase). That way, the application can be downloaded and used on various platforms, from Android, iOS, website, to desktop.

Applications developed by flutter use the Dart programming language and widgets that are already in this framework. Flutter itself is a widget. Widgets in flutter also support animation and gestures. The logic of the application is based on reactive programming. Widgets in flutter can also have state. By changing the state of the flutter widget, it will automatically perform a comparison between the new and old states. Then the widget will be rendered with the required changes rather than re-rendering the entire widget.

Flutter has been utilized by large companies to create their own mobile applications, such as Alibaba, BMW, and eBay. So, more and more developers are using flutter to help their work to create mobile applications.

Why is Flutter so important?

Flutter is a popular framework used by developers since 2019 although it has actually been developed from 2015. Flutter's popularity is because it uses a programming language called Dart. Dart is easy to understand, learn, and fast. By using flutter, developers can customize the UI appearance and design as desired, so that it can be designed well and uniquely and different from other mobile applications.

Currently, the trend of flutter among developers reaches 46% as a multi-platform framework. In 2023 alone, there were more than 36,889 projects developed using flutter and 11 billion downloads. according to Google Trends analysis, flutter continues to gain popularity year by year, with a significantly increasing trend compared to Kotlin, a native programming language for Android apps. This indicates that developers are increasingly inclined to adopt flutter as their primary choice in mobile app development.

One of the advantages of flutter is that it runs very fast. Flutter is an application development framework that uses the Skia-2D graphic engine which is also used in Chrome and Android. Skia-2D is a powerful and efficient graphics engine, which allows flutter to produce sharp and smooth images on various devices. Flutter's code uses the Dart language, which is a fast, efficient, and easy-to-learn programming language. Dart allows flutter to be compiled to native 32-bit and 64-bit ARM code for iOS and Android. This makes flutter a fast, efficient, and reusable framework.

Flutter is known to be a very productive framework. This is because flutter has a hot-reload feature that allows developers to view compilation results in real-time. This feature is very useful for speeding up application development, because it allows developers to see code changes instantly on the device without the need to wait for the application to restart. The hot-reload feature works by monitoring code changes in flutter files. When any code changes are detected, flutter will compile the changed code and apply the changes to the running application. This process takes place very quickly, so developers can see their code changes instantly on the device.

Flutter is also open source with a BSD license. A BSD (Berkeley Source Distribution) license is one that allows free use, modification, and distribution of software. The BSD license allows commercial use and without any restrictions. The code in flutter comes from the contributions of hundreds of developers from around the world. Many plugins have been created by developers. Each developer can contribute to the development of flutter by reporting bugs/issues or improving existing code. The source code of flutter can be found at the link.

Feature on Flutter

Flutter is one of the most popular frameworks which includes numerous features when it comes to multi-platform development. Flutter has so many special features that are very useful and popular, but the following will be the most popular features.

- Hot reload
- Single codebase, multi-platform reach
- Expressive UI & Animations
- Native rendering for each platform,
- Rich Ecosystem & Community
- Declarative Programming
- Fast Development & Prototyping
- Future-proof & Evolving

These features enable flutter developers to create beautiful, high-performance, and reusable apps for multiple platforms.

What are the differences and similarities between Java and JavaScript?

Flutter and JavaScript are two popular app development frameworks. Both have some things in common. First, both Flutter and JavaScript embrace a familiar landscape. Dart, flutter's language, has similarities to JavaScript in terms of syntax and object-oriented nature. This familiarity makes it easier for JavaScript developers to learn Dart, thus lowering the barrier to entry into the flutter world. Conversely, developers who are already familiar with Dart can utilize their existing knowledge to understand core JavaScript concepts, thereby gaining knowledge for web development.

Flutter and JavaScript also have similarities in terms of cross-platform capabilities. Both can be used to create applications that can run on various devices, such as iOS, Android, web, and desktop. This saves application development time and costs, as developers only need to write code once to run on multiple platforms.

Flutter and JavaScript prioritize rapid iteration and prototyping. Flutter's hot reload feature allows developers to see changes instantly on their devices. Similarly, the dynamic nature of JavaScript enables rapid prototyping and testing in web browsers.

Difference between Flutter and JavaScript

1. Mobile App Development

- **Flutter:** Uses Dart programming and widgets, which can be used to create complex and responsive UIs. Flutter also supports smooth and responsive animations.
- **JavaScript:** HTML is used to define the structure and content of the page, while CSS is used to define the style of the page. JavaScript can also be used to create animations, but not as easily as flutter.

2. Syntax

- **Flutter:** Similar to JavaScript, but there are some differences, such as the use of data types, operators, and functions.
- **JavaScript:** JavaScript is a scripting language that has a simple and easy-to-learn syntax.

3. Compilation

- **Flutter:** Ahead-of-time (AOT) compilation, so Dart code will be compiled into native code for the target platform.
- **JavaScript:** JavaScript uses interpretation, so JavaScript code will be interpreted by the browser or virtual machine when the application is running.

4. Learning Curve

- **Flutter:** Has extensive documentation and community, so developers can easily learn flutter.
- **JavaScript:** Has a low learning curve, as it is a simple and easy-to-learn scripting language.

Basic Syntax Dart Programming

1. A simple variable declaration

```
var name = 'Dart';
```

Dart allows straightforward variable declaration using "var" or explicit type annotations.

2. Null Safety

Dart embraces null safety, ensuring variables explicitly handle potential nothingness.

Null safety introduces 3 key change :

- a. When you specify a type for a variable, parameter, or another relevant component, you can control whether the type allows **null**. To enable nullability, you add a **?** or **!** (not recommended to use because will force null variable to non-nullable variable) to the end of the type declaration.

```
String? name //Nullable type.  
String name //Non-nullable type.
```

- b. Must initialize variables before using them.
- c. Can't access properties or call methods on an expression with a nullable type.

3. Default value

With null safety, you must initialize the values of non-nullable variables before you use them. You don't have to initialize a local variable where it's declared, but you do need to assign it a value before it's used.

```
int tapCount = 0;
```

4. Final and const

A final variable can be set only once.

```
final name = 'Bob'; //Without a type annotation.  
final String longName = 'Bobby';
```

A const variable is a compile-time constant.

```
const bar = 1000000; //Unit of pressure (dynes/cm2)  
var foo = const [];  
foo = [1, 2, 3]; //Was const []
```

5. Comments usage

A single-line comment begins with **//**. A multi-line comment begins with **/*** and ends with ***/**.

```
void main() {  
  // TODO: refactor  
  print('Welcome back');  
  /*
```

```
Cars car = Cars();
car.engine();
car.clean();
*/
}
```

6. Output

In Dart, you have several options for "displaying" data,

- **Text widget:** Update the text content of a [Text](#) widget to show your data.
- **Log:** Use the [print](#) function to write data to the console. This is similar to [console.log\(\)](#) in JavaScript and is helpful for debugging and inspecting values.
- **Debug tools:** Leverage Flutter's DevTools to inspect variables and data structures interactively.
- **Toasts:** Display temporary messages containing your data using the [Toast](#) widget.
- **Custom formatting:** You can format your data before displaying it using Dart's string interpolation and formatting features.

Dart Programming Data Types

The Dart language has special support for the following:

- Numbers (int, double)
- Strings (String)
- Booleans (bool)
- Records ((value1, value2))
- Lists (List, also known as arrays)
- Sets (Set)
- Maps (Map, associates keys and values.)
- Runes (Runes; often replaced by the characters API)
- Symbols (Symbol)
- The value null (Null)

Dart allows you to build objects directly from simple values. Strings like "this is a string" and booleans like true are examples of these "value builders" called literals.

Since everything in Dart is ultimately an object, you can also use constructors to set up variables with specific details. Certain built-in types like maps even have their own dedicated constructors, like Map().

Some other types also have special roles in the Dart language:

- **Object:** The superclass of all Dart classes except Null.
- **Enum:** The superclass of all enums.

- **Future and Stream:** Used in asynchrony support.
- **Iterable:** Used in for-in loops and in synchronous generator functions.
- **Never:** Indicates that an expression can never successfully finish evaluating. Most often used for functions that always throw an exception.
- **dynamic:** Indicates that you want to disable static checking. Usually you should use **Object** or **Object?** instead.
- **void:** Indicates that a value is never used. Often used as a return type.

In Dart also have generic types. The `<...>` syntax denotes "List" as a parameterized type, allowing it to take on various concrete types through type arguments placed within the brackets. For lists and sets, add `<type>` before the bracket, and for maps, use `<keyType, valueType>`. These parameterized literals define the data types allowed within, enhancing both clarity and type safety.

```
var name = <String>['Seto', 'Kathy', 'Lark'];
var views = Map<int, View>();
```

We also used enumerated types. Often called enums, used to represent a fixed number of constant values.

```
enum Color { red, green, blue }; //to declare.
//Using enum like any other static variable.
final favoriteColor = Color.blue;
//get list of all enums value, can also access their index.
List<Color> colors = Color.value;
```

Dart Programming Operators

This table offers a general guide to Dart's operator relationships, showcasing how operators combine values (associativity) and their order of execution (precedence).

Description	Operator	Associativity
unary postfix	<code>expr++ expr-- () [] ?[] . ?. !</code>	None
unary prefix	<code>-expr !expr ~expr ++expr --expr await expr</code>	None
multiplicative	<code>* / % ~/</code>	Left
additive	<code>+ -</code>	Left
shift	<code><< >> >>></code>	Left
bitwise AND	<code>&</code>	Left
bitwise XOR	<code>^</code>	Left
bitwise OR	<code> </code>	Left
relational and type test	<code>>= > <= < as is is!</code>	None
equality	<code>== !=</code>	None

Description	Operator	Associativity
logical AND	<code>&&</code>	Left
logical OR	<code> </code>	Left
if null	<code>??</code>	Left
conditional	<code>expr1 ? expr2 : expr3</code>	Right
cascade	<code>.. ?..</code>	Left
assignment	<code>= *= /= += -= &= ^= etc.</code>	Right

1. Arithmetic operators

Operator	Meaning
<code>+</code>	Add
<code>-</code>	Subtract
<code>-expr</code>	Unary minus, also known as negation (reverse the sign of the expression)
<code>*</code>	Multiply
<code>/</code>	Divide
<code>~/</code>	Divide, returning an integer result
<code>%</code>	Get the remainder of an integer division (modulo)

Dart also supports both prefix and postfix increment and decrement operators.

Operator	Meaning
<code>++var</code>	<code>var = var + 1</code> (expression value is <code>var + 1</code>)
<code>var++</code>	<code>var = var + 1</code> (expression value is <code>var</code>)
<code>--var</code>	<code>var = var - 1</code> (expression value is <code>var - 1</code>)
<code>var--</code>	<code>var = var - 1</code> (expression value is <code>var</code>)

2. Equality and relational operators

Operator	Meaning
<code>==</code>	Equal; see discussion below
<code>!=</code>	Not equal
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

This operators will return with bool type of value

3. Type test operator

Operator	Meaning
<code>as</code>	Typecast (also used to specify library prefixes)
<code>is</code>	True if the object has the specified type
<code>is!</code>	True if the object doesn't have the specified type

- Use the `as` operator to cast an object to a particular type if and only if you are sure that the object is of that type.
- If you aren't sure that the object is of type `T`, then use `is T` to check the type before using the object.

4. Assignment operator

<code>=</code>	<code>*=</code>	<code>%=</code>	<code>>>>=</code>	<code>^=</code>
<code>+=</code>	<code>/=</code>	<code><<=</code>	<code>&=</code>	<code> =</code>
<code>-=</code>	<code>~/=</code>	<code>>>=</code>		

Here's how compound assignment operators work:

	Compound assignment	Equivalent expression
For an operator <i>op</i> :	<code>a op= b</code>	<code>a = a op b</code>
Example:	<code>a += b</code>	<code>a = a + b</code>

5. Logical operators

Operator	Meaning
<code>!expr</code>	inverts the following expression (changes false to true, and vice versa)
<code> </code>	logical OR
<code>&&</code>	logical AND

Can invert or combine boolean expressions using the logical operators.

6. Bitwise and shift operators

Operator	Meaning
<code>&</code>	AND
<code> </code>	OR
<code>^</code>	XOR

Operator	Meaning
<code>~expr</code>	Unary bitwise complement (0s become 1s; 1s become 0s)
<code><<</code>	Shift left
<code>>></code>	Shift right
<code>>>></code>	Unsigned shift right

7. Cascade notation

Dart's cascades (`..` and `?..`) let you chain operations on a single object, accessing its properties and calling methods in a neat flow. This avoids storing the object in a temporary variable, making your code smoother and more elegant.

The previous example :

```
var painting = Paint();
painting.color = Colors.black;
painting.strokeWidth = 5.0;
painting.strokeCap = StrokeCap.round;
```

After cascades :

```
var painting = Paint()
  ..color = Colors.black
  ..strokeWidth = 5.0
  ..strokeCap = StrokeCap.round;
```

For objects that might be null, initiate your cascade with a null-shorting operator (`?..`) on the first operation to avoid unnecessary evaluations. Can also nest multiple chains of operations on the same object for even more conciseness. Make sure your cascade starts with a function that actually returns an object, or it will runtime errors.

8. Other operators

Operator	Name	Meaning
<code>()</code>	Function application	Represents a function call
<code>[]</code>	Subscript access	Represents a call to the overridable <code>[]</code> operator; example: <code>fooList[1]</code> passes the int <code>1</code> to <code>fooList</code> to access the element at index <code>1</code>
<code>?[]</code>	Conditional subscript access	Like <code>[]</code> , but the leftmost operand can be null; example: <code>fooList?[1]</code> passes the int <code>1</code> to <code>fooList</code> to access the element at index <code>1</code> unless <code>fooList</code> is null (in which case the expression evaluates to null)
<code>.</code>	Member access	Refers to a property of an expression; example: <code>foo.bar</code> selects property <code>bar</code> from expression <code>foo</code>
<code>?.</code>	Conditional member access	Like <code>.</code> , but the leftmost operand can be null; example: <code>foo?.bar</code> selects property <code>bar</code> from expression <code>foo</code> unless <code>foo</code> is null (in which case the value of <code>foo?.bar</code> is null)
<code>!</code>	Null assertion operator	Casts an expression to its underlying non-nullable type, throwing a runtime exception if the cast fails; example: <code>foo!.bar</code> asserts <code>foo</code> is non-null and selects the property <code>bar</code> , unless <code>foo</code> is null in which case a runtime exception is thrown

Dart Programming Decision Making and Loops

Conditional statement

In Dart we have several conditional statements:

1. If...

Using `if` to specify code that needs to be executed if a certain condition is true.

2. else..

Using `else` to specify code that needs to be executed if a certain condition is false.

3. else if..

Using `else if` to specify a new condition that needs to be executed if the first condition is false.

```
if (isRaining()) {  
  bringRainCoat();  
} else if (isSnowing()) {  
  wearCoat();  
} else {  
  useSunscreen();  
}
```

4. switch..case

Using `switch` to select one of many kinds of code specifications that need to be executed.

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

In expressing `if` and `else` in dart, there are conditional expressions that allow you to concisely evaluate expressions that may require `if-else` statements:

`condition ? expr1 : expr2`

If the condition is true, evaluates `expr1` (and returns its value). Otherwise, evaluates and returns the value of `expr2`.

`expr1 ?? expr2`

If `expr1` is non-null, it returns its value. Otherwise, evaluates and returns the value of `expr2`.

```
var activate = isActive ? 'activate' : 'deactivate';  
String playerName(String? name) => name ?? 'Anonymous';
```

Loops

In Dart we have several control flows (loops) :

1. for loops

You can iterate with the standard `for` loop. For example:

```
for(int i=0; i<5; i++) {  
    print('**');  
}
```

Navigating lists and sets (Iterables) doesn't always require explicit index tracking. For a concise and efficient approach, leverage the `for-in` loop.

```
for(final player in players) {  
    player.training();  
}
```

Iterable classes also offer the `forEach()` method for functional-style iteration, providing additional flexibility.

```
var someCollection = [1, 2, 3];  
someCollection.forEach(print);
```

2. while and do while loops

A while loop evaluates the condition before the loop :

```
while(!isPlayed()) {  
    doSomething();  
}
```

A do-while loop evaluates the condition after the loop :

```
do {  
    printLine();  
} while(!atEndOfPage());
```

3. break and continue

Use break to stop looping :

```
while(isStopped()) {  
    if (shutdownRequested()) break;  
    processIncomingReq();  
}
```

Use continue to skip to the next loop iteration :

```
for(int i = 0; i<players.length; i++) {  
    var player=players[i];  
    if(player.joinedMonth < 5) {  
        continue;  
    }  
    player.training();  
}
```

If you're using an iterable such as a list or set, how you write the previous example might differ:


```
players
  .where((p) => p.joinedMonth >= 5)
  .forEach((p) => p.training());
```

Dart Programming Function

Functions in Dart are objects, so developers can assign them to variables, send them as arguments, or even call them like classes. They possess a dedicated type ("Function") and can be manipulated like any other data entity.

```
bool isNoble(int atomicNum) {
  return _nobleGases[atomicNum] != null
}
```

For functions that contain just one expression, you can use a shorthand syntax:

```
bool isNoble(int atomicNum) => _nobleGases[atomicNum] != null
```

The `=> expr` syntax is a shorthand for `{ return expr; }`. The `=>` notation is sometimes referred to as *arrow* syntax.

```
void main() {
  print("Hello World!");
}
```

Every app must have a top-level `main()` function, which serves as the entrypoint to the app. The `main()` function returns `void` and has an optional `List<String>` parameter for arguments.

Introduction to Flutter Widget

Widgets are the most important thing about flutter. Flutter's UI construction adopts a modern framework heavily influenced by React. Flutter embraces a widget-based approach. Each widget is a self-contained building block with its own visual representation and state, empowering you to build complex UIs from simple, reusable components.

The minimal flutter app simply calls the `runApp()` function with a widget. The `runApp()` function takes the given `Widget` and makes it the root of the widget tree.

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

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
```

```

    ),
  ),
);
}

```

Flutter has majorly two types of widget - Stateless and Stateful widgets.

- **Stateless widgets:** No internal state, update only with parent changes. Build function (responsible to build the widget) is triggered only when the constructor is called.
- **Stateful widgets:** Have internal state, update with internal or parent changes. Build function is triggered when the constructor is called, also when `initState()` and `setState()` are called.

Widgets also can be grouped into multiple categories based on the features :

1. Platform specific widgets

Flutter caters to each platform's unique design language. Android embraces Material widgets based on Google's Material Design guidelines, while iOS utilizes Cupertino widgets built in accordance with Apple's Human Interface Guidelines.

a. Examples of Material Widgets :

- **AppBar** : A Material Design app bar with various options for navigation and actions.
- **FloatingActionButton** : A Material Design floating action button used for primary actions.
- **RaisedButton and ElevatedButton** : Buttons styled according to Material Design.
- **Card** : A Material Design card for displaying content.
- **TextField** : A text input field with Material Design styling.
- **ListTile** : A single fixed-height row typically used in lists.
- **BottomNavigationBar** : A bottom navigation bar for app navigation.

b. Examples of Cupertino Widgets :

- **CupertinoNavigationBar** : A navigation bar with iOS-style navigation.
- **CupertinoButton** : A button styled according to iOS guidelines.
- **CupertinoTextField** : A text input field with iOS styling.
- **CupertinoPicker** : A wheel-like picker control often used in iOS.
- **CupertinoActionSheet** : An iOS-style action sheet for presenting options.

2. Layout widgets

Flutter facilitates the creation of complex user interfaces through the composition of individual widgets. Many widgets offer built-in layout functions, allowing the arrangement and organization of multiple widgets within a single parent widget.

a. Single-child layout widgets (Align, Center, Container, Padding, SizedBox, etc)

- b. Multi-child layout widgets (Column, Row, GridView, ListView, etc)
 - c. Sliver widgets (SliverGridDelegate, etc)
3. Platform independent/basic widgets

Flutter provides a large number of basic widgets to create both simple and complex user interfaces in a platform-independent manner. Let's see some of the basic widgets.

MaterialApp

MaterialApp is a widget that is used as the root of the entire flutter app interface. It is the widget that is first built in the widget tree, and governs much of the configuration that affects the entire app. Its key properties :

- title : gives the app a title
- theme : defines the overall theme of the app with ThemeData
- home : defines the widget that will appear first when the application runs.
- routes : defining the various routes in the application
- initialRoute: determines the route that will appear first when the application is running.

Scaffold

Scaffold is a widget used to create a general interface framework that follows Material Design design guidelines. Its key properties :

- appBar: adds an app bar to the page with the AppBar widget.
- body: places the main widget or page content.
- floatingActionButton: adds a floating action button with the FloatingActionButton widget.
- bottomNavigationBar: adds a bottom navigation bar with the BottomNavigationBar widget.
- drawer : adds a side drawer or navigation drawer, which is a panel that can be dragged from the side of the screen to display additional navigation.

AppBar

AppBar is the topmost (or sometimes bottommost) component, contains the toolbar and action buttons, also can provide navigation and information. Its key properties :

- actions : contains a list of widgets as parameters that will be displayed on the app bar behind the title in the form of a row.
- title : adds text or logo in the form of widgets as parameters that will be displayed on the app bar. Can also center the title by making true centerTitle.
- backgroundColor : gives the background color to the app bar.
- elevation : controls the shadow effect displayed below the app bar.
- shape : sets the shape of the app bar and sets the shadow as well.

Bottom Navigation Bar

BottomNavigationBar is a widget that displays a row of small widgets at the bottom of a flutter app. BottomNavigationBar allows you to select one item at a time and quickly navigate to a given page. Its key properties :

- items : required property contains a list of widgets as parameters that will be displayed, using widget of a type BottomNavigationBarItem which contains icon, label and onTap.
- type : sets the type of the bottom navigation bar. There are fixed (when there are less than 4 items) and shifting (when there are 4 or more items).
- currentIndex : initiate the index of the item for the current active BottomNavigationBarItem.

```
MaterialApp(  
  title: 'Flutter Demo',  
  home: Scaffold(  
    appBar: AppBar(  
      title: const Text("Home"),  
      centerTitle: true,  
      actions: const [  
        Icon(Icons.add),  
        Icon(Icons.menu),  
      ],  
      elevation: 1,  
    ),  
    bottomNavigationBar: BottomNavigationBar(  
      currentIndex: 0,  
      type: BottomNavigationBarType.fixed,  
      items: const [  
        BottomNavigationBarItem(  
          icon: Icon(Icons.home),  
          label: 'home',  
        ),  
        BottomNavigationBarItem(  
          icon: Icon(Icons.list_alt),  
          label: 'catalog',  
        ),  
        BottomNavigationBarItem(  
          icon: Icon(Icons.person),  
          label: 'profile',  
        ),  
      ],  
    ),  
    body: Container(),  
  ),  
);
```

Text

The Text widget displays a string of text with a single style. The style argument is optional. When omitted, the text will use the style from the closest enclosing DefaultTextStyle. Its key properties :

- String data: the content of the text that you want to display on the page.
- style : set the style of the string with the style property and TextStyle class. Can adjust color, font style, font size, font weight, line height, etc.
- textAlign : set the alignment of the text that wants to be displayed.

Widget Text also has a special constructor, Text.rich to display text with various styles and colors in one paragraph.

```
const Text(  
  "Let's study Flutter",  
  style: TextStyle(  
    fontSize: 24,  
    fontWeight: FontWeight.bold,  
    fontStyle: FontStyle.italic,  
    letterSpacing: 5.0,  
    color: Colors.redAccent  
  ),  
  textAlign: TextAlign.justify,  
)
```

Container

The Container serves as a versatile utility widget in flutter. It handles painting, positioning, and sizing, allowing to easily group and arrange other widgets within the overall UI. Its key properties :

- child : stores its children. The child class can be any widget.
- color : set the background color of the entire container.
- height and width : by default, the container takes as much space as its child needs, but we can also specify the size ourselves.
- margin : to create empty white space around the container. Here we use EdgeInsets to set the margin.
- padding : to create space from the container border to its child. Here we also use EdgeInsets to set it.
- alignment : set the position of the child inside the container. We can align in different ways: bottom, center left, left, right, etc.
- decoration : add decoration to the container by using BoxDecoration class. We can set border, border radius, color, shadow, image, shape, etc.
- transform : helps to rotate the container by using the Transform class.

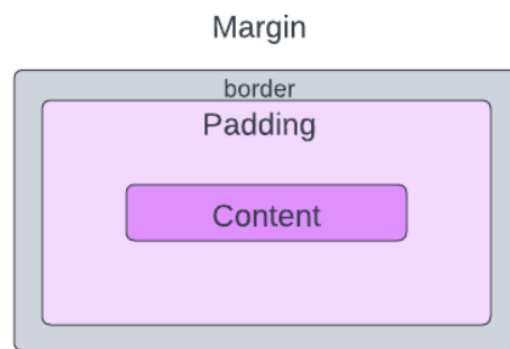
```
Container(  
  child: Text(  
    "Let's study Flutter",  
    style: TextStyle(  
      fontSize: 24,  
      fontWeight: FontWeight.bold,  
      fontStyle: FontStyle.italic,  
      letterSpacing: 5.0,  
      color: Colors.redAccent  
    ),  
    textAlign: TextAlign.justify,  
  ),  
)
```

```

height: 300, width: 300,
margin: const EdgeInsets.symmetric(vertical: 10),
padding: const EdgeInsets.all(30),
alignment: Alignment.topCenter,
decoration: BoxDecoration(
  color: Colors.blueAccent,
  border: Border.all(width: 2.0, color: Colors.black),
  borderRadius: BorderRadius.circular(20.0),
)
child: const Text("Hello Flutter!"),
)

```

Container have properties margin, border, padding, and child (content), which illustrated like this :



Image

Image widgets are used to display images in the application. Image widgets provide various constructors to load images from various sources.

- Image : Generic image loader using ImageProvider class.
- Image.asset : Load image from flutter project's assets.
- Image.file : Load image from system folder
- Image.memory : Load image from memory
- Image.Network : Load image from network

The easiest way to load and display images is to put the images into the assets project.

1. Create an assets folder in the flutter project folder and insert the files you want to use.
2. Assign the assets into pubspec.yaml as follows:

```

flutter:
  assets:
    - assets/car_yellow.png

```

3. Can load and display images to the application by entering the url of the assets.

```

Image.asset('assets/car_yellow.png')

```

output :



TextField

TextField and TextFormField are the most used widgets for user input. They handle various forms of input like email, password, address, etc. To read the user's input, must to initiate the TextEditingController variable and set it as a controller property of TextField widget.

```
final _controller = TextEditingController();
```

Its key properties :

- controller : controls the text being edited. TextEditingController can be used to retrieve the contents of this widget with controller.text, or delete the contents with controller.clear(), etc.
- keyboard type : set the type of input text with TextInputType, can be streetAddress, emailAddress, password etc.
- decoration : add decoration to the text form field by using InputDecoration class. We can set hint text, color, icon, label, error text, etc.
- onChanged : used when the user initiates a change of value in the text field when he has filled or deleted text from the field.

```
TextFormField(  
  controller: _controller,  
  keyboardType: TextInputType.phone,  
  decoration: const InputDecoration(  
    hintText: 'Type your phone number',  
    icon: Icon(Icons.phone),  
  ),  
)
```

Button

Buttons are control elements that provide user-triggered events. The standard feature present on every flutter button is :

- Easy addition of different child widgets for different purposes
- Easy assignment of button themes
- Easy addition of themed text and icons
- Providing action functions

Flutter has various kinds of buttons, which are:

- Elevated Button
- Floating Action Button
- Outlined Button
- Icon Button
- Text Button
- Dropdown Button
- PopUp Menu Button

The basic button on a flutter is an elevated button. When an elevated button is pressed, its height will be raised to a certain value. Its key properties :

- child : stores its children. The child class can be any widget. Basically use text widgets or icon widgets.
- onPressed : called when the button is tapped or activated.
- enabled : used to initiate whether the button is enabled or disabled.
- style : customizes the appearance of the button with the ButtonStyle class or the button styleFrom() methods.

```
ElevatedButton(  
  onPressed: () {},  
  style: const ButtonStyle(  
    backgroundColor: MaterialStatePropertyAll<Color>(Colors.teal),  
    shadowColor: MaterialStatePropertyAll<Color>(Colors.red),  
    elevation: MaterialStatePropertyAll<double>(5.0)  
  ),  
  child: const Text('Woolha.com'),  
)
```

Icon

Icon widget is used to display a glyph from a font described in IconData class using Material icons. Search and find the perfect icon on the [Google Fonts](#) website. To use this class, make sure you set uses-material-design: true in your project's pubspec.yaml file in the flutter section.

```
const Icon(  
  Icons.access_time_filled,  
  size: 50.0,  
  color: Colors.blueGrey,  
  weight: 3.0,  
)
```

Card

Card widget is a built-in flutter widget that derives its design from Google's Material Design Library. This widget makes it easy to create a bland space or panel with rounded corners and a slight elevation at the bottom. Its key properties :

- child : takes in a widget as an object to show inside the widget.
- color : set background color to the card.
- elevation : controls the shadow effect displayed below the card.
- margin : to create empty white space around the card. Here we use EdgeInsets to set the margin.
- shape : takes ShapeBorder class as the object to decide the shape of the card.
- borderOnForeground : to set card borders or borderless.

```
Card(
  shape: RoundedRectangleBorder(
    borderRadius: BorderRadius.circular(10.0),
  ),
  elevation: 5.0,
  child: ListTile(
    title: const Text('This is a card with a list tile'),
    leading: const Icon(Icons.card_giftcard),
    trailing: IconButton(
      icon: const Icon(Icons.delete),
      onPressed: () {},
    ),
  ),
)
```

ListTile

ListTile widget is a ui element that is used to display a package of information to make it more compact. Divided into 3 sections, namely start, center, and end. The Start section contains the leading widget, the Center section includes the title and subtitle, and the End section contains the trailing widget. Its key properties :

- title : The primary content of the list tile.
- subtitle : Additional content displayed below the title.
- leading : A widget to display before the title. It can be anything, most will provide an image or icon.
- trailing : A widget to display after the title. It can be anything, most will provide a button.
- isThreeLine : Whether this list tile is intended to display three lines of text.
- onTap : called when the user taps this list tile.

```
ListTile(
  leading: const Icon(Icons.car_rental),
  title: const Text("Car"),
  subtitle: const Text("For car controller"),
  trailing: IconButton(
    icon: const Icon(Icons.more_vert),
    onPressed: () {},
  ),
)
```

DropDownButton

DropDownButton widget is a widget that provides the option to select an item from a list of items. The button shows the currently selected item as well as an arrow that opens a menu for selecting another item. All entries in a given menu must represent values of a consistent type. Usually, an enum is used. Each DropdownMenuItem in the item must be specialized with an argument of the same type. Its key properties :

- items : define various items that are to be defined in our dropdown menu/list. It is a list of items that users can select.
- value : currently selected item.
- style : style text in the dropdown menu/list like color, fontsize, fontweight, etc.
- alignment : defines how the hint or selected item is positioned within the button.
- icon : display an icon to the dropdown button.
- selectedItemBuilder : if we want to display some other text instead of the option selected on the button, we will use selectedItemBuilder.

```
//declare dropdown content in list
final List<String> _items = ['Option 1', 'Option 2', 'Option 3'];
//create dropdown and access _items with .map
DropDownButton(
  items: _items.map((item) {
    return DropdownMenuItem(
      child: Text(item),
      value: item,
    );
  }).toList(),
  onChanged: (value) {
    print('Selected item: $value');
  },
  value: _items[0],
)
```

RadioButton

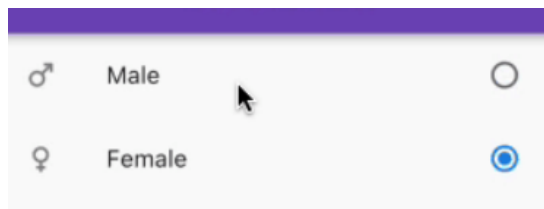
RadioListTile widget is a widget that provides a group of buttons that can select only one of them. This widget uses a generic type, so it needs to provide the generic type before the bracket, usually using enums. Its key properties :

- value : The value represented by this radio button.
- groupValue : currently selected value for this group of radio buttons.
- onChanged : called when the user selects this radio button.
- title & subtitle : The primary content of the list tile.
- activeColor : The color to use when this radio button is selected.

```
//declare radio list with enum
enum Gender { male, female }
Gender? _gender = Gender.male;
```

```
//create radio button inside of ListView
ListView(
  children: [
    RadioListTile<Gender>(
      secondary: const Icon(Icons.male),
      controlAffinity: ListTileControlAffinity.trailing,
      title: const Text("Male"),
      value: Gender.male, //assign value
      groupValue: _gender, //assign group
      onChanged: (Gender? value) {
        setState(() => _gender=value); //for change the state
      },
    ),
    RadioListTile<Gender>(
      secondary: const Icon(Icons.female),
      controlAffinity: ListTileControlAffinity.trailing,
      title: const Text("Female"),
      value: Gender.female,
      groupValue: _gender,
      onChanged: (Gender? value) {
        setState(() => _gender=value);
      },
    ),
  ],
)
)
```

Output :



Checkbox

The CheckboxListTile widget is similar to the Radio button, except it allows the user to choose multiple options from several selections. Usually, to save the selected items, we insert them into an empty set. Its key properties :

- value : whether this checkbox is checked or not.
- onChanged : called when the value of the checkbox should change. This is where we give the command to insert the selected items into the created set.
- title & subtitle : the primary content of the list tile.
- secondary : widget to display on the opposite side of the tile from the checkbox.
- controlAffinity : where to place the control relative to the text, it can be [ListTile.leading](#) or [ListTile.trailing](#).
- checkColor : color to use for the check icon when this checkbox is checked.

```
//declare check box content in list and selected item on set
List<String> languages = ["English", "French", "German"];
```

```

Set<String> selectedItems = {};
//create radio button inside of ListView
ListView(
  children: languages.map((language) {
    bool isChecked = selectedItems.contains(language);
    return CheckboxListTile(
      title: Text(language),
      controlAffinity: ListTileControlAffinity.trailing,
      value: isChecked,
      onChanged: (bool? value) {
        setState(() {
          if(value==true) {
            selectedItems.add(language);
          } else {
            selectedItems.remove(language);
          }
        });
      },
    );
  }).toList(),
)

```

Progress Bar

Progress bars are used to observe how much processing has been done. Flutter provides 2 kinds of progress bars, which are `LinearProgressIndicator` and `CircularProgressIndicator`. Basically, the syntax of both is the same, it's just that the shape is different so there are properties to adjust the height that are different. Its key properties :

- value : if non-null, the value of this progress indicator.
- valueColor : progress indicator's color as an animated value.
- `LinearProgressIndicator` :
 - minHeight : minimum height of the line used to draw the linear indicator.
- `CircularProgressIndicator` :
 - strokeWidth : width of the line used to draw the circle.

```

CircularProgressIndicator(
  strokeWidth: 10.0,
  value: _progress,
  valueColor: AlwaysStoppedAnimation<Color>(Colors.red),
  backgroundColor: Colors.orange,
)

```

Introduction to Flutter Layouts

Flutter brings the main concept that is entirely using widgets, so the interface of the layout is the widget itself. Flutter develops many widgets that can help in making interfaces. Developers can use layout widgets when composing other widgets.

Types of layout widgets

Layout widgets can be divided into two categories based on the number of child:

- widgets that can only accommodate one child
- widgets that can accommodate multiple children

Single-child layout widgets

A widget will only have one other widget as its child and each widget will have specific functionality. Single-child widgets are very useful for developing high quality widgets having single functionality such as buttons, labels, etc. There are very important single-child layout widgets provided by flutter :

Padding

Padding widget is used to set its child widgets to be given space from surrounding widgets. Padding uses EdgeInsetssets to set it.

```
Padding(  
  padding: EdgeInsets.symmetric(horizontal: 20.0),  
  child: Container(  
    height: 200, width: 200,  
    color: Colors.blueAccent,  
    child: const Text("Activate")  
  ),  
)
```

Align

Align widget is used to set the alignment of its child widget using the value of alignment property. It can also be replaced with the FractionalOffset class, so that it can set a more specific distance. For example: FractionalOffset(1.0, 0.0) represents the top right.

```
Align(  
  alignment: Alignment.topRight,  
  child: Container(  
    height: 200, width: 200,  
    color: Colors.blueAccent,  
    child: const Text("Activate")  
  ),  
)
```

SizeBox

SizeBox widget is a basic container defining its child's size, enabling precise layout control from snug placement to strategic spacing.

```
const SizedBox(  
  width: 200.0,  
  height: 300.0,  
  child: const Text("Deactivate"),  
)
```

Multi-child layout widgets

Multi-child layout widgets are widgets that have more than one child widget, and each child widget has a different layout. This widget is used to arrange child widgets in a flexible and complex way. There are very important multi-child layout widgets provided by flutter :

Column

Column widgets are widgets that organize children widgets vertically. This widget doesn't scroll (will occur an error if there are more children in the Column than available room). Its key properties :

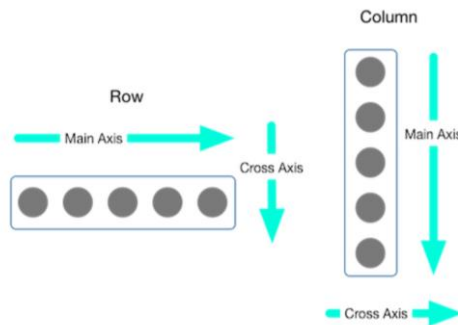
- children : contains a list of all widget that want to display vertically.
- crossAxisAlignment : how the children should be placed along the cross axis. Use the CrossAxisAlignment enum to set this, there are a constructor available :
 - o start : place the children with their start edge aligned
 - o end : place the children as close to the end of the cross axis as possible
 - o center : place the children so that their centers align with the middle of the cross axis.
 - o stretch : require the children to fill the cross axis.
 - o baseline : place the children along the cross axis such that their baselines match.
- mainAxisAlignment : how the children should be placed along the main axis. Use the MainAxisAlignment enum to set this, there are a constructor available :
 - o start : place the children as close to the start of the main axis as possible.
 - o end : place the children as close to the end of the main axis as possible.
 - o spaceBetween : place the free space evenly between the children.
 - o spaceAround : place the free space evenly between the children as well as half of that space before and after the first and last child.
 - o spaceEvenly : place the free space evenly between the children as well as before and after the first and last child.
- mainAxisAlignment : how much space should be occupied in the main axis. Use the MainAxisSize enum to set this, there are 2 values min and max.

Row

Row widget is literally similar to the column widget, except that the row widget organizes the child widgets horizontally. This widget also doesn't scroll (will occur an error if there are more children in the Row than available room). Its key properties :

- children : contains a list of all widget that want to display vertically.
- crossAxisAlignment : how the children should be placed along the cross axis. Use the CrossAxisAlignment enum to set this.
- mainAxisAlignment : how the children should be placed along the main axis. Use the MainAxisAlignment enum to set this.
- mainAxisAlignment : how much space should be occupied in the main axis. Use the MainAxisSize enum to set this, there are 2 values min and max.

This is a visualization of aligning children widgets for Column and Row. For row, the main axis runs horizontally and the cross axis runs vertically. For column, the main axis runs vertically and the cross axis runs horizontally.



```
Column(  
  crossAxisAlignment: CrossAxisAlignment.center,  
  children: [  
    const Text("Win Hotel"),  
    const SizedBox(height: 30),  
    Image.asset('assets/win_hotel.jpg'),  
    const SizedBox(height: 30),  
    Row(  
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
      children: const [  
        Icon(Icons.message),  
        Icon(Icons.phone),  
        Icon(Icons.email)  
      ],  
    ),  
  ],  
)
```

ListView

ListView is most often used for scrolling widgets. If you have many children in a column then an error can occur due to the screen not having enough space to accommodate them, so

instead of using Column, using ListView widget is the solution. There are 4 various ListView based on their constructor :

- common ListView : creates a scrollable, linear array of widgets from an explicit List.
- ListView.builder : creates a scrollable, linear array of widgets that are created on demand. Often used when retrieving data from the internet or API.
- ListView.custom : creates a scrollable, linear array of widgets with a custom child model.
- ListView.separated : creates a fixed-length scrollable linear array of list "items" separated by list item "separators".

Its key properties :

- childrenDelegate : delegate that provides the children for the ListView. Use the SliverChildDelegate class to set this.
- controller : object that can be used to control the position to which this scroll view is scrolled.
- shrinkWrap : take up only as much vertical space as needed to display its children.
- itemBuilder : callback function is called for each item in the list. It takes two arguments: BuildContext and int index. You're responsible for returning the widget for the item at the given index.
- itemCount : defines the number of items in the list. It specifies how many times the itemBuilder function will be called.

```
//declare contents data in list
final List<String> _data = ['item 1', 'item 2', 'item 3', 'item
4'];
ListView.builder(
  itemCount: _data.length,
  itemBuilder: (context, index) {
    return Container(
      height: 50.0,
      color: Colors.primaryes[index % Colors.primaryes.length],
      child: Center(
        child: Text(_data[index]),
      ),
    ),
  ),
);
```

GridView

GridView is also the same as ListView which is used for scrolling widgets. Provides us a widget called GridView which is used to create a scrollable grid of widgets. There are 5 various GridView based on their constructor :

- common GridView : creates a scrollable, 2D array of widgets with a custom SliverGridDelegate.

- `GridView.builder` : creates a scrollable, 2D array of widgets that are created on demand. Often used when retrieving data from the internet or API.
- `GridView.count` : creates a scrollable, 2D array of widgets with a fixed number of tiles in the cross axis.
- `GridView.custom` : creates a scrollable, 2D array of widgets with both a custom `SliverGridDelegate` and a custom `SliverChildDelegate`.
- `GridView.extent` : creates a scrollable, 2D array of widgets with tiles that each have a maximum cross-axis extent.

Its key properties :

- `childrenDelegate` : delegate that provides the children for the `GridView`. Use the `SliverChildDelegate` class to set this.
- `controller` : object that can be used to control the position to which this scroll view is scrolled.
- `gridDelegate` : delegate that controls the layout of the children within the `GridView`. Use the `SliverGridDelegate` class to set this.
- `shrinkWrap` : take up only as much horizontal space as needed to display its children.
- `itemBuilder` : callback function is called for each item in the grid. It takes two arguments: `BuildContext` and `int index`. You're responsible for returning the widget for the item at the given index.
- `itemCount` : defines the number of items in the grid. It specifies how many times the `itemBuilder` function will be called.
- `crossAxisCount` : defines the number of columns we want to have in a grid.
- `maxCrossAxisExtent` : defines maximum item width.

```
//declare contents data in list
final List<String> _data = ['item 1', 'long item 2', 'item 3',
'item 4', 'item 5', 'item 6'];
GridView.builder(
  gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(
    maxCrossAxisExtent: 150.0, // Max width of each item
    mainAxisSpacing: 10.0, // Spacing between rows
    crossAxisSpacing: 10.0, // Spacing between columns
  ),
  itemCount: _data.length,
  itemBuilder: (context, index) {
    return Container(
      color: Colors.grey[200],
      child: Center(
        child: Text(_data[index]),
      ),
    );
  },
);
```

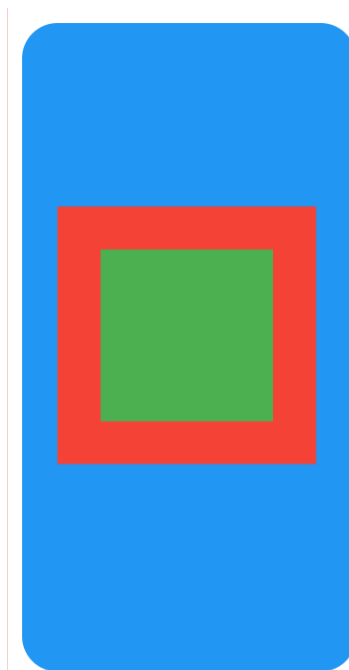
Stack

Stack widgets are used to overlap widgets or layer widgets. The positioned children are then placed relative to the stack according to its top, right, bottom, and left properties. The stack paints its children in order with the first child being at the bottom. If you want to change the order of its children, you can rebuild the stack with the children in a new order. Its key properties :

- children : contains a list of all widget that want to display.
- alignment : align the non-positioned and partially-positioned children in the stack. Use the Alignment enum to set this.
- fit : size the non-positioned children in the stack.
- clipBehavior : content will be clipped (or not) according to this option.

```
Stack(  
  alignment: Alignment.center,  
  children: [  
    Container(  
      color: Colors.blue,  
    ),  
    Container(  
      width: 300, height: 300,  
      color: Colors.red,  
    ),  
    Container(  
      width: 200, height: 200,  
      color: Colors.green,  
    ),  
  ],  
)
```

Output :



ScrollView

If there are not enough widgets on one page but cannot be separated on another page, then a scroll view is needed, so that it can place other widgets below. Flutter has many widgets that support scrolling. In addition to GridView and ListView there are also various other ScrollViews :

- SingleChildScrollView : automatically scrolls its child when necessary.
- CustomScrollView : directly utilize Slivers to create scrolling effects such as lists, grids, and expanding headers.
- NestedScrollView : scrolling view inside of which can be nested other scrolling positions being intrinsically linked.

Here we only need to wrap the widgets that we want to scroll using one of these scrollview widgets. Its key properties :

- child : contains a widget that scrolls.
- controller : object that can be used to control the position to which this scroll view is scrolled.
- physics : how the scroll view should respond to user input. Use ScrollPhysics class to set this.
- reverse : whether the scroll view scrolls in the reading direction.
- scrollDirection : axis along which the scroll view's offset increases.

```
SingleChildScrollView(  
  child: Column(  
    children: [  
      ListView.builder(  
        physics: const NeverScrollableScrollPhysics(),  
        shrinkWrap: true,  
        itemCount: 50,  
        itemBuilder: (context, index) {  
          return Center(  
            child: Container(  
              height: 50, width: 200,  
              color: Colors.red,  
            ),  
          );  
        },  
      ),  
    ],  
  ),  
)
```

In addition to dealing with layout configuration, we can also adjust the size of a widget according to the device used without the need to initiate what device is being used first.

```
Container(  
  height: MediaQuery.sizeOf(context).height/3,  
  width: MediaQuery.sizeOf(context).width,
```

```
color: Colors.blueGrey,  
child: const Text("Media Query"),  
)
```

Introduction to Flutter Gestures

Developing a program, especially a mobile, website or desktop application, will always expect interaction from users to operate the application. As a framework for creating a multi-platform application, flutter also provides a way to handle gestures from users. Some of the widely used gestures are mentioned here :

- **Tap** : a simple gesture used to select or perform an action. This is done by briefly touching the screen with one finger.
- **Double Tap** : a gesture similar to tap, but performed twice quickly. This is often used to zoom in or out on objects.
- **Drag** : a gesture used to move an object. This is done by touching the screen and then dragging your finger in the desired direction.
- **Flick** : a gesture used to scroll the screen up or down. This is done by touching the screen and then swiping your finger in the desired direction.
- **Pinch** : a gesture used to zoom in or out on an object by bringing two fingers together or apart.
- **Spread/Zoom** : a gesture similar to pinch, but done by moving two fingers separately or together.
- **Panning** : a gesture used to move an object freely. This is done by touching the screen and then dragging your finger in the desired direction without releasing the screen.

Flutter offers a special widget to handle gestures from users, the GestureDetector widget. This widget works by recognizing gestures that have callbacks set and responding accordingly. If a gesture is to be disabled, a null value will be passed to the callback.

There are common gestures captured by the GestureDetector widget and their corresponding events :

- Tap
 - onTapDown : triggered when user makes contact with screen, might be a tap.
 - onTapUp : triggered when the user stops making contact with the screen.
 - onTap : triggered when the user briefly touches the screen.
 - onTapCancel : triggered when the event that fired onTapDown is not a tap.
- Double tap
 - onDoubleTap : triggered when user taps the screen at the same location twice in quick succession.

- Long press
 - `onLongPress` : triggered when a long press has been detected.
- Vertical drag
 - `onVerticalDragStart` : triggered when user has made contact with screen and began to move vertically.
 - `onVerticalDragUpdate` : triggered when contact that is moving vertically has moved in a vertical direction once again.
 - `onVerticalDragEnd` : triggered when the end of a vertical drag has been detected.
- Horizontal drag
 - `onHorizontalDragStart` : triggered when user has made contact with screen and began to move horizontally.
 - `onHorizontalDragUpdate` : triggered when contact that is moving horizontally has moved in a horizontal direction once again.
 - `onHorizontalDragEnd` : triggered when the end of a horizontal drag has been detected.
- Pan
 - `onPanDragStart` : triggered when touching the screen and may start moving horizontally or vertically. This callback crashes if `onHorizontalDragStart` or `onVerticalDragStart` is set.
 - `onPanDragUpdate` : triggered when touching the screen and moves the vertical or horizontal direction. This callback crashes if `onHorizontalDragUpdate` or `onVerticalDragUpdate` is set.
 - `onPanDragEnd` : triggered when the previous screen contact is no longer in contact with the screen and moves at a certain speed when it stops contacting the screen. This callback crashes if `onHorizontalDragEnd` or `onVerticalDragEnd` is set.

This is an example of using the GestureDetector widget :

```
GestureDetector(
  onTap: () {
    print("You've tapped");
  },
  onDoubleTap: () {
    print("You've double tapped");
  },
  onLongPress: () {
    print("You've long pressed");
  },
  child: Container(
    width: MediaQuery.sizeOf(context).width,
```

```

    height: 200,
    color: Colors.yellow,
  ),
)

```

Flutter also offers a low-level gesture detection mechanism through Listener widget. It will detect all user interactions and dispatches these events :

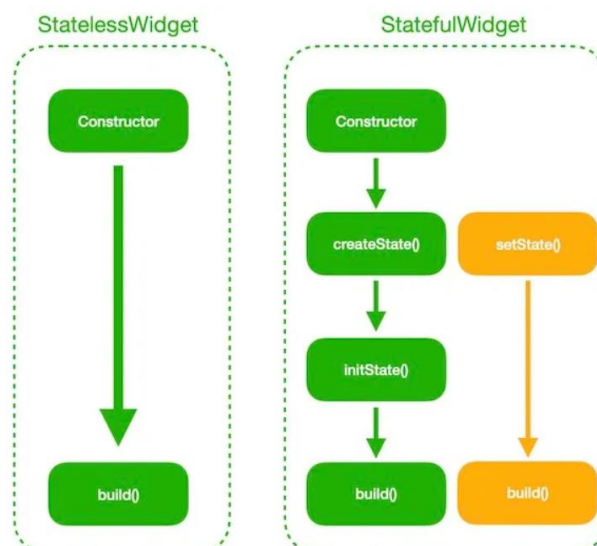
- PointerDownEvent
- PointerMoveEvent
- PointerUpEvent
- PointerCancelEvent

There are also small set of widgets to do specific or advanced gestures.

- Dismissible : support flick gesture to close the widget.
- Draggable : supports dragging gestures to move the widget.
- LongPressDraggable : supports dragging to move the widget, if its parent widget can also be dragged.
- DragTarget : accepts draggable widgets.
- IgnorePointer : hides the widget and its children from the gesture detection process.
- AbsorbPointer : stops the motion detection process itself so that the overlapping widget cannot also participate in the motion detection process and hence, no event is raised.
- Scrollable : supports scrolling of the content available inside the widget

Introduction to Flutter State Management

We certainly know that in flutter two kinds of large groups of widgets are Stateless and Stateful widgets. Both are distinguished by different application lifecycle processes. Here is a visualization of the stateless and stateful lifecycle.



From visualization, it can be taken that stateful has an internal state that is used for internal updates or parent changes.

Before we move on to state management, we want to review the `initState()` method first. The `initState()` method is basically the entry point for `StatefulWidget`. It is only called once and is used in general to initialize the predefined variables of the `StatefulWidget`. The `initState()` method is widely overridden because as mentioned earlier, it is only called once during its lifetime.

```
@override
void initState() {
  print("initState Called");
  super.initState();
}
@override
Widget build(BuildContext context) {...}
```

When running the application for the first time, expect to see the `initState` method triggered, i.e. this method is called first and after that, the control goes into the Build Context.

```
I/flutter (11535): initState Called
I/flutter (11535): Build method called
```

State management can be divided into two categories based on the duration a particular state lasts in an application.

- **Ephemeral state**

Lasts for a few seconds like the current state of an animation or a single page like the current product rating. Flutter supports it through `StatefulWidget`. `StatefulWidget` provides an option for widgets to create state. `State<T>` (where T is the inherited widget) when the widget is created for the first time through `createState` method and then `setState` method to change the state whenever required. The state change will be done through gestures.

For the examples we want to create some `tabBox` class :

The `tabBox` class :

- Manage state for `TapBoxx`.
- Defines the `active` boolean which determines the box's current color.
- Defines the `handleTap()` function, which updates `active` when the box is tapped and calls the `setState()` function to update the UI.
- Implements all interactive behavior for the widget.

The `TapBoxx` class manages its own state, so it overrides `createState()` to create a `State` object. The framework calls `createState()` when it wants to build the widget. In this class `createState()` returns an instance of `_TapBoxxState`.

```
class TapBoxx extends StatefulWidget {
```

```

const TapBoxx({super.key});

@override
State<TapBoxx> createState() => _TapBoxxState();
}

```

The `_TapBoxxState` class stores the mutable data that can change over the lifetime of the widget.

```

class _TapBoxxState extends State<TapBoxx> {
  bool active = false;

  // ...
}

```

The class also defines a `build()` method, which creates a `GestureDetector` and `Container` for its child. `onTap` property that defines the callback functions (`handleTap`) for handling a tap.

```

class _TapBoxxState extends State<TapBoxx> {
  // ...
  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: handleTap,
      child: Container(
        width: 250, height: 250,
        decoration: BoxDecoration(
          color: active ? Colors.green : Colors.grey
        ),
        alignment: Alignment.center,
        child: Text(
          active ? "Active" : "Inactive",
          style: const TextStyle(
            fontSize: 32, color: Colors.white
          )
        ),
      ),
    );
  }
}

```

The `handleTap()` method, which is called when the `Container` is tapped, calls `setState()`. Calling `setState()` is critical, because this tells the framework that the widget's state has changed and the widget should be redrawn.

```

void handleTap() {
  setState(() {
    active = !active;
  });
}

```

This is complete code of the `TapBoxx` class.


```

import 'package:flutter/material.dart';

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

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: TapBoxx(),
    );
  }
}

class TapBoxx extends StatefulWidget {
  const TapBoxx({super.key});

  @override
  State<TapBoxx> createState() => _TapBoxxState();
}

class _TapBoxxState extends State<TapBoxx> {
  bool active = false;
  void handleTap() {
    setState(() {
      active = !active;
    });
  }
  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: handleTap,
      child: Container(
        width: 250, height: 250,
        decoration: BoxDecoration(
          color: active ? Colors.green : Colors.grey
        ),
        alignment: Alignment.center,
        child: Text(
          active ? "Active" : "Inactive",
          style: const TextStyle(
            fontSize: 32, color: Colors.white
          )
        ),
      ),
    );
  }
}

```

- **App state**

Persists for the entire app like logged in user details, cart information, etc. For managing app state, you'll want to research your options. For example, you can use provider, BLoC (Business Logic Component) Pattern with RxDart, GetX, Riverpod, etc.

Introduction to Flutter Navigation and Routing

In all functional applications that must have many pages to run the workflow, there is a way to switch from one page to another. This method is called Routing. Flutter offers a basic routing class to support workflow in applications.

MaterialPageRoute

The MaterialPageRoute class is used to render the UI by replacing the entire page with a platform-appropriate transition. By default, when a modal route is replaced by another, the previous route remains in memory.

```
MaterialPageRoute(builder: (context) => OtherPage())
```

The most important property is the builder, as it will receive the functionality to build its content by supplying the current application context.

Navigation.push

Navigation.push is used to navigate to a new page using the MaterialPageRoute class.

```
Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => OtherPage()),  
);
```

Navigation.pop

Navigation.pop is used to navigate to the previous page. So if the current page is the very first page of the application, it is not recommended to use Navigate.pop because it will give an error because it does not have the previous page.

```
Navigator.pop(  
  context,  
  MaterialPageRoute(builder: (context) => OtherPage()),  
);
```

Since we are still on the topic of page navigation, we would like to explain two more widgets that are closely related to page navigation.

Dialog

Dialog Widgets in flutter appear on the screen to get confirmation regarding important/non-cancellable tasks or inform users about important information about the app. There are two types of flutter dialog :

- AlertDialog : used to get confirmation from users for any critical action they have asked to perform.
- SimpleDialog : used to display a list of options that users can select.

These two flutter dialogs will be wrapped with the showDialog function which requires context and builder arguments. context is taken from the BuildContext in the application while a builder is needed to return the Dialog widget. In addition, the barrierDismissible argument is used to indicate whether tapping on the barrier will dismiss the dialog. It is true by default and can not be null.

```
showDialog(  
  context: context,  
  builder: (context) => AlertDialog(  
    title: const Text("Next Step"),  
    content: const Text("Are you sure you want to continue?"),  
    actions: [  
      TextButton(  
        onPressed: () => Navigator.pop(context),  
        child: const Text("Cancel"),  
      ),  
      TextButton(  
        onPressed: () {  
          print("Continue!");  
          Navigator.pop(context);  
        },  
        child: const Text("Yes"),  
      ),  
    ],  
  ),  
)
```

SnackBar

SnackBar widget is a light message with an optional action that is briefly displayed at the bottom of the screen. SnackBar widget should be inside the Scaffold widget because it ensures that important widgets do not overlap. Its key properties :

content : the primary content of the snack bar.

- margin & padding : create empty space to surround the snack bar, also amount of padding to apply to snack bar's content and optional action.
- width : set width of the snack bar.
- action : action that the user can take based on the snack bar.
- duration : amount of time the snack bar should be displayed.
- showCloseIcon : whether to include a "close" icon widget.

```
Scaffold(  
  body: Center(  
    child: ElevatedButton(  
      onPressed: () {  
        final snackBar = SnackBar(  
          content: const Text("SnackBar shows"),  
          action: SnackBarAction(  
            label: "Done",  
            onPressed: () {},  
          ),  
        ),  
      },  
    ),  
  ),  
)
```

```
        ),  
      );  
      ScaffoldMessenger.of(context).showSnackBar(snackBar);  
    },  
    child: const Text('Show SnackBar'),  
  ),  
),  
)
```