|  |
| --- |
|  |
| Mastering Flutter App Development |
| A Step-by-Step Guide for Beginners |
|  |
| **Nikhil Kumar Mishra** |
| **3/14/2024** |

|  |
| --- |
|  |

Table of Contents

[Dart Tutorial 5](#_Toc161163686)

[Introduction and Basics 5](#_Toc161163687)

[Introduction to Dart 5](#_Toc161163688)

[Install Dart 7](#_Toc161163689)

[Basic Dart Program 9](#_Toc161163690)

[Variables in Dart 11](#_Toc161163691)

[Data Types in Dart 14](#_Toc161163692)

[Comments in Dart 23](#_Toc161163693)

[User Input in Dart 32](#_Toc161163694)

[String in Dart 33](#_Toc161163695)

[Conditions and Loops 39](#_Toc161163696)

[Conditions in Dart 39](#_Toc161163697)

[Switch Case in Dart 44](#_Toc161163698)

[Ternary Operator in Dart 49](#_Toc161163699)

[For Loop in Dart 52](#_Toc161163700)

[For Each Loop in Dart 54](#_Toc161163701)

[While Loop in Dart 57](#_Toc161163702)

[Do While Loop in Dart 59](#_Toc161163703)

[Break and Continue in Dart 61](#_Toc161163704)

[Exception Handling in Dart 65](#_Toc161163705)

[Exception In Dart 65](#_Toc161163706)

[Try & Catch In Dart 66](#_Toc161163707)

[Finally In Dart 67](#_Toc161163708)

[Throwing An Exception 68](#_Toc161163709)

[Why Is Exception Handling Needed? 68](#_Toc161163710)

[How To Create Custom Exception In Dart 69](#_Toc161163711)

[Functions in Dart 73](#_Toc161163712)

[Functions in Dart 73](#_Toc161163713)

[Types of Functions in Dart 77](#_Toc161163714)

[Function Parameter 83](#_Toc161163715)

[Arrow Function in Dart 88](#_Toc161163716)

[Math in Dart 91](#_Toc161163717)

[Collections in Dart 94](#_Toc161163718)

[List in Dart 95](#_Toc161163719)

[Set in Dart 104](#_Toc161163720)

[Map in Dart 110](#_Toc161163721)

[Where in Dart 117](#_Toc161163722)

[File Handling in Dart 119](#_Toc161163723)

[Read File in Dart 119](#_Toc161163724)

[Write File in Dart 122](#_Toc161163725)

[Delete File in Dart 125](#_Toc161163726)

[OOP in Dart 126](#_Toc161163727)

[OOP in Dart 126](#_Toc161163728)

[Class in Dart 127](#_Toc161163729)

[Object in Dart 129](#_Toc161163730)

[Constructor in Dart 130](#_Toc161163731)

[Default Constructor in Dart 138](#_Toc161163732)

[Parameterized Constructor in Dart 140](#_Toc161163733)

[Named Constructor in Dart 143](#_Toc161163734)

[Constant Constructor in Dart 147](#_Toc161163735)

[Encapsulation in Dart 149](#_Toc161163736)

[Getter in Dart 155](#_Toc161163737)

[Setter in Dart 160](#_Toc161163738)

[Inheritance in Dart 163](#_Toc161163739)

[Super in Dart 177](#_Toc161163740)

[Polymorphism in Dart 180](#_Toc161163741)

[Static in Dart 184](#_Toc161163742)

[Enum in Dart 189](#_Toc161163743)

[Abstract Class 193](#_Toc161163744)

[Interface in Dart 198](#_Toc161163745)

[Mixin in Dart 206](#_Toc161163746)

[Factory Constructor in Dart 210](#_Toc161163747)

[Null Safety in Dart 219](#_Toc161163748)

[Advantage Of Null Safety 219](#_Toc161163749)

[Type Promotion in Dart 226](#_Toc161163750)

[Late Keyword in Dart 228](#_Toc161163751)

[Null Safety Exercise 233](#_Toc161163752)

[Asynchronous Programming 238](#_Toc161163753)

[Synchronous Programming 238](#_Toc161163754)

[Asynchronous Programming 239](#_Toc161163755)

[Why We Need Asynchronous 239](#_Toc161163756)

[Future In Dart 240](#_Toc161163757)

[Async and Await In Dart 242](#_Toc161163758)

[Streams In Dart 245](#_Toc161163759)

[Final Vs Const 255](#_Toc161163760)

[Const In Dart 255](#_Toc161163761)

[Final In Dart 256](#_Toc161163762)

[Datetime In Dart 257](#_Toc161163763)

[Flutter Tutorial 261](#_Toc161163764)

[Introduction and Setting up the Environment 261](#_Toc161163765)

[Creating a Sample App 263](#_Toc161163766)

[Widgets in Flutter 264](#_Toc161163767)

[Understanding the Sample Code 264](#_Toc161163768)

[Types of Widgets in Flutter 265](#_Toc161163769)

[Stateless Widgets 265](#_Toc161163770)

[Stateful Widgets 266](#_Toc161163771)

[List of Widgets 270](#_Toc161163772)

[Responsive Layouts in Flutter 309](#_Toc161163773)

[Creating Custom Widgets 312](#_Toc161163774)

[Custom Themes and Animations 314](#_Toc161163775)

[Using Custom Theme 314](#_Toc161163776)

[Using Animation 317](#_Toc161163777)

[Flutter Navigation – How to Add Stack, Tab, and Drawer Navigators to Your Apps 322](#_Toc161163778)

[Types of Navigation 322](#_Toc161163779)

[How to Build the Stack Navigation 323](#_Toc161163780)

[How to Build the Tab Navigation 328](#_Toc161163781)

[How to Build the Drawer Navigation 332](#_Toc161163782)

[Dependency Management 337](#_Toc161163783)

[Data persistence in Flutter 338](#_Toc161163784)

[Using Shared Preferences 338](#_Toc161163785)

[Using SQLite 341](#_Toc161163786)

[Networking in Flutter 353](#_Toc161163787)

[Study HTTP requests and how to make them using the http package 353](#_Toc161163788)

[Learn how to parse JSON data in Dart 353](#_Toc161163789)

[Use the http package to make GET and POST requests to a web API 354](#_Toc161163790)

[Parse the JSON data returned by the API into Dart objects 354](#_Toc161163791)

[Resources 355](#_Toc161163792)

[Using JSON Server 355](#_Toc161163793)

[Guide 355](#_Toc161163794)

[Complete Example 357](#_Toc161163795)

[State Management in Flutter 360](#_Toc161163796)

[Using GetX 360](#_Toc161163797)

[Using BLOC 380](#_Toc161163798)

[Using Flutter Plugins 393](#_Toc161163799)

[Take a picture using the camera 393](#_Toc161163800)

[Play and pause a video 401](#_Toc161163801)

[Building Flutter APK 408](#_Toc161163802)

# Dart Tutorial

## Introduction and Basics

### Introduction to Dart

**Dart**

* Dart is a client-optimized, object-oriented, modern programming language to build apps fast for many platforms like android, iOS, web, desktop, etc.
  + Client optimized means optimized for crafting a beautiful user interface and high-quality experiences.
  + Google developed Dart as a programming language.
    - A solid understanding of Dart is necessary to develop high-quality apps with flutter.

**Dart Features**

* Free and open-source.
  + Object-oriented programming language.
* Used to develop android, iOS, web, and desktop apps fast.
  + - Can compile to either native code or javascript.
      * Offers modern programming features like null safety and asynchronous programming.
* You can even use Dart for servers and backend.

**Difference Between Dart & Flutter**

* **Dart** is a client optimized, object-oriented programming language. It is popular nowadays because of flutter. It is difficult to build complete apps only using Dart because you have to manage many things yourself.
* **Flutter** is a framework that uses dart programming language. With the help of flutter, you can build apps for android, iOS, web, desktop, etc. The framework contains ready-made tools to make apps faster.

**Which Is The Best Code Editor For Dart Programming**

The best code editor is VS Code if you want to run the dart program from a computer or laptop. You can download the dart extension from VS Code and start coding. You will learn more about [installing dart](https://dart-tutorial.com/introduction-and-basics/dart-install/) in the next topic. You can also use [DartPad](https://dartpad.dev/) to run simple dart programs without installing anything.

**Dart History**

* Google developed Dart in 2011 as an alternative to javascript.
  + Dart 1.0 was released on November 14, 2013.
    - Dart 2.0 was released in August 2018.
      * Dart 3.0 was released in May 2023.
    - Dart gained popularity in recent days because of flutter.

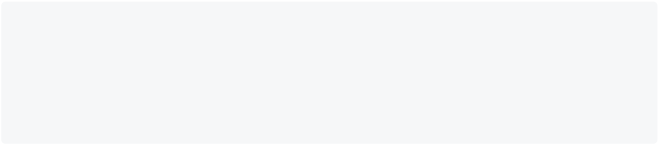
**Basic Programming Terms**

Important words that you often hear while learning programming languages.

**Statements:** A statement is a command that tells a computer to do something. In Dart, you can end most statements with a semicolon **;**.

**Expressions:** An Expression is a value or something that can be calculated as a value. The expression can be numbers, text, or some other type. For E.g.

1. 52



1. 5+5
2. 'Hello World.'
3. num

**Keywords:** Keywords are reserved words that give special meaning to the dart compiler. For E.g. **int**, **if**, **var**, **String**, **const**, etc.

**Identifiers:** Identifiers are names created by the programmer to define variables, functions, classes, etc. Identifiers shouldn’t be keywords and must have a unique name. For E.g. **int age =19;**, here age is an identifier. You will learn more about identifiers later in this course.

**High-Level Programming Language:** High-Level Programming Language is easy to learn, user-friendly, and uses English-like-sentence. For E.g. dart,c,java,etc.

**Low-Level Programming Language:** Low-level programming language is hard to learn, non-user friendly, and deals with computer hardware components, e.g.,

machine and assembly language. Info

Note: Low-level languages are faster than high-level but hard to understand and debug.

**Compiler:** A compiler is a computer program that translates the high-level programming language into machine-level language.

**Syntax:** The Syntax is a programming language’s pattern or rules that give the concept to code.

**Key Points**

* Dart is a free and open-source programming language. You don’t need to pay any money to run dart programs.
  + Dart is a platform-independent language and supports almost every operating system such as windows, mac, and Linux.
  + Dart is an object-oriented programming language and supports all oops features such as encapsulation, inheritance, polymorphism, interface, etc.
    - Dart comes with a **dart2js** compiler which translates dart code to javascript code that runs on all modern browsers.
      * Dart is a programming language used by flutter, the world’s most popular framework for building apps.

### Install Dart

**Dart Installation**

There are multiple ways to install a dart on your system. You can install Dart on **Windows, Mac, and Linux** or run it from the browser.

**Requirements**

* **Dart SDK**,
  + **VS code or other editors** like Intellij [We will use VS Code here].

**Dart Windows Installation**

Follow the below instructions to install a dart on the windows operating system. **Steps:**

* Download Dart SDK from [here](https://dart.dev/get-dart/archive).
  + Copy **dart-sdk** folder to your C drive.
* Add **C:\dart-sdk\bin** to your environment variable.
  + - Open the command prompt and type **dart --version** to check it.
* Install [VS Code](https://code.visualstudio.com/download) and Add Dart Extension.

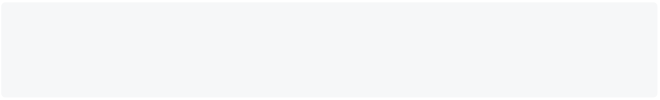
**Note**: Dart SDK provides the tools to compile and run dart program. **Dart Mac Installation**

* Install Homebrew From [here](https://brew.sh/).
  + Type brew tap dart-lang/dart in the terminal.
* Type brew install dart in the terminal.

**Homebrew Install Command**

Copy and paste this command on your terminal to install Homebrew.

/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"



To set the homebrew path, copy and paste this command on your terminal.

export PATH=/opt/homebrew/bin:$PATH



**Check Dart Installation**

Open your command prompt and type **dart --version**. The dart is successfully installed on your system if it gives you a version code. If not, watch the video above.

**Some Useful Commands**

|  |  |
| --- | --- |
| **Command** | **Description** |
| dart --help | Show all available commands. |
| dart filename.dart | Run the dart file. |
| dart create | Create a dart project. |
| dart fix | Update dart project to new syntax. |
| **Command** | **Description** |
| dart compile exe bin/dart.dart | Compile dart code. |
| dart compile js bin/dart.dart | Compile dart to javascript. You can run this file with Node.js. |

**Run Dart On Web**

You can run the dart program on your browser without installing any software. Dartpad is a web tool to write and run your dart code.

* [Run Dart Programming on Web](https://dartpad.dev/)

**Install Dart Official Link**

[Install Dart Official Link](https://dart.dev/get-dart)



**Can You Run Dart From Mobile?**

Yes, you can use [DartPad](https://dartpad.dev/) to run simple dart programs from your phone without installing any software. For bigger projects, using DartPad is not recommended.

### Basic Dart Program

This is a simple dart program that prints **Hello World** on screen. Most programmers write the Hello World program as their first program.

void main() {



print("Hello World!"); }

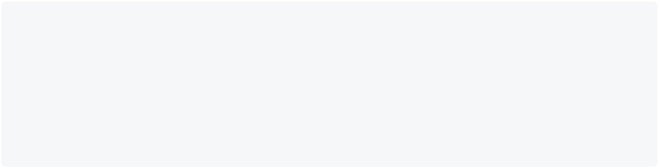
[Run Online](https://dartpad.dev/?id=2a27a92364b348df4953f880518af7a3)

**Basic Dart Program Explained**

* void main() is the starting point where the execution of your program begins.
  + Every program starts with a main function.
    - The curly braces {} represent the beginning and the ending of a block of code.
* print(“Hello World!”); prints Hello World! on screen.
  + Each code statement must end with a semicolon.

**Basic Dart Program For Printing Name**

void main()



{

var name = "John"; print(name);

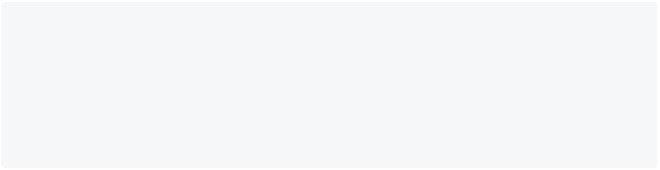
}

[Run Online](https://dartpad.dev/?id=52502e861f491ed1b28a6da73b53efba)

**Dart Program To Join One Or More Variables**

Here **$variableName** is used to join variables. This joining process in dart is called string interpolation.

void main(){



var firstName = "John";

var lastName = "Doe";

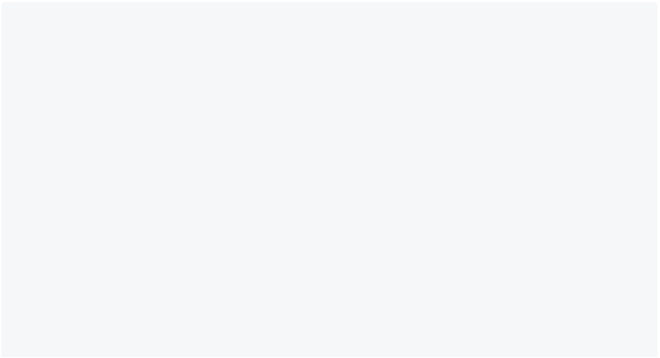
print("Full name is $firstName $lastName"); }

[Run Online](https://dartpad.dev/?id=caa95bbad26818e23a292938ceba4d3a)

**Dart Program For Basic Calculation**

Performing addition, subtraction, multiplication, and division in dart.

void main() {



int num1 = 10; //declaring number1 int num2 = 3; //declaring number2

// Calculation

int sum = num1 + num2;

int diff = num1 - num2;

int mul = num1 \* num2;

double div = num1 / num2; // It is double because it outputs number with decimal.

// displaying the output print("The sum is $sum"); print("The diff is $diff"); print("The mul is $mul"); print("The div is $div");



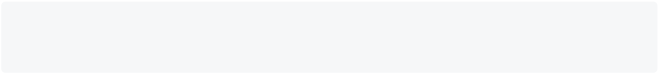
}

[Run Online](https://dartpad.dev/?id=5de9dc2148637e9d9ecbd98632b457e6)

**Create Full Dart Project**

It’s nice to work on a single file, but if your project gets bigger, you need to manage configurations, packages, and assets files. So creating a dart project will help you to manage this all.

dart create <project\_name>



This will create a simple dart project with some ready-made code. **Steps To Create Dart Project**

* Open folder location on command prompt/terminal.
  + Type dart create project\_name (For E.g. dart create first\_app)
* Type cd first\_app
  + - Type code . to open project with visual studio code
      * To check the main dart file go to **bin/first\_app.dart** and edit your code.

**Run Dart Project**

First, open the project location on the command/terminal and run the project with this command.

dart run



**Convert Dart Code To Javascript**

|  |  |
| --- | --- |
| **Command** | **Description** |
| dart compile js filename.dart | Compile dart to javascript. You can run this file with Node.js. |

### Variables in Dart

**Variables**

Variables are containers used to store value in the program. There are different types of variables where you can keep different kinds of values. Here is an example of creating a variable and initializing it.

// here variable name contains value John. var name = "John";



**Variable Types**

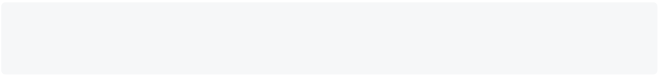
They are called data types. We will learn more about data types later in this dart tutorial.

* **String**: For storing text value. E.g. “John” [Must be in quotes]
  + **int**: For storing integer value. E.g. 10, -10, 8555 [Decimal is not included]
    - **double**: For storing floating point values. E.g. 10.0, -10.2, 85.698 [Decimal is included]
* **num**: For storing any type of number. E.g. 10, 20.2, -20 [both int and double]
  + **bool**: For storing true or false. E.g. true, false [Only stores true or false values]
    - **var**: For storing any value. E.g. ‘Bimal’, 12, ‘z’, true

**Syntax**

This is syntax for creating a variable in dart.

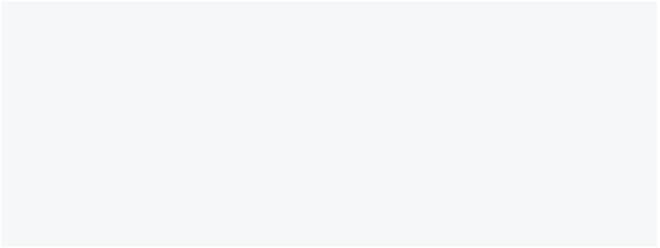
type variableName = value;



**Example 1: Using Variables In Dart**

In this example, you will learn how to declare variables and print their values.

void main() {



// declaring variables

String name = "John";

String address = "USA";

num age = 20; // used to store any types of numbers num height = 5.9;

bool isMarried = false;

// printing variables value

print("Name is $name"); print("Address is $address"); print("Age is $age");



print("Height is $height"); print("Married Status is $isMarried");

}

[Run Online](https://dartpad.dev/?id=e050476fb9a1afa27732106f52d38d68)

**Note**: Always use the descriptive variable name. Don’t use a variable name like a, b, c because this will make your code more complex.

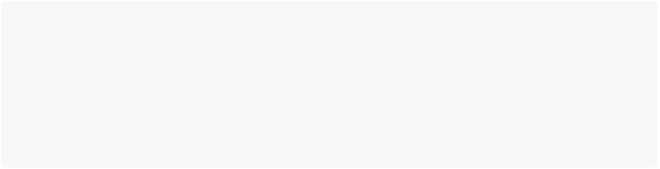
**Rules For Creating Variables In Dart**

* Variable names are case sensitive, i.e., a and A are different.
  + A variable name can consist of letters and alphabets.
* A variable name cannot start with a number.
  + - Keywords are not allowed to be used as a variable name.
      * Blank spaces are not allowed in a variable name.
        + Special characters are not allowed except for the underscore (\_) and the dollar ($) sign.

**Dart Constant**

Constant is the type of variable whose value never changes. In programming, changeable values are **mutable** and unchangeable values are **immutable**. Sometimes, you don’t need to change the value once declared. Like the value of PI=3.14, it never changes. To create a constant in Dart, you can use the const keyword.

void main(){



const pi = 3.14;

pi = 4.23; // not possible

print("Value of PI is $pi"); }

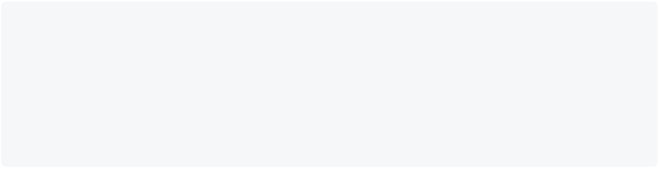
[Run Online](https://dartpad.dev/?id=7fd7914f845a1c8ec59b89e00ead5916)

**Naming Convention For Variables In Dart**

It is a good habit to follow the naming convention. In Dart Variables, the variable name should start with lower-case, and every second word’s first letter will be upper-case like num1, fullName, isMarried, etc. Technically, this naming convention is called **lowerCamelCase**.

**Naming Convention Example**

// Not standard way



var fullname = "John Doe"; // Standard way

var fullName = "John Doe"; const pi = 3.14;

### Data Types in Dart

**Data types** help you to categorize all the different types of data you use in your code. **For e.g. numbers, texts, symbols, etc**. The data type specifies what type of value will be stored by the variable. Each variable has its data type. Dart supports the following built-in data types :

1. Numbers
2. Strings
3. Booleans
4. Lists
5. Maps
6. Sets
7. Runes
8. Null

**Built-In Types**

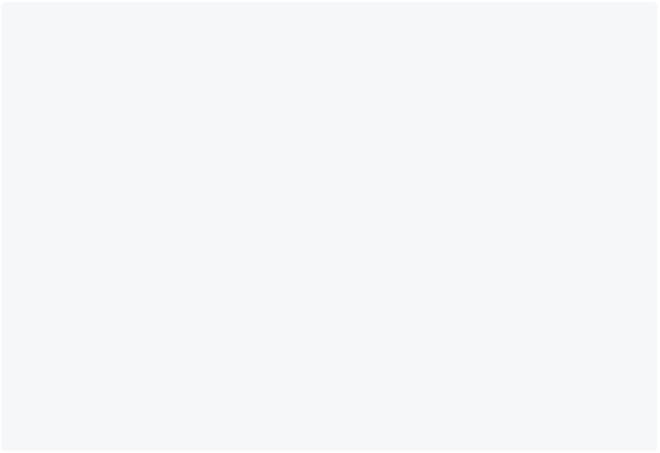
In Dart language, there is the type of values that can be represented and manipulated. The data type classification is as given below:

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Keyword** | **Description** |
| Numbers | int, double, num | It represents numeric values |
| Strings | String | It represents a sequence of characters |
| **Data Type** | **Keyword** | **Description** |
| Booleans | bool | It represents Boolean values true and false |
| Lists | List | It is an ordered group of items |
| Maps | Map | It represents a set of values as key-value pairs |
| Sets | Set | It is an unordered list of unique values of same types |
| Runes | runes | It represents Unicode values of String |
| Null | null | It represents null value |

**Numbers**

When you need to store numeric value on dart, you can use either int or double. Both int and double are subtypes of **num**. You can use num to store both int or double value.

void main() {



// Declaring Variables

int num1 = 100; // without decimal point. double num2 = 130.2; // with decimal point. num num3 = 50;

num num4 = 50.4;

// For Sum

num sum = num1 + num2 + num3 + num4;

// Printing Info print("Num 1 is $num1"); print("Num 2 is $num2"); print("Num 3 is $num3"); print("Num 4 is $num4"); print("Sum is $sum");

}

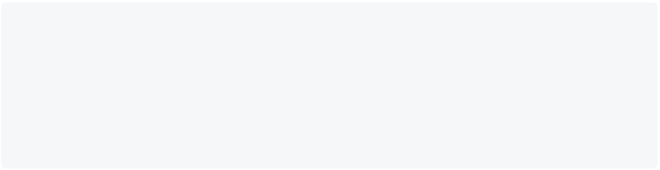
[Run Online](https://dartpad.dev/?id=7be7e0aa5918419c03b55d27222d4820)

**Round Double Value To 2 Decimal Places**

The .toStringAsFixed(2) is used to round the double value upto 2 decimal places in dart. You can round to any decimal places by entering numbers like 2, 3,

4, etc.

void main() {



// Declaring Variables

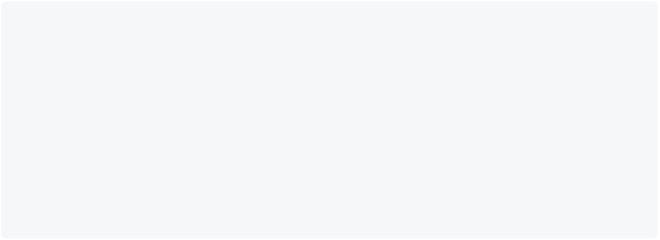
double price = 1130.2232323233233; // valid. print(price.toStringAsFixed(2));

}

[Run Online](https://dartpad.dev/?id=cf1299853b81c55dad6539147bab4bd4) **String**

String helps you to store text data. You can store values like **I love dart**, **New York 2140** in String. You can use single or double quotes to store string in dart.

void main() {



// Declaring Values

String schoolName = "Diamond School"; String address = "New York 2140";

// Printing Values

print("School name is $schoolName and address is $address");

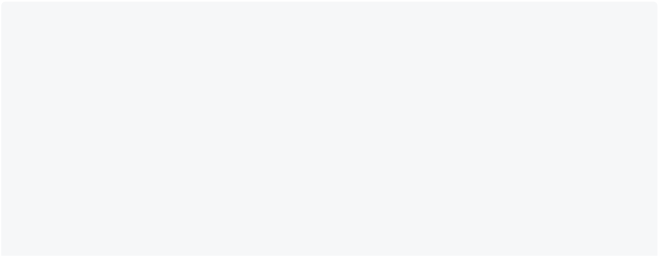
}

[Run Online](https://dartpad.dev/?id=babf76424f9daaafcb0c522b39fafdfe)

**Create A Multi-Line String In Dart**

If you want to create a multi-line String in dart, then you can use triple quotes with either single or double quotation marks.

void main() {



// Multi Line Using Single Quotes String multiLineText = '''

This is Multi Line Text

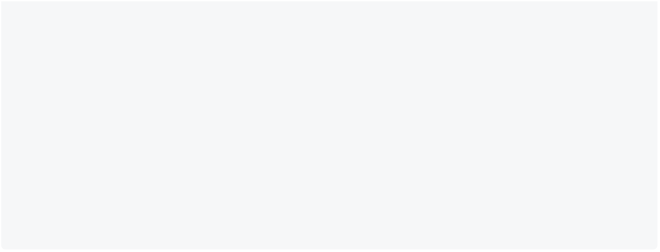
with 3 single quote

I am also writing here.

''';

// Multi Line Using Double Quotes String otherMultiLineText = """

This is Multi Line Text with 3 double quote



I am also writing here. """;

// Printing Information

print("Multiline text is $multiLineText"); print("Other multiline text is $otherMultiLineText");

}

[Run Online](https://dartpad.dev/?id=d97095b4bf9822a4838cc6c3571cc457)

**Special Character In String**

|  |  |
| --- | --- |
| **Special Character** | **Work** |
| \n | New Line |
| \t | Tab |

void main() {



// Using \n and \t

print("I am from \nUS.");

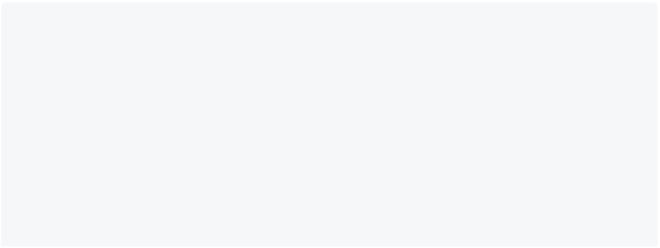
print("I am from \tUS."); }

[Run Online](https://dartpad.dev/?id=0fd1cb69933acb4686e9523ef8e839c6)

**Create A Raw String In Dart**

You can also create raw string in dart. Special characters won’t work here. You must write **r** after equal sign.

void main() {



// Set price value

num price = 10;

String withoutRawString = "The value of price is \t $price"; //

regular String

String withRawString =r"The value of price is \t $price"; // raw String

print("Without Raw: $withoutRawString"); // regular result

print("With Raw: $withRawString"); // with raw result



}

[Run Online](https://dartpad.dev/?id=d2d0263d6e80d92a1e5f845e1074963b)

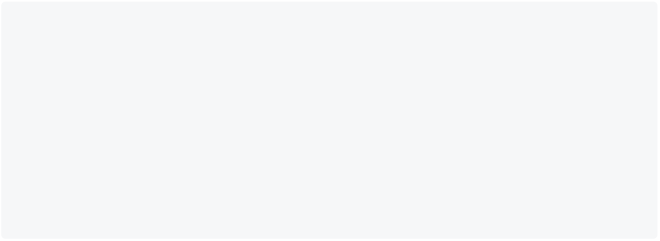
**Type Conversion In Dart**

In dart, type conversion allows you to convert one data type to another type. For e.g. to convert String to int, int to String or String to bool, etc.

**Convert String To Int In Dart**

You can convert String to int using int.parse() method. The method takes String as an argument and converts it into an integer.

void main() {



String strvalue = "1";

print("Type of strvalue is ${strvalue.runtimeType}"); int intvalue = int.parse(strvalue);

print("Value of intvalue is $intvalue");

// this will print data type

print("Type of intvalue is ${intvalue.runtimeType}");

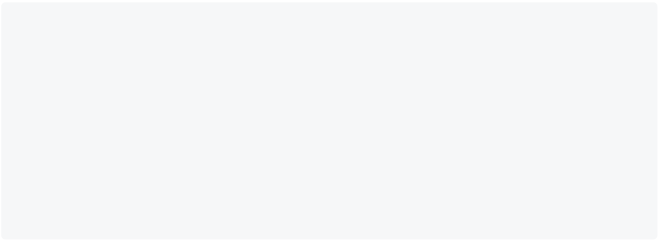
}

[Run Online](https://dartpad.dev/?id=93c9156125508d1344a8324dd337aa6a)

**Convert String To Double In Dart**

You can convert String to double using double.parse() method. The method takes String as an argument and converts it into a double.

void main() {



String strvalue = "1.1";

print("Type of strvalue is ${strvalue.runtimeType}"); double doublevalue = double.parse(strvalue);

print("Value of doublevalue is $doublevalue");

// this will print data type

print("Type of doublevalue is ${doublevalue.runtimeType}");

}

[Run Online](https://dartpad.dev/?id=35e0e9928eac53ff2f15b75ec690942a)

**Convert Int To String In Dart**

You can convert int to String using the toString() method. Here is example:

void main() {



int one = 1;

print("Type of one is ${one.runtimeType}");

String oneInString = one.toString();

print("Value of oneInString is $oneInString");

// this will print data type

print("Type of oneInString is ${oneInString.runtimeType}");

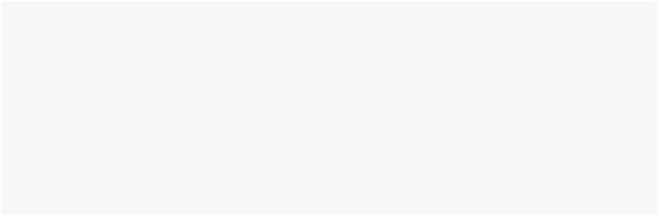
}

[Run Online](https://dartpad.dev/?id=b70894081c9642a9c5626cfc5ecb14c9)

**Convert Double To Int In Dart**

You can convert double to int using the toInt() method.

void main() {



double num1 = 10.01;

int num2 = num1.toInt(); // converting double to int

print("The value of num1 is $num1. Its type is ${num1.runtimeType}"); print("The value of num2 is $num2. Its type is ${num2.runtimeType}"); }

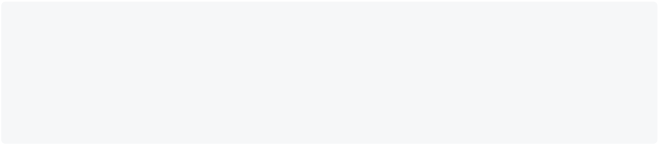
[Run Online](https://dartpad.dev/?id=91723c90e7aa0c5e55b3f15866e9dbc0) **Booleans**

In Dart, boolean holds either true or false value. You can write the **bool** keyword to define the boolean data type. You can use boolean if the answer is true or false. Consider the answer to the following questions:

* Are you married?
  + Is the door open?
    - Does a cat fly?
      * Is the traffic light green?
* Are you older than your father?

**These all are yes/no questions. Its a good idea to store them in boolean.**

void main() {



bool isMarried = true;

print("Married Status: $isMarried"); }

[Run Online](https://dartpad.dev/?id=affed489d713e7b1a193dbc01c080425) **Lists**

The list holds multiple values in a single variable. It is also called arrays. If you want to store multiple values without creating multiple variables, you can use a list.

void main() {



List<String> names = ["Raj", "John", "Max"]; print("Value of names is $names");

print("Value of names[0] is ${names[0]}"); // index 0 print("Value of names[1] is ${names[1]}"); // index 1 print("Value of names[2] is ${names[2]}"); // index 2

// Finding Length of List

int length = names.length;

print("The Length of names is $length");

}

[Run Online](https://dartpad.dev/?id=ef8ac2962eafb87626cff2d0a72631ef)

**Note**: List index always starts with 0. Here names[0] is Raj, names[1] is John and names[2] is Max.

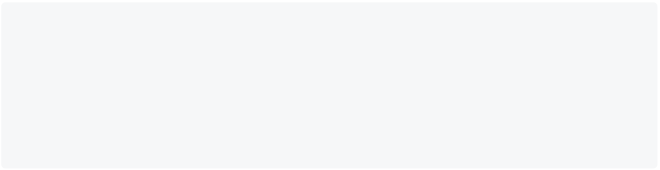
**Sets**

An unordered collection of unique items is called set in dart. You can store unique data in sets.

Info

Note: Set doesn’t print duplicate items.

void main() {



Set<String> weekday = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

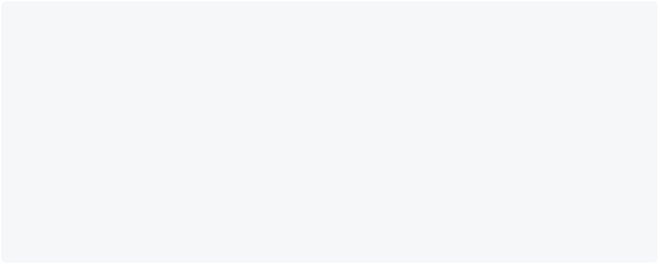
print(weekday);

}

[Run Online](https://dartpad.dev/?id=19097581fa56c7d5322da870e121a33c) **Maps**

In Dart, a map is an object where you can store data in key-value pairs. Each key occurs only once, but you can use same value multiple times.

void main() {



Map<String, String> myDetails = { 'name': 'John Doe',

'address': 'USA',

'fathername': 'Soe Doe'

};

// displaying the output print(myDetails['name']);

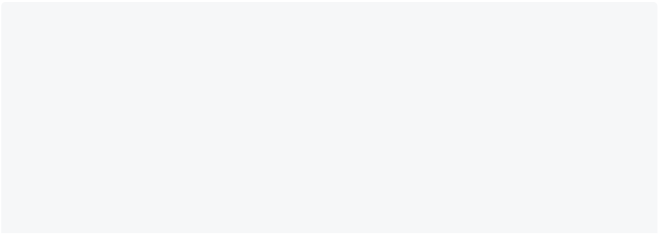
}

[Run Online](https://dartpad.dev/?id=38f9825722404dd6f31d259ea52605d6)

**Var Keyword In Dart**

In Dart, **var** automatically finds a data type. In simple terms, var says if you don’t want to specify a data type, I will find a data type for you.

void main(){



var name = "John Doe"; // String var age = 20; // int

print(name); print(age);

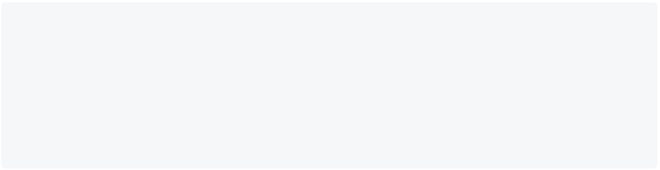
}



[Run Online](https://dartpad.dev/?id=ba9debaeebf20409ddf8e59b3f4daa22) **Runes In Dart**

With runes, you can find Unicode values of String. The Unicode value of **a** is **97**, so runes give 97 as output.

void main() {



String value = "a";

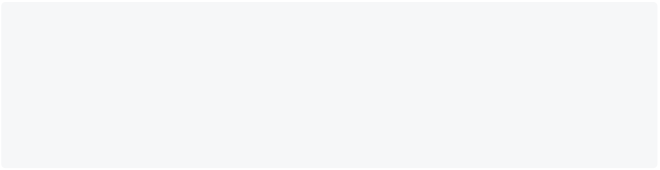
print(value.runes); }

[Run Online](https://dartpad.dev/?id=eb53d4079e1051f22bc2950de78ac241)

**How To Check Runtime Type**

You can check runtime type in dart with .runtimeType after the variable name.

void main() {



var a = 10;

print(a.runtimeType); print(a is int); // true }

[Run Online](https://dartpad.dev/?id=b15e41105a5b65c6ca0daaf8d00bf595)

**Optionally Typed Language**

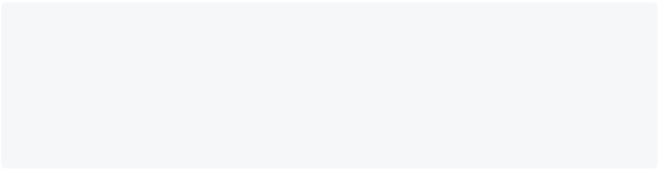
You may have heard of the **statically-typed** language. It means the data type of variables is known at compile time. Similarly, **dynamically-typed** language means data types of variables are known at run time. Dart supports dynamic and static types, so it is called optionally-typed language.

**Statically Typed**

A language is statically typed if the data type of variables is known at compile time. Its main advantage is that the compiler can quickly check the issues and

detect bugs.

void main() {



var myVariable = 50; // You can also use int instead of var myVariable = "Hello"; // this will give error

print(myVariable);

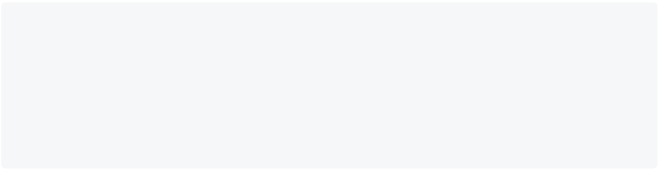
}

[Run Online](https://dartpad.dev/?id=9d93d16d1fd115524698bdd9510f84c7)

**Dynamically Typed Example**

A language is dynamically typed if the data type of variables is known at run time.

void main() {



dynamic myVariable = 50; myVariable = "Hello";

print(myVariable);

}

[Run Online](https://dartpad.dev/?id=7db07c7c42b1408649cae9276817a1bf)

**Note**: Using static type helps you to prevent writing silly mistakes in code. It’s a good habit to use static type in dart.

### Comments in Dart

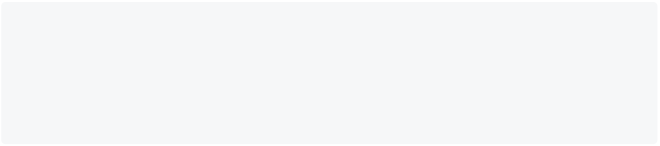
**Comments** are the set of statements that are ignored by the dart compiler during program execution. They are used to explain the code so that you or other people can understand it easily.

* You can describe your code.
  + Other people will understand your code more clearly.
* **Single-Line Comment**: For commenting on a single line of code. E.g. // This is a single-line comment.
* **Multi-Line Comment**: For commenting on multiple lines of code. E.g. /\* This is a multi-line comment. \*/
  + - **Documentation Comment**: For generating documentation or reference for a project/software package. E.g. /// This is a documentation comment

Single line comments start with // in dart. You can write // and your text.



void main() {



// This is single-line comment.

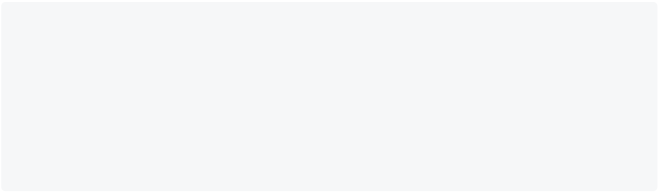
print("Welcome to Technology Channel."); }

[Run Online](https://dartpad.dev/?id=f7b3ff31a94bb2bb0ddbbc4c57d08ab7)

Multi-line comments start with /\* and end with \*/ . You can write your comment inside /\* and \*/.



void main(){



/\*

This is a multi-line comment.

\*/

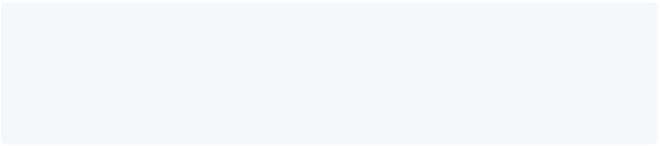
print("Welcome to Technology Channel."); }

[Run Online](https://dartpad.dev/?id=2eb2cb81bf3019e0838e5019656f0e25)

Documentation comments are helpful when you are writing documentation for your code. Documentation comments start with /// in dart.



void main(){



/// This is documentation comment

print("Welcome to Technology Channel."); }

[Run Online](https://dartpad.dev/?id=ce68f8c0b69d6ef7a0b7b7f8c0f1bd14) Operators in Dart

Operators are used to perform mathematical and logical operations on the variables. Each operation in dart uses a symbol called the operator to denote the type of operation it performs. Before learning operators in the dart, you must understand the following things.

* **Operands** : It represents the data.

**Operator** : It represents how the operands will be processed to produce a value.



Info

**Note**: Suppose the given expression is 2 + 3. Here 2 and 3 are operands, and + is the operator.



**Types Of Operators**

There are different types of operators in dart. They are as follows:

* **Arithmetic Operators**
  + **Increment and Decrement Operators**
* **Assignment Operators**
  + - **Logical Operators**
      * **Type Test Operators**

**Arithmetic Operators**

Arithmetic operators are the most common types of operators. They perform operations like addition, subtraction, multiplication, division, etc.

|  |  |  |
| --- | --- | --- |
| **Operator Symbol** | **Operator Name** | **Description** |
| + | Addition | For adding two operands |
| - | Subtraction | For subtracting two operands |
| -expr | Unary Minus | For reversing the sign of the expression |
| \* | Multiplication | For multiplying two operands |
| / | Division | For dividing two operands and give output in double |
| ~/ | Integer Division | For dividing two operands and give output in integer |
| % | Modulus | Remainder After Integer Division |

Let’s look at how to perform arithmetic calculations in dart.

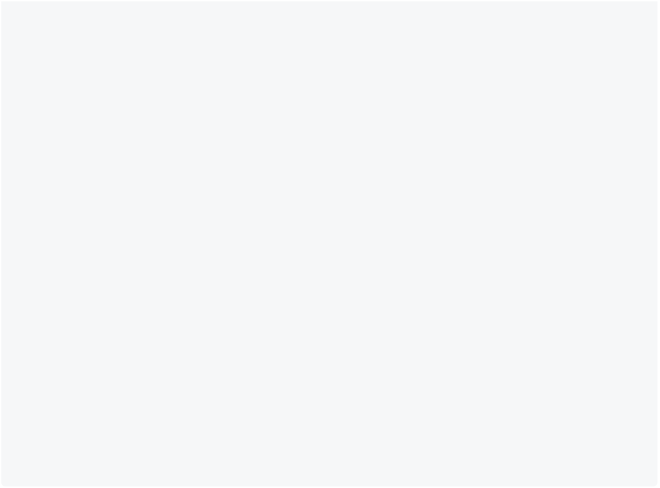
void main() {



// declaring two numbers int num1=10;

int num2=3;

// performing arithmetic calculation



int sum=num1+num2; // addition

int diff=num1-num2; // subtraction

int unaryMinus = -num1; // unary minus int mul=num1\*num2; // multiplication

double div=num1/num2; // division

int div2 =num1~/num2; // integer division int mod=num1%num2; // show remainder

//Printing info

print("The addition is $sum.");

print("The subtraction is $diff.");

print("The unary minus is $unaryMinus."); print("The multiplication is $mul.");

print("The division is $div.");

print("The integer division is $div2."); print("The modulus is $mod.");

}

[Run Online](https://dartpad.dev/?id=cb631ee189ccf2f8d5e215f6b1834746)

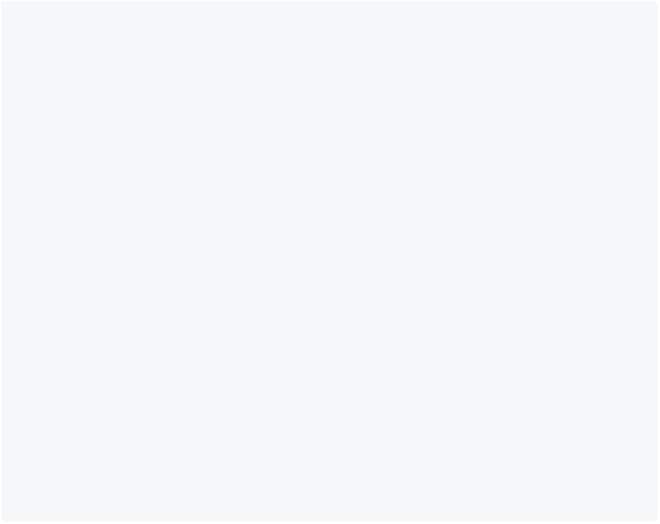
**Increment and Decrement Operators**

With increment and decrement operators, you can increase and decrease values. If ++ is used at the beginning, then it is a prefix. If it is used at last, then it is postfix.

|  |  |  |
| --- | --- | --- |
| **Operator Symbol** | **Operator Name** | **Description** |
| ++var | Pre Increment | Increase Value By 1. var = var + 1 Expression value is var+1 |
| --var | Pre Decrement | Decrease Value By 1. var = var - 1 Expression value is var-1 |
| var++ | Post Increment | Increase Value By 1. var = var + 1 Expression value is var |
| var-- | Post Decrement | Decrease Value By 1. var = var - 1 Expression value is var |

**Note**: ++var increases the value of operands, whereas var++ returns the actual value of operands before the increment.

void main() {



// declaring two numbers int num1=0;

int num2=0;

// performing increment / decrement operator

// pre increment

num2 = ++num1;

print("The value of num2 is $num2");

// reset value to 0 num1 = 0;

num2 = 0;

// post increment

num2 = num1++;

print("The value of num2 is $num2");

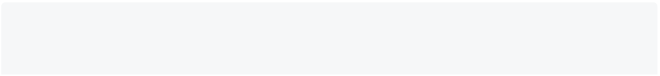
}

[Run Online](https://dartpad.dev/?id=a22b435de274f81b7553046482ecd084) **Assignment Operators**

It is used to assign some values to variables. Here, we are assigning 24 to the age variable.

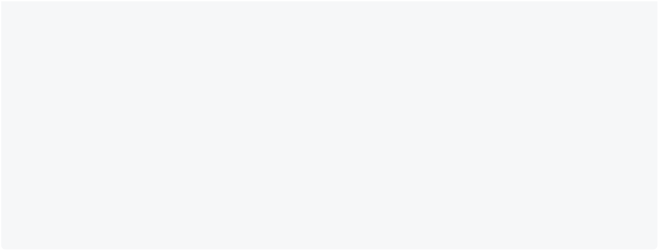
|  |  |
| --- | --- |
| **Operator Type** | **Description** |
| = | Assign a value to a variable |
| += | Adds a value to a variable |
| -= | Reduces a value to a variable |
| \*= | Multiply value to a variable |
| /= | Divided value by a variable |

void main() {



double age = 24;

age+= 1; // Here age+=1 means age = age + 1. print("After Addition Age is $age");



age-= 1; //Here age-=1 means age = age - 1. print("After Subtraction Age is $age");

age\*= 2; //Here age\*=2 means age = age \* 2. print("After Multiplication Age is $age");

age/= 2; //Here age/=2 means age = age / 2. print("After Division Age is $age");

}

[Run Online](https://dartpad.dev/?id=80bd0e4ffe646c1456ce93f22f013ff5) **Relational Operators**

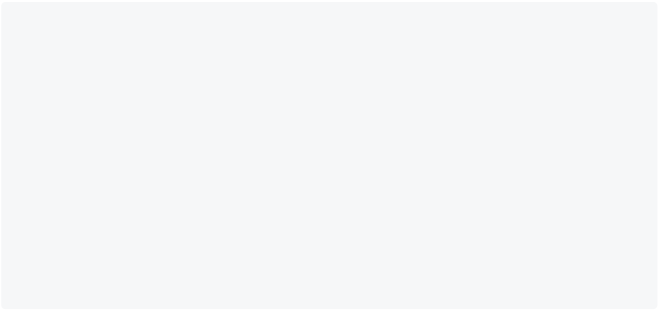
Relational operators are also called comparison operators. They are used to make a comparison.

* Operator Symbol: >
  + - Operator Name: Greater than
      * Description: Used to check which operand is bigger and gives result as boolean
* Operator Symbol: <
  + - Operator Name: Less than
      * + Description: Used to check which operand is smaller and gives result as boolean
* Operator Symbol: >=
  + - Operator Name: Greater than or equal to
      * Description: Used to check which operand is bigger or equal and gives result as boolean
* Operator Symbol: <=
  + - Operator Name: Less than or equal to
      * Description: Used to check which operand is smaller or equal and gives result as boolean
* Operator Symbol: ==
  + - Operator Name: Equal to

Description: Used to check operands are equal to each other and gives result as boolean

* + Operator Symbol: !=
* Operator Name: Not equal to
  + Description: Used to check operand are not equal to each other and gives result as boolean

void main() {



int num1=10;

int num2=5;

//printing info

print(num1==num2); print(num1<num2); print(num1>num2); print(num1<=num2); print(num1>=num2);

}

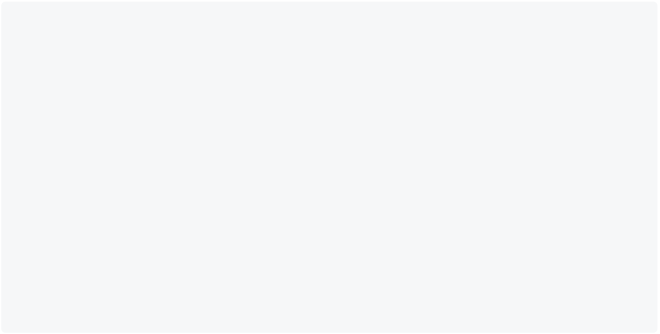
[Run Online](https://dartpad.dev/?id=751428eeb99566d37edd02325f446ca8)

**Logical Operators**

It is used to compare values.

|  |  |
| --- | --- |
| **Operator Type** | **Description** |
| && | This is ‘and’, return true if all conditions are true |
|  |  |
| ! | This is ’not’. return false if the result is true and vice versa |

void main(){



int userid = 123; int userpin = 456;

// Printing Info

print((userid == 123) && (userpin== 456)); // print true

print((userid == 1213) && (userpin== 456)); // print false. print((userid == 123) || (userpin== 456)); // print true. print((userid == 1213) || (userpin== 456)); // print true print((userid == 123) != (userpin== 456));//print false

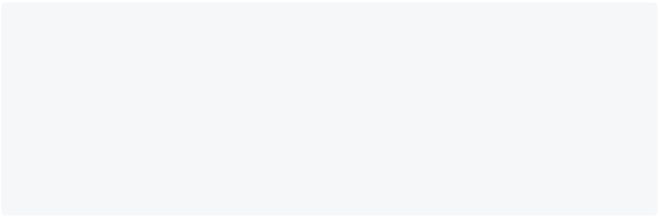
[Run Online](https://dartpad.dev/?id=567ef2df99b30af3352e71dd74d92b90)

**Type Test Operators**

In Dart, type test operators are useful for checking types at runtime.

|  |  |  |
| --- | --- | --- |
| **Operator Symbol** | **Operator Name** | **Description** |
| is | is | Gives boolean value true if the object has a specific type |
| is! | is not | Gives boolean value false if the object has a specific type |

void main() {



String value1 = "Dart Tutorial"; int age = 10;

print(value1 is String); print(age is !int);

}

[Run Online](https://dartpad.dev/?id=13717bb8ae63da6004b4bb4a44bf0fb8)

### User Input in Dart

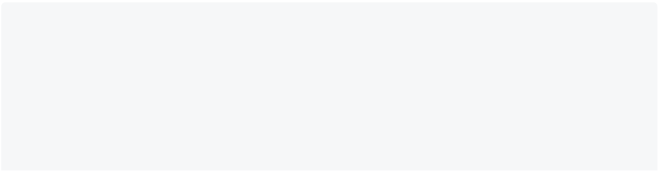
You must import the package import 'dart:io'; for user input.

**Note**: You won’t be able to take input from users using dartpad. You need to run a program from your computer.

**String User Input**

They are used for storing textual user input. If you want to keep values like somebody’s name, address, description, etc., you can take string input from the user.

import 'dart:io';



void main() {

print("Enter name:");

String? name = stdin.readLineSync(); print("The entered name is ${name}");

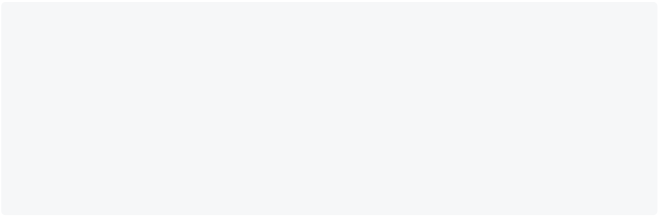
}



**Integer User Input**

You can take integer input to get a numeric value from the user without the decimal point. E.g. 10, 100, -800 etc.

import 'dart:io';



void main() {

print("Enter number:");

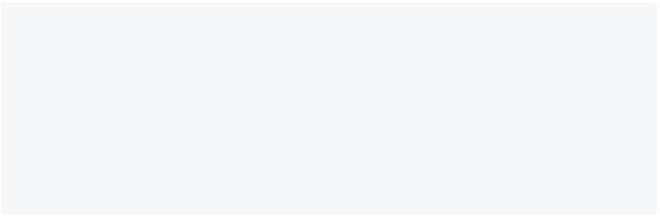
int? number = int.parse(stdin.readLineSync()!); print("The entered number is ${number}");

}

**Floating Point User Input**

You can use float input if you want to get a numeric value from the user with the decimal point. E.g. 10.5, 100.5, -800.9 etc.

import 'dart:io';



void main() {

print("Enter a floating number:");

double number = double.parse(stdin.readLineSync()!); print("The entered num is $number");

}

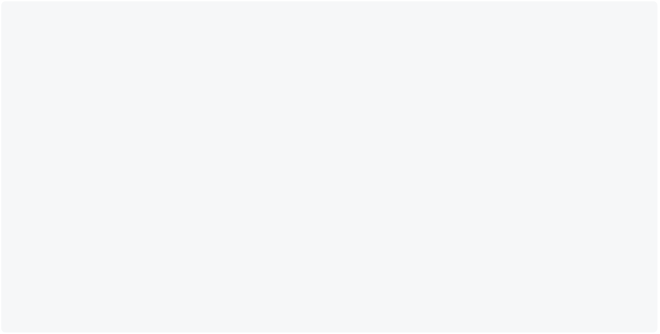
### String in Dart

**String** helps you to store text based data. In String, you can represent your name, address, or complete book. It holds a series or sequence of characters – letters, numbers, and special characters. You can use single or double, or triple quotes to represent String.

**Example: String In Dart**

Single line String is written in single or double quotes, whereas multi-line strings are written in triple quotes. Here is an example of it:

void main() {



String text1 = 'This is an example of a single-line string.'; String text2 = "This is an example of a single line string using double quotes.";

String text3 = """This is a multiline line

string using the triple-quotes.

This is tutorial on dart strings.

""";

print(text1);

print(text2);

print(text3);

}

[Run Online](https://dartpad.dev/?id=c2023963b37b2664ea4b40650e1c0037)

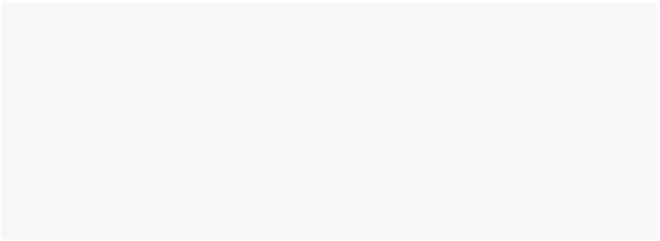
**String Concatenation**

You can combine one String with another string. This is called concatenation. In Dart, you can use the + operator or use **interpolation** to concatenate the String. Interpolation makes it easy to read and understand the code.



**String Concatenation In Dart**

void main() {



String firstName = "John";

String lastName = "Doe";

print("Using +, Full Name is "+firstName + " " + lastName+"."); print("Using interpolation, full name is $firstName

$lastName."); }

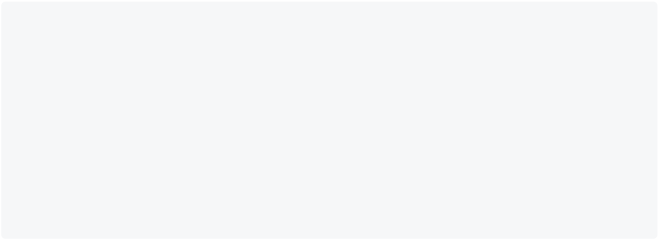
[Run Online](https://dartpad.dev/?id=e4abbe14213951d7526cb280eddc47c2)

**Properties Of String**

* **codeUnits**: Returns an unmodifiable list of the UTF-16 code units of this string.
* **isEmpty**: Returns true if this string is empty.
  + **isNotEmpty**: Returns false if this string is empty.
* **length**: Returns the length of the string including space, tab, and newline characters.

**String Properties Example In Dart**

void main() {



String str = "Hi";

print(str.codeUnits); //Example of code units

print(str.isEmpty); //Example of isEmpty

print(str.isNotEmpty); //Example of isNotEmpty

print("The length of the string is: ${str.length}"); //Example of Length

}

[Run Online](https://dartpad.dev/?id=e2fdfcfe0fcfbd11df3644acc24b531a)

**Methods Of String**

* **toLowerCase()**: Converts all characters in this string to lowercase.
  + **toUpperCase()**: Converts all characters in this string to uppercase.
    - **trim()**: Returns the string without any leading and trailing whitespace.
      * **compareTo()**: Compares this object to another.
    - **replaceAll()**: Replaces all substrings that match the specified pattern with a given value.
    - **split()**: Splits the string at matches of the specified delimiter and returns a list of substrings.
* **toString()**: Returns a string representation of this object.
  + **substring()**: Returns the text from any position you want.
    - **codeUnitAt()**: Returns the 16-bit UTF-16 code unit at the given index.

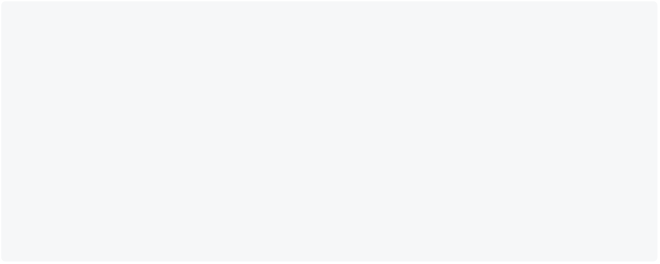
**String Methods Example In Dart**

Here you will see various string methods that can help your work a lot better and faster.

**Converting String To Uppercase and Lowercase**

You can convert your text to lower case using .toLowerCase() and convert to uppercase using .toUpperCase() method.

//Example of toUpperCase() and toLowerCase()



void main() {

String address1 = "Florida"; // Here F is capital

String address2 = "TexAs"; // Here T and A are capital

print("Address 1 in uppercase: ${address1.toUpperCase()}"); print("Address 1 in lowercase: ${address1.toLowerCase()}"); print("Address 2 in uppercase: ${address2.toUpperCase()}"); print("Address 2 in lowercase: ${address2.toLowerCase()}");

}

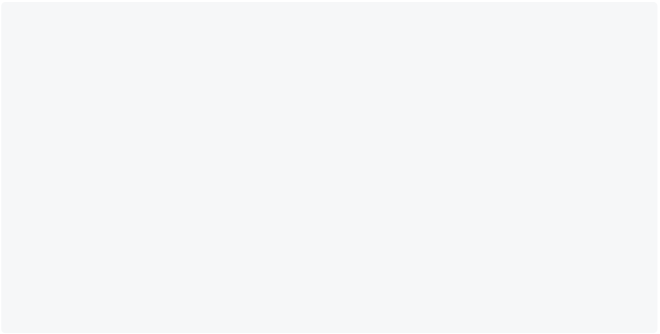
[Run Online](https://dartpad.dev/?id=13ba9115b8702616179459da7c189cbe)

**Trim String In Dart**

Trim is helpful when removing leading and trailing spaces from the text. This trim method will remove all the starting and ending spaces from the text. You can also use **trimLeft()** and **trimRight()** methods to remove space from left and right, respectively.

**Note**: The trim() method in Dart doesn’t remove spaces in the middle.

//Example of trim()



void main() {

String address1 = " USA"; // Contain space at leading.

String address2 = "Japan "; // Contain space at trailing. String address3 = "New Delhi"; // Contains space at middle.

print("Result of address1 trim is ${address1.trim()}"); print("Result of address2 trim is ${address2.trim()}");

print("Result of address3 trim is ${address3.trim()}");

print("Result of address1 trimLeft is ${address1.trimLeft()}"); print("Result of address2 trimRight is ${address2.trimRight()}"); }

[Run Online](https://dartpad.dev/?id=9ce2e4517d297d4f22c9c055917c2617)

**Compare String In Dart**

In Dart, you can compare two strings. It will give the result 0 when two texts are equal, 1 when the first String is greater than the second, and -1 when the first String is smaller than the second.

//Example of compareTo() void main() {



String item1 = "Apple"; String item2 = "Ant";

String item3 = "Basket";

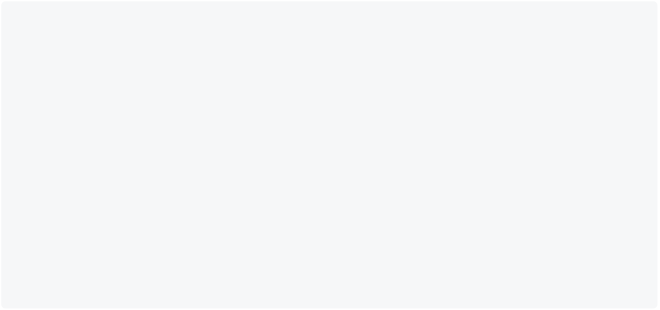
print("Comparing item 1 with item 2: ${item1.compareTo(item2)}"); print("Comparing item 1 with item 3: ${item1.compareTo(item3)}"); print("Comparing item 3 with item 2: ${item3.compareTo(item2)}"); }

[Run Online](https://dartpad.dev/?id=964cfc29d289b6949283c4fc7ad846ea)

**Replace String In Dart**

You can replace one value with another with the replaceAll(“old”, “new”) method in Dart. It will replace all the “old” words with “new”. Here in this example, this will replace milk with water.

//Example of replaceAll()



void main() {

String text = "I am a good boy I like milk. Doctor says milk is good for health.";

String newText = text.replaceAll("milk", "water");

print("Original Text: $text"); print("Replaced Text: $newText");

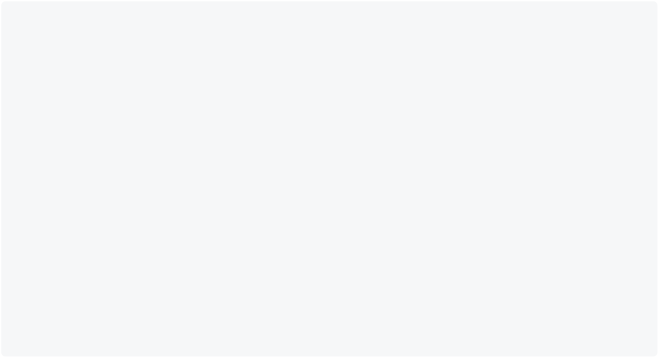
}

[Run Online](https://dartpad.dev/?id=47f1cae2e9a0e2f4502947f486fbe260)

**Split String In Dart**

You can use the dart split method if you want to split String by comma, space, or other text. It will help you to split String to list.

//Example of split()



void main() {

String allNames = "Ram, Hari, Shyam, Gopal";

List<String> listNames = allNames.split(","); print("Value of listName is $listNames");

print("List name at 0 index ${listNames[0]}"); print("List name at 1 index ${listNames[1]}"); print("List name at 2 index ${listNames[2]}"); print("List name at 3 index ${listNames[3]}");

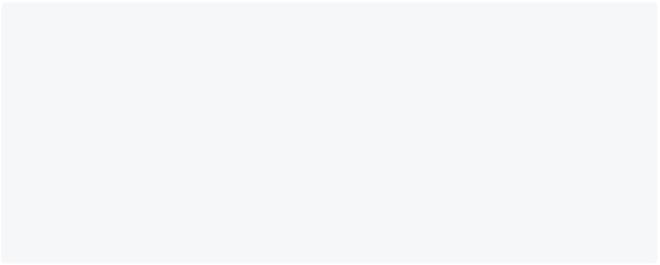
}

[Run Online](https://dartpad.dev/?id=e2084d37c1c4153c687157f0ab528731)

**ToString In Dart**

In dart, toString() represents String representation of the value/object.

//Example of toString()



void main() {

int number = 20;

String result = number.toString();

print("Type of number is ${number.runtimeType}"); print("Type of result is ${result.runtimeType}");

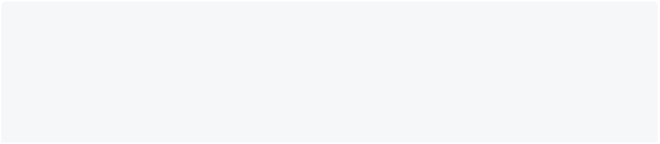
}

[Run Online](https://dartpad.dev/?id=85c061d07c69063fb819da5042ae9f03)

**SubString In Dart**

You can use substring in Dart when you want to get a text from any position.

//Example of substring()

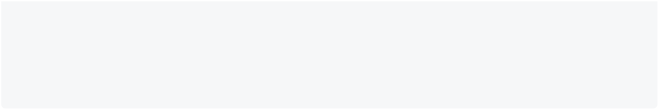


void main() {

String text = "I love computer";

print("Print only computer: ${text.substring(7)}"); // from index 6 to the last index

print("Print only love: ${text.substring(2,6)}");// from index 2 to the 6th index



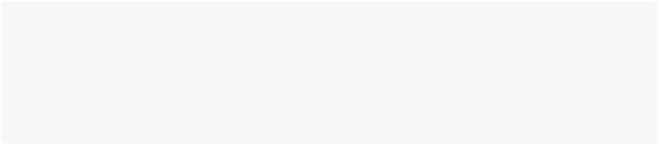
}

[Run Online](https://dartpad.dev/?id=e9b00023c94645700208883bf320aa39)

**Reverse String In Dart**

If you want to reverse a String in Dart, you can reverse it using a different solution. One solution is here.

void main() {



String input = "Hello";

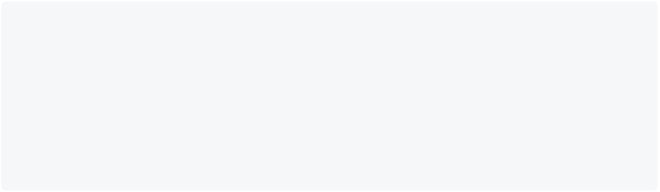
print("$input Reverse is ${input.split('').reversed.join()}"); }

[Run Online](https://dartpad.dev/?id=bdc13769cc0522c03ae999a6db4330a3)

**How To Capitalize First Letter Of String In Dart**

If you want to capitalize the first letter of a String in Dart, you can use the following code.

//Example of capitalize first letter of String void main() {



String text = "hello world";

print("Capitalized first letter of String: ${text[0].toUpperCase()}${text.substring(1)}"); }

[Run Online](https://dartpad.dev/?id=f0f335372ad3bad70a651a5e8967500b)

## Conditions and Loops

### Conditions in Dart

With conditions, you can control the flow of the dart program.

**Types Of Condition**

You can use following conditions to control the flow of your program.

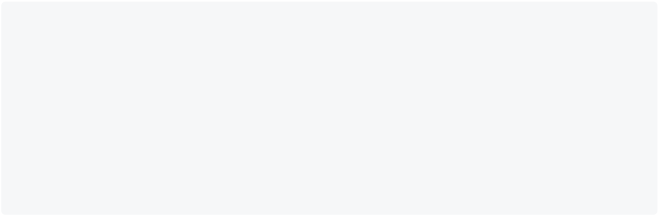
* **If Condition**
  + **If-Else Condition**
* **If-Else-If Condition**
  + - **Switch case**

**If Condition**

The easy and most common way of controlling the flow of a program is through the use of an *if statement*. If statement allow us to execute a code block when the given condition is true. Conditions evaluate boolean values.

**Syntax**

if(condition) { Statement 1; Statement 2; .



.

Statement n; }

**Example Of If Condition**

It prints whether the person is a voter. If the person’s age is greater and equal to 18, it will print, You are a voter.

void main()



{

var age = 20;

if(age >= 18){

print("You are voter."); }

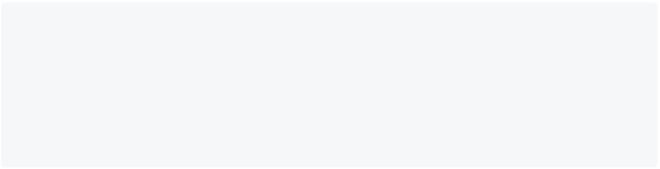
}

[Run Online](https://dartpad.dev/?id=6d8cac25d23cfd6e3d344ea3d7d551f6) **If-Else Condition**

If the result of the condition is true, then the body of the if-condition is executed. Otherwise, the body of the else-condition is executed.

**Syntax**

if(condition){



statements;

}else{

statements; }

**Example Of If-Else Condition**

Dart program prints whether the person is a voter or not based on age.

void main(){



int age = 12;

if(age >= 18){

print("You are voter.");

}else{

print("You are not voter."); }

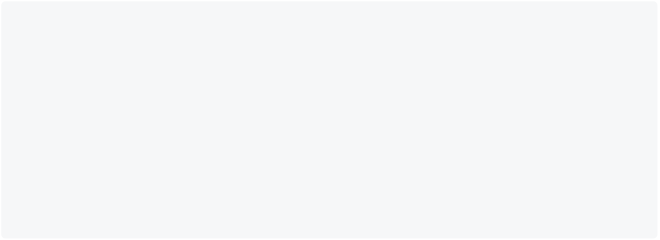
}

[Run Online](https://dartpad.dev/?id=976ee1d061eb24ba3ce26688bec33f37)

**Condition Based On Boolean Value**

If the married status is false, it prints you are single; otherwise, it will print you are married.

void main(){



bool isMarried = false;

if(isMarried){

print("You are married."); }else{

print("You are single."); }

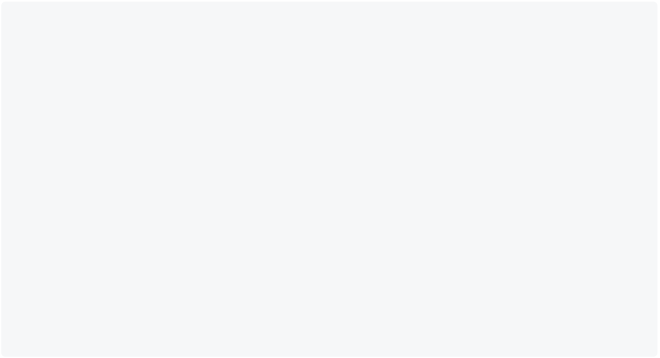
}

**If-Else-If Condition**

When you have multiple if conditions, then you can use if-else-if. You can learn more in the example below. When you have more than two conditions, you can use if, else if, else in dart.

**Syntax**

if(condition1){



statements1; }else if(condition2){

statements2; }else if(condition3){

statements3; }

.

.

.

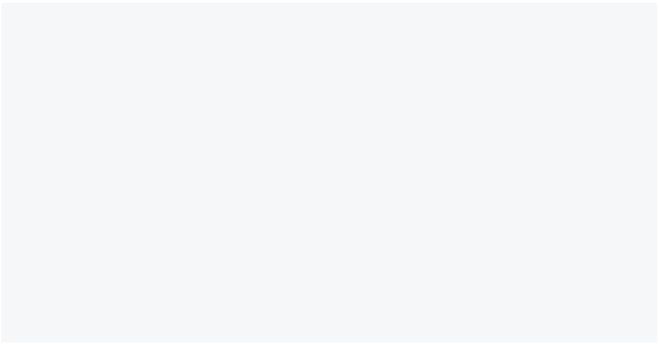
else(conditionN){ statementsN;

}

**Example Of If-Else-If Condition**

This program prints the month name based on the numeric value of that month. You will get a different result if you change the number of month.

void main() {



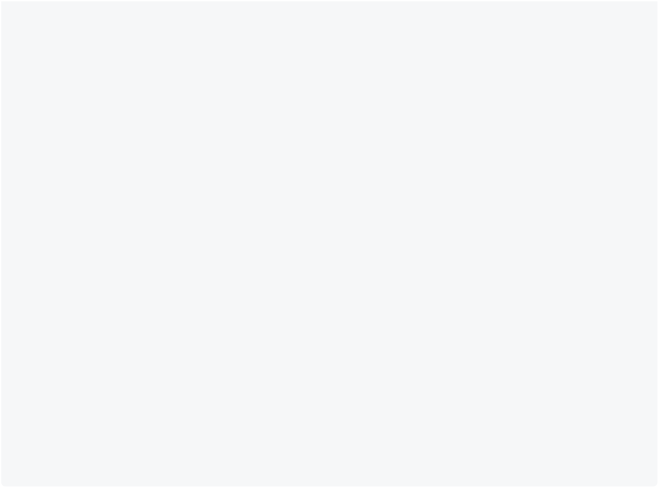
int noOfMonth = 5;

// Check the no of month

if (noOfMonth == 1) {

print("The month is jan"); } else if (noOfMonth == 2) { print("The month is feb"); } else if (noOfMonth == 3) { print("The month is march"); } else if (noOfMonth == 4) { print("The month is april"); } else if (noOfMonth == 5) {

print("The month is may"); } else if (noOfMonth == 6) { print("The month is june"); } else if (noOfMonth == 7) { print("The month is july"); } else if (noOfMonth == 8) { print("The month is aug"); } else if (noOfMonth == 9) { print("The month is sep"); } else if (noOfMonth == 10) { print("The month is oct"); } else if (noOfMonth == 11) { print("The month is nov"); } else if (noOfMonth == 12) { print("The month is dec");



} else {

print("Invalid option given."); }

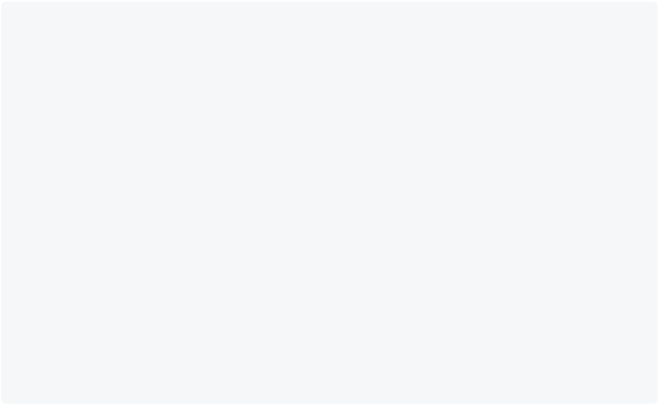
}

[Run Online](https://dartpad.dev/?id=6b8d479d070a1cdff239851785bf6935)

**Find Greatest Number Among 3 Numbers**

Dart program, which finds the greatest number among three numbers.

void main(){



int num1 = 1200; int num2 = 1000; int num3 = 150;

if(num1 > num2 && num1 > num3){

print("Num 1 is greater: i.e $num1"); }

if(num2 > num1 && num2 > num3){

print("Num2 is greater: i.e $num2"); }

if(num3 > num1 && num3 > num2){

print("Num3 is greater: i.e $num3"); }

}

### Switch Case in Dart

A Switch case is used to execute the code block based on the condition.

switch(expression) { case value1:



// statements break;

case value2:

// statements break;

case value3:

// statements break;

default:

// default statements }

**How does switch-case statement work in dart**

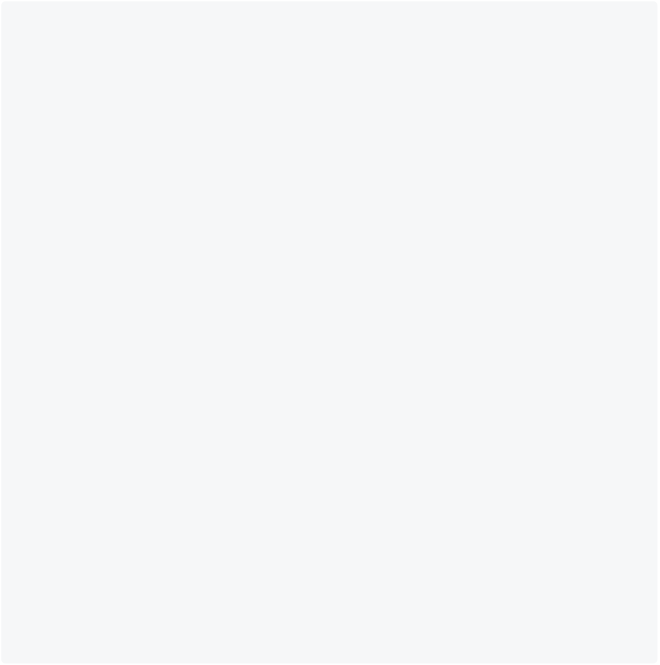
* The **expression** is evaluated once and compared with each case value.
  + If **expression** matches with case value1, the statements of case value1 are executed. Similarly, case value 2 will be executed if the expression matches case value2. If the expression matches the case value3, the statements of case value3 are executed.
* The **break** keywords tell dart to exit the switch statement because the statements in the case block are finished.
  + - If there is no match, **default statements** are executed.

**Note**: You can use a Switch case as an alternative to the **if-else-if** condition. **Replace If Else If With Switch In Dart**

Here you can see the same program using **if else if** and **switch** in dart. **Example: Using If Else If**

This example prints the day name based on the numeric day of the week using a if else if.

void main(){



var dayOfWeek = 5;

if (dayOfWeek == 1) {

print("Day is Sunday.");

}

else if (dayOfWeek == 2) {

print("Day is Monday.");

}

else if (dayOfWeek == 3) {

print("Day is Tuesday.");

}

else if (dayOfWeek == 4) {

print("Day is Wednesday."); }

else if (dayOfWeek == 5) {

print("Day is Thursday.");

}

else if (dayOfWeek == 6) {

print("Day is Friday."); }

else if (dayOfWeek == 7) {

print("Day is Saturday."); }else{

print("Invalid Weekday."); }

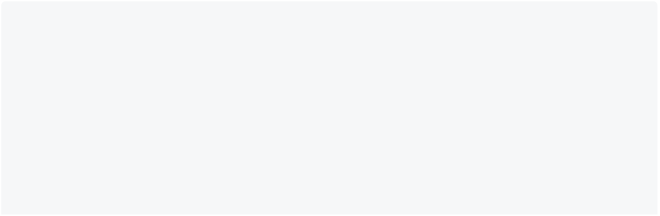
}

[Run Online](https://dartpad.dev/?id=ee6d90177af379b0a6fc7917130249fc)

**Example Of Switch Statement**

This example prints the day name based on the numeric day of the week using a switch case.

void main() {



var dayOfWeek = 5;

switch (dayOfWeek) {

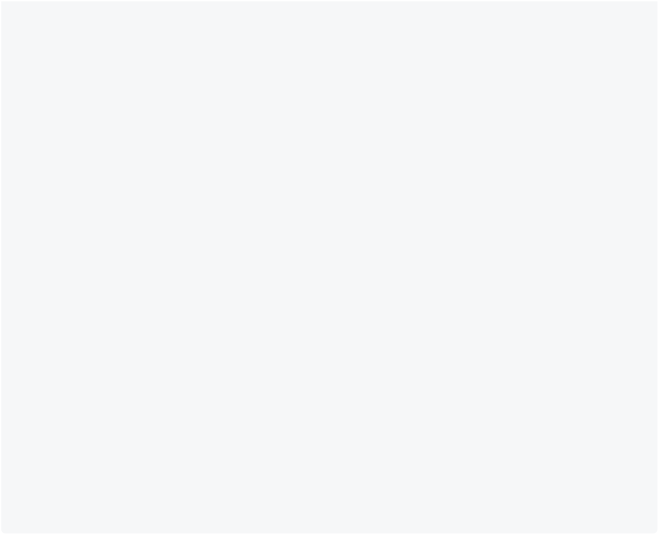
case 1:

print("Day is Sunday."); break;

case 2:

print("Day is Monday.");

break;



case 3:

print("Day is Tuesday.");

break;

case 4:

print("Day is Wednesday."); break;

case 5:

print("Day is Thursday."); break;

case 6:

print("Day is Friday.");

break;

case 7:

print("Day is Saturday."); break;

default:

print("Invalid Weekday."); break;

}

}

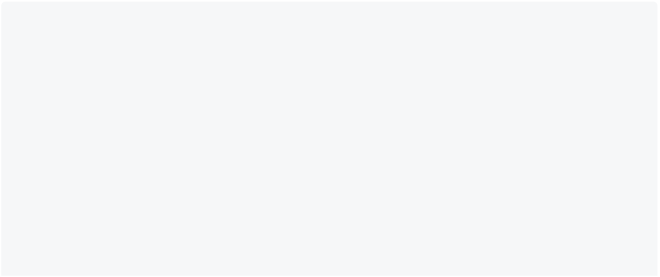
[Run Online](https://dartpad.dev/?id=01f56818a1754280968c0b6546321478)

**Note**: The syntax of switch statements is cleaner and much easier to read and write.

**Switch Case On Strings**

You can also use a switch case with strings. This program prints information based on weather value.

void main() {



const weather = "cloudy";

switch (weather) {

case "sunny":

print("Its a sunny day. Put sunscreen."); break;

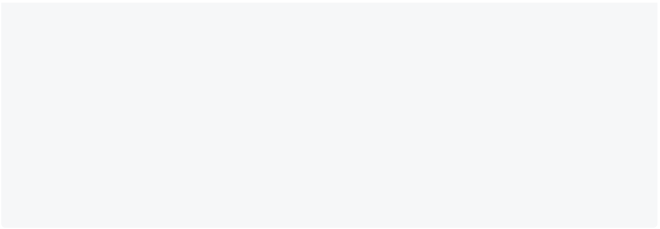
case "snowy":

print("Get your skis.");

break;

case "cloudy":

case "rainy":



print("Please bring umbrella.");

break;

default:

print("Sorry I am not familiar with such weather."); break;

}

}

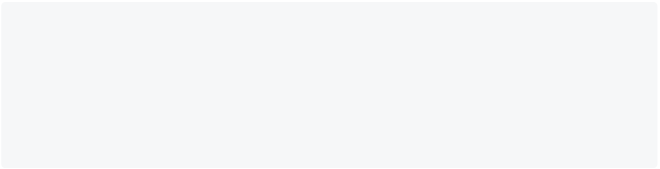
[Run Online](https://dartpad.dev/?id=ace41d17f7bc6b5f68a767cccc218000)

**Switch Case On Enum**

An [**enum**](https://dart-tutorial.com/object-oriented-programming/enum-in-dart/) or enumeration is used for defining value according to you. You can define your own type with a finite number of options. Here is the syntax for defining enum.

**Syntax**

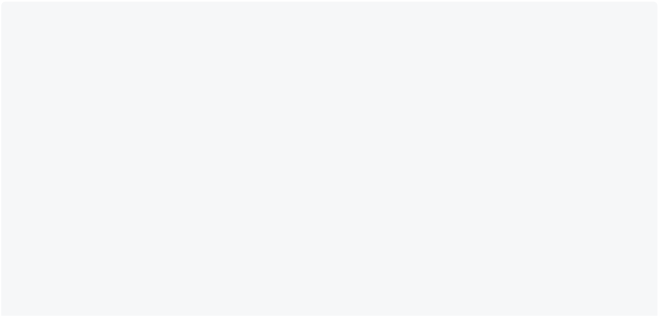
enum enum\_name { constant\_value1, constant\_value2, constant\_value3 }



**Example of Switch Using Enum In Dart**

Enum plays well with switch statements. Let’s see an example using enum.

// define enum outside main function



enum Weather{ sunny, snowy, cloudy, rainy} // main method

void main() {

const weather = Weather.cloudy;

switch (weather) {

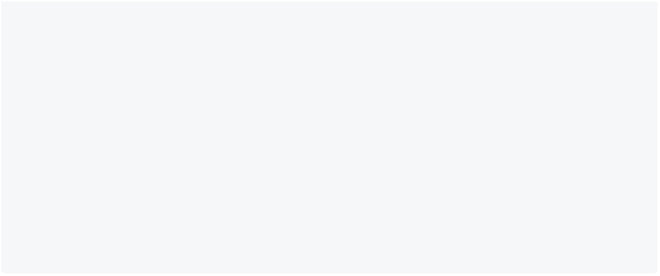
case Weather.sunny:

print("Its a sunny day. Put sunscreen."); break;

case Weather.snowy:

print("Get your skis.");

break;



case Weather.rainy:

case Weather.cloudy:

print("Please bring umbrella.");

break;

default:

print("Sorry I am not familiar with such weather."); break;

}

}

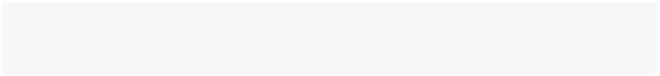
[Run Online](https://dartpad.dev/?id=e954c4fc1f335d0c06cb66d6c4493392)

### Ternary Operator in Dart

The ternary operator is like if-else statement. This is a one-liner replacement for the if-else statement. It is used to write a conditional expression, where based on the result of a boolean condition, one of the two values is selected.

**Syntax**

condition ? exprIfTrue : exprIfFalse



**Note**: The ternary operator takes a condition and returns one of two values, depending upon the condition’s boolean value, i.e., true or false.

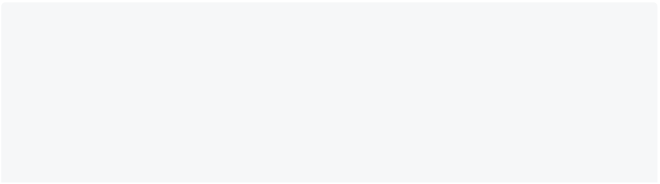
**Ternary Operator Vs If Else**

We already learned if-else in dart. Let us see the same example using the if-else and ternary operator.

**Example Using If Else**

This program finds greatest number between two numbers using if else.

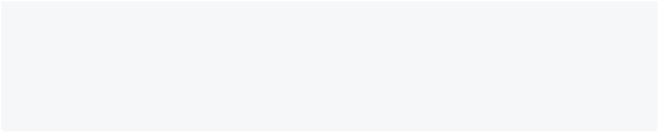
void main() {



int num1 = 10; int num2 = 15; int max = 0;

if(num1> num2){ max = num1; }else {

max = num2;



}

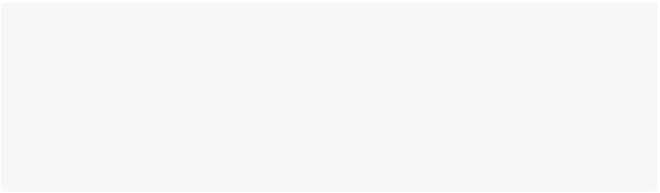
print("The greatest number is $max"); }

[Run Online](https://dartpad.dev/?id=2a5b5e4364c0b8a960a7048dcf848d5c)

**Example 1: Using Ternary Operator**

This program finds greatest number between two numbers using ternary operator.

void main() {



int num1 = 10;

int num2 = 15;

int max = (num1 > num2) ? num1 : num2; print("The greatest number is $max"); }

[Run Online](https://dartpad.dev/?id=d8828e24baa9ad7c4e80cdda2fc9811a)

**Note**: Ternary operator makes if-else code much shorter and readable. If you have problems with ternary, you can always use if-else.

**Example 2: Ternary Operator Dart**

If the selection value is 2 then it will set output as Apple otherwise, Banana.

void main() {



var selection = 2;

var output = (selection == 2) ? 'Apple' : 'Banana'; print(output);

}

[Run Online](https://dartpad.dev/?id=4c3782133a8a3e3279dc14a1de2357f5)

**Example 3 Ternary Operator Dart**

This is a dart program to print whether the person is a voter or not using a ternary operator.

void main() {



var age = 18;

var check = (age >= 18) ? 'You ara a voter.' : 'You are not a voter.'; print(check);

}

[Run Online](https://dartpad.dev/?id=51c2bc6014a529fb0e8415e59c629fe5)

### For Loop in Dart

This is the most common type of loop. You can use **for loop** to run a code block multiple times according to the condition. The syntax of for loop is:

for(initialization; condition; increment/decrement){ statements;



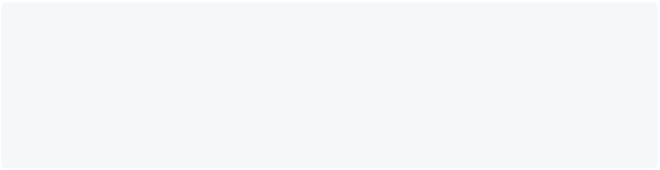
}

* Initialization is executed (one time) before the execution of the code block.
  + Condition defines the condition for executing the code block.
    - Increment/Decrement is executed (every time) after the code block has been executed.

**Example 1: To Print 1 To 10 Using For Loop**

This example prints 1 to 10 using for loop. Here **int i = 1;** is initialization, **i<=10** is condition and **i++** is increment/decrement.

void main() {



for (int i = 1; i <= 10; i++) { print(i);

}

}

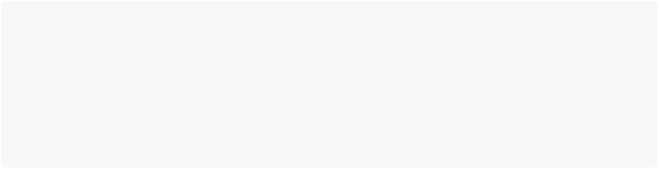
[Run Online](https://dartpad.dev/?id=264feb7ba11737142520e4b5f23351c0)

**Example 2: To Print 10 To 1 Using For Loop**

This example prints 10 to 1 using for loop. Here **int i = 10;** is initialization, **i>=1** is condition and **i--** is increment/decrement.



void main() {



for (int i = 10; i >= 1; i--) { print(i);

}

}

[Run Online](https://dartpad.dev/?id=f82a4a43964c0086b4aab3e5894e4194)

**Example 3: Print Name 10 Times Using For Loop**

This example prints the name 10 times using for loop. Based on the condition, the body of the loop executes 10 times.

void main() {



for (int i = 0; i < 10; i++) { print("John Doe");

}

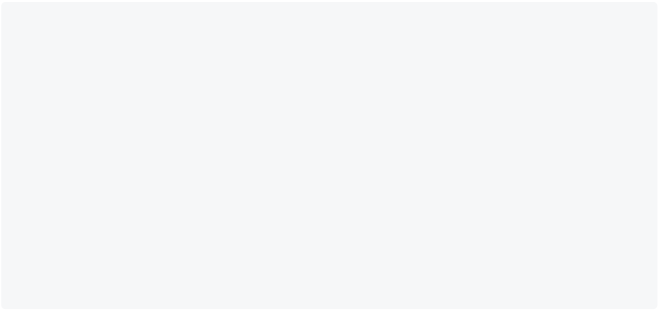
}

[Run Online](https://dartpad.dev/?id=ee30fb7c24a66cfd219624363ceaba44)

**Example 4: Display Sum of n Natural Numbers Using For Loop**

Here, the value of the **total** is **0** initially. Then, the for loop is iterated from **i = 1 to 100**. In each iteration, **i** is added to the **total**, and the value of **i** is increased by 1. Result is **1+2+3+….+99+100**.

void main(){



int total = 0;

int n = 100; // change as per required

for(int i=1; i<=n; i++){ total = total + i;

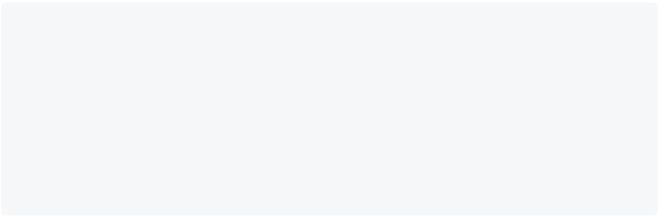
}

print("Total is $total");

[Run Online](https://dartpad.dev/?id=c2a8725e7265d0913de7df0457481fbe)

**Example 5: Display Even Numbers Between 50 to 100 Using For Loop** This program will print even numbers between 50 to 100 using for loop.

void main(){



for(int i=50; i<=100; i++){ if(i%2 == 0){

print(i);

}

}

}

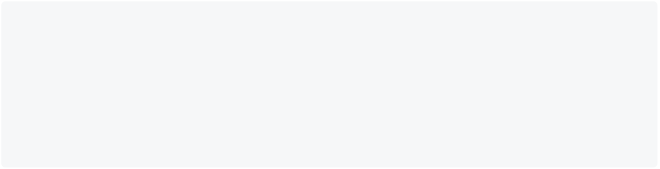
[Run Online](https://dartpad.dev/?id=045b8c1efb218cc4b6a3faf5cda9ac7c)

**Infinite Loop In Dart**

If the condition never becomes false in looping, it is called an infinite loop. It uses more resources on your computer. The task is done repeatedly until the memory runs out.

This program prints 1 to infinite because the condition is **i>=1**, which is always true with i++.

void main() {



for (int i = 1; i >= 1; i++) { print(i);

}

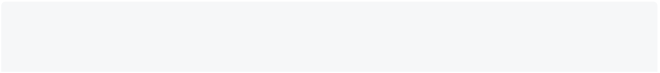
}

**Note**: Infinite loops take your computer resources continuously, use more power, and slow your computer. So always check your loop before use.

### For Each Loop in Dart

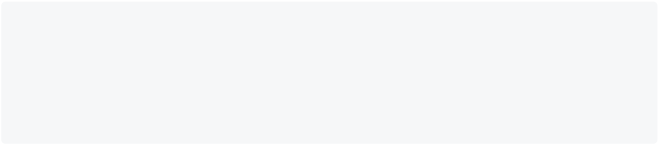
The **for each** loop iterates over all list elements or variables. It is useful when you want to loop through **list/collection**. The syntax of for-each loop is:

collection.forEach(void f(value));



**Example 1: Print Each Item Of List Using Foreach** This will print each name of football players.

void main(){



List<String> footballplayers=['Ronaldo','Messi','Neymar','Hazard']; footballplayers.forEach( (names)=>print(names));

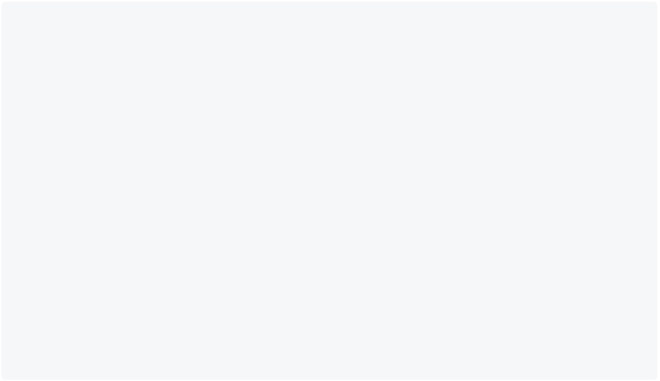
}

[Run Online](https://dartpad.dev/?id=3546b6f86e5fa6d692259a57bb2b31c3)

**Example 2: Print Each Total and Average Of Lists**

This program will print the total sum of all numbers and also the average value from the total.

void main(){



List<int> numbers = [1,2,3,4,5];

int total = 0;

numbers.forEach( (num)=>total= total+ num); print("Total is $total.");

double avg = total / (numbers.length);

print("Average is $avg.");

}

[Run Online](https://dartpad.dev/?id=398279cbea47b32e742d90523ede34a8)

**For In Loop In Dart**

There is also another for loop, i.e., **for in loop**. It also makes looping over the list very easily.

void main(){



List<String> footballplayers=['Ronaldo','Messi','Neymar','Hazard'];

for(String player in footballplayers){ print(player);

}

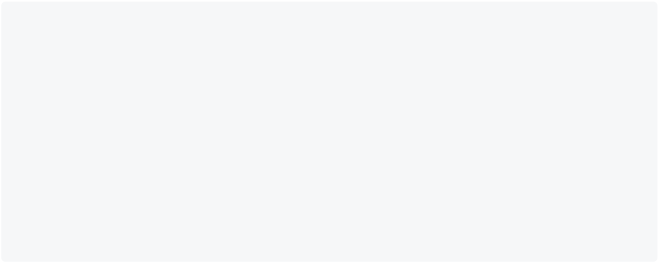
}

[Run Online](https://dartpad.dev/?id=5e76bb057666cf7c4e5c1cc3c45fa918)

**How to Find Index Value Of List**

In dart, asMap method converts the list to a map where the keys are the index and values are the element at the index.

void main(){



List<String> footballplayers= ['Ronaldo','Messi','Neymar','Hazard'];

footballplayers.asMap().forEach((index, value) => print("$value index is $index"));

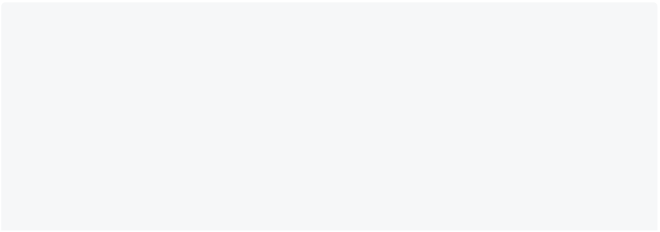
}

[Run Online](https://dartpad.dev/?id=cc3495b4811d8ee0c52618ec6551614e)

**Example 3: Print Unicode Value of Each Character of String**

This will split the name into Unicode values and then find characters from the Unicode value.

void main(){



String name = "John";

for(var codePoint in name.runes){

print("Unicode of ${String.fromCharCode(codePoint)} is

$codePoint.");

}

}



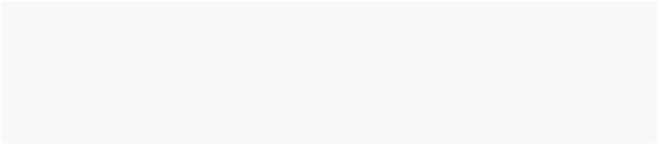
[Run Online](https://dartpad.dev/?id=72c0263e605c022d0134c4e58480f205)

### While Loop in Dart

In **while loop**, the loop’s body will run until and unless the condition is true. You must write conditions first before statements. This loop checks conditions on every iteration. If the condition is true, the code inside {} is executed, if the condition is false, then the loop stops.

**Syntax**

while(condition){



//statement(s);

// Increment (++) or Decrement (--) Operation; }

* A while loop evaluates the condition inside the parenthesis ().
  + If the condition is true, the code inside {} is executed.
    - The condition is re-checked until the condition is false.
      * When the condition is false, the loop stops.

**Example 1: To Print 1 To 10 Using While Loop** This program prints 1 to 10 using while loop.

void main() {



int i = 1;

while (i <= 10) { print(i);

i++;

}

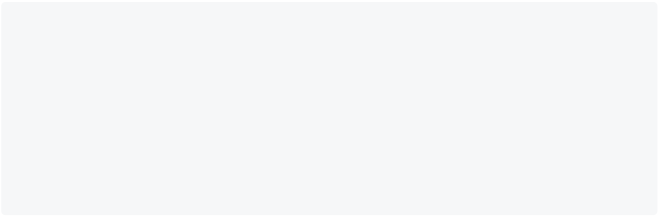
}

[Run Online](https://dartpad.dev/?id=4bd92d159957b654e5f8e0703eb6a620)

**Note**: Do not forget to increase the variable used in the condition. Otherwise, the loop will never end and becomes an infinite loop.

**Example 2: To Print 10 To 1 Using While Loop** This program prints 10 to 1 using while loop.

void main() {



int i = 10;

while (i >= 1) { print(i);

i--;

}

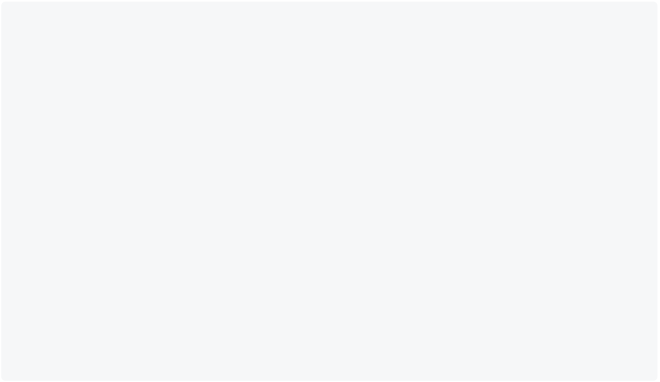
}

[Run Online](https://dartpad.dev/?id=d625da929c5550ba0f65397010922653)

**Example 3: Display Sum of n Natural Numbers Using While Loop**

Here, the value of the total is 0 initially. Then, the while loop is iterated from **i = 1 to 100**. In each iteration, **i** is added to the total, and the value of **i** is increased by 1. Result is **1+2+3+….+99+100**.

void main(){



int total = 0;

int n = 100; // change as per required int i =1;

while(i<=n){

total = total + i; i++;

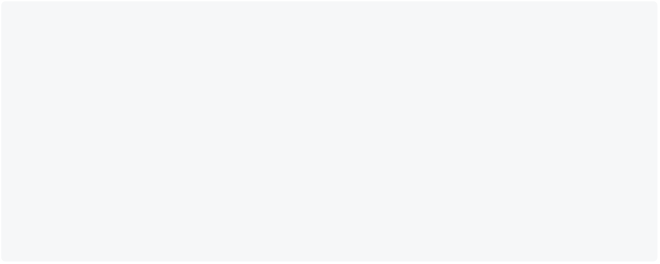
}

print("Total is $total"); }

[Run Online](https://dartpad.dev/?id=d7212f76e2c02a141c001b86136f92fd)

**Example 4: Display Even Numbers Between 50 to 100 Using While Loop** This program will print even numbers between 50 to 100 using while loop.

void main(){



int i = 50;

while(i<=100){ if(i%2 == 0){ print(i); }

i++;

}

}

[Run Online](https://dartpad.dev/?id=88474097465d8039d38c35f68f886a70)

### Do While Loop in Dart

Do while loop is used to run a block of code multiple times. The loop’s body will be executed first, and then the condition is tested. The syntax of do while loop is:

do{



statement1;

statement2;

.

.

.

statementN; }while(condition);

* First, it runs statements, and finally, the condition is checked.
  + If the condition is true, the code inside {} is executed.
    - The condition is re-checked until the condition is false.
      * When the condition is false, the loop stops.

**Note**: In a do-while loop, the statements will be executed at least once time, even if the condition is false. It is because the statement is executed before checking the condition.

**Example 1: To Print 1 To 10 Using Do While Loop**

void main() { int i = 1; do {



print(i);



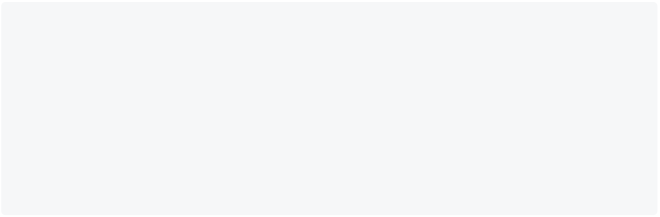
i++;

} while (i <= 10); }

[Run Online](https://dartpad.dev/?id=3ed5f1b22915886443ecc157480e6bbc)

**Example 2: To Print 10 To 1 Using Do While Loop**

void main() {



int i = 10;

do {

print(i);

i--;

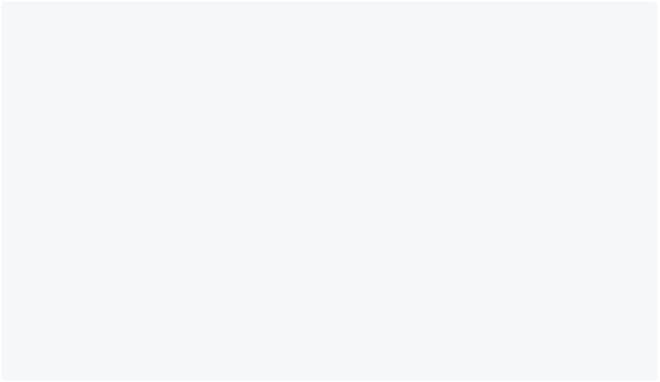
} while (i >= 1); }

[Run Online](https://dartpad.dev/?id=5f8af035a8fa74c794c82bbaa589d03a)

**Example 3: Display Sum of n Natural Numbers Using Do While Loop**

Here, the value of the **total** is 0 initially. Then, the do-while loop is iterated from **i = 1 to 100**. In each iteration, **i** is added to the total, and the value of **i** is increased by 1. Result is **1+2+3+….+99+100**.

void main(){



int total = 0;

int n = 100; // change as per required int i =1;

do{

total = total + i; i++;

}while(i<=n);

print("Total is $total"); }

[Run Online](https://dartpad.dev/?id=491f3761c6e018e314945490efc86bfb)

**When The Condition Is False**

Let’s make one condition false and see the demo below. **Hello** got printed if the condition is false.

void main(){



int number = 0;

do{

print("Hello"); number--;

}while(number >1);

}

[Run Online](https://dartpad.dev/?id=7c3c2a8a4e750a3f2a8169f86d5d04e4)

### Break and Continue in Dart

In this tutorial, you will learn about the **break and continue** in dart. While working on loops, we need to skip some elements or terminate the loop immediately without checking the condition. In such a situation, you can use the break and continue statement.

**Break Statement**

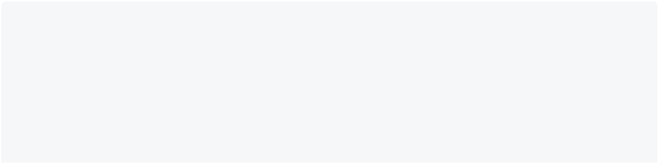
Sometimes you will need to break out of the loop immediately without checking the condition. You can do this using break statement.

The break statement is used to exit a loop. It stops the loop immediately, and the program’s control moves outside the loop. Here is syntax of break:

**Example 1: Break In Dart For Loop**

Here, the loop condition is true until the value of i is less than or equal to 10. However, the break says to go outside the loop when the value of i becomes 5.

void main() {



for (int i = 1; i <= 10; i++) { if (i == 5) {

break;

}

print(i);

} }

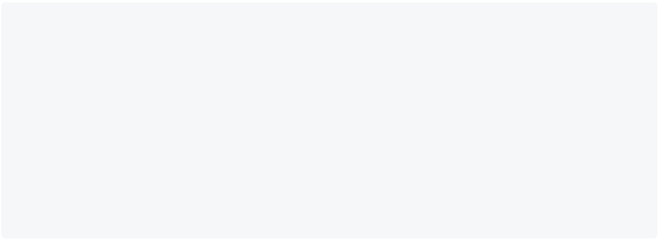


[Run Online](https://dartpad.dev/?id=50ac9046a365fef6dde49b5db51619df)

**Example 2: Break In Dart Negative For Loop**

Here, the loop condition is true until the value of i is more than or equal to 1. However, the break says to go outside the loop when the value of i becomes 7.

void main() {



for (int i = 10; i >= 1; i--) { if (i == 7) {

break;

}

print(i);

}

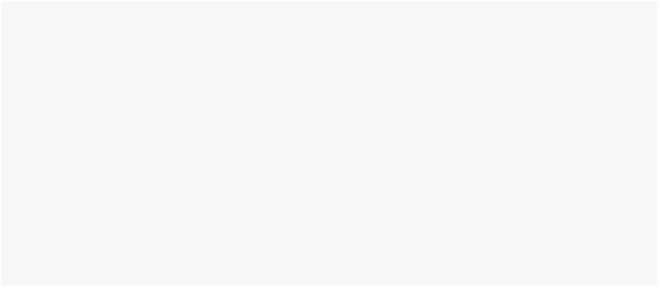
}

[Run Online](https://dartpad.dev/?id=3018972104eb8b763dfc0963ab20f378)

**Example 3: Break In Dart While Loop**

Here, this while loop condition is true until the value of i is less than or equal to 10. However, the break says to go outside the loop when the value of i becomes 5.

void main() {



int i =1;

while(i<=10){ print(i);

if (i == 5) { break;

}

i++;

}

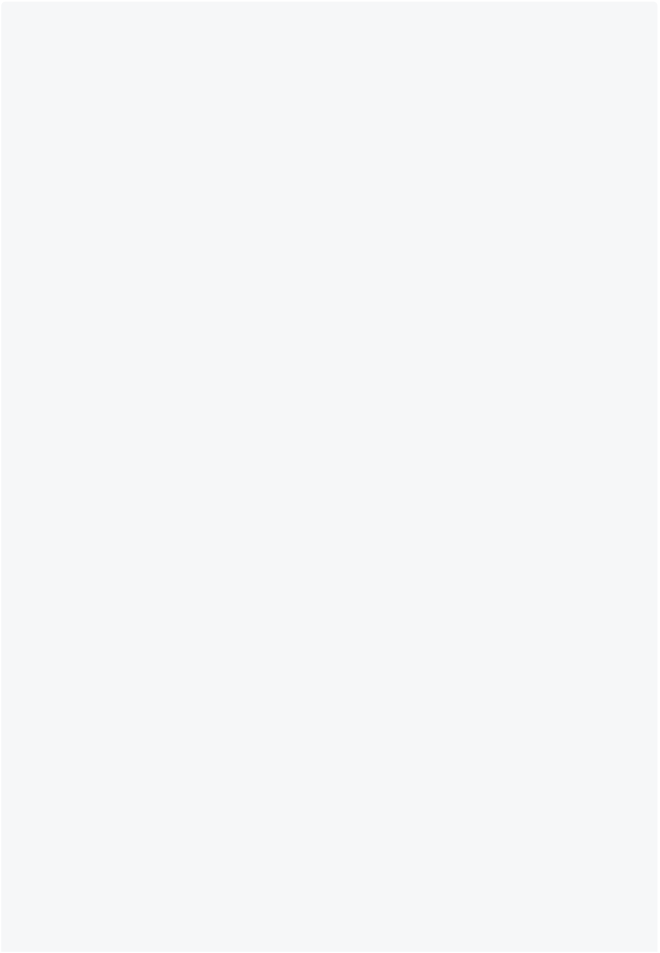
}

[Run Online](https://dartpad.dev/?id=fc56b90939f7b2f0071143cc57108b94)

**Example 4: Break In Switch Case**

As we already learn in dart switch case, it is important to add **break** keyword in switch statement. This example prints the month name based on the number of the month using a switch case.

void main() {



var noOfMoneth = 5;

switch (noOfMoneth) {

case 1:

print("Selected month is January."); break;

case 2:

print("Selected month is February."); break;

case 3:

print("Selected month is march.");

break;

case 4:

print("Selected month is April.");

break;

case 5:

print("Selected month is May.");

break;

case 6:

print("Selected month is June.");

break;

case 7:

print("Selected month is July.");

break;

case 8:

print("Selected month is August.");

break;

case 9:

print("Selected month is September."); break;

case 10:

print("Selected month is October."); break;

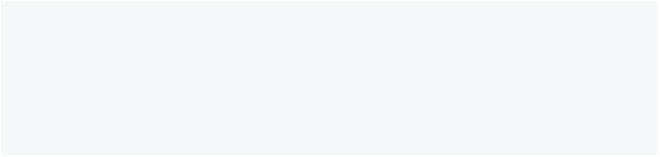
case 11:

print("Selected month is November."); break;

case 12:

print("Selected month is December."); break;

default:



print("Invalid month."); break;

}

}

[Run Online](https://dartpad.dev/?id=089433333e62977d5957c4b3d2882930) **Continue Statement**

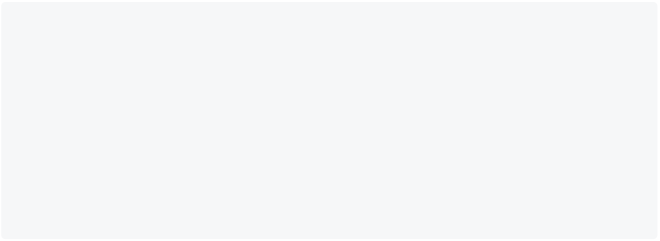
Sometimes you will need to skip an iteration for a specific condition. You can do this utilizing continue statement.

The continue statement skips the current iteration of a loop. It will bypass the statement of the loop. It does not terminate the loop but rather continues with the next iteration. Here is the syntax of continue statement:

**Example 1: Continue In Dart**

Here, the loop condition is true until the value of i is less than or equal to 10. However, the continue says to go to the next iteration of the loop when the value of i becomes 5.

void main() {



for (int i = 1; i <= 10; i++) { if (i == 5) {

continue;

}

print(i);

}

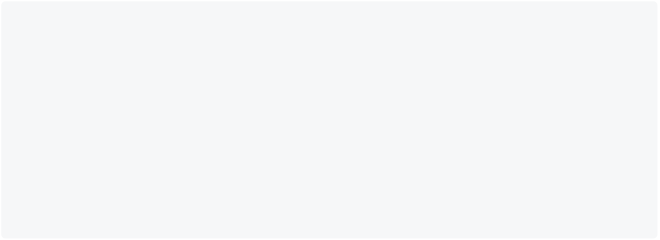
}

[Run Online](https://dartpad.dev/?id=c5f2878678e363a80626ba578c53bdb9)

**Example 2: Continue In For Loop Dart**

Here, the loop condition is true until the value of i is more than or equal to 1. However, the continue says to go to the next iteration of the loop when the value of i becomes 4.

void main() {



for (int i = 10; i >= 1; i--) { if (i == 4) {

continue;

}

print(i);

}

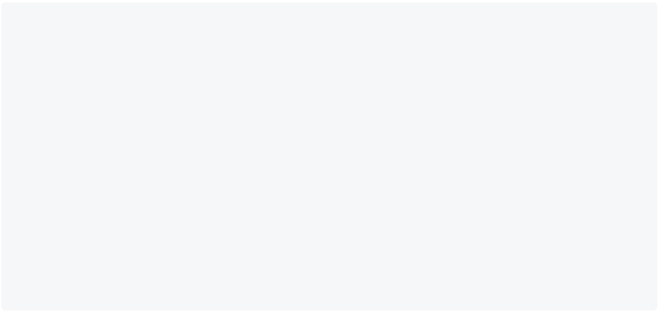
}

[Run Online](https://dartpad.dev/?id=822da2a7cb4e2b6fd4bae2c249c7aa52)

**Example 3: Continue In Dart While Loop**

Here, this while loop condition is true until the value of i is less than or equal to 10. However, the continue says to go to the next iteration of the loop when the value of i becomes 5.

void main() {



int i = 1;

while (i <= 10) { if (i == 5) { i++;

continue;

}

print(i);

i++;

}

}

[Run Online](https://dartpad.dev/?id=74b1837417fe0f7f53207072e58f45d1)

## Exception Handling in Dart

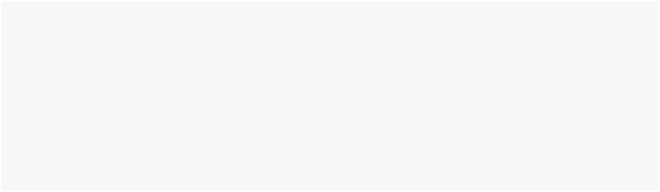
### Exception In Dart

An exception is an error that occurs at runtime during program execution. When the exception occurs, the flow of the program is interrupted, and the program terminates abnormally. There is a high chance of crashing or terminating the program when an exception occurs. Therefore, to save your program from crashing, you need to catch the exception.

**Note**: If you are attempting a task that might result in an error, it’s a good habit to use the try-catch statement.

**Syntax**

try {



// Your Code Here }

catch(ex){

// Exception here }

### Try & Catch In Dart

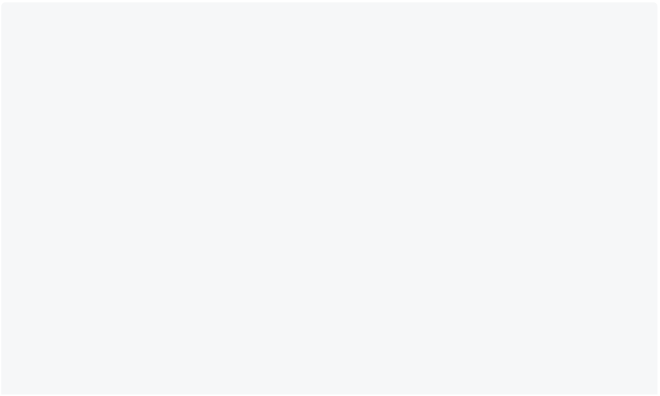
**Try** You can write the logical code that creates exceptions in the try block.

**Catch** When you are uncertain about what kind of exception a program produces, then a catch block is used. It is written with a try block to catch the general exception.

**Example 1: Try Catch In Dart**

In this example, you will see how to handle the exception using the try-catch block.

void main() { int a = 18;



int b = 0; int res;

try {

res = a ~/ b;

print("Result is $res");

}

// It returns the built-in exception related to the occurring exception

catch(ex) {

print(ex);

}

}



[Run Online](https://dartpad.dev/?id=fbe990056aa6798dd04a0c4d1cd38dc3)

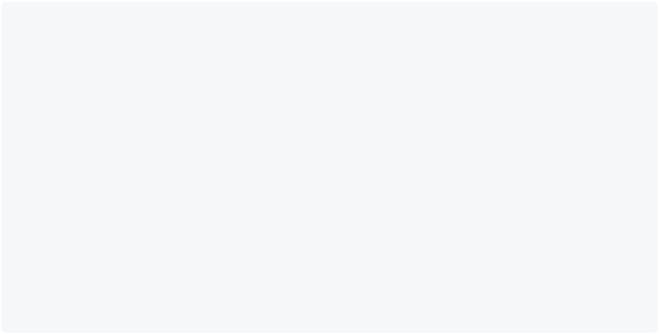
### Finally In Dart

The **finally** block is always executed whether the exceptions occur or not. It is optional to include the final block, but if it is included, it should be after the try and catch block is over.

**On** block is used when you know what types of exceptions are produced by the program.

**Syntax**

try {



.....

}

on Exception1 {

....

}

catch Exception2 {

....

}

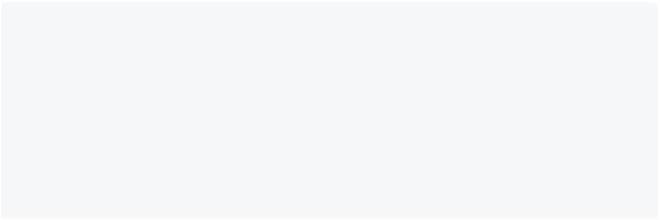
finally {

// code that should always execute whether an exception or not. }

**Example 2: Finally In Dart Try Catch**

In this example, you will see how to handle the exception using the try-catch block with the finally block.

void main() {



int a = 12;

int b = 0;

int res;

try {

res = a ~/ b;

} on UnsupportedError {

print('Cannot divide by zero');

} catch (ex) {



print(ex);

} finally {

print('Finally block always executed'); }

}

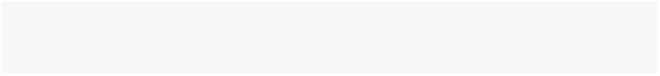
[Run Online](https://dartpad.dev/?id=76649da1be9329e5adb737ab6537b02e)

### Throwing An Exception

The throw keyword is used to raise an exception explicitly. A raised exception should be handled to prevent the program from exiting unexpectedly.

**Syntax**

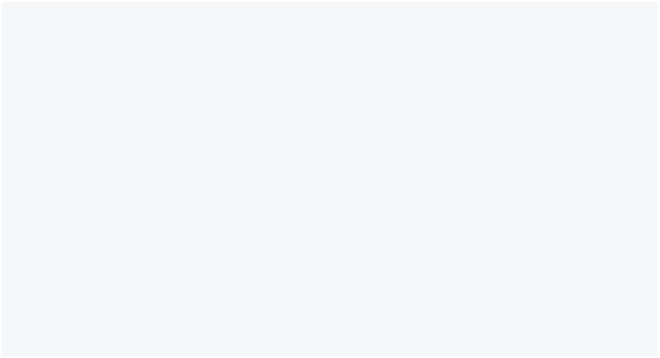
throw new Exception\_name()



**Example 3: Throwing An Exception**

In this example, you will see how to throw an exception using the throw keyword.

void main() {



try {

check\_account(-10);

} catch (e) {

print('The account cannot be negative'); }

}

void check\_account(int amount) {

if (amount < 0) {

throw new FormatException(); // Raising explanation externally }

}

[Run Online](https://dartpad.dev/?id=bf9e42c2a20df5759490c0e7ee5b1d27)

### Why Is Exception Handling Needed?

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. Therefore, exceptions must be handled to prevent the application from unexpected termination. Here are some reasons why exception handling is necessary:

* To avoid abnormal termination of the program.
  + To avoid an exception caused by logical error.
    - To avoid the program from falling apart when an exception occurs.
      * To reduce the vulnerability of the program.
        + To maintain a good user experience.

To try providing aid and some debugging in case of an exception.

### How To Create Custom Exception In Dart

As you go advance, you need to create your exception; Dart enables you to create your exception.

**Syntax**

class YourExceptionClass implements Exception{ // constructors, variables & methods

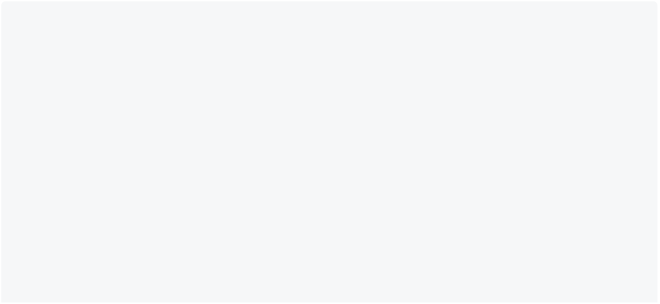


}

**Example 4: How to Create & Handle Exception**

This program throws an exception when a student’s mark is negative. You will understand **implements** in the object-oriented programming section.

class MarkException implements Exception {



String errorMessage() {

return 'Marks cannot be negative value.'; }

}

void main() {

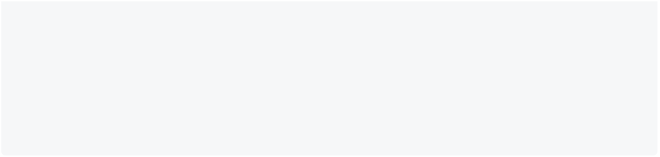
try {

checkMarks(-20);

} catch (ex) {

print(ex.toString());

}



void checkMarks(int marks) {

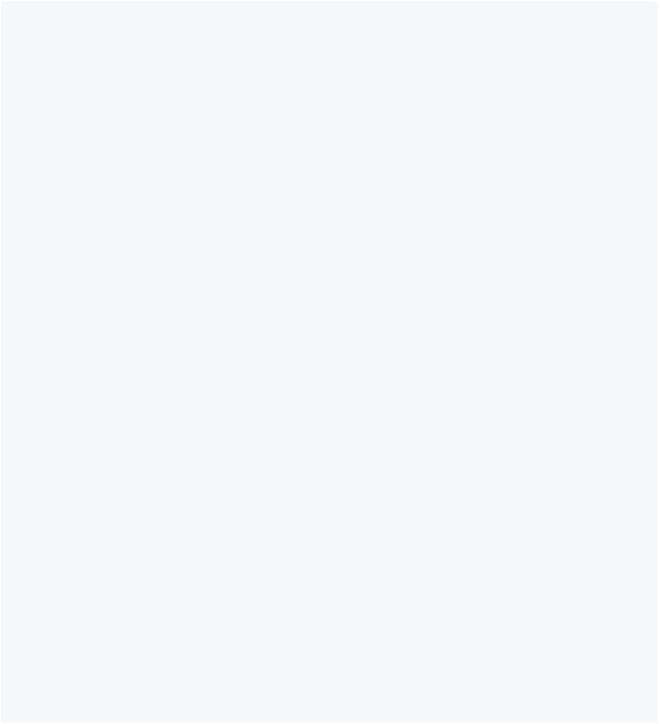
if (marks < 0) throw MarkException().errorMessage(); }

[Run Online](https://dartpad.dev/?id=38e624859b1671f840556ce827140079)

**Example 5: How to Create & Handle Exception**

This program throws an exception when you find the square root of a negative number.

import 'dart:math';



// custom exception class

class NegativeSquareRootException implements Exception {

@override

String toString() {

return 'Sqauare root of negative number is not allowed here.'; }

}

// get square root of a positive number

num squareRoot(int i) {

if (i < 0) {

// throw `NegativeSquareRootException` exception throw NegativeSquareRootException();

} else {

return sqrt(i);

}

}

void main() {

try {

var result = squareRoot(-4);

print("result: $result");

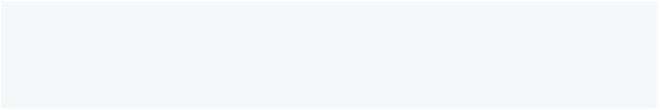
} on NegativeSquareRootException catch (e) { print("Oops, Negative Number: $e");

} catch (e) {

print(e);

} finally {

print('Job Completed!'); }



}

[Run Online](https://dartpad.dev/?id=10e7acfcc31f524a1e7d45d2b768df36)

## Functions in Dart

### Functions in Dart

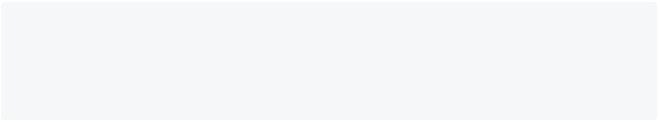
**Functions** are the block of code that performs a specific task. They are created when some statements are repeatedly occurring in the program. The function helps reusability of the code in the program.

**Note**: The main objective of the function is **DRY(Don’t Repeat Yourself)**. **Function Advantages**

* Avoid Code Repetition
  + Easy to divide the complex program into smaller parts
    - Helps to write a clean code

**Syntax**

returntype functionName(parameter1,parameter2, ...){ // function body



}

**Return type**: It tells you the function output type. It can be void, String, int, double, etc. If the function doesn’t return anything, you can use void as the return type.

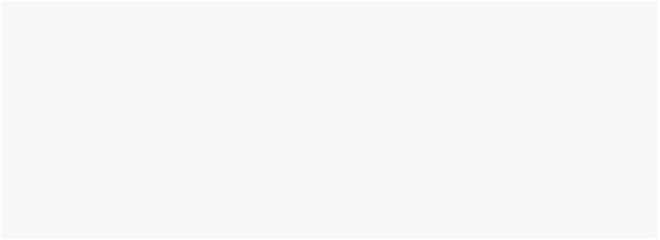
**Function Name**: You can name functions by almost any name. Always follow a lowerCamelCase naming convention like void printName().

**Parameters**: Parameters are the input to the function, which you can write inside the bracket (). Always follow a lowerCamelCase naming convention for your function parameter.

**Example 1: Function That Prints Name**

This is a simple program that prints name using function. The name of function is **printName()**.

// writing function outside main function.



void printName(){

print("My name is Raj Sharma. I am from function."); }

// this is our main function.

void main(){

printName();

}

[Run Online](https://dartpad.dev/?id=1a1843e8361cdf3ff84007aa35a716dc)

**Example 2: Function To Find Sum of Two Numbers**

This function finds the sum of two numbers. Here, the function accepts two parameters. i.e., **num1 and num2**, and the return type is void.

void add(int num1, int num2){ int sum = num1 + num2;



print("The sum is $sum"); }

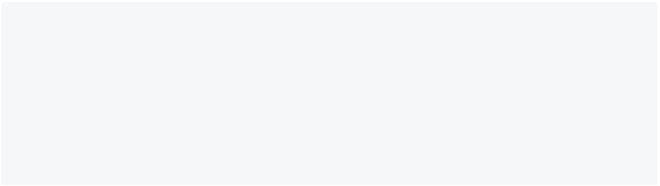
void main(){ add(10, 20); }

[Run Online](https://dartpad.dev/?id=751cd393eb50f6be93ad6c5469275e49)

**Example 3: Function That Find Simple Interest**

This function finds simple interest from principal, time and rate and display result.

// function that calculate interest



void calculateInterest(double principal, double rate, double time) { double interest = principal \* rate \* time / 100;

print("Simple interest is $interest");

}

void main() {

double principal = 5000;



double time = 3;

double rate = 3;

calculateInterest(principal, rate, time); }

[Run Online](https://dartpad.dev/?id=7650d5a07147369ce515a5ca809560b7) **Key Points**

* In dart function are also objects.
  + You should follow the **lowerCamelCase** naming convention while naming function.
    - You should follow the **lowerCamelCase** naming convention while naming function parameters.

**About lowerCamelCase**

Name should start with lower-case, and every second word’s first letter will be upper-case like num1, fullName, isMarried, etc. Technically, this naming convention is called lowerCamelCase.

**Function Parameters Vs Arguments**

Many programmers are often confused about parameters and arguments. Let’s have a look at this example.

// Here num1 and num2 are parameters void add(int num1, int num2){



int sum;

sum = num1 + num2;

print("The sum is $sum"); }

void main(){

// Here 10 and 20 are arguments add(10, 20);

}

[Run Online](https://dartpad.dev/?id=b869453c5a183426f4e3414ce3f15ede)

* Here in **add(int num1, int num2)**, num1 and num2 are parameters and in **add(10, 20)**, 10 and 20 are arguments.
  + Parameter is the name and data type you define as an input for your function.
    - Argument is the actual value that you passed in.

**Note**: In dart, if you don’t write the return type of function. It will automatically understand.

### Types of Functions in Dart

**Functions** are the block of code that performs a specific task. Here are different types of functions:

* No Parameter And No Return Type
  + Parameter And No Return Type
    - No Parameter And Return Type
      * Parameter And Return Type

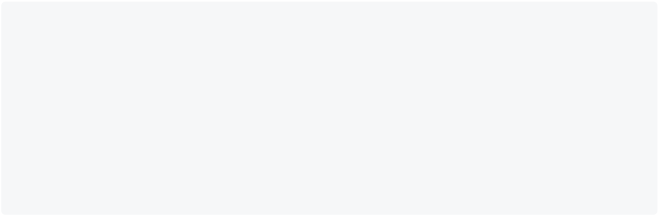
**Function With No Parameter And No Return Type**

In this function, you do not pass any parameter and expect no return type. Here is an example of it:

**Example 1: No Parameter & No Return Type**

Here **printName()** is a function which prints name on screen.

void main() { printName(); }



void printName() {

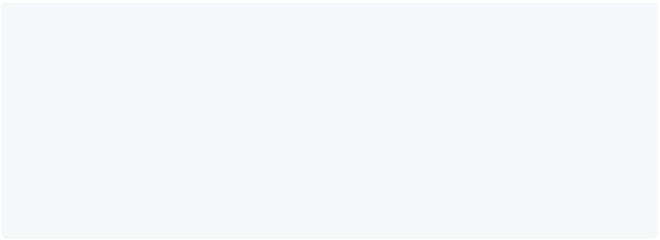
print("My name is John Doe."); }

[Run Online](https://dartpad.dev/?id=d7894e31281ec69f141433769c81e7d7)

In this program, **printName()** is the function which has keyword **void**. It means it has **no return type**, and the empty pair of parentheses implies that there is **no parameter** that is passed to the function.

Here **printPrimeMinisterName()** is a function which prints prime minister name on screen.

void main() {



print("Function With No Parameter and No Return Type"); printPrimeMinisterName();

}

void printPrimeMinisterName() { print("John Doe.");

}

[Run Online](https://dartpad.dev/?id=220db13fab191bff038d75c3effafc63)

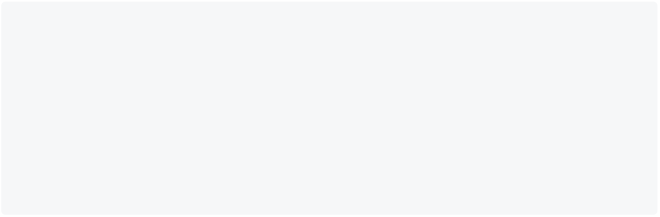
**Function With Parameter And No Return Type**

In this function, you do pass the parameter and expect no return type. Here is an example of it:

**Example 1: Parameter & No Return Type**

Here **printName(String name)** is a function which welcome person.

void main() {



printName("John"); }

void printName(String name) { print("Welcome, ${name}."); }

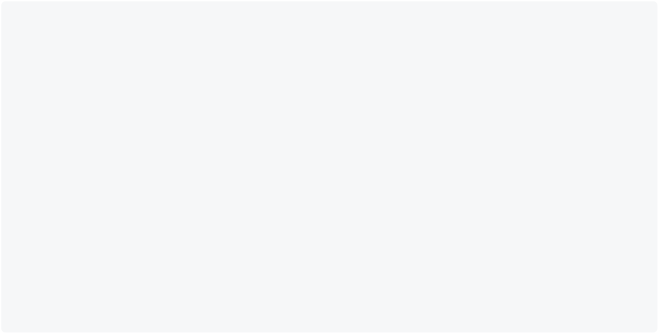
[Run Online](https://dartpad.dev/?id=ad0b462f855964a5832b9cca0a006456)

In this program, **printName(String name)** is the function which has keyword **void**. It means it has **no return type**, and the pair of parentheses is not empty but this time that suggests it to accept an **parameter**.

**Example 2: Parameter & No Return Type**

Here **add(int a, int b)** is a function that finds and prints the sum of two numbers.

// This function add two numbers void add(int a, int b) {



int sum = a + b;

print("The sum is $sum");

}

void main() {

int num1 = 10; int num2 = 20;

add(num1, num2); }

[Run Online](https://dartpad.dev/?id=9b3295128019ab7b80fc78b0f0c03fbf)

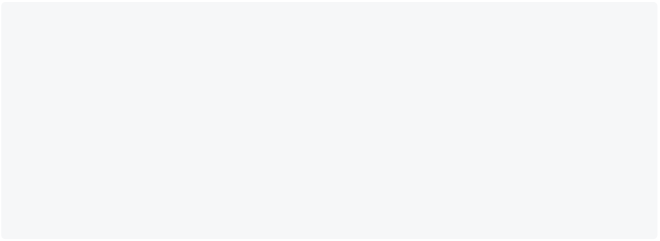
**Function With No Parameter And Return Type**

In this function, you do not pass any parameter but expect return type. Here is an example of it:

**Example 1: No Parameter & Return Type**

Here **primeMinisterName()** is a function which returns prime minister name. In the entire program, anyone can use this function to find the name of the prime minister.

void main() {



// Function With No Parameter & Return Type String name = primeMinisterName();

print("The Name from function is $name."); }

String primeMinisterName() {

return "John Doe";

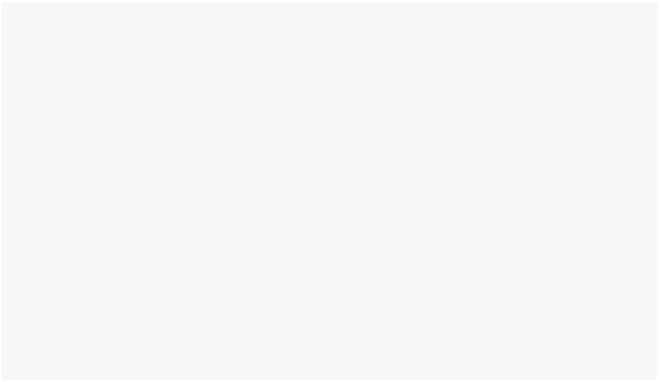
}

[Run Online](https://dartpad.dev/?id=c96e8149702197004a330b4b558a048e)

In this program, **primeMinisterName()** is the function which has **String** keyword before function name, means it **return** String value, and the empty pair of parentheses suggests that there is **no parameter** that is passed to the function.

Here **voterAge()** is a function which returns minimum voter age.

// Function With No Parameter & Return Type void main() {



int personAge = 17;

if (personAge >= voterAge()) {

print("You can vote.");

} else {

print("Sorry, you can't vote."); }

}

int voterAge() { return 18;

}

[Run Online](https://dartpad.dev/?id=0114d061a25e1e26b8d3bf105b24cb29)

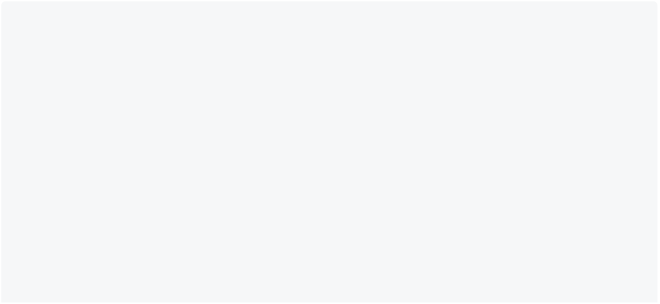
**Function With Parameter And Return Type**

In this function, you do pass the parameter and also expect return type. Here is an example of it:

**Example 1: Parameter & Return Type**

Here **add(int a, int b)** is a function that returns its sum in integer. We can display results in our main function.

// this function add two numbers int add(int a, int b) {



int sum = a + b;

return sum;

}

void main() {

int num1 = 10; int num2 = 20;

int total = add(num1, num2); print("The sum is $total.");

}



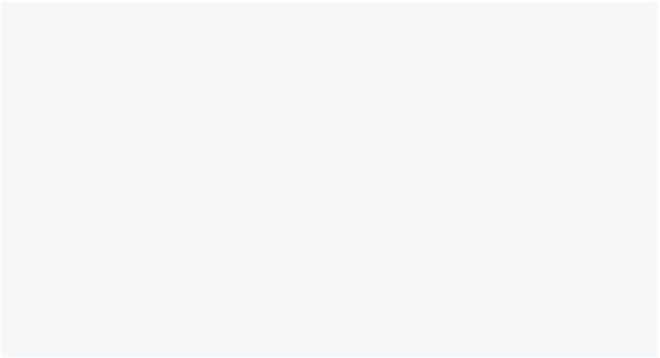
[Run Online](https://dartpad.dev/?id=f9631f623653b1990513c8e7698509e2)

In this program, **int add(int a, int b)** is the function with **int** as the return type, and the pair of parenthesis has two **parameters**, i.e., a and b.

**Example 2: Parameter & Return Type**

Here **calculateInterest(double principal, double rate, double time)** is a function that returns its simple interest in double. We can display results in our main function.

// function that calculate interest



double calculateInterest(double principal, double rate, double time) { double interest = principal \* rate \* time / 100;

return interest;

}

void main() {

double principal = 5000;

double time = 3;

double rate = 3;

double result = calculateInterest(principal, rate, time); print("The simple interest is $result.");

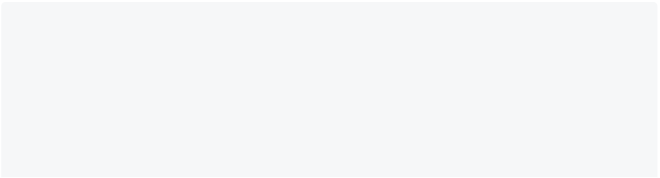
}

[Run Online](https://dartpad.dev/?id=da88ac059ff1947abe46bced02f0af0b)

**Note**: void is used for no return type as it is a non value-returning function. **\*\*Complete Example \*\***

Here is the program, which includes all types of functions we studied earlier.

// parameter and return type int add(int a, int b) {



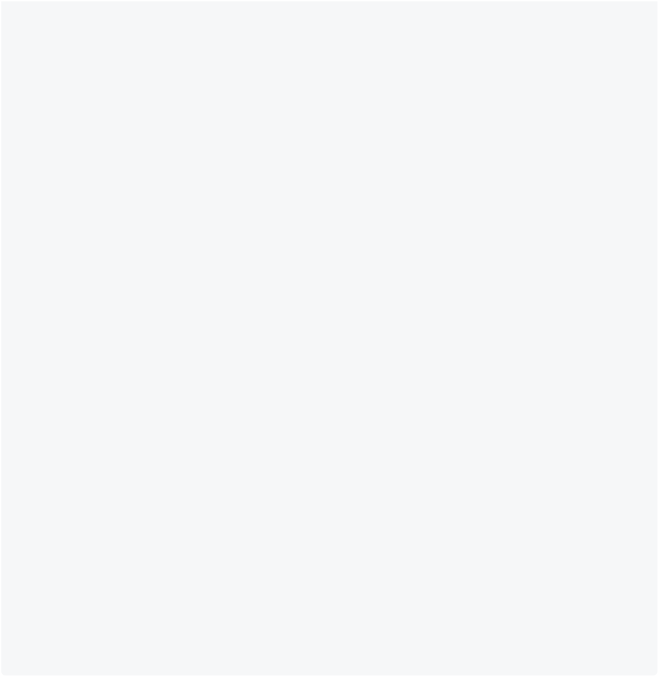
var total;

total = a + b;

return total;

}

// parameter and no return type



void mul(int a, int b) {

var total;

total = a \* b;

print("Multiplication is : $total"); }

// no parameter and return type String greet() {

String greet = "Welcome";

return greet;

}

// no parameter and no return type void greetings() {

print("Hello World!!!");

}

void main() {

var total = add(2, 3);

print("Total sum: $total"); mul(2, 3);

var greeting = greet();

print("Greeting: $greeting"); greetings();

}

[Run Online](https://dartpad.dev/?id=acf8ce1d1d325f6783bbce80f7c6b14b)

### Function Parameter

**Parameter In Dart**

The parameter is the process of passing values to the function. The values passed to the function must match the number of parameters defined. A function can have any number of parameters.

// here a and b are parameters void add(int a, int b) {



}

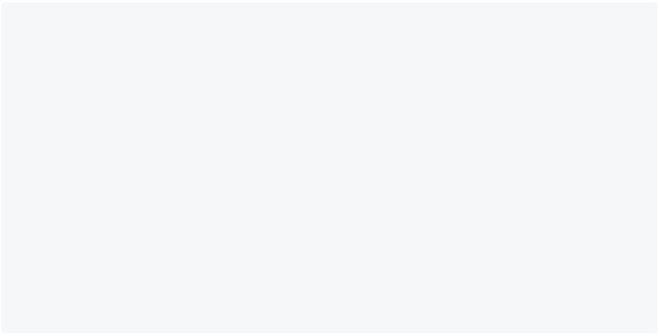
**Positional Parameter In Dart**

In positional parameters, you must supply the arguments in the same order as you defined on parameters when you wrote the function. If you call the function with the parameter in the wrong order, you will get the wrong result.

**Example 1: Use Of Positional Parameter**

In the example below, the function **printInfo** takes two parameters. You must pass the person’s name and gender in the same order. If you pass values in the wrong order, you will get the **wrong result**.

void printInfo(String name, String gender) {



print("Hello $name your gender is $gender."); }

void main() {

// passing values in wrong order printInfo("Male", "John");

// passing values in correct order printInfo("John", "Male");

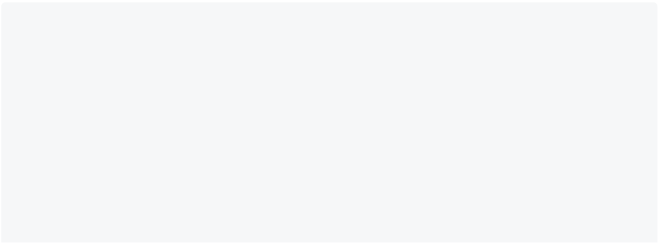
}

[Run Online](https://dartpad.dev/?id=8d5bdeb2d1eae817658f9fd141b00655)

**Example 2: Providing Default Value On Positional Parameter**

In the example below, function **printInfo** takes two positional parameters and one optional parameter. The title parameter is optional here. If the user doesn’t pass the title, it will automatically set the title value to **sir/ma’am**.

void printInfo(String name, String gender, [String title = "sir/ma'am"]) {



print("Hello $title $name your gender is $gender.");

}

void main() {

printInfo("John", "Male");

printInfo("John", "Male", "Mr.");

printInfo("Kavya", "Female", "Ms.");

}

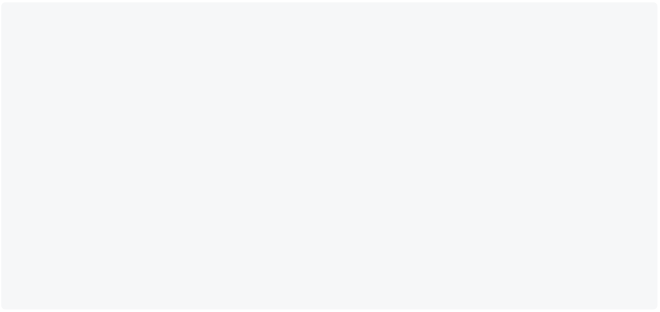


[Run Online](https://dartpad.dev/?id=965979f3c88c980a0b7454b458f50eb5)

**Example 3: Providing Default Value On Positional Parameter**

In the example below, function **add** takes two positional parameters and one optional parameter. The **num3** parameter is **optional** here with default value **0**.

void add(int num1, int num2, [int num3=0]){ int sum;



sum = num1 + num2 + num3;

print("The sum is $sum"); }

void main(){

add(10, 20);

add(10, 20, 30); }

[Run Online](https://dartpad.dev/?id=b59f4dbc5355d207ce289fd2593359a4)

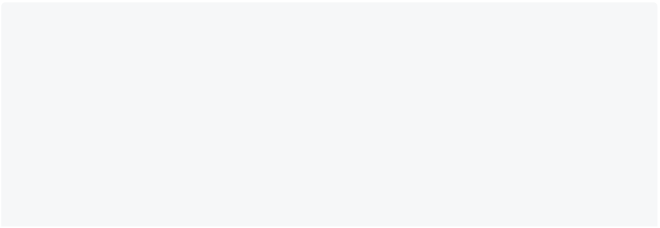
**Named Parameter In Dart**

Dart allows you to use named parameters to clarify the parameter’s meaning in function calls. **Curly braces {}** are used to specify named parameters.

**Example 1: Use Of Named Parameter**

In the example below, function **printInfo** takes two named parameters. You can pass value in any order. You will learn about **?** in **null safety** section.

void printInfo({String? name, String? gender}) { print("Hello $name your gender is $gender."); }



void main() {

// you can pass values in any order in named parameters. printInfo(gender: "Male", name: "John");

printInfo(name: "Sita", gender: "Female");

printInfo(name: "Reecha", gender: "Female"); printInfo(name: "Reecha", gender: "Female"); printInfo(name: "Harry", gender: "Male");



printInfo(gender: "Male", name: "Santa");

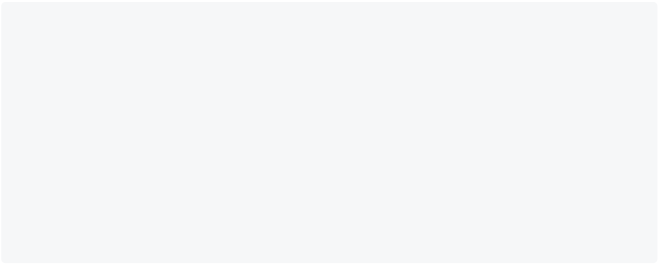
}

[Run Online](https://dartpad.dev/?id=6a8792f6f343fd5062b28bc83f3eddfd)

**Example 2: Use Of Required In Named Parameter**

In the example below, function **printInfo** takes two named parameters. You can see a **required** keyword, which means you must pass the person’s name and gender. If you don’t pass it, it won’t work.

void printInfo({required String name, required String gender}) { print("Hello $name your gender is $gender.");



}

void main() {

// you can pass values in any order in named parameters. printInfo(gender: "Male", name: "John");

printInfo(gender: "Female", name: "Suju");

}

[Run Online](https://dartpad.dev/?id=b2ce48d6cd282a9d1113b59b0d71c976)

**Note**: You can pass the value in any order in the named parameter. **?** is used to remove null safety, which we will discuss in the coming chapter.

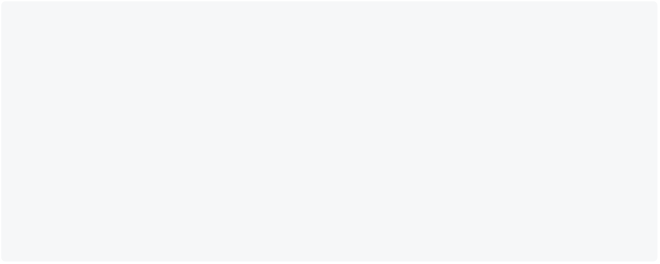
**Optional Parameter In Dart**

Dart allows you to use optional parameters to make the parameter optional in function calls. **Square braces []** are used to specify optional parameters.

**Example: Use Of Optional Parameter**

In the example below, function **printInfo** takes two **positional parameters** and one **optional parameter**. First, you must pass the person’s name and gender. The title parameter is optional here. Writing **[String? title]** makes **title** optional.

void printInfo(String name, String gender, [String? title]) { print("Hello $title $name your gender is $gender.");



}

void main() {

printInfo("John", "Male");

printInfo("John", "Male", "Mr.");

printInfo("Kavya", "Female", "Ms."); }

[Run Online](https://dartpad.dev/?id=ec0d52e0f1d15f7e9aeebe1a8f9bcb01)

**Anonymous Function in Dart**

This tutorial will teach you the anonymous function and how to use it. You already saw function like **main()**, **add()**, etc. These are the **named** functions, which means they have a certain name.

But not every function needs a name. If you remove the return type and the function name, the function is called **anonymous function**.

**Syntax**

Here is the syntax of the anonymous function.

(parameterList){ // statements

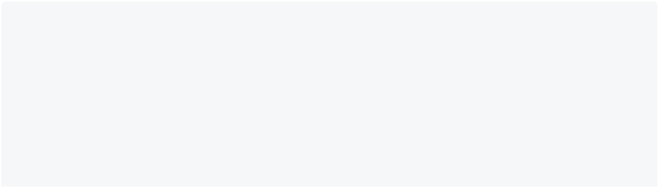


}

**Example 1: Anonymous Function In Dart**

In this example, you will learn to use an anonymous function to print all list items. This function invokes each fruit without having a function name.

void main() {



const fruits = ["Apple", "Mango", "Banana", "Orange"];

fruits.forEach((fruit) { print(fruit);

});

}

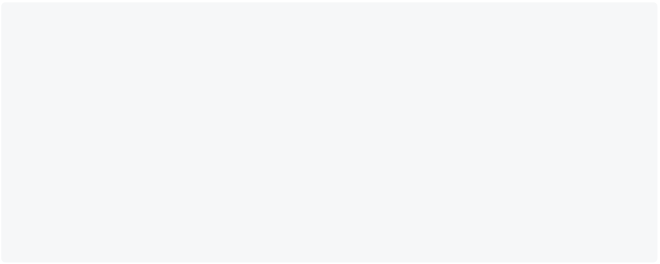


[Run Online](https://dartpad.dev/?id=eed29975b352c0f1f61760dc0c0d6bfe)

**Example 2: Anonymous Function In Dart**

In this example, you will learn to find the cube of a number using an anonymous function.

void main() {



// Anonymous function

var cube = (int number) {

return number \* number \* number; };

print("The cube of 2 is ${cube(2)}"); print("The cube of 3 is ${cube(3)}"); }

[Run Online](https://dartpad.dev/?id=b57bd7884e109904eaf227e53148c464)

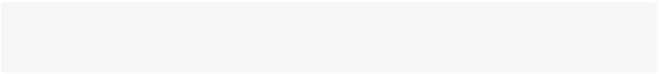
### Arrow Function in Dart

Dart has a special syntax for the function body, which is only one line. The arrow function is represented by **=>** symbol. It is a shorthand syntax for any function that has only one expression.

**Syntax**

The syntax for the dart arrow function.

returnType functionName(parameters...) => expression;

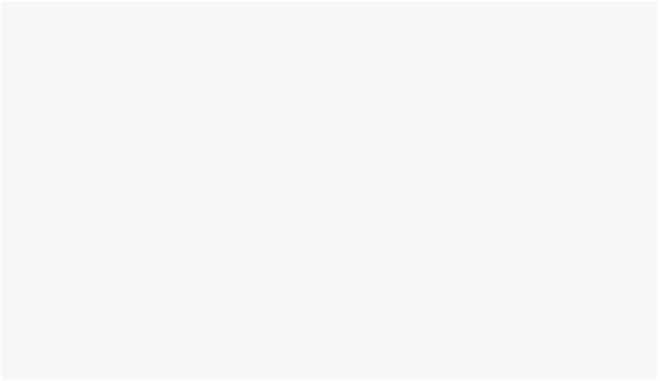


**Note**: The arrow function is used to make your code short.**=> expr** syntax is a shorthand for **{ return expr; }**.

**Example 1: Simple Interest Without Arrow Function**

This program finds simple interest without using the arrow function.

// function that calculate interest



double calculateInterest(double principal, double rate, double time) { double interest = principal \* rate \* time / 100;

return interest;

}

void main() {

double principal = 5000; double time = 3;

double rate = 3;

double result = calculateInterest(principal, rate, time); print("The simple interest is $result.");

}

[Run Online](https://dartpad.dev/?id=7766c775f91c378bb5d0efec23ecb183)

**Example 2: Simple Interest With Arrow Function**

This program finds simple interest using the arrow function.

// arrow function that calculate interest



double calculateInterest(double principal, double rate, double time) => principal \* rate \* time / 100;

void main() {

double principal = 5000; double time = 3;

double rate = 3;

double result = calculateInterest(principal, rate, time); print("The simple interest is $result.");

}

[Run Online](https://dartpad.dev/?id=657009baa484bbaac0bc4f92adb2f7f2)

**Example 3: Simple Calculation Using Arrow Function**

This program finds the sum, difference, multiplication, and division of two numbers using the arrow function.

int add(int n1, int n2) => n1 + n2; int sub(int n1, int n2) => n1 - n2; int mul(int n1, int n2) => n1 \* n2; double div(int n1, int n2) => n1 / n2;



void main() {

int num1 = 100; int num2 = 30;

print("The sum is ${add(num1, num2)}"); print("The diff is ${sub(num1, num2)}"); print("The mul is ${mul(num1, num2)}"); print("The div is ${div(num1, num2)}"); }

[Run Online](https://dartpad.dev/?id=516bd6e463ff93ead5edf7b69d7b3e2e) Scope in Dart

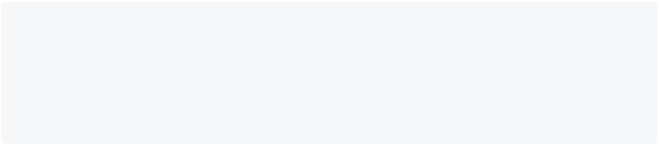
The scope is a concept that refers to where values can be accessed or referenced. Dart uses curly braces **{}** to determine the scope of variables. If you define a variable inside curly braces, you can’t use it outside the curly braces.

**Method Scope**

If you created variables inside the method, you can use them inside the method block but not outside the method block.

**Example 1: Method Scope**

void main() {



String text = "I am text inside main. Anyone can't access me."; print(text);

}

[Run Online](https://dartpad.dev/?id=1d97729350575e25c79cbdf8b5ef59b1)

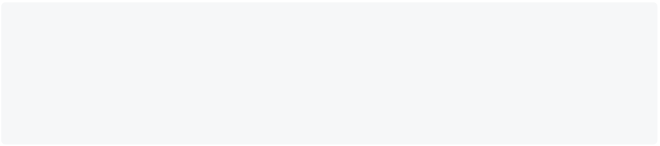
In this program, **text** is a String type where you can access and print method only inside the main function but not outside the main function.

**Global Scope**

You can define a variable in the global scope to use the variable anywhere in your program.

**Example 1: Global Scope**

String global = "I am Global. Anyone can access me."; void main() {



print(global);

}

[Run Online](https://dartpad.dev/?id=5cb7155e0cfe835bce865764a1da3674)

In this program, the variable named **global** is a top-level variable; you can access it anywhere in the program.

Info

**Note**: Define your variable as much as close **Local** as you can. It makes your code clean and prevents you from using or changing them where you shouldn’t.

**Lexical Scope**

Dart is lexically scoped language, which means you can find the scope of variables with the help of **braces {}**.

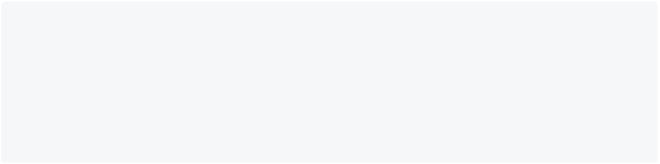
### Math in Dart

Math helps you to perform mathematical calculations efficiently. With dart math, you can **generate random number**, **find square root**, **find power of number**, or **round specific numbers**. To use math in dart, you must import 'dart:math';.

**How To Generate Random Numbers In Dart**

This example shows how to generate random numbers from **0 - 9** and also **1 to 10**. After watching this example, you can generate a random number between your choices.

import 'dart:math'; void main()

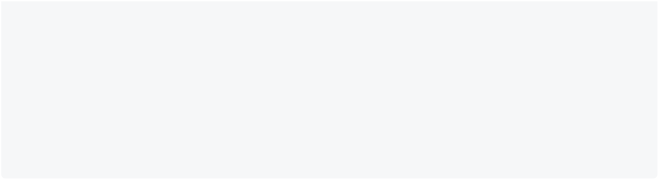


{

Random random = new Random();

int randomNumber = random.nextInt(10); // from 0 to 9 included print("Generated Random Number Between 0 to 9: $randomNumber");

int randomNumber2 = random.nextInt(10)+1; // from 1 to 10 included



print("Generated Random Number Between 1 to 10: $randomNumber2");

}

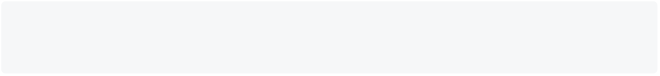
[Run Online](https://dartpad.dev/?id=bc906521f231d51ec3c5a4ee81f9d4c4)

* In this program, **random.nextInt(10)** function is used to generate a random number between **0 and 9** in which the value is stored in a variable **randomNumber**.
  + The **random.nextInt(10)+1** function is used to generate random number between **1 to 10** in which the value is stored in a variable **randomNumber2**.

**Generate Random Number Between Any Number**

Use this formula to generate a random number between any numbers in the dart.

min + Random().nextInt((max + 1) - min);



**Example: Random Number In Dart Between 10 - 20**

This program generates random numbers between 10 to 20.

import 'dart:math'; void main()



{

int min = 10; int max = 20;

int randomnum = min + Random().nextInt((max + 1) - min); print("Generated Random number between $min and $max is:

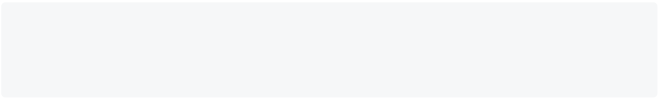
$randomnum"); }

[Run Online](https://dartpad.dev/?id=fcc351a04341474cbc963937ed860275)

**Random Boolean And Double Value**

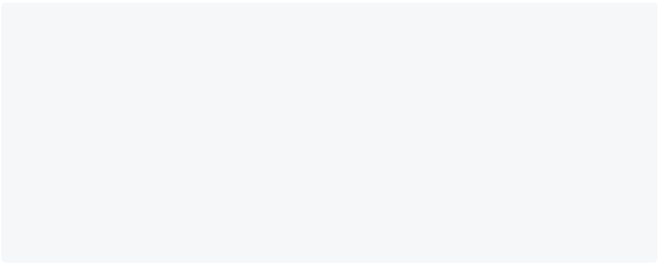
Here you will learn how to generate random boolean and double values in dart.

Random().nextBool(); // return true or false Random().nextDouble(); // return 0.0 to 1.0



**Example 1: Generate Random Boolean And Double Values** This example below generate random and boolean value.

import 'dart:math'; void main()



{

double randomDouble = Random().nextDouble(); bool randomBool = Random().nextBool();

print("Generated Random double value is: $randomDouble"); print("Generated Random bool value is: $randomBool");

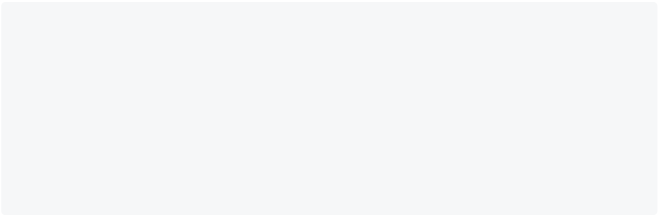
}

[Run Online](https://dartpad.dev/?id=d51f0a1dc4cf3a6afabd87510e089ae5)

**Example 2: Generate a List Of Random Numbers In Dart**

This example will generate a list of 10 random numbers between 1 to 100.

import 'dart:math';



void main()

{

List<int> randomList = List.generate(10, (\_) => Random().nextInt(100)+1);

print(randomList);

}

[Run Online](https://dartpad.dev/?id=773b8fe29b427e94cffebe3882f32634)

**Useful Math Function In Dart**

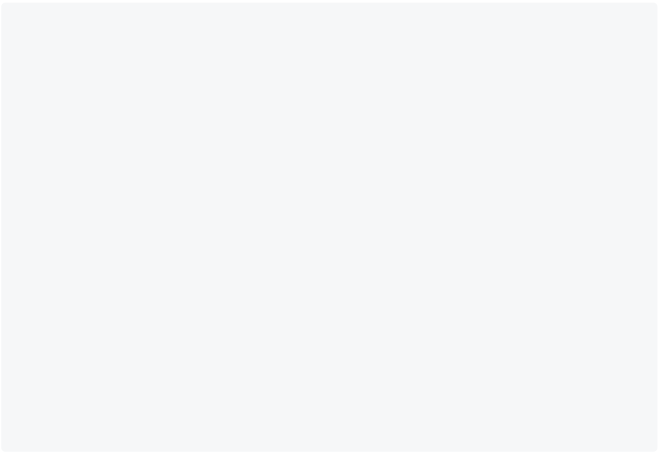
You can use some useful math functions to perform your daily task with dart programming.

|  |  |  |
| --- | --- | --- |
| **Function Name** | **Output** | **Description** |
| pow(10,2) | 100 | 10 to the power 2 is 10\*10 |
| max(10,2) | 10 | Maximum number is 10 |
| min(10,2) | 2 | Minimum number is 2 |
| sqrt(25) | 5 | Square root of 25 is 5 |

**Example: Math In Dart**

This example below finds the power of a number, a minimum and maximum value between two numbers, and the square root of a number.

import 'dart:math'; void main()



{

int num1 = 10;

int num2 = 2;

num powernum = pow(num1,num2);

num maxnum = max(num1,num2);

num minnum = min(num1,num2);

num squareroot = sqrt(25); // Square root of 25

print("Power is $powernum");

print("Maximum is $maxnum");

print("Minimum is $minnum");

print("Square root is $squareroot");

}

[Run Online](https://dartpad.dev/?id=5f165f8153240f2ea69654d8ea283d7f)

* In this program, **pow(num1, num2)** is a function where num1 is a digit and num2 is a power.
  + **max(num1,num2)** is a function which give the maximum number between num1 and num2.
    - **min(num1,num2)** is a function which give the mininum number between num1 and num2.
      * **sqrt(25)** is a function that gives the square root of 25.

## Collections in Dart

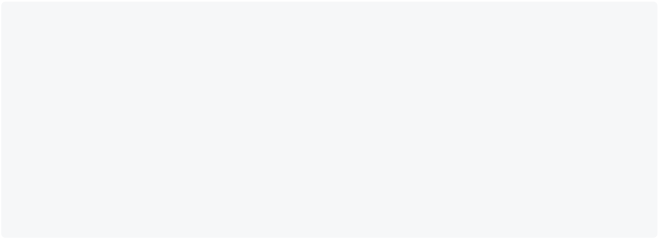
### List in Dart

If you want to store multiple values in the same variable, you can use **List**. List in dart is similar to **Arrays** in other programming languages. E.g. to store the names of multiple students, you can use a List. The List is represented by **Square Braces[].**

**How To Create List**

You can create a List by specifying the initial elements in a square bracket. Square bracket **[]** is used to represent a List.

// Integer List



List<int> ages = [10, 30, 23];

// String List

List<String> names = ["Raj", "John", "Rocky"];

// Mixed List

var mixed = [10, "John", 18.8];

**Types Of Lists**

* Fixed Length List
  + Growable List [**Mostly Used**]

**Fixed Length List**

The fixed-length Lists are defined with the specified length. You cannot change the size at runtime. This will create List of 5 integers with the value 0.

void main() {



var list = List<int>.filled(5,0); print(list);

}

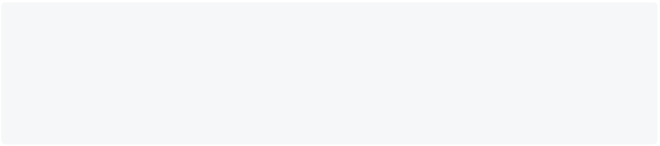
[Run Online](https://dartpad.dev/?id=435b64ef1dc3a1956bec1f90f3524bfd)

Note: You cannot add a new item to **Fixed Length List**, but you can change the values of List.

**Growable List**

A List defined without a specified length is called Growable List. The length of the growable List can be changed in runtime.

void main() {



var list1 = [210,21,22,33,44,55]; print(list1);

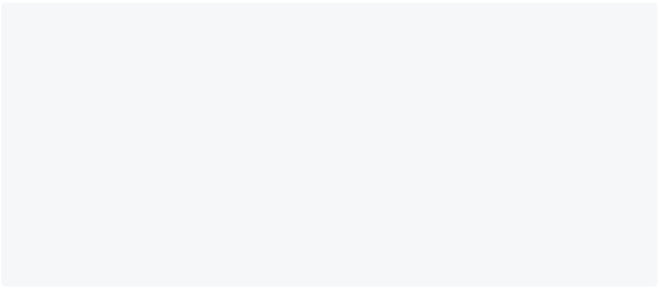
}

[Run Online](https://dartpad.dev/?id=fcbb0ff5a18b58da07c651cb18a9f678)

**Access Item Of List**

You can access the List item by **index**. Remember that the List index always starts with **0**.

void main() {



var list = [210, 21, 22, 33, 44, 55];

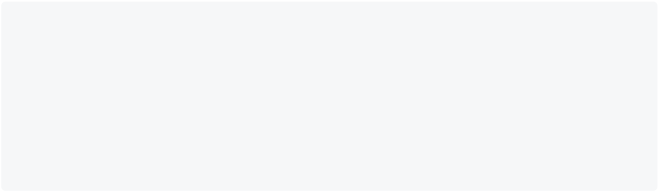
print(list[0]); print(list[1]); print(list[2]); print(list[3]); print(list[4]); print(list[5]); }

[Run Online](https://dartpad.dev/?id=4df857aa907cc94bd39aa358efcccef9)

**Get Index By Value**

You can also get the index by value.

void main() {



var list = [210, 21, 22, 33, 44, 55];

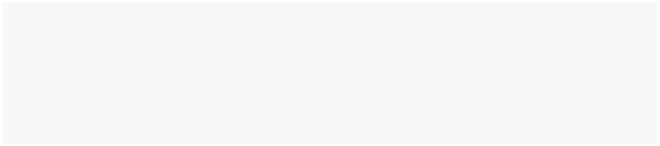
print(list.indexOf(22)); print(list.indexOf(33)); }

[Run Online](https://dartpad.dev/?id=6ae37c408e11aab38a25f6c32678fa4e)

**Find The Length Of The List**

You can find the length of List by using **.length** property.

void main(){



List<String> names = ["Raj", "John", "Rocky"]; print(names.length);

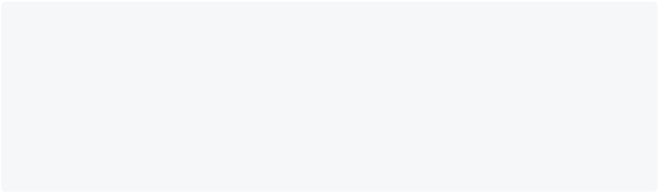
}

[Run Online](https://dartpad.dev/?id=e9ab9ed13874c12aee3f755626d6235e)

Note: Remember that List **index** starts with **0** and length always starts with **1**. **Changing Values Of List**

You can also change the value of List. You can do it by **listName[index]=value;**. For more, see the example below.

void main(){



List<String> names = ["Raj", "John", "Rocky"]; names[1] = "Bill";

names[2] = "Elon";

print(names);

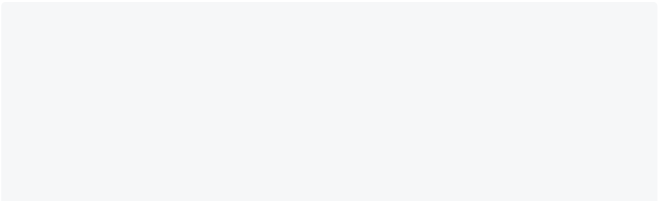
}

[Run Online](https://dartpad.dev/?id=12055628434885a94d851cc4ef7e2dcd)

**Mutable And Immutable List**

A mutable List means they can change after the declaration, and an immutable List means they can’t change after the declaration.

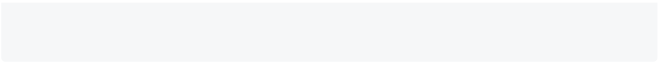
List<String> names = ["Raj", "John", "Rocky"]; // Mutable List names[1] = "Bill"; // possible



names[2] = "Elon"; // possible

const List<String> names = ["Raj", "John", "Rocky"]; // Immutable List names[1] = "Bill"; // not possible

names[2] = "Elon"; // not possible



**List Properties In Dart**

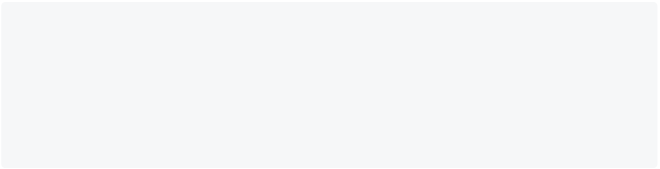
* **first**: It returns the first element in the List.
  + **last**: It returns the last element in the List.
    - **isEmpty**: It returns **true** if the List is empty and **false** if the List is not empty.
      * **isNotEmpty**: It returns **true** if the List is not empty and **false** if the List is empty.
  + **length**: It returns the length of the List.
    - * + **reversed**: It returns a List in reverse order.

**single**: It is used to check if the List has only one element and returns it.

**Access First And Last Elements Of List**

You can access the first and last elements in the List by:

void main() {



List<String> drinks = ["water", "juice", "milk", "coke"]; print("First element of the List is: ${drinks.first}"); print("Last element of the List is: ${drinks.last}");

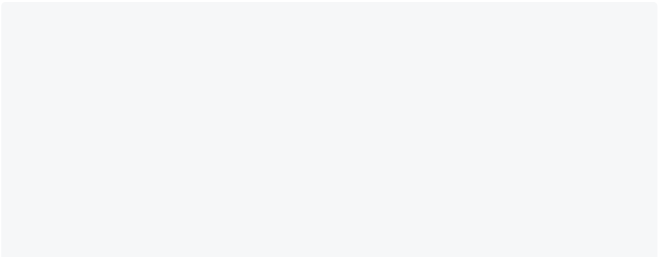
}

[Run Online](https://dartpad.dev/?id=a9e9571dc9be52e88985fd80b96d1143)

**Check The List Is Empty Or Not**

You can also check List contain any elements inside it or not. It will give result either in **true** or in **false**.

void main() {



List<String> drinks = ["water", "juice", "milk", "coke"];

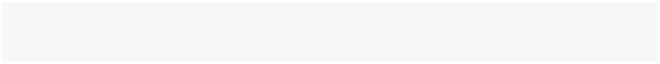
List<int> ages = [];

print("Is drinks Empty: "+drinks.isEmpty.toString());

print("Is drinks not Empty: "+drinks.isNotEmpty.toString()); print("Is ages Empty: "+ages.isEmpty.toString());

print("Is ages not Empty: "+ages.isNotEmpty.toString());

}

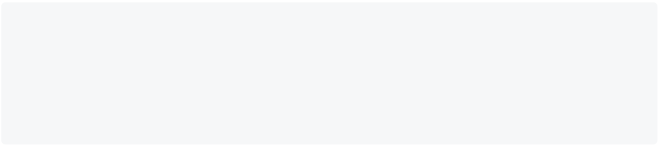


[Run Online](https://dartpad.dev/?id=48641b91ad78114471165aa2a7af7dc1)

**Reverse List In Dart**

You can easily reverse List by using **.reversed** properties. Here is an example below:

void main() {



List<String> drinks = ["water", "juice", "milk", "coke"]; print("List in reverse: ${drinks.reversed}");

}

[Run Online](https://dartpad.dev/?id=5501d43466e1b7c7c049b4dc323f7d9c)

**Adding Item To List**

Dart provides four methods to insert the elements into the Lists. These methods are given below.

* Method: add()
  + - * + Description: Add one element at a time and returns the modified List object.
  + Method: addAll()

Description: Insert the multiple values to the given List, and each value is separated by the commas and enclosed with a square bracket ([]).

* + - Method: insert()

Description: Provides the facility to insert an element at a specified index position.

* + - * Method: insertAll()

Description: Insert the multiple value at the specified index position.

**Example 1: Add Item To List**

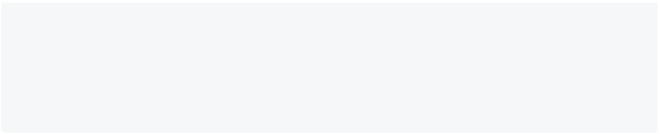
In this example below, we are adding an item to evenList using **add()** method.

void main() {



var evenList = [2,4,6,8,10];

print(evenList); evenList.add(12); print(evenList); }

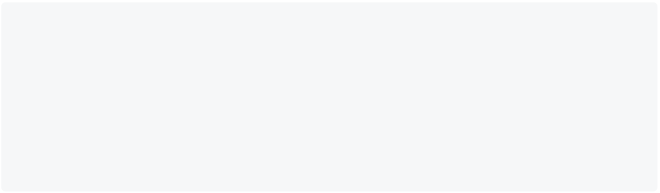


[Run Online](https://dartpad.dev/?id=6e9d51b33ef774c98bea789e5d6d5024)

**Example 2: Add Items To List**

In this example below, we are adding items to evenList using **addAll()** method.

void main() {



var evenList = [2, 4, 6, 8, 10]; print(evenList);

evenList.addAll([12, 14, 16, 18]); print(evenList);

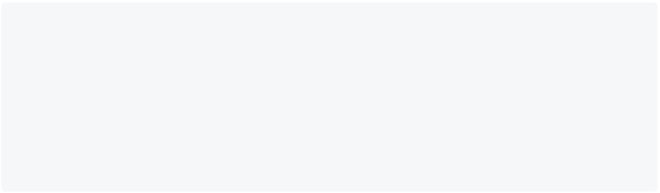
}

[Run Online](https://dartpad.dev/?id=4574deeb6dbfa2a667d3ff67e34ad50f)

**Example 3: Insert Item To List**

In this example below, we are adding an item to myList using **insert()** method.

void main() {



List myList = [3, 4, 2, 5]; print(myList);

myList.insert(2, 15);

print(myList);

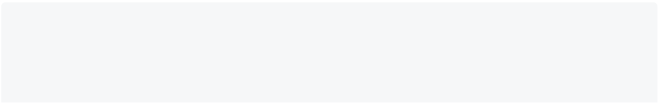
}

[Run Online](https://dartpad.dev/?id=6ad3b5f29e6dcacfde34ddc923a0c2a8)

**\*\*Example 4: Insert Items To List \*\***

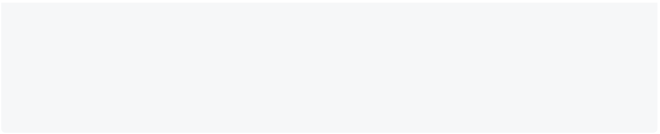
In this example below, we are adding items to myList using **insertAll()** method.

void main() {



var myList = [3, 4, 2, 5]; print(myList);

myList.insertAll(1, [6, 7, 10, 9]); print(myList);



}

[Run Online](https://dartpad.dev/?id=30e5e6960d2e05d99c17b517e088532d)

**Replace Range Of List**

You can also replace the range of the List. For more, see the example below.

void main() {



var list = [10, 15, 20, 25, 30];

print("List before updation: ${list}");

list.replaceRange(0, 4, [5, 6, 7, 8]);

print("List after updation using replaceAll() function : ${list}"); }

[Run Online](https://dartpad.dev/?id=422e4c7cb0305806ccd251e1b80acbd4)

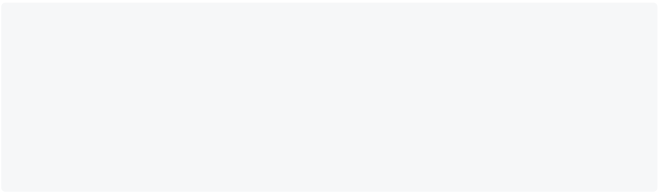
**Removing List Elements**

|  |  |
| --- | --- |
| **Method** | **Description** |
| remove() | Removes one element at a time from the given List. |
| removeAt() | Removes an element from the specified index position and returns it. |
| removeLast() | Remove the last element from the given List. |
| removeRange() | Removes the item within the specified range. |

**Example 1: Removing List Item From List**

In this example below, we are removing item of List using **remove()** method.

void main() {



var list = [10, 20, 30, 40, 50];

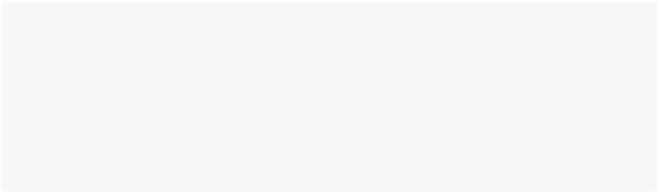
print("List before removing element : ${list}"); list.remove(30);

print("List after removing element : ${list}"); }

**Example 2: Removing List Item From List**

In this example below, we are removing item of List using **removeAt()** method.

void main() {



var list = [10, 11, 12, 13, 14];

print("List before removing element : ${list}"); list.removeAt(3);

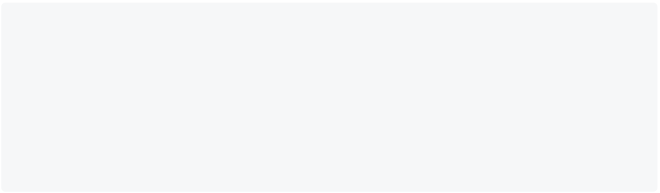
print("List after removing element : ${list}"); }

[Run Online](https://dartpad.dev/?id=4eb32e9382bd8f43e4967d74dea459c5)

**Example 3: Removing Last Item From List**

In this example below, we are removing last item of List using **removeLast()** method.

void main() {



var list = [10, 20, 30, 40, 50];

print("List before removing element:${list}");

list.removeLast();

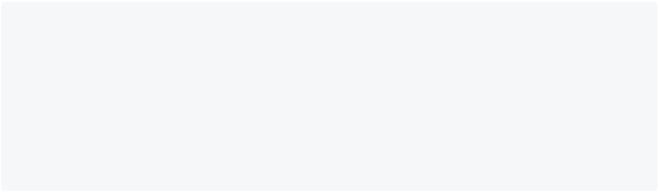
print("List after removing last element:${list}"); }

[Run Online](https://dartpad.dev/?id=57bb1b31fd02a0c0fe220402d39dc9a2)

**Example 4: Removing List Range From List**

In this example below, we are removing the range of items of List using **removeRange()** method.

void main() {



var list = [10, 20, 30, 40, 50];

print("List before removing element:${list}");

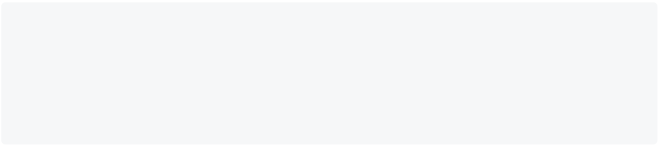
list.removeRange(0, 3);

print("List after removing range element:${list}"); }

**Loops In List**

You can use for loop, for each loop, or any other type of loop.

void main() {



List<int> list = [10, 20, 30, 40, 50]; list.forEach((n) => print(n));

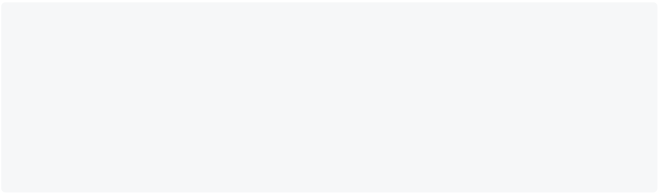
}

[Run Online](https://dartpad.dev/?id=0d29000e7ff3fa97970ae3277b1ace8a)

**Multiply All Value By 2 Of All List**

This example below multiply value of List item by 2.

void main() {



List<int> list = [10, 20, 30, 40, 50]; var douledList = list.map((n) => n \* 2);

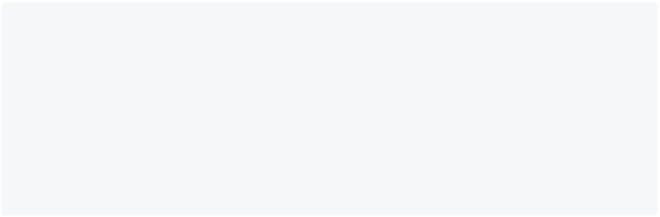
print((douledList)); }

[Run Online](https://dartpad.dev/?id=206c821dfd78247b87f354a05ef128fa)

**Combine Two Or More List In Dart**

You can combine two or more Lists in dart by using **spread** syntax.

void main() {



List<String> names = ["Raj", "John", "Rocky"];

List<String> names2 = ["Mike", "Subash", "Mark"];

List<String> allNames = [...names, ...names2]; print(allNames);

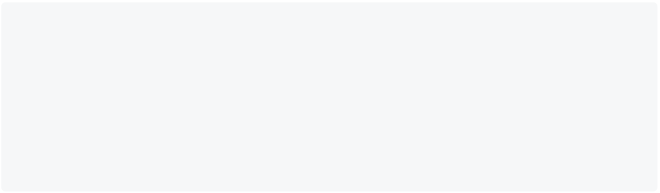
}

[Run Online](https://dartpad.dev/?id=b1c3e94a9ddbd9b981b8eb31384e3dc4)

**Conditions In List**

You can also use conditions in List. Here **sad = false** so cart doesn’t contain **Beer** in it.

void main() {



bool sad = false;

var cart = ['milk', 'ghee', if (sad) 'Beer']; print(cart);

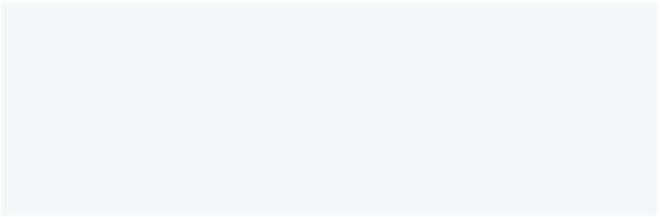
}

[Run Online](https://dartpad.dev/?id=7f5860b2b6ddb098be0b5de777ca1d5a)

**Where In List Dart**

You can use where with List to filter specific items. Here in this example, even numbers are only filtered.

void main(){



List<int> numbers = [2,4,6,8,10,11,12,13,14];

List<int> even = numbers.where((number)=> number.isEven).toList();

print(even);

}

[Run Online](https://dartpad.dev/?id=0acad1be5fbeac37ba35b7d6089e99b8)

**Note**: Choose Lists if order matters. You can easily add items to the end. Searching can be slow when the List size is big.

### Set in Dart

Set is a unique collection of items. You cannot store duplicate values in the Set. It is unordered, so it can be faster than lists while working with a large amount of data. Set is useful when you need to store unique values without considering the order of the input. E.g., fruits name, months name, days name, etc. It is represented by **Curley Braces{}.**

**Note**:The list allows you to add**duplicate items**, but the Set doesn’t allow it.

**Syntax**

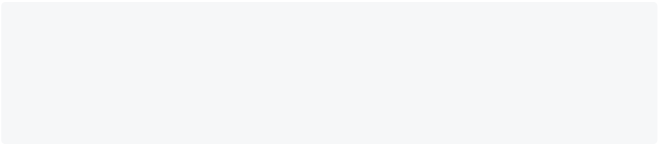
Set <variable\_type> variable\_name = {};



**How To Create A Set In Dart**

You can create a Set in Dart using the**Set**type annotation.

void main(){



Set<String> fruits = {"Apple", "Orange", "Mango"}; print(fruits);

}

[Run Online](https://dartpad.dev/?id=a81291eb0617b7bd9db9d5f82d94838e)

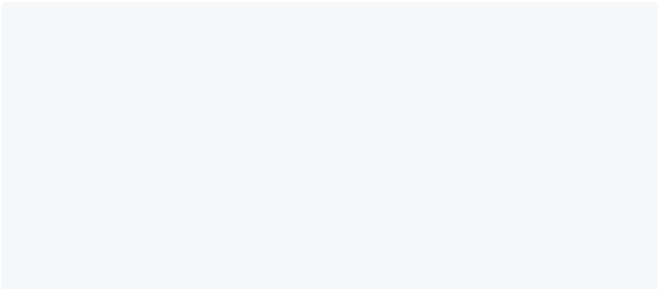
**Set Properties In Dart**

|  |  |
| --- | --- |
| **Properties** | **Work** |
| first | To get first value of Set. |
| last | To get last value of Set. |
| isEmpty | Return true or false. |
| isNotEmpty | Return true or false. |
| length | It returns the length of the Set. |

**Example of Set Properties Dart**

This example finds the first and last element of the Set, checks whether it is empty or not, and finds its length.

void main() {



// declaring fruits as Set

Set<String> fruits = {"Apple", "Orange", "Mango", "Banana"};

// using different properties of Set

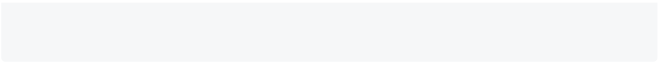
print("First Value is ${fruits.first}");

print("Last Value is ${fruits.last}");

print("Is fruits empty? ${fruits.isEmpty}");

print("Is fruits not empty? ${fruits.isNotEmpty}"); print("The length of fruits is ${fruits.length}");

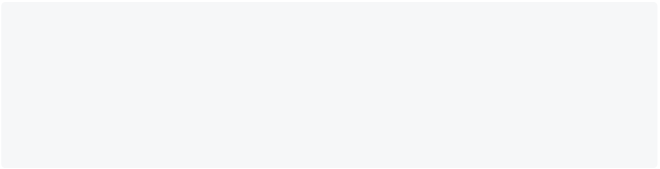
}



**Check The Available Value**

If you want to see whether the Set contains specific items or not, you can use the **contains** method, which returns true or false.

void main(){



Set<String> fruits = {"Apple", "Orange", "Mango"}; print(fruits.contains("Mango"));

print(fruits.contains("Lemon"));

}

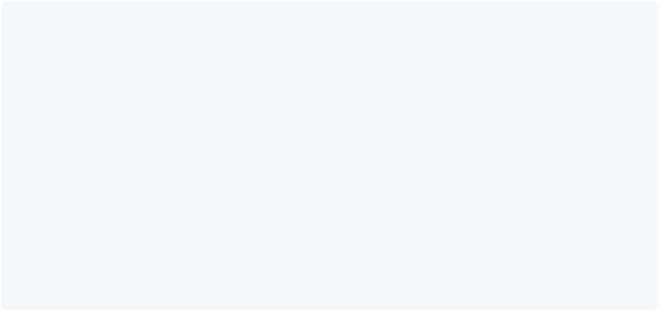
[Run Online](https://dartpad.dev/?id=e158381f766d9be511b06aa5780a0f3c)

**Add & Remove Items In Set**

Like lists, you can add or remove items in a Set. To add items use **add()** method and to remove use **remove()** method.

|  |  |
| --- | --- |
| **Method** | **Description** |
| add() | Add one element to Set. |
| remove() | Removes one element from Set. |

void main(){



Set<String> fruits = {"Apple", "Orange", "Mango"};

fruits.add("Lemon"); fruits.add("Grape");

print("After Adding Lemon and Grape: $fruits");

fruits.remove("Apple");

print("After Removing Apple: $fruits"); }

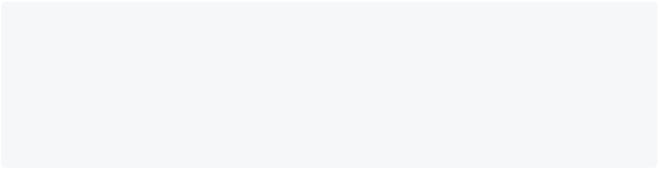
[Run Online](https://dartpad.dev/?id=5adb9c8bb832520d8cbeebb09b859f83)

**Adding Multiple Elements**

You can use **addAll()** method to add multiple elements from the list to Set.

|  |  |
| --- | --- |
| **Method** | **Description** |
| addAll() | Insert the multiple values to the given Set. |

void main(){



Set<int> numbers = {10, 20, 30};

numbers.addAll([40,50]);

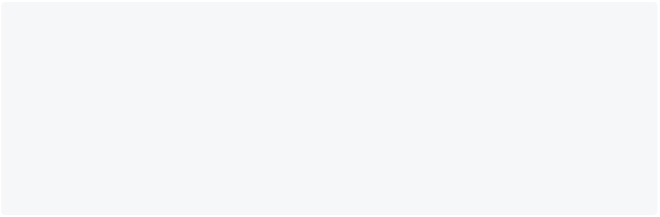
print("After adding 40 and 50: $numbers"); }

[Run Online](https://dartpad.dev/?id=a4c3327c1f7aba19407f61637e7dc71f)

**Printing All Values In Set**

You can print all Set items by using loops. [Click here](https://dart-tutorial.com/conditions-and-loops/loops-in-dart/) if you want to learn loop in dart.

void main(){



Set<String> fruits = {"Apple", "Orange", "Mango"};

for(String fruit in fruits){ print(fruit);

}

}

[Run Online](https://dartpad.dev/?id=be626d8991c500692b39daac328d9a18)

**Set Methods In Dart**

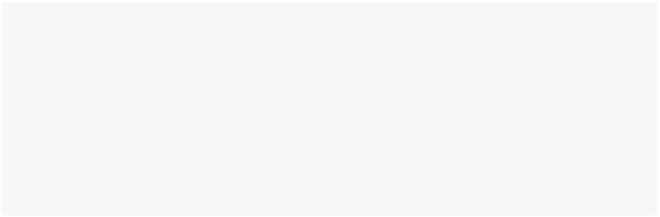
Some other helpful Set methods in dart.

|  |  |
| --- | --- |
| **Method** | **Description** |
| clear() | Removes all elements from the Set. |
| difference() | Creates a new Set with the elements of this that are not in other. |
| elementAt() | Returns the index value of element. |
| intersection() | Find common elements in two sets. |

**Clear Set In Dart**

In this example, you can see how to remove all items from the Set in dart.

void main() {



Set<String> fruits = {"Apple", "Orange", "Mango"}; // to clear all items

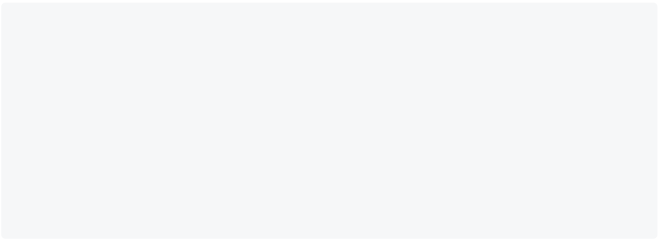
fruits.clear();

print(fruits); }

[Run Online](https://dartpad.dev/?id=ff1f296632223f0bb4cec6a6903b071f) **Difference In Set**

In Dart, the difference method creates a new Set with the elements that are not in the other.

void main() {



Set<String> fruits1 = {"Apple", "Orange", "Mango"}; Set<String> fruits2 = {"Apple", "Grapes", "Banana"};

final differenceSet = fruits1.difference(fruits2);

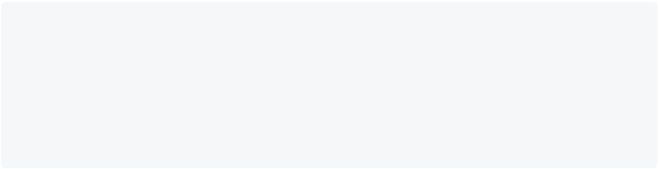
print(differenceSet); }

[Run Online](https://dartpad.dev/?id=92152b6f5447d0c18f8bf29e5c83c070)

**Element At Method In Dart**

In Dart you can find the Set value by its index number. The index number starts with 0.

void main() {



Set<String> days = {"Sunday", "Monday", "Tuesday"}; // index starts from 0 so 2 means Tuesday

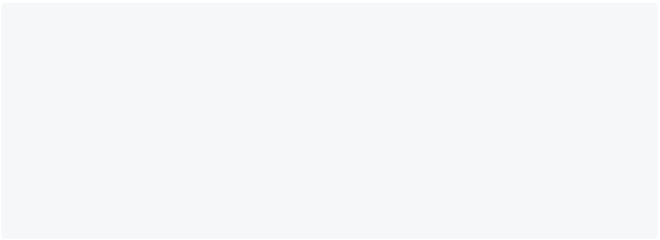
print(days.elementAt(2));

[Run Online](https://dartpad.dev/?id=c590abf0c2bbbb7df6e0c3ac13cc9cda)

**Intersection Method In Dart**

In Dart, the intersection method creates a new Set with the common elements in 2 Sets. Here Apple is available in both Sets.

void main() {



Set<String> fruits1 = {"Apple", "Orange", "Mango"}; Set<String> fruits2 = {"Apple", "Grapes", "Banana"};

final intersectionSet = fruits1.intersection(fruits2);

print(intersectionSet); }

[Run Online](https://dartpad.dev/?id=dd1d9fea77619307a9de26282c02e369)

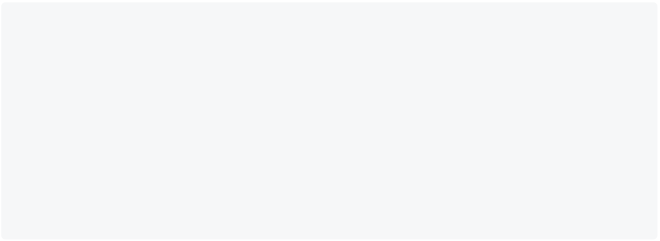
Map in Dart

In a Map, data is stored as keys and values. In Map, each key must be unique. They are similar to HashMaps and Dictionaries in other languages.

**How To Create Map In Dart**

Here we are creating a Map for **String** and **String**. It means keys and values must be the type of String. You can create a Map of any kind as you like.

void main(){



Map<String, String> countryCapital = { 'USA': 'Washington, D.C.',

'India': 'New Delhi',

'China': 'Beijing'

};

print(countryCapital);

}

[Run Online](https://dartpad.dev/?id=c8ff56092d2128b0a80ae75cb14bc979)

**Note**: Here **Usa**, **India**, and **China** are keys, and it must be **unique**. **Access Value From Key**

You can find the value of Map from its key. Here we are printing **Washington, D.C.** by its key, i.e., **USA**.

void main(){



Map<String, String> countryCapital = { 'USA': 'Washington, D.C.',

'India': 'New Delhi',

'China': 'Beijing'

};

print(countryCapital["USA"]);

}

[Run Online](https://dartpad.dev/?id=48d1e56c5b29d4a2cfe84f77f54e9c01)

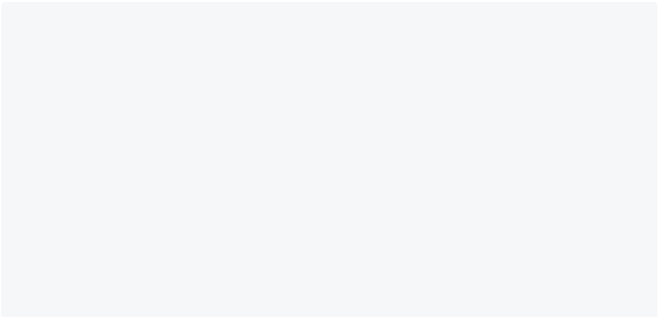
**Map Properties In Dart**

|  |  |
| --- | --- |
| **Properties** | **Work** |
| keys | To get all keys. |
| values | To get all values. |
| isEmpty | Return true or false. |
| isNotEmpty | Return true or false. |
| length | It returns the length of the Map. |

**Example Of Map Properties In Dart**

This example finds all keys/values of Map, the first and last element, checks whether it is empty or not, and finds its length.

void main() {



Map<String, double> expenses = { 'sun': 3000.0,

'mon': 3000.0,

'tue': 3234.0,

};

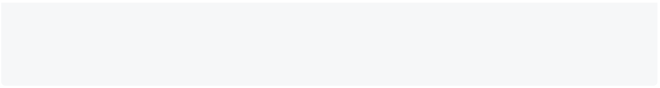
print("All keys of Map: ${expenses.keys}");

print("All values of Map: ${expenses.values}");

print("Is Map empty: ${expenses.isEmpty}");

print("Is Map not empty: ${expenses.isNotEmpty}");

print("Length of map is: ${expenses.length}"); }

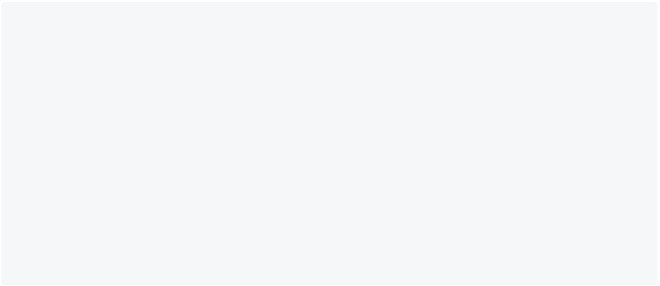


[Run Online](https://dartpad.dev/?id=018e421ba69575f674d94eb5af45bc05)

**Adding Element To Map**

If you want to add an element to the existing Map. Here is the way for you:

void main(){



Map<String, String> countryCapital = { 'USA': 'Washington, D.C.',

'India': 'New Delhi',

'China': 'Beijing'

};

// Adding New Item

countryCapital['Japan'] = 'Tokio'; print(countryCapital);

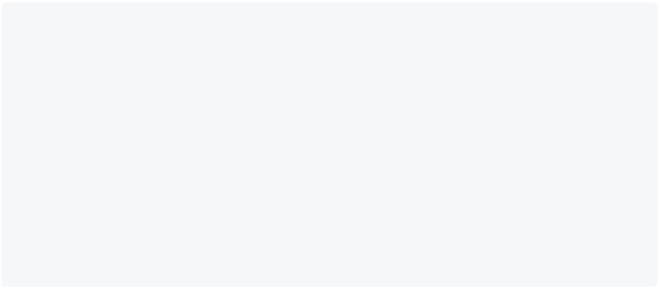
}

[Run Online](https://dartpad.dev/?id=b31bbfd3e49ccdf41cf07728939b18cc)

**Updating An Element Of Map**

If you want to update an element of the existing Map. Here is the way for you:

void main(){



Map<String, String> countryCapital = {

'USA': 'Nothing',

'India': 'New Delhi',

'China': 'Beijing'

};

// Updating Item

countryCapital['USA'] = 'Washington, D.C.'; print(countryCapital);

}

[Run Online](https://dartpad.dev/?id=d8190e762e8d657a43490064763a4e0a)

**Map Methods In Dart**

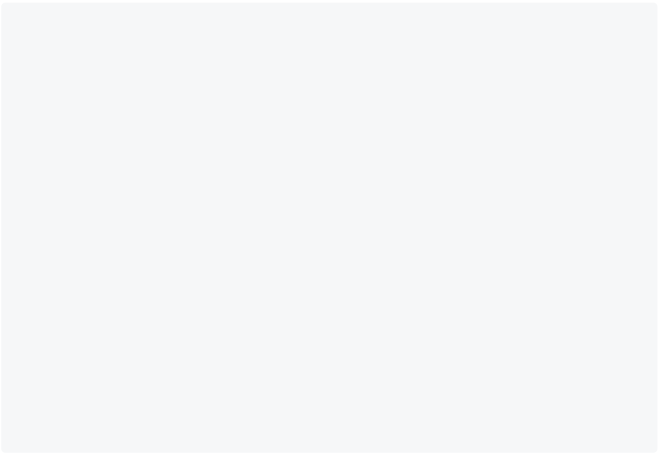
Some useful Map methods in dart.

|  |  |
| --- | --- |
| **Properties** | **Work** |
| keys.toList() | Convert all Maps keys to List. |
| values.toList() | Convert all Maps values to List. |
| containsKey(‘key’) | Return true or false. |
| containsValue(‘value’) | Return true or false. |
| clear() | Removes all elements from the Map. |
| removeWhere() | Removes all elements from the Map if condition is valid. |

**Convert Maps Keys & Values To List**

Let’s convert keys and values of Map to List.

void main() {



Map<String, double> expenses = { 'sun': 3000.0,

'mon': 3000.0,

'tue': 3234.0,

};

// Without List

print("All keys of Map: ${expenses.keys}");

print("All values of Map: ${expenses.values}");

// With List

print("All keys of Map with List: ${expenses.keys.toList()}");

print("All values of Map with List: ${expenses.values.toList()}");

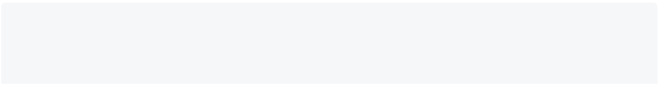
}

[Run Online](https://dartpad.dev/?id=0c0c31b4de26e1fd50f12a254bbc5da8)

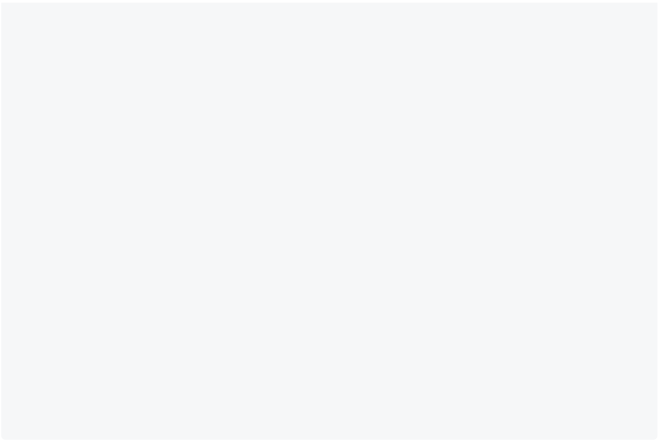
**Check Map Contains Specific Key/Value Or Not?**

Let’s check whether the Map contains a specific key/value in it or not.

void main() {



Map<String, double> expenses = { 'sun': 3000.0,



'mon': 3000.0,

'tue': 3234.0,

};

// For Keys

print("Does Map contain key sun: ${expenses.containsKey("sun")}"); print("Does Map contain key abc: ${expenses.containsKey("abc")}");

// For Values

print("Does Map contain value 3000.0: ${expenses.containsValue(3000.0)}");

print("Does Map contain value 100.0: ${expenses.containsValue(100.0)}");

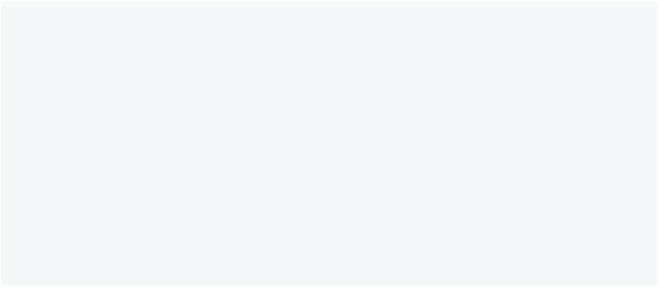
}

[Run Online](https://dartpad.dev/?id=ce64c93cc951a8d969954a98e02052d1)

**Removing Items From Map**

Suppose you want to remove an element of the existing Map. Here is the way for you:

void main(){



Map<String, String> countryCapital = { 'USA': 'Nothing',

'India': 'New Delhi',

'China': 'Beijing'

};

countryCapital.remove("USA"); print(countryCapital);

}

[Run Online](https://dartpad.dev/?id=bcddb1eaa4fb7ec897fb3d4318c27e31)

**Looping Over Element Of Map**

You can use any loop in Map to print all keys/values or to perform operations in its keys and values.

void main(){



Map<String, dynamic> book = { 'title': 'Misson Mangal', 'author': 'Kuber Singh',

'page': 233

};

// Loop Through Map

for(MapEntry book in book.entries){

print('Key is ${book.key}, value ${book.value}'); }

}

[Run Online](https://dartpad.dev/?id=d51dfb383cc42c9099afdc500de70078)

**Looping In Map Using For Each**

In this example, you will see how to use a loop to print all the keys and values in Map.

void main(){



Map<String, dynamic> book = { 'title': 'Misson Mangal', 'author': 'Kuber Singh',

'page': 233

};

// Loop Through For Each

book.forEach((key,value)=> print('Key is $key and value is $value'));

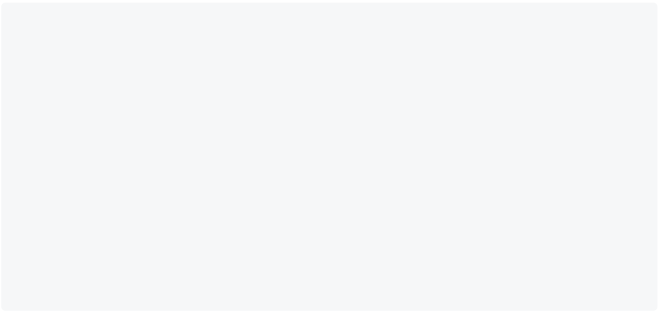
}

[Run Online](https://dartpad.dev/?id=29f7e04ab59ce860fb034b69e0d7d075)

**Remove Where In Dart Map**

In this example, you will see how to get students whose marks are greater or equal to 32 using where method.

void main() {



Map<String, double> mathMarks = {

"ram": 30,

"mark": 32,

"harry": 88,

"raj": 69,

"john": 15,

};

mathMarks.removeWhere((key, value) => value < 32); print(mathMarks);

}

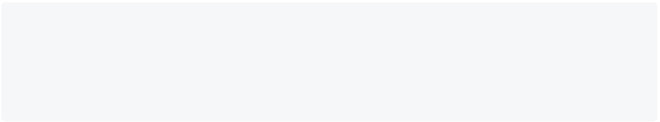
[Run Online](https://dartpad.dev/?id=ed459a19dbd14d4dc99aee6771fe172e)

Where in Dart

You can use where in list, set, map to **filter specific items**. It returns a new list containing all the elements that satisfy the condition. This is also called **Where Filter** in dart. Let’s see the syntax below:

**Syntax**

Iterable<E> where(



bool test(E element) )

**Example 1: Filter Only Odd Number From List**

In this example, you will get only odd numbers from a list.

void main() {

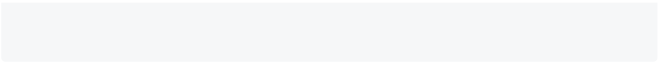


List<int> numbers = [2, 4, 6, 8, 10, 11, 12, 13, 14];

List<int> oddNumbers = numbers.where((number) => number.isOdd).toList();

print(oddNumbers);

}

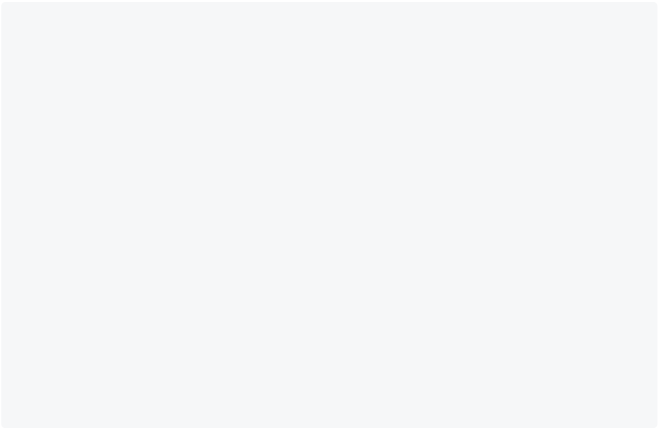


[Run Online](https://dartpad.dev/?id=c3c3e9ed5adad4ec4ae46886c26433b2)

**Example 2: Filter Days Start With S**

In this example, you will get only days that start with alphabet s.

void main() {



List<String> days = [ "Sunday",

"Monday",

"Tuesday",

"Wednesday",

"Thursday",

"Friday",

"Saturday"

];

List<String> startWithS =

days.where((element) => element.startsWith("S")).toList();

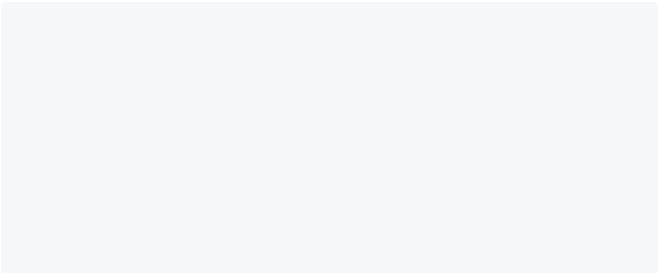
print(startWithS); }

[Run Online](https://dartpad.dev/?id=a3b6fe72bb23b3a7ee06166e301ca371)

**Example 3: Where Filter In Map**

In this example, you will get students whose marks are greater or equal to 32.

void main() {



Map<String, double> mathMarks = { "ram": 30,

"mark": 32,

"harry": 88,

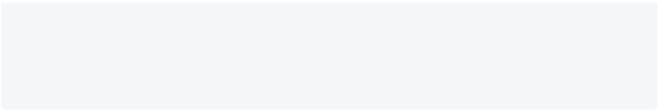
"raj": 69,

"john": 15,

};

mathMarks.removeWhere((key, value) => value < 32);

print(mathMarks); }



[Run Online](https://dartpad.dev/?id=3468132eab9dba6639a823cab27da514)

## File Handling in Dart

### Read File in Dart

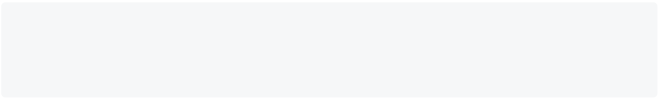
**Introduction To File Handling**

File handling is an important part of any programming language. In this section, you will learn how to read the file in a dart programming language.

**Read File In Dart**

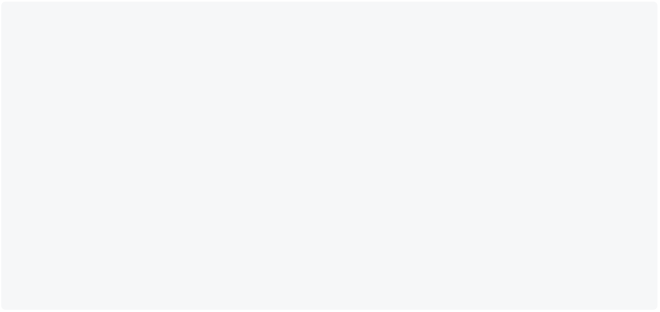
Assume that you have a file named test.txt in the same directory of your dart program.

Welcome to test.txt file. This is a test file.



Now, you can read this file using **File** class and **readAsStringSync()** method.

// dart program to read from file import 'dart:io';



void main() {

// creating file object

File file = File('test.txt');

// read file

String contents = file.readAsStringSync(); // print file

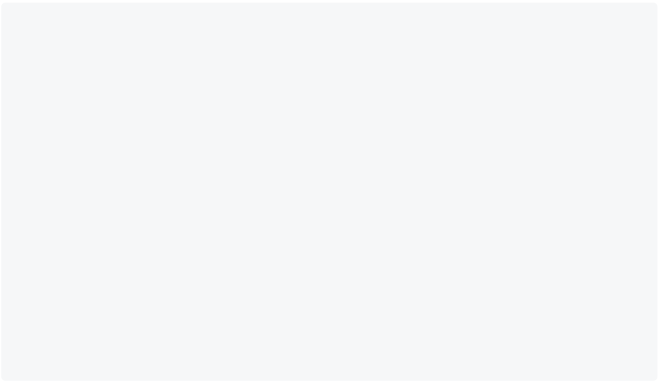
print(contents);

}

**Get File Information**

In this example below, you will learn how to get file information like file location, file size, and last modified time.

import 'dart:io';



void main() {

// open file

File file = File('test.txt');

// get file location

print('File path: ${file.path}');

// get absolute path

print('File absolute path: ${file.absolute.path}'); // get file size

print('File size: ${file.lengthSync()} bytes');

// get last modified time

print('Last modified: ${file.lastModifiedSync()}');

}

**Note**: If you try to get information of a file that does not exist, then it will throw an exception.

**CSV File**

A CSV (**Comma Separated Values**) file is a plain text file that contains data organized in a table format, where columns are separated by commas and rows are separated by line breaks. CSV files are used for:

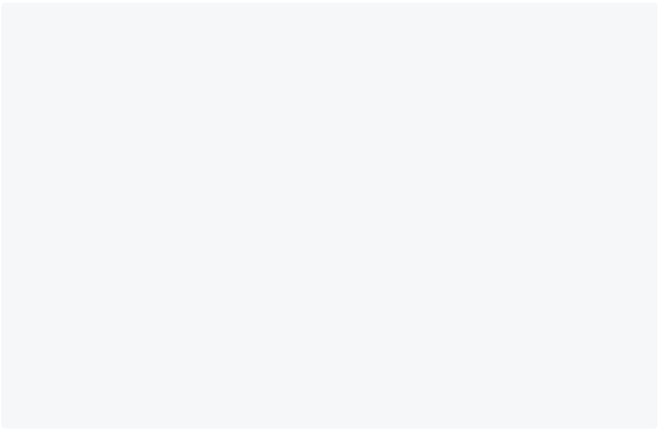
* Data exchange between different applications.
  + Data backup and restore.
    - Importing and exporting data from databases.
      * Automation of data processing tasks.

**Read CSV File In Dart**

Assume that you have a CSV file named test.csv in the same directory of your dart program.

Now, you can read this file using **File** class and **readAsStringSync()** method. We will use **split()** method to split the string into a list of strings.

// dart program to read from csv file import 'dart:io';



void main() {

// open file

File file = File('test.csv');

// read file

String contents = file.readAsStringSync(); // split file using new line

List<String> lines = contents.split('\n'); // print file

print('---------------------');

for (var line in lines) {

print(line);

}

}

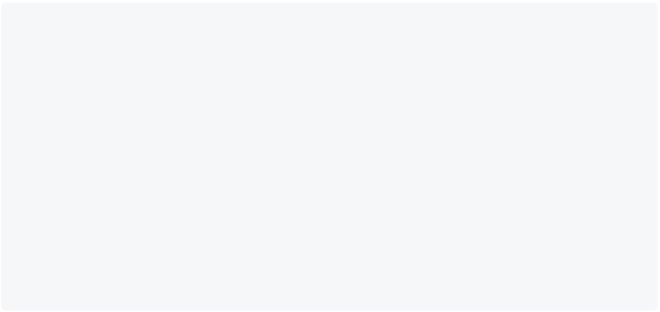
**Read Only Part Of File**

You can read only part of file using **substring()** method. Here is an example to read only first 10 characters of file. Make sure that you have a file named **test.txt** in the same directory of your dart program.

Welcome to test.txt file This is a test file.



// dart program to read from file import 'dart:io';



void main() {

// open file

File file = new File('test.txt');

// read only first 10 characters

String contents = file.readAsStringSync().substring(0, 10); // print file

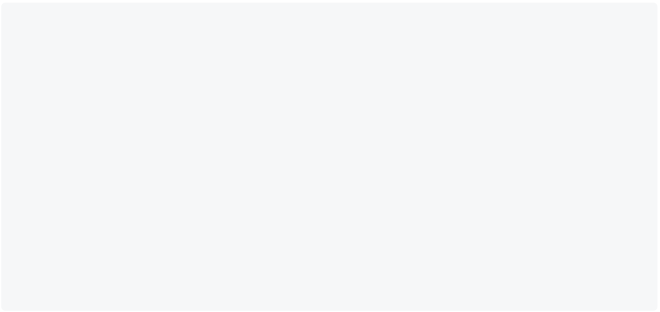
print(contents);

}

**Read File From Specific Directory**

To read a file from a specific directory, you need to provide the full path of the file. Here is an example to read file from a specific directory.

// dart program to read from file import 'dart:io';



void main() {

// open file

File file = File('C:\\Users\\test.txt'); // read file

String contents = file.readAsStringSync(); // print file

print(contents);

}

### Write File in Dart

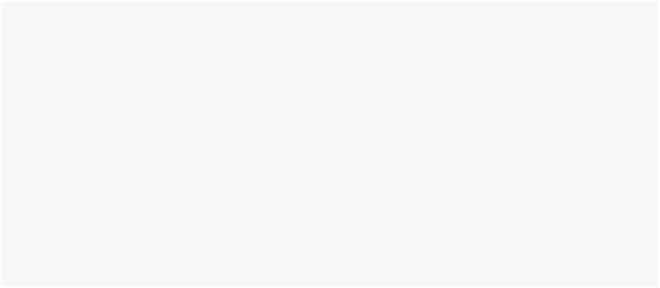
**Introduction**

In this section, you will learn how to write file in dart programming language by using **File** class and **writeAsStringSync()** method.

**Write File In Dart**

Let’s create a file named **test.txt** in the same directory of your dart program and write some text in it.

// dart program to write to file import 'dart:io';



void main() {

// open file

File file = File('test.txt');

// write to file

file.writeAsStringSync('Welcome to test.txt file.'); print('File written.');

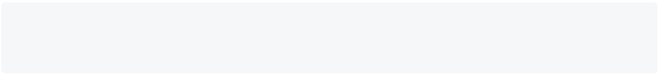
}

**Note**: If you have already some content in **test.txt** file, then it will be removed and replaced with new content.

**Add New Content To Previous Content**

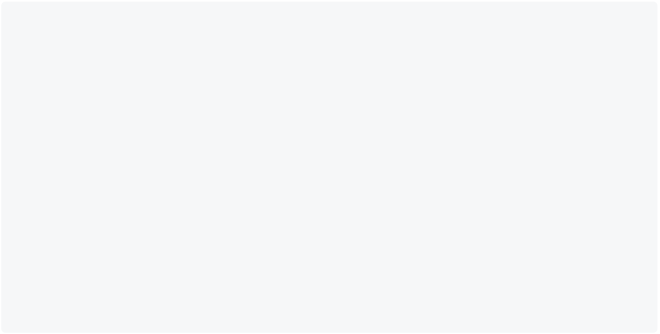
You can use **FileMode.append** to add new content to previous content. Assume that **test.txt** file already contains some text.

Welcome to test.txt file.



Now, let’s add new content to it.

// dart program to write to existing file import 'dart:io';



void main() {

// open file

File file = File('test.txt');

// write to file

file.writeAsStringSync('\nThis is a new content.', mode: FileMode.append);

print('Congratulations!! New content is added on top of previous content.');

}

**Write CSV File In Dart**

In the example below, we will ask user to enter **name** and **phone** of 3 students and write it to a csv file named **students.csv**.

// dart program to write to csv file import 'dart:io';



void main() {

// open file

File file = File("students.csv");

// write to file

file.writeAsStringSync('Name,Phone\n');

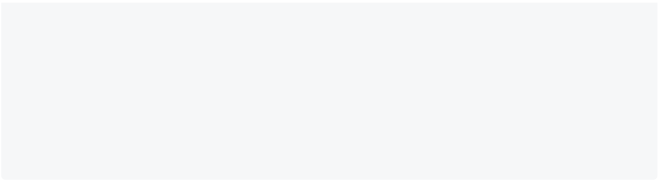
for (int i = 0; i < 3; i++) {

// user input name

stdout.write("Enter name of student ${i + 1}: "); String? name = stdin.readLineSync();

stdout.write("Enter phone of student ${i + 1}: ");

// user input phone



String? phone = stdin.readLineSync();

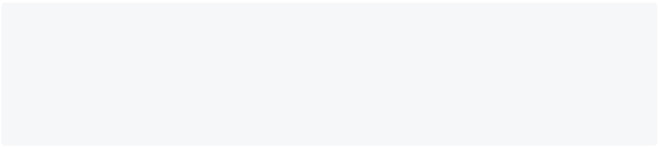
file.writeAsStringSync('$name,$phone\n', mode: FileMode.append); }

print("Congratulations!! CSV file written successfully.");

}

**students.csv** file will look like this:

Name,Phone John,1234567890 Mark,0123456789 Elon,0122112322



**Note**: You can create any type of file using **writeAsStringSync()** method. For example, **.html**, **.json**, **.xml**, etc.

### Delete File in Dart

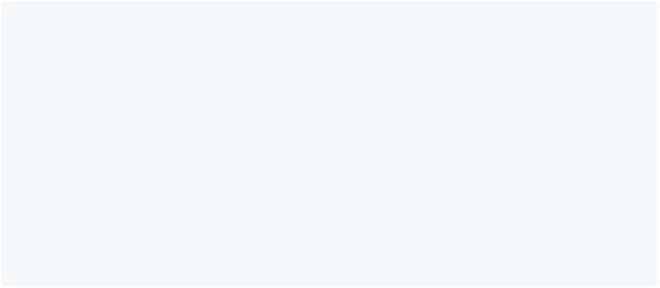
**Introduction**

In this section, you will learn how to delete file in dart programming language using **File** class and **deleteSync()** method.

**Delete File In Dart**

Assume that you have a file named **test.txt** in the same directory of your dart program. Now, let’s delete it.

// dart program to delete file import 'dart:io';



void main() {

// open file

File file = File('test.txt'); // delete file

file.deleteSync();

print('File deleted.');

}

**Note**: If you try to delete a file that does not exist, then it will throw an exception. **Delete File If Exists**

You can use **File.existsSync()** method to check if a file exists or not. If it exists, then you can delete it.

// dart program to delete file if exists import 'dart:io';



void main() {

// open file

File file = File('test.txt'); // check if file exists

if (file.existsSync()) {

// delete file

file.deleteSync();

print('File deleted.');

} else {

print('File does not exist.'); }

}

## OOP in Dart

### OOP in Dart

**Object-oriented programming (OOP)** is a programming method that uses objects and their interactions to design and program applications. It is one of the most popular programming paradigms and is used in many programming languages, such as Dart, Java, C++, Python, etc.

In **OOP**, an object can be anything, such as a person, a bank account, a car, or a house. Each object has its attributes (or properties) and behavior (or methods). For example, a person object may have the attributes **name**, **age** and **height**, and the behavior **walk** and **talk**.

**Advantages**

* It is easy to understand and use.
  + It increases reusability and decreases complexity.
    - The productivity of programmers increases.
* It makes the code easier to maintain, modify and debug.
  + It promotes teamwork and collaboration.
    - It reduces the repetition of code.

**Features Of OOP**

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction

Note: The main purpose of OOP is to break complex problems into smaller objects. You will learn all these OOPs features later in this dart tutorial.

**Key Points**

* Object Oriented Programming (OOP) is a programming paradigm that uses objects and their interactions to design and program applications.
  + OOP is based on objects, which are data structures containing data and methods.
    - OOP is a way of thinking about programming that differs from traditional procedural programming.
      * OOP can make code more modular, flexible, and extensible.
        + OOP can help you to understand better and solve problems.

### Class in Dart

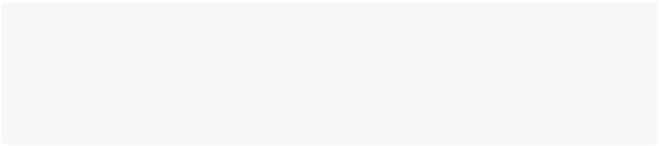
In object-oriented programming, a class is a blueprint for creating objects. A class defines the properties and methods that an object will have. For example, a class called **Dog** might have properties like **breed**, **color** and methods like **bark**, **run**.

**Declaring Class In Dart**

You can declare a class in dart using the **class** keyword followed by class name and braces {}. It’s a good habit to write class name in **PascalCase**. For example, **Employee**, **Student**, **QuizBrain**, etc.

**Syntax**

class ClassName {



// properties or fields // methods or functions }

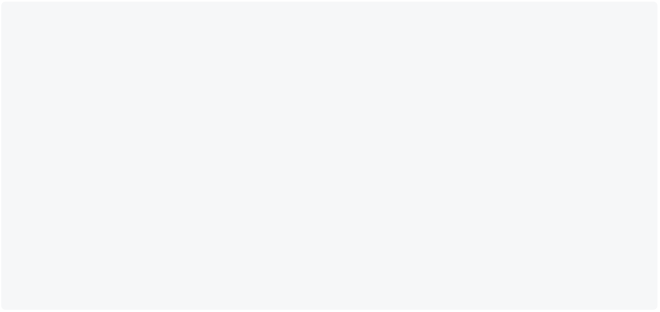
In the above syntax:

* The **class** keyword is used for defining the class.
  + **ClassName** is the name of the class and must start with capital letter.
    - Body of the class consists of **properties** and **functions**.
      * **Properties** are used to store the data. It is also known as **fields** or **attributes**.
        + **Functions** are used to perform the operations. It is also known as **methods**.

**Example : Declaring A Class In Dart**

In this example below, there is class **Animal** with three properties: **name**, **numberOfLegs**, and **lifeSpan**. The class also has a method called **display**, which prints out the values of the three properties.

class Animal {



String? name;

int? numberOfLegs; int? lifeSpan;

void display() {

print("Animal name: $name.");

print("Number of Legs: $numberOfLegs."); print("Life Span: $lifeSpan.");

}

}

**Note: This program will not print anything** because we have not created any object of the class. You will learn about the **object** later. The **?** is used for null safety. You will also learn about **null safety** later.

**Key Points**

* The class is declared using the **class** keyword.
  + The class is a blueprint for creating objects.
* The class body consists of properties and methods.
  + The properties are also known as fields, attributes, or data members.
    - The methods are also known as behaviors, or member functions.

### Object in Dart

**In object-oriented programming**, an object is a self-contained unit of code and data. Objects are created from templates called classes. An object is made up of properties(variables) and methods(functions). An object is an instance of a class.

**For example**, a bicycle object might have attributes like color, size, and current speed. It might have methods like changeGear, pedalFaster, and brake.

Info

**Note**: To create an object, you must create a class first. It’s a good practice to declare the object name in lower case.

**Instantiation**

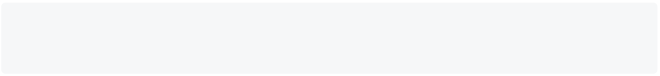
In object-oriented programming, instantiation is the process of creating an instance of a class. In other words, you can say that instantiation is the process of creating an object of a class. For example, if you have a class called **Bicycle**, then you can create an object of the class called **bicycle**.

**Declaring Object In Dart**

Once you have created a class, it’s time to declare the object. You can declare an object by the following syntax:

**Syntax**

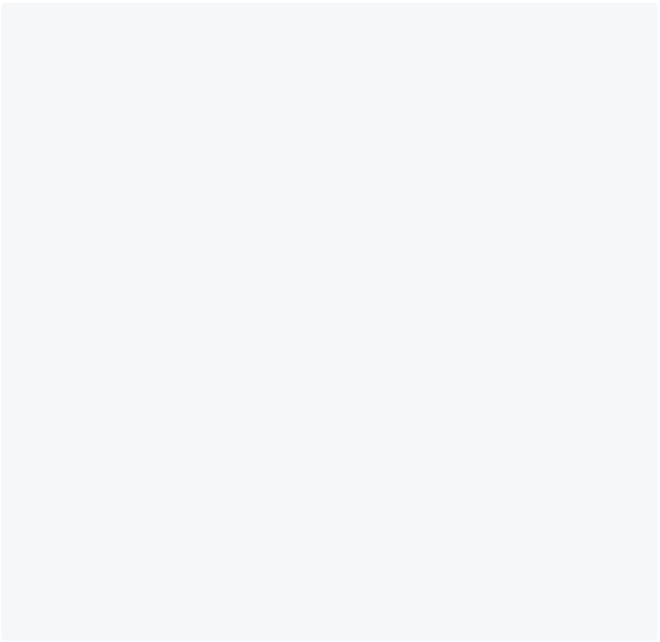
ClassName objectName = ClassName();



**Example : Declaring An Object In Dart**

In this example below, there is class **Bycycle** with three properties: **color**, **size**, and **currentSpeed**. The class has two methods. One is **changeGear**, which changes the gear of the bicycle, and **display** method prints out the values of the three properties. We also have an object of the class **Bycycle** called **bicycle**.

class Bicycle {



String? color;

int? size;

int? currentSpeed;

void changeGear(int newValue) { currentSpeed = newValue;

}

void display() {

print("Color: $color");

print("Size: $size");

print("Current Speed: $currentSpeed"); }

}

void main(){

// Here bicycle is object of class Bicycle. Bicycle bicycle = Bicycle();

bicycle.color = "Red";

bicycle.size = 26;

bicycle.currentSpeed = 0;

bicycle.changeGear(5);

bicycle.display();

}

[Run Online](https://dartpad.dev/?id=ac6ab46115153304cc99e2054c93cc17)

**Note**: Once you create an object, you can access the properties and methods of the object using the dot(.) operator.

**Key Points**

* The main method is the program’s entry point, so it is always needed to see the result.
  + The **new** keyword can be used to create a new object, but it is unnecessary.

### Constructor in Dart

**Introduction**

In this section, you will learn about constructor in Dart programming language and how to use constructors with the help of examples. Before learning about the

constructor, you should have a basic understanding of the class and object in dart. **Constructor In Dart**

**A constructor** is a special method used to initialize an object. It is called automatically when an object is created, and it can be used to set the initial values for the object’s properties. For example, the following code creates a **Person** class object and sets the initial values for the **name** and **age** properties.

Person person = Person("John", 30);



**Without Constructor**

If you don’t define a constructor for class, then you need to set the values of the properties manually. For example, the following code creates a **Person** class object and sets the values for the **name** and **age** properties.

Person person = Person(); person.name = "John"; person.age = 30;

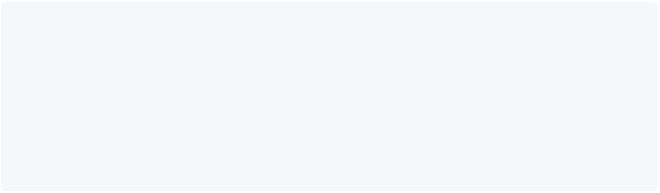


**Things To Remember**

* The constructor’s name should be the same as the class name.
  + Constructor doesn’t have any return type.

**Syntax**

class ClassName {



// Constructor declaration: Same as class name ClassName() {

// body of the constructor

}

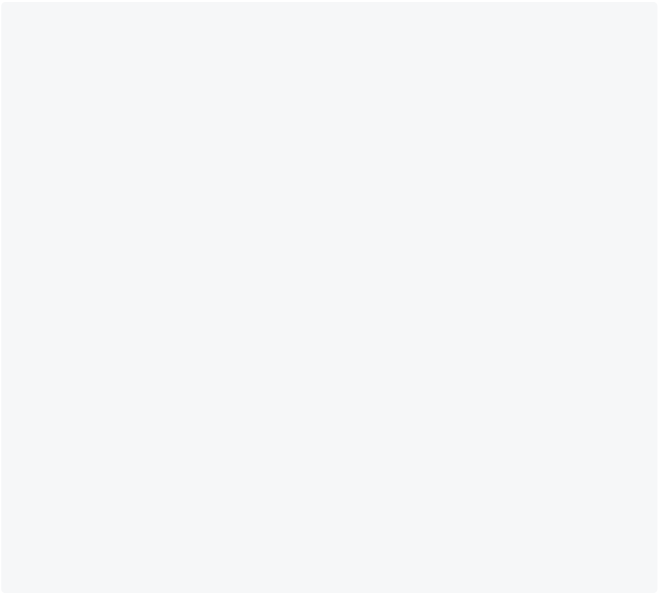
}

**Note**: When you create a object of a class, the constructor is called automatically. It is used to initialize the values when an object is created.

**Example 1: How To Declare Constructor In Dart**

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also created an object of the class **Student** called **student**.

class Student {



String? name;

int? age;

int? rollNumber;

// Constructor

Student(String name, int age, int rollNumber) {

print(

"Constructor called"); // this is for checking the constructor is called or not.

this.name = name;

this.age = age;

this.rollNumber = rollNumber;

}

}

void main() {

// Here student is object of class Student. Student student = Student("John", 20, 1);

print("Name: ${student.name}");

print("Age: ${student.age}");

print("Roll Number: ${student.rollNumber}"); }

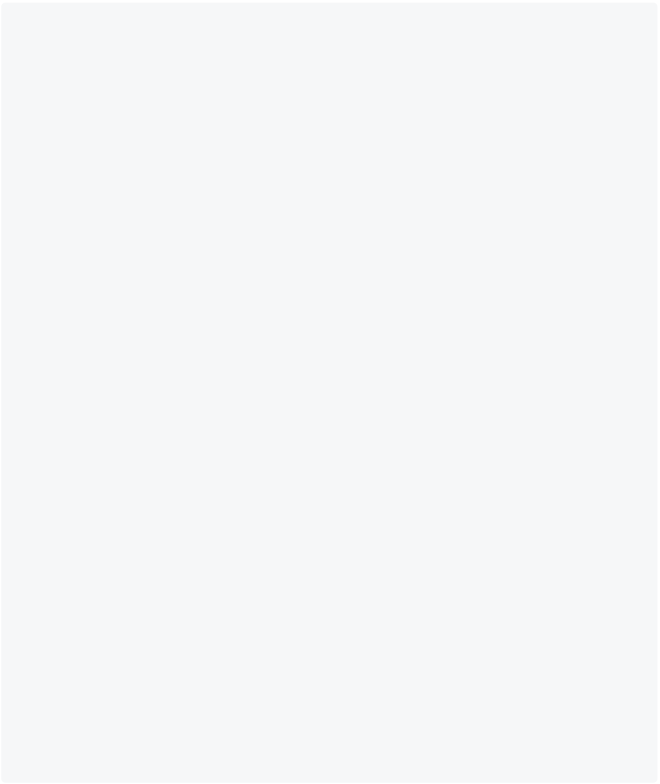
[Run Online](https://dartpad.dev/?id=272408e52010155b508050093eda4c91)

**Note**: The **this** keyword is used to refer to the current instance of the class. It is used to access the current class properties. In the example above, parameter names and class properties of constructor **Student** are the same. Hence to avoid confusion, we use the **this** keyword.

**Example 2: Constructor In Dart**

In this example below, there is a class **Teacher** with four properties: **name**, **age**, **subject**, and **salary**. Class has one constructor for initializing the values of the properties. Class also contain method **display()** which is used to display the values of the properties. We also created 2 objects of the class **Teacher** called **teacher1** and **teacher2**.

class Teacher {



String? name;

int? age;

String? subject; double? salary;

// Constructor

Teacher(String name, int age, String subject, double salary) { this.name = name;

this.age = age;

this.subject = subject;

this.salary = salary;

}

// Method

void display() {

print("Name: ${this.name}");

print("Age: ${this.age}");

print("Subject: ${this.subject}");

print("Salary: ${this.salary}\n"); // \n is used for new line

} }

void main() {

// Creating teacher1 object of class Teacher

Teacher teacher1 = Teacher("John", 30, "Maths", 50000.0); teacher1.display();

// Creating teacher2 object of class Teacher

Teacher teacher2 = Teacher("Smith", 35, "Science", 60000.0); teacher2.display();

}

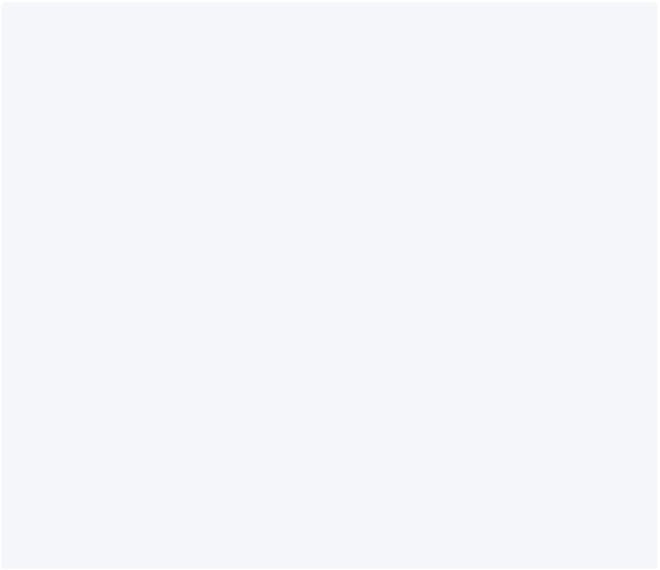
[Run Online](https://dartpad.dev/?id=feca294afc69f2a6e40c72e94bacff4f)

**Note**: You can create many objects of a class. Each object will have its own copy of the properties.

**Example 3: Constructor In Dart**

In this example below, there is a class **Car** with two properties: **name** and **price**. The class has one constructor for initializing the values of the properties. The class also contains method **display()**, which is used to display the values of the properties. We also created an object of the class **Car** called **car**.

class Car {



String? name; double? price;

// Constructor

Car(String name, double price) { this.name = name;

this.price = price;

}

// Method

void display() {

print("Name: ${this.name}"); print("Price: ${this.price}"); }

}

void main() {

// Here car is object of class Car. Car car = Car("BMW", 500000.0);

car.display();

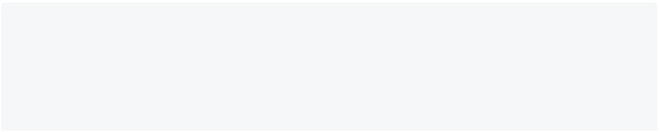
}

[Run Online](https://dartpad.dev/?id=df9f4ebe50ab75c11e9f70c1dd9781bb)

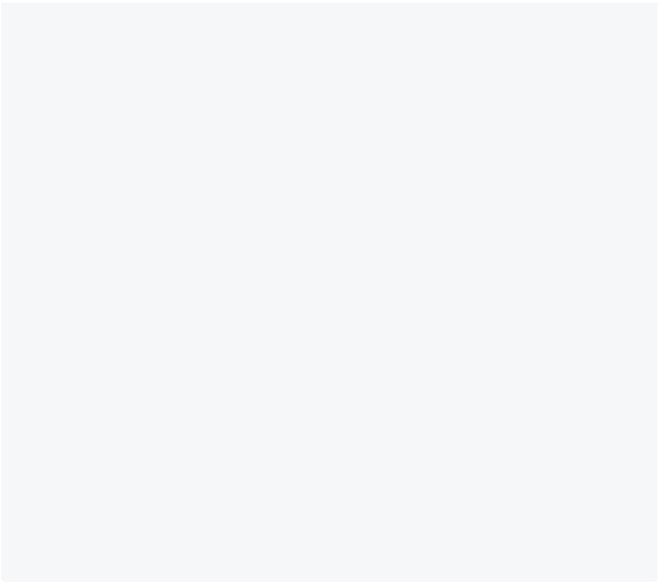
**Example 4: Constructor In Dart**

In this example below, there is a class **Staff** with four properties: **name**, **phone1**, **phone2**, and **subject** and one method **display()**. Class has one constructor for initializing the values of only **name**, **phone1** and **subject**. We also created an object of the class **Staff** called **staff**.

class Staff { String? name; int? phone1; int? phone2;



String? subject;



// Constructor

Staff(String name, int phone1, String subject) { this.name = name;

this.phone1 = phone1;

this.subject = subject;

}

// Method

void display() {

print("Name: ${this.name}");

print("Phone1: ${this.phone1}"); print("Phone2: ${this.phone2}"); print("Subject: ${this.subject}"); }

}

void main() {

// Here staff is object of class Staff.

Staff staff = Staff("John", 1234567890, "Maths"); staff.display();

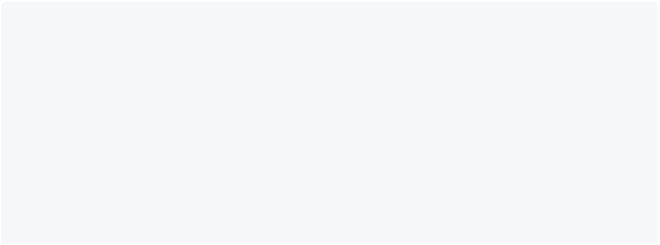
}

[Run Online](https://dartpad.dev/?id=b1c03fbb1b0169c623df475cd31ce4c8)

**Example 5: Write Constructor Single Line**

In the avobe section, you have written the constructor in long form. You can also write the constructor in short form. You can directly assign the values to the properties. For example, the following code is the short form of the constructor in one line.

class Person{



String? name;

int? age;

String? subject; double? salary;

// Constructor in short form

Person(this.name, this.age, this.subject, this.salary);

// display method



void display(){

print("Name: ${this.name}");

print("Age: ${this.age}");

print("Subject: ${this.subject}"); print("Salary: ${this.salary}"); }

}

void main(){

Person person = Person("John", 30, "Maths", 50000.0); person.display();

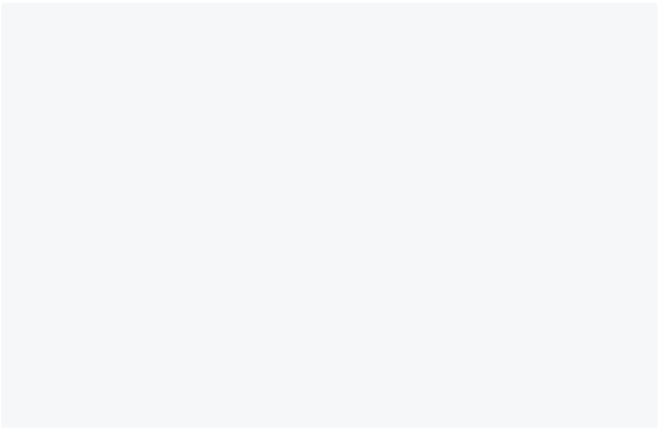
}

[Run Online](https://dartpad.dev/?id=28abe898777d13770dd239aa836c83e3)

**Example 6: Constructor With Optional Parameters**

In the example below, we have created a class **Employee** with four properties: **name**, **age**, **subject**, and **salary**. Class has one constructor for initializing the all properties values. For **subject** and **salary**, we have used optional parameters. It means we can pass or not pass the values of **subject** and **salary**. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Employee** called **employee**.

class Employee { String? name;



int? age;

String? subject; double? salary;

// Constructor

Employee(this.name, this.age, [this.subject = "N/A", this.salary=0]);

// Method

void display() {

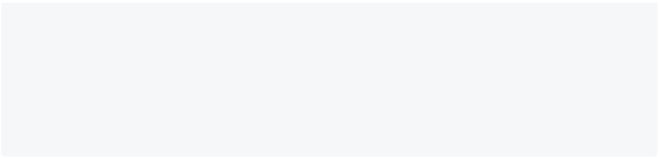
print("Name: ${this.name}");

print("Age: ${this.age}");

print("Subject: ${this.subject}"); print("Salary: ${this.salary}"); }

}

void main(){



Employee employee = Employee("John", 30); employee.display();

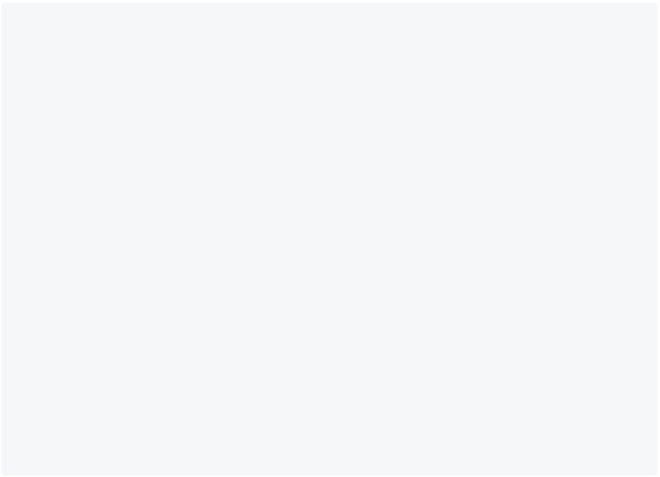
}

[Run Online](https://dartpad.dev/?id=841e4c019446527d2fddcb7a92c3fa3e)

**Example 7: Constructor With Named Parameters**

In the example below, we have created a class **Chair** with two properties: **name** and **color**. Class has one constructor for initializing the all properties values with named parameters. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Chair** called **chair**.

class Chair { String? name; String? color;



// Constructor Chair({this.name, this.color});

// Method

void display() {

print("Name: ${this.name}"); print("Color: ${this.color}"); }

}

void main(){

Chair chair = Chair(name: "Chair1", color: "Red"); chair.display();

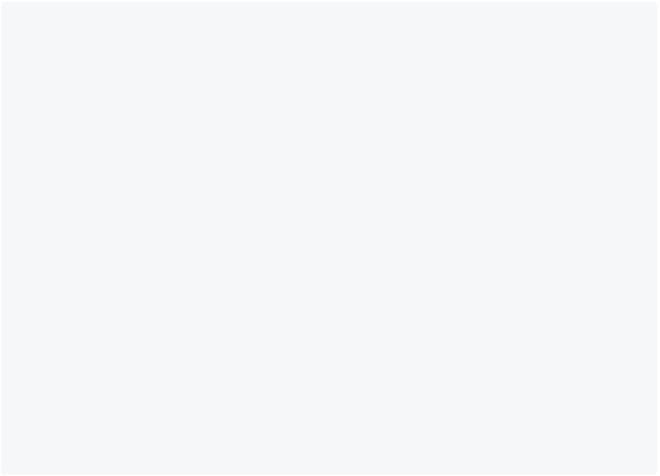
}

[Run Online](https://dartpad.dev/?id=9aa428d7c5e498d5233345470cf299c4)

**Example 8: Constructor With Default Values**

In the example below, we have created a class **Table** with two properties: **name** and **color**. Class has one constructor for initializing the all properties values with default values. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Table** called **table**.

class Table {



String? name; String? color;

// Constructor

Table({this.name = "Table1", this.color = "White"});

// Method

void display() {

print("Name: ${this.name}"); print("Color: ${this.color}"); }

}

void main(){

Table table = Table(); table.display();

}

[Run Online](https://dartpad.dev/?id=a6e04e8b75208952220b899e13e60feb) **Key Points**

* The constructor’s name should be the same as the class name.
  + Constructor doesn’t have any return type.
    - Constructor is only called once at the time of the object creation.
      * Constructor is called automatically when an object is created.
        + Constructor is used to initialize the values of the properties of the class.

### Default Constructor in Dart

The constructor which is automatically created by the dart compiler if you don’t create a constructor is called a default constructor. A default constructor has no parameters. A default constructor is declared using the class name followed by parentheses ().

**Example 1: Default Constructor In Dart**

In this example below, there is a class **Laptop** with two properties: **brand**, and **price**. Lets create constructor with no parameter and print something from the constructor. We also have an object of the class **Laptop** called **laptop**.

class Laptop { String? brand; int? price;



// Constructor

Laptop() {

print("This is a default constructor"); }

}

void main() {

// Here laptop is object of class Laptop. Laptop laptop = Laptop();

}

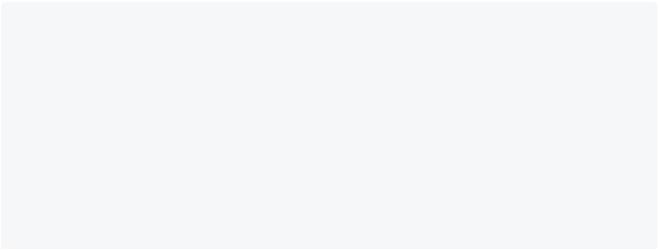
[Run Online](https://dartpad.dev/?id=5e6ea41c667f47ea9114142c4059a2fa)

**Note**: The default constructor is called automatically when you create an object of the class. It is used to initialize the instance variables of the class.

**Example 2: Default Constructor In Dart**

In this example below, there is a class **Student** with four properties: **name**, **age**, **schoolname** and **grade**. The default constructor is used to initialize the values of the school name. The reason for this is that the school name is the same for all the students. We also have an object of the class **Student** called **student**. The default constructor is called automatically when you create an object of the class.

class Student {



String? name;

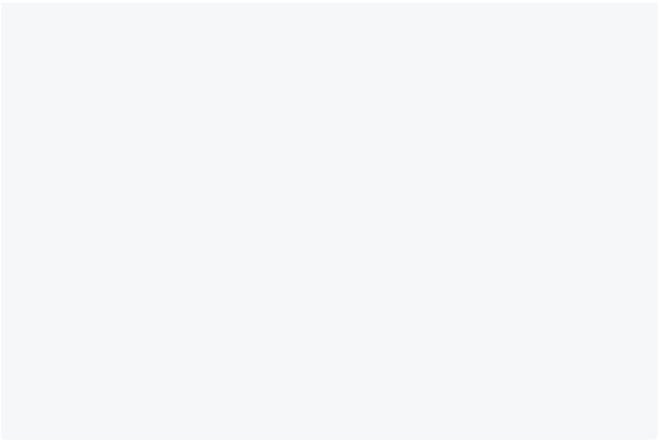
int? age;

String? schoolname; String? grade;

// Default Constructor Student() {

print(

"Constructor called"); // this is for checking the constructor is called or not.



schoolname = "ABC School";

}

}

void main() {

// Here student is object of class Student. Student student = Student();

student.name = "John";

student.age = 10;

student.grade = "A";

print("Name: ${student.name}");

print("Age: ${student.age}");

print("School Name: ${student.schoolname}"); print("Grade: ${student.grade}");

}

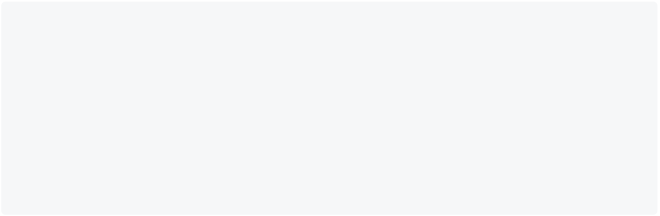
[Run Online](https://dartpad.dev/?id=59d9e16d1c95bf8f2e23e88ce77e5992)

### Parameterized Constructor in Dart

Parameterized constructor is used to initialize the instance variables of the class. Parameterized constructor is the constructor that takes parameters. It is used to pass the values to the constructor at the time of object creation.

**Syntax**

class ClassName {



// Instance Variables

int? number;

String? name;

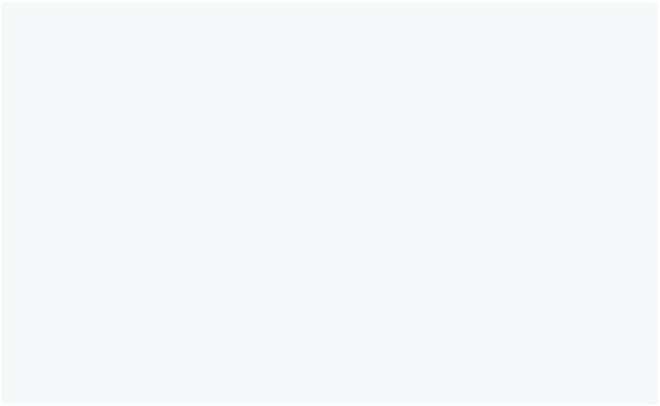
// Parameterized Constructor

ClassName(this.number, this.name); }

**Example 1: Parameterized Constructor In Dart**

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

class Student {



String? name;

int? age;

int? rollNumber;

// Constructor

Student(this.name, this.age, this.rollNumber); }

void main(){

// Here student is object of class Student. Student student = Student("John", 20, 1);

print("Name: ${student.name}");

print("Age: ${student.age}");

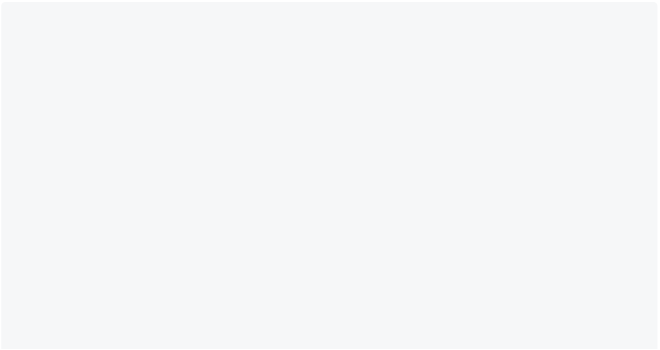
print("Roll Number: ${student.rollNumber}"); }

[Run Online](https://dartpad.dev/?id=099d5b4383b42ffc1277811e87378f35)

**Example 2: Parameterized Constructor With Named Parameters In Dart**

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

class Student {



String? name;

int? age;

int? rollNumber;

// Constructor

Student({String? name, int? age, int? rollNumber}) { this.name = name;

this.age = age;

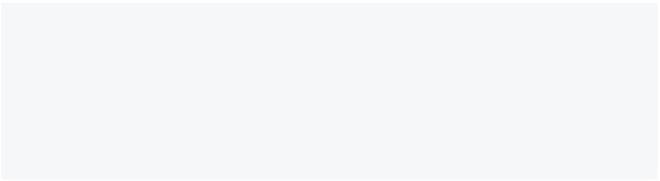
this.rollNumber = rollNumber;

}

}

void main(){

// Here student is object of class Student.



Student student = Student(name: "John", age: 20, rollNumber: 1); print("Name: ${student.name}");

print("Age: ${student.age}");

print("Roll Number: ${student.rollNumber}");

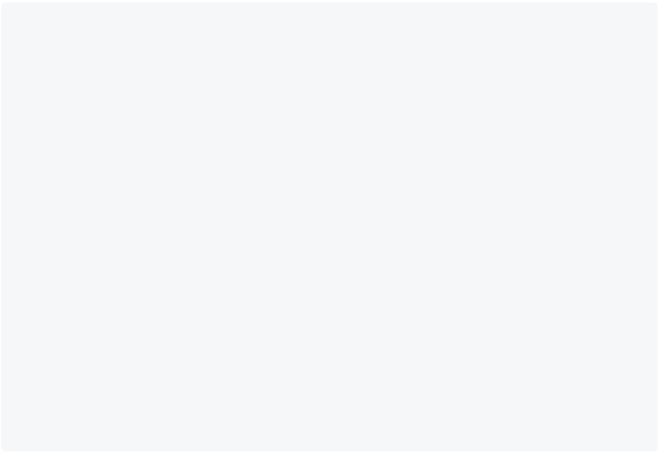
}

[Run Online](https://dartpad.dev/?id=5f9afa5783945e2135f36500641defba)

**Example 3: Parameterized Constructor With Default Values In Dart**

In this example below, there is class **Student** with two properties: **name**, and **age**. The class has parameterized constructor with default values. The constructor is used to initialize the values of the two properties. We also have an object of the class **Student** called **student**.

class Student { String? name; int? age;



// Constructor

Student({String? name = "John", int? age = 0}) { this.name = name;

this.age = age;

}

}

void main(){

// Here student is object of class Student. Student student = Student();

print("Name: ${student.name}");

print("Age: ${student.age}");

}

[Run Online](https://dartpad.dev/?id=9518729429347d37ec7344b9967377fa)

**Note**: In parameterized constructor, at the time of object creation, you must pass the parameters through the constructor which initialize the variables value, avoiding the null values.

### Named Constructor in Dart

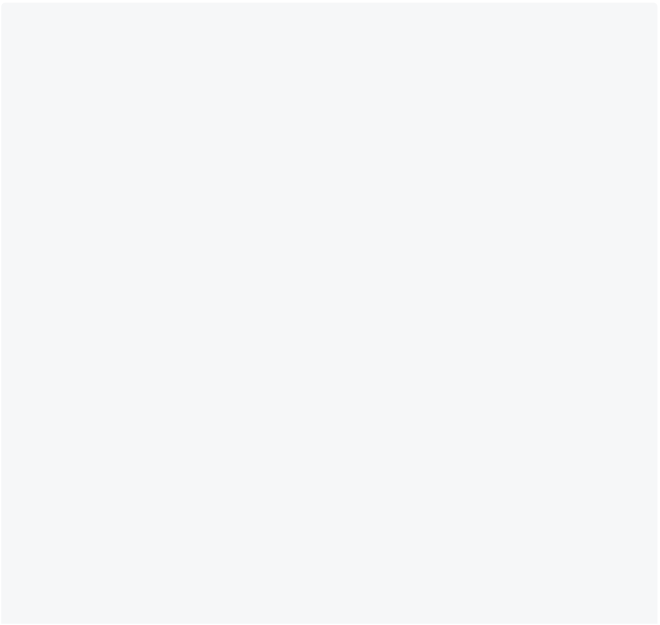
In most programming languages like java, c++, c#, etc., we can create multiple constructors with the same name. But in Dart, this is not possible. Well, there is a way. We can create multiple constructors with the same name using **named constructors**.

**Note**: Named constructors improves code readability. It is useful when you want to create multiple constructors with the same name.

**Example 1: Named Constructor In Dart**

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has two constructors. The first constructor is a default constructor. The second constructor is a named constructor. The named constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

class Student {



String? name;

int? age;

int? rollNumber;

// Default Constructor

Student() {

print("This is a default constructor"); }

// Named Constructor

Student.namedConstructor(String name, int age, int rollNumber) { this.name = name;

this.age = age;

this.rollNumber = rollNumber;

}

}

void main() {

// Here student is object of class Student.

Student student = Student.namedConstructor("John", 20, 1); print("Name: ${student.name}");

print("Age: ${student.age}");

print("Roll Number: ${student.rollNumber}");

}

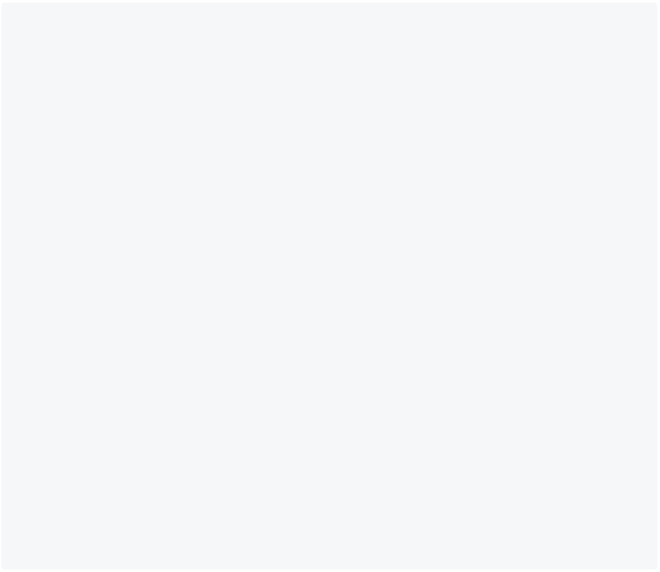


[Run Online](https://dartpad.dev/?id=01cbd57409dec39915c1011f79cc166a)

**Example 2: Named Constructor In Dart**

In this example below, there is class **Mobile** with three properties **name**, **color**, and **price**. The class has one method **display** which prints out the values of the three properties. We also have an object of the class **Mobile** called **mobile**. There is also constructor **Mobile** which takes all the three properties as parameters. Named constructor **Mobile.namedConstructor** is used to create an object of the class **Mobile** with name, color and optional price. The default value of the price is 0. If the price is not passed, then the default value is used.

class Mobile { String? name; String? color; int? price;



Mobile(this.name, this.color, this.price);

// here Mobile() is a named constructor

Mobile.namedConstructor(this.name, this.color, [this.price = 0]);

void displayMobileDetails() {

print("Mobile name: $name."); print("Mobile color: $color."); print("Mobile price: $price"); }

}

void main() {

var mobile1 = Mobile("Samsung", "Black", 20000);

mobile1.displayMobileDetails();

var mobile2 = Mobile.namedConstructor("Apple", "White");

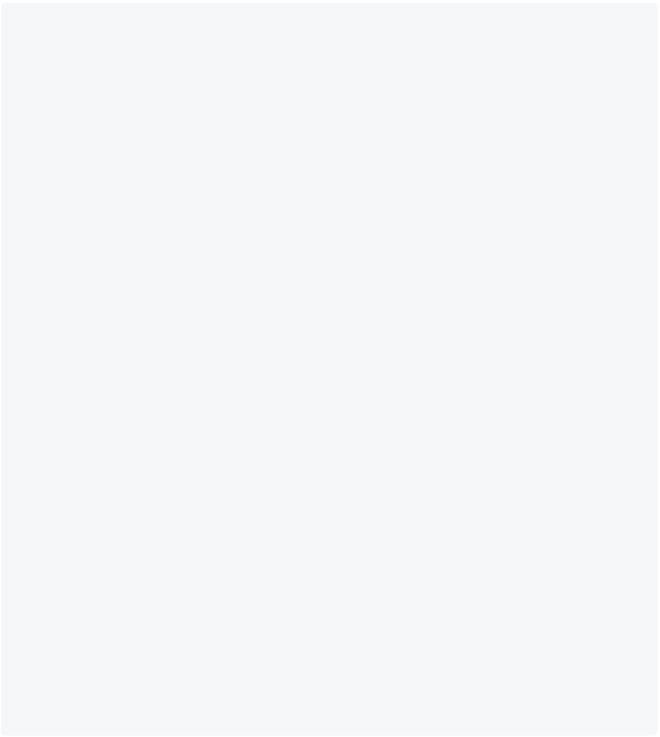
mobile2.displayMobileDetails(); }

[Run Online](https://dartpad.dev/?id=4bbc78cfd447ac22205b312e261ce881)

**Example 3: Named Constructor In Dart**

In this example below, there is a class **Animal** with two properties **name** and **age**. The class has three constructors. The first constructor is a default constructor. The second and third constructors are named constructors. The second constructor is used to initialize the values of name and age, and the third constructor is used to initialize the value of name only. We also have an object of the class **Animal** called **animal**.

class Animal { String? name; int? age;



// Default Constructor

Animal() {

print("This is a default constructor"); }

// Named Constructor

Animal.namedConstructor(String name, int age) { this.name = name;

this.age = age;

}

// Named Constructor

Animal.namedConstructor2(String name) {

this.name = name;

}

}

void main(){

// Here animal is object of class Animal.

Animal animal = Animal.namedConstructor("Dog", 5); print("Name: ${animal.name}");

print("Age: ${animal.age}");

Animal animal2 = Animal.namedConstructor2("Cat"); print("Name: ${animal2.name}");

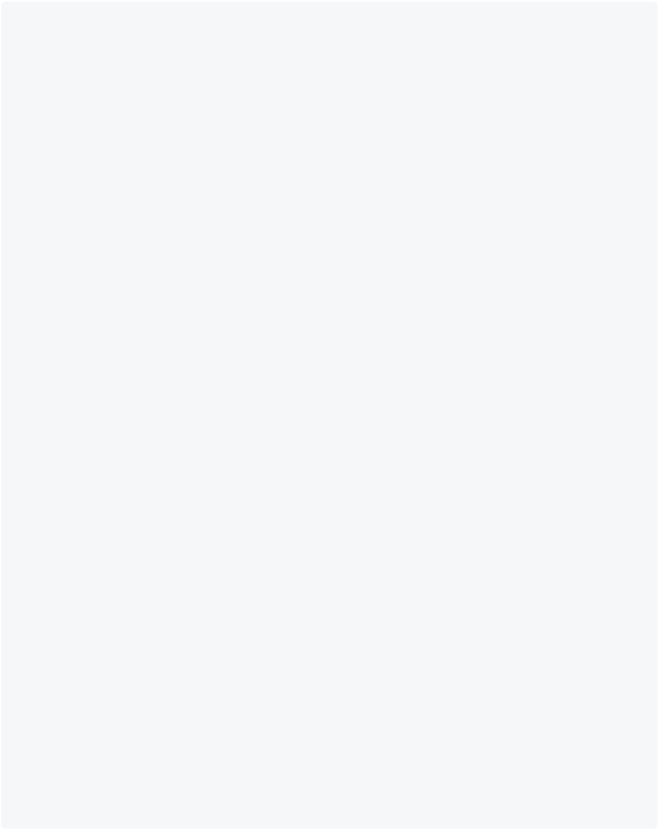
}

[Run Online](https://dartpad.dev/?id=9daa38f8a39ff46dd222adfbd0c4e2c7)

**Example 4: Real Life Example Of Named Constructor In Dart**

In this example below, there is a class **Person** with two properties **name** and **age**. The class has three constructors. The first is a parameterized constructor which takes two parameters **name** and **age**. The second and third constructors are named constructors. Second constructor fromJson is used to create an object of the class **Person** from a JSON. The third fromJsonString is used to create an object of the class **Person** from a JSON string. We also have an object of the class **Person** called **person**.

import 'dart:convert';



class Person { String? name; int? age;

Person(this.name, this.age);

Person.fromJson(Map<String, dynamic> json) { name = json['name'];

age = json['age'];

}

Person.fromJsonString(String jsonString) {

Map<String, dynamic> json = jsonDecode(jsonString); name = json['name'];

age = json['age'];

}

}

void main() {

// Here person is object of class Person.

String jsonString1 = '{"name": "Bishworaj", "age": 25}'; String jsonString2 = '{"name": "John", "age": 30}';

Person p1 = Person.fromJsonString(jsonString1); print("Person 1 name: ${p1.name}");

print("Person 1 age: ${p1.age}");

Person p2 = Person.fromJsonString(jsonString2); print("Person 2 name: ${p2.name}");

print("Person 2 age: ${p2.age}");

}

[Run Online](https://dartpad.dev/?id=4e57d5b04142ae9ac3ceb963cf1ea3fc)

### Constant Constructor in Dart

**Constant constructor** is a constructor that creates a constant object. A constant object is an object whose value cannot be changed. A constant constructor is declared using the keyword **const**.

**Note**: **Constant Constructor** is used to create a object whose value cannot be changed. It Improves the performance of the program.

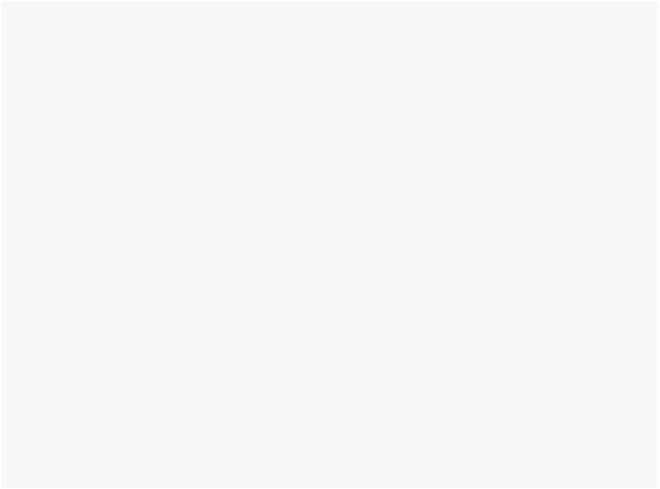
**Rule For Declaring Constant Constructor In Dart**

* All properties of the class must be final.
  + It does not have any body.
    - Only class containing **const** constructor is initialized using the **const** keyword.

**Example 1: Constant Constructor In Dart**

In this example below, there is a class **Point** with two final properties: **x** and **y**. The class also has a constant constructor that initializes the two properties. The class also has a method called **display**, which prints out the values of the two properties.

class Point { final int x; final int y;



const Point(this.x, this.y); }

void main() {

// p1 and p2 has the same hash code.

Point p1 = const Point(1, 2);

print("The p1 hash code is: ${p1.hashCode}");

Point p2 = const Point(1, 2);

print("The p2 hash code is: ${p2.hashCode}"); // without using const

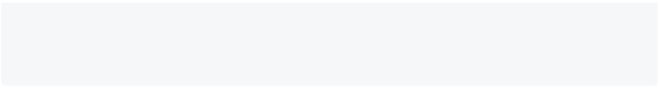
// this has different hash code.

Point p3 = Point(2, 2);

print("The p3 hash code is: ${p3.hashCode}");

Point p4 = Point(2, 2);

print("The p4 hash code is: ${p4.hashCode}"); }



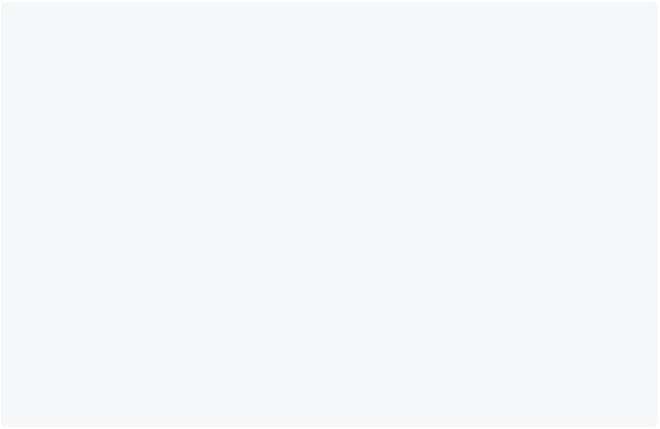
[Run Online](https://dartpad.dev/?id=45137d89664562d2a4dfaac1b1fc18bb)

**Note**: Here p1 and p2 has the same hash code. This is because p1 and p2 are constant objects. The hash code of a constant object is the same. This is because the hash code of a constant object is computed at compile time. The hash code of a non-constant object is computed at run time. This is why p3 and p4 have different hash code.

**Example 2: Constant Constructor In Dart**

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constant constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

class Student {



final String? name;

final int? age;

final int? rollNumber;

// Constant Constructor

const Student({this.name, this.age, this.rollNumber}); }

void main() {

// Here student is object of Student.

const Student student = Student(name: "John", age: 20, rollNumber: 1); print("Name: ${student.name}");

print("Age: ${student.age}");

print("Roll Number: ${student.rollNumber}");

}

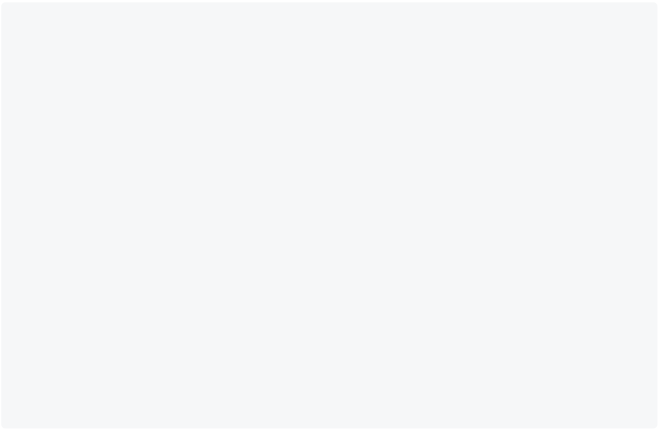
[Run Online](https://dartpad.dev/?id=e90b731d35c72aa9f171b8895f26f6c1)

**Example 3: Constant Constructor With Named Parameters In Dart**

In this example below, there is a class **Car** with three properties: **name**, **model**, and **price**. The class has one constructor. The constructor is used to initialize the values

of the three properties. We also have an object of the class **Car** called **car**.

class Car {



final String? name; final String? model; final int? price;

// Constant Constructor

const Car({this.name, this.model, this.price}); }

void main() {

// Here car is object of class Car.

const Car car = Car(name: "BMW", model: "X5", price: 50000); print("Name: ${car.name}");

print("Model: ${car.model}");

print("Price: ${car.price}");

}

[Run Online](https://dartpad.dev/?id=98af5674f23e052615b129554eb170a7)

**Benefits Of Constant Constructor In Dart**

Improves the performance of the program.



### Encapsulation in Dart

In this section, you will learn about **encapsulation in Dart** programming language with examples. Encapsulation is one of the important concepts of object-oriented programming. Before learning about dart encapsulation, you should have a basic understanding of the class and object in dart.

**Encapsulation In Dart**

In Dart, **Encapsulation** means **hiding data** within a library, preventing it from outside factors. It helps you control your program and prevent it from becoming too complicated.

**What Is Library In Dart?**

By default, every **.dart** file is a library. A library is a collection of functions and classes. A library can be imported into another library using the **import** keyword.

**How To Achieve Encapsulation In Dart?** Encapsulation can be achieved by:

* Declaring the class properties as **private** by using **underscore(\_)**.
  + Providing public **getter** and **setter** methods to access and update the value of private property.

**Note:** Dart doesn’t support keywords like **public**, **private**, and **protected**. Dart uses **\_** (underscore) to make a property or method private. The encapsulation happens at library level, not at class level.

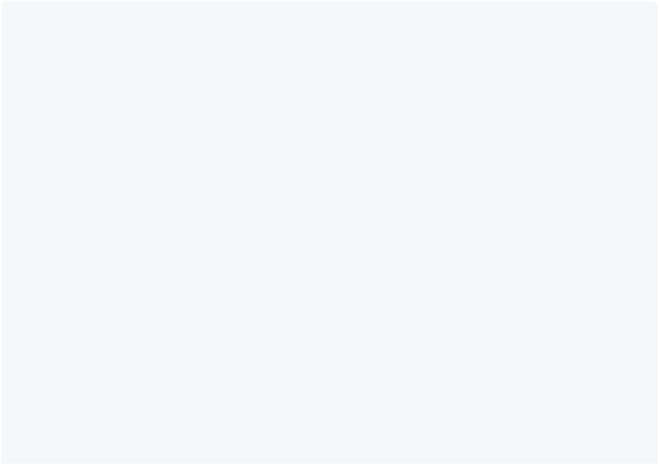
**Getter and Setter Methods**

**Getter** and **setter** methods are used to access and update the value of private property. **Getter** methods are used to access the value of private property. **Setter** methods are used to update the value of private property.

**Example 1: Encapsulation In Dart**

In this example, we will create a class named **Employee**. The class will have two private properties **\_id** and **\_name**. We will also create two public methods **getId()** and **getName()** to access the private properties. We will also create two public methods **setId()** and **setName()** to update the private properties.

class Employee {



// Private properties int? \_id;

String? \_name;

// Getter method to access private property \_id int getId() {

return \_id!;

}

// Getter method to access private property \_name String getName() {

return \_name!;

}

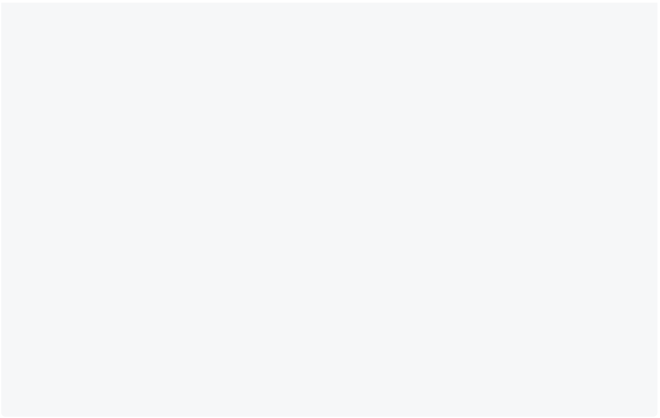
// Setter method to update private property \_id void setId(int id) {

this.\_id = id;

}

// Setter method to update private property \_name void setName(String name) {

this.\_name = name; }



}

void main() {

// Create an object of Employee class

Employee emp = new Employee();

// setting values to the object using setter emp.setId(1);

emp.setName("John");

// Retrieve the values of the object using getter print("Id: ${emp.getId()}");

print("Name: ${emp.getName()}");

}

[Run Online](https://dartpad.dev/?id=536723b1febc22e47a41b2053e29ca8d)

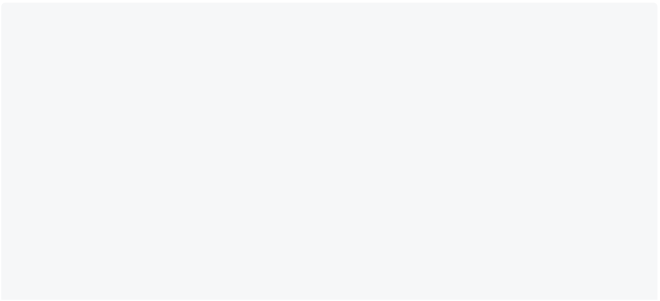
**Private Properties**

**Private property** is a property that can only be accessed from same **library**. Dart does not have any keywords like **private** to define a private property. You can define it by prefixing an **underscore (\_)** to its name.

**Example 2: Private Properties In Dart**

In this example, we will create a class named **Employee**. The class has one private property **\_name**. We will also create a public method **getName()** to access the private property.

class Employee {



// Private property var \_name;

// Getter method to access private property \_name String getName() {

return \_name;

}

// Setter method to update private property \_name void setName(String name) {

this.\_name = name;

} }



void main() {

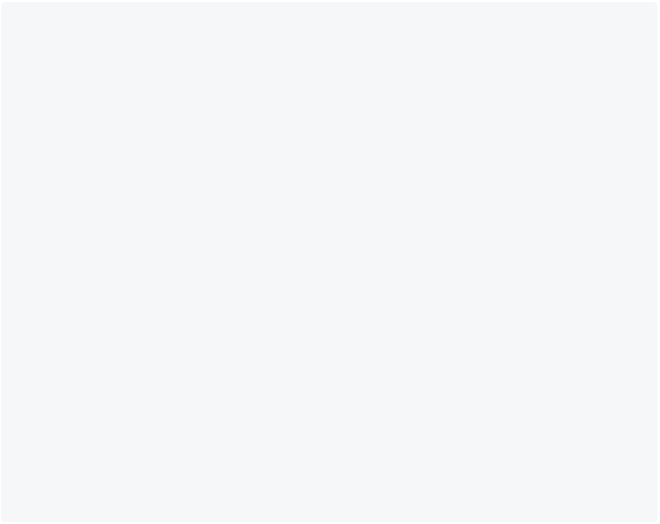
var employee = Employee(); employee.setName("Jack"); print(employee.getName()); }

[Run Online](https://dartpad.dev/?id=cbf4b6fcfd4bb0cffabb8f7eb352331e)

**Why Aren’t Private Properties Private?**

In the main method, if you write the following code, it will compile and run without any error. Let’s see why it is happening.

class Employee {



// Private property var \_name;

// Getter method to access private property \_name String getName() {

return \_name;

}

// Setter method to update private property \_name void setName(String name) {

this.\_name = name;

}

}

void main() {

var employee = Employee();

employee.\_name = "John"; // It is working, but why? print(employee.getName());

}

[Run Online](https://dartpad.dev/?id=d165c18b438ea6d57240577c88116e23) **Reason**

The reason is that using **underscore (\_)** before a variable or method name makes it **library private** not **class private**. It means that the variable or method is only visible to the library in which it is declared. It is not visible to any other library. In simple words, library is one file. If you write the main method in a separate file, this will not work.

**Solution**

To see private properties in action, you must create a separate file for the class and import it into the main file.



**Read-only Properties**

You can control the properties’s access and implement the encapsulation in the dart by using the read-only properties. You can do that by adding the **final** keyword before the properties declaration. Hence, you can only access its value, but you cannot change it.

**Note:** Properties declared with the **final** keyword must be initialized at the time of declaration. You can also initialize them in the constructor.

class Student {



final \_schoolname = "ABC School";

String getSchoolName() { return \_schoolname;

}

}

void main() {

var student = Student();

print(student.getSchoolName());

// This is not possible

//student.\_schoolname = "XYZ School"; }

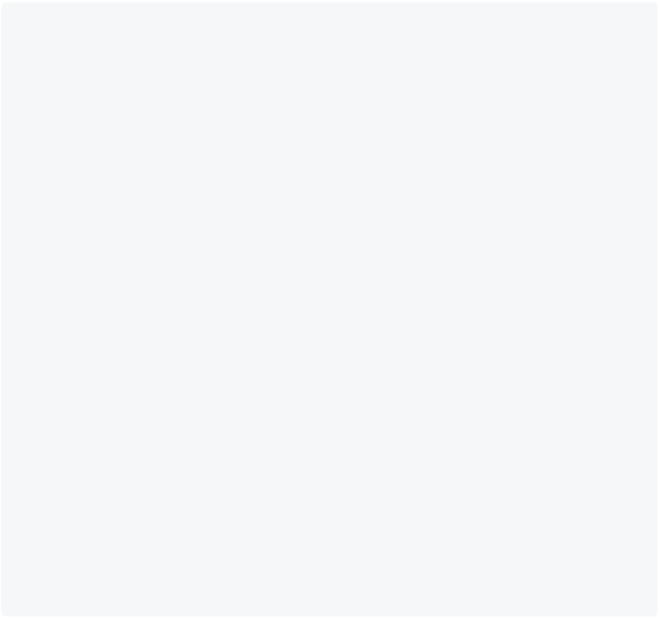
[Run Online](https://dartpad.dev/?id=f340b4a448d078f9a195af7a76541fb9)

**Note:** You can also define **getter** and **setter** using **get** and **set** keywords. For more see this example below.

**How To Create Getter and Setter Methods?**

You can create getter and setter methods by using the **get** and **set** keywords. In this example below, we have created a class named **Vehicle**. The class has two private properties **\_model** and **\_year**. We have also created two getter and setter methods for each property. The getter and setter methods are named **model** and **year**. The getter and setter methods are used to access and update the value of the private properties.

class Vehicle { String \_model; int \_year;



// Getter method

String get model => \_model;

// Setter method

set model(String model) => \_model = model;

// Getter method

int get year => \_year;

// Setter method

set year(int year) => \_year = year; }

void main() {

var vehicle = Vehicle(); vehicle.model = "Toyota"; vehicle.year = 2019;

print(vehicle.model);

print(vehicle.year);

}

[Run Online](https://dartpad.dev/?id=59c98ac34e8e994fc483e353c5f1b43c)

**Note:** In dart, any identifier like (class, class properties, top-level function, or variable) that starts with an underscore \_ it is private to its library.

**Why Encapsulation Is Important?**

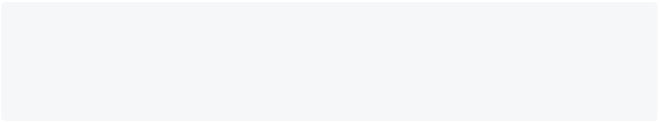
* **Data Hiding**: Encapsulation hides the data from the outside world. It prevents the data from being accessed by the code outside the class. This is known as data hiding.
  + **Testability**: Encapsulation allows you to test the class in isolation. It will enable you to test the class without testing the code outside the class.
    - **Flexibility**: Encapsulation allows you to change the implementation of the class without affecting the code outside the class.
      * **Security**: Encapsulation allows you to restrict access to the class members. It will enable you to limit access to the class members from the code outside the library.

### Getter in Dart

**Getter** is used to get the value of a property. It is mostly used to access a **private property’s** value. Getter provide explicit read access to an object properties.

**Syntax**

return\_type get property\_name { // Getter body



}

**Note:** Instead of writing { } after the property name, you can also write **=>** (fat arrow) after the property name.

**Example 1: Getter In Dart**

In this example below, there is a class named **Person**. The class has two properties **firstName** and **lastName**. There is getter **fullName** which is responsible to get full name of person.

class Person {



// Properties

String? firstName; String? lastName;

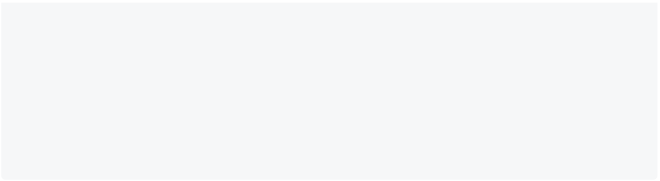
// Constructor

Person(this.firstName, this.lastName);

// Getter

String get fullName => "$firstName $lastName";

}



void main() {

Person p = Person("John", "Doe"); print(p.fullName);

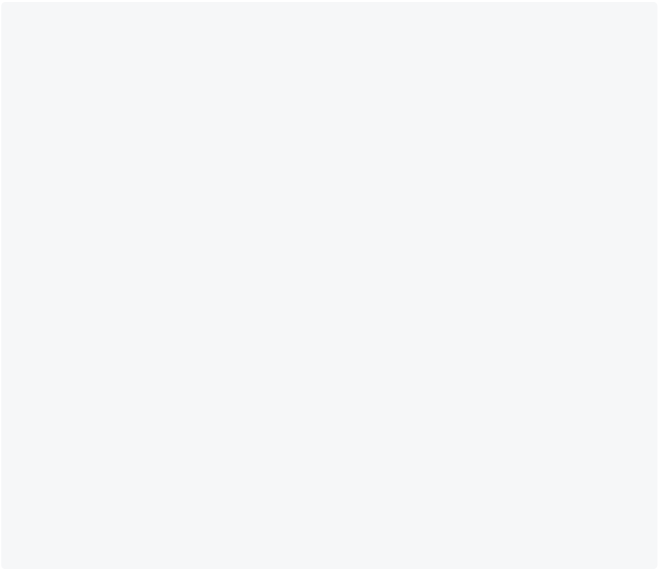
}

[Run Online](https://dartpad.dev/?id=09842075da5b538616d6cafc57066174)

**Example 2: Getter In Dart**

In this example below, there is a class named **NoteBook**. The class has two private properties **\_name** and **\_prize**. There are two getters **name** and **price** to access the value of the properties.

class NoteBook {



// Private properties String? \_name;

double? \_prize;

// Constructor

NoteBook(this.\_name, this.\_prize);

// Getter method to access private property \_name String get name => this.\_name!;

// Getter method to access private property \_prize double get price => this.\_prize!;

}

void main() {

// Create an object of NoteBook class

NoteBook nb = new NoteBook("Dell", 500); // Display the values of the object

print(nb.name);

print(nb.price);

}

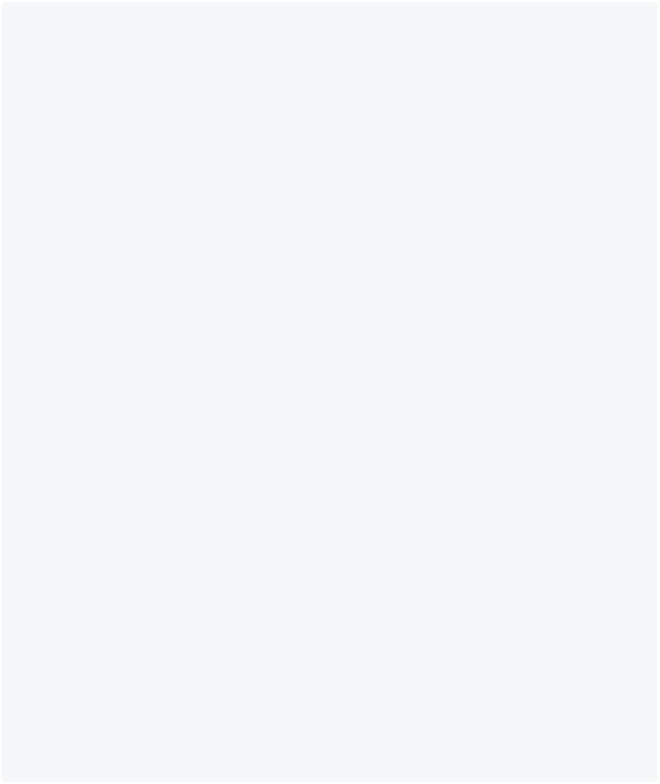
[Run Online](https://dartpad.dev/?id=d9b2a23ec157805d7c44ed6b5e382098)

**Note:** In the above example, a getter **name** and **price** are used to access the value of the properties **\_name** and **\_prize**.

**Example 3: Getter In Dart With Data Validation**

In this example below, there is a class named **NoteBook**. The class has two private properties **\_name** and **\_prize**. There are two getters **name** and **price** to access the value of the properties. If you provide a blank name, then it will return **No Name**.

class NoteBook {



// Private properties String \_name;

double \_prize;

// Constructor

NoteBook(this.\_name, this.\_prize);

// Getter to access private property \_name String get name {

if (\_name == "") {

return "No Name";

}

return this.\_name;

}

// Getter to access private property \_prize double get price {

return this.\_prize;

}

}

void main() {

// Create an object of NoteBook class

NoteBook nb = new NoteBook("Apple", 1000);

print("First Notebook name: ${nb.name}");

print("First Notebook price: ${nb.price}"); NoteBook nb2 = new NoteBook("", 500);

print("Second Notebook name: ${nb2.name}"); print("Second Notebook price: ${nb2.price}");

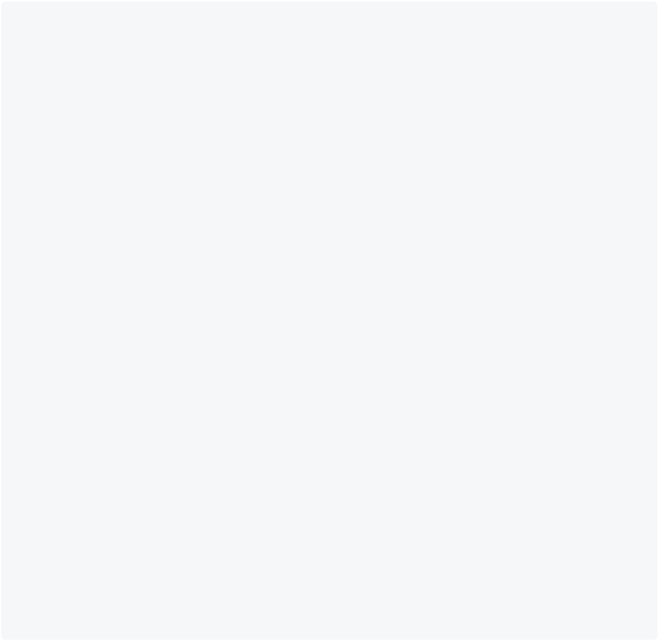
}

[Run Online](https://dartpad.dev/?id=dc3151d967367e0a5d3e85f2378188ae)

**Example 4: Getter In Dart**

In this example below, there is a class named **Doctor**. The class has three private properties **\_name**, **\_age** and **\_gender**. There are three getters **name**, **age**, and **gender** to access the value of the properties. It has **map** getter to get [**Map**](https://dart-tutorial.com/collections/map-in-dart/) of the object.

class Doctor {



// Private properties String \_name;

int \_age;

String \_gender;

// Constructor

Doctor(this.\_name, this.\_age, this.\_gender);

// Getters

String get name => \_name;

int get age => \_age;

String get gender => \_gender;

// Map Getter

Map<String, dynamic> get map {

return {"name": \_name, "age": \_age, "gender": \_gender}; }

}

void main() {

// Create an object of Doctor class

Doctor d = Doctor("John", 41, "Male"); print(d.map);

}

[Run Online](https://dartpad.dev/?id=9d2fe00e4026977942d8496f6658f513)

**Why Is Getter Important In Dart?**

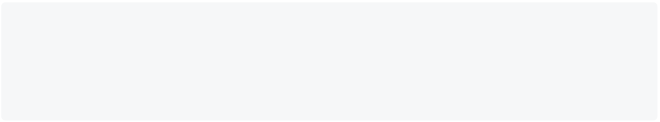
* To access the value of private property.
  + To restrict the access of data members of a class.

### Setter in Dart

**Setter** is used to set the value of a property. It is mostly used to update a **private property’s** value. Setter provide explicit write access to an object properties.

**Syntax**

set property\_name (value) { // Setter body



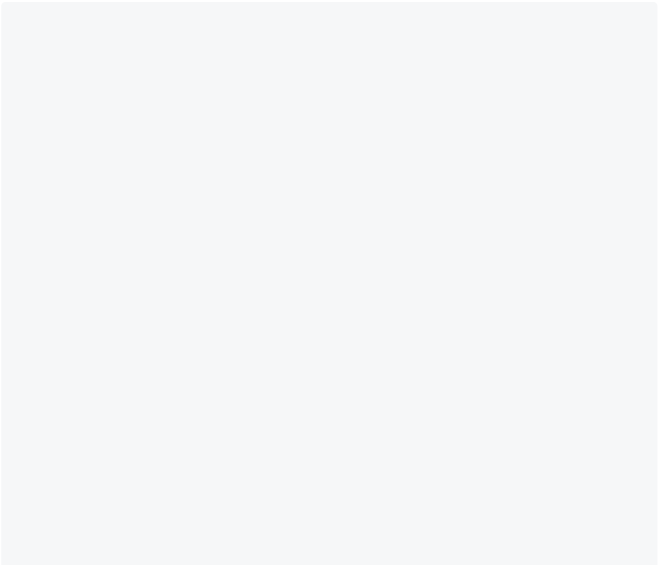
}

**Note:** Instead of writing { } after the property name, you can also write **=>** (fat arrow) after the property name.

**Example 1: Setter In Dart**

In this example below, there is a class named **NoteBook**. The class has two private properties **\_name** and **\_prize**. There are two setters **name** and **price** to update the value of the properties. There is also a method **display** to display the value of the properties.

class NoteBook {



// Private Properties String? \_name;

double? \_prize;

// Setter to update private property \_name set name(String name) => this.\_name = name;

// Setter to update private property \_prize

set price(double price) => this.\_prize = price;

// Method to display the values of the properties void display() {

print("Name: ${\_name}");

print("Price: ${\_prize}");

}

}

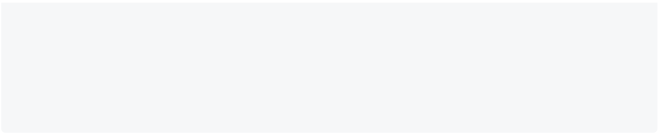
void main() {

// Create an object of NoteBook class

NoteBook nb = new NoteBook();

// setting values to the object using setter nb.name = "Dell";

nb.price = 500.00;



// Display the values of the object nb.display();

}

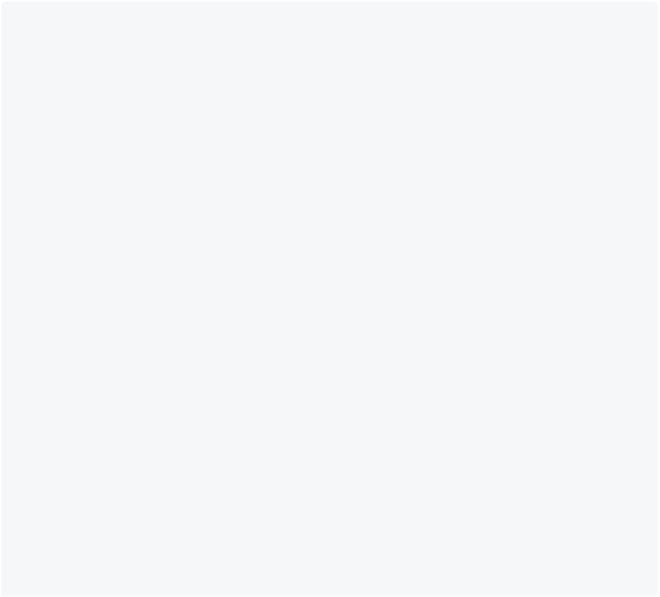
[Run Online](https://dartpad.dev/?id=1b5d019550b1964430157edcabf932fe)

**Note:** In the above example, a setter **name** and **price** are used to update the value of the properties **\_name** and **\_prize**.

**Example 2: Setter In Dart With Data Validation**

In this example, there is a class named **NoteBook**. The class has two private properties **\_name** and **\_prize**. If the value of **\_prize** is less than 0, we will throw an exception. There are also two setters **name** and **price** to update the value of the properties. The class also has a method **display()** to display the values of the properties.

class NoteBook {



// Private properties String? \_name;

double? \_prize;

// Setter to update the value of name property set name(String name) => \_name = name;

// Setter to update the value of price property

set price(double price) {

if (price < 0) {

throw Exception("Price cannot be less than 0"); }

this.\_prize = price;

}

// Method to display the values of the properties void display() {

print("Name: $\_name");

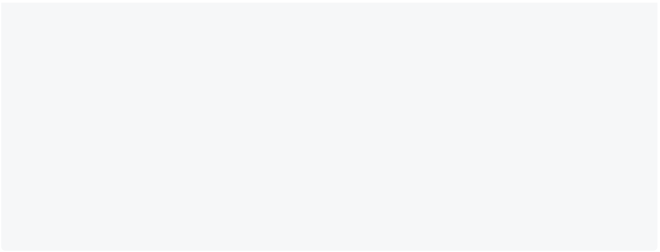
print("Price: $\_prize");

}

}

void main() {

// Create an object of NoteBook class



NoteBook nb = new NoteBook();

// setting values to the object using setter nb.name = "Dell";

nb.price = 250;

// Display the values of the object nb.display();

}

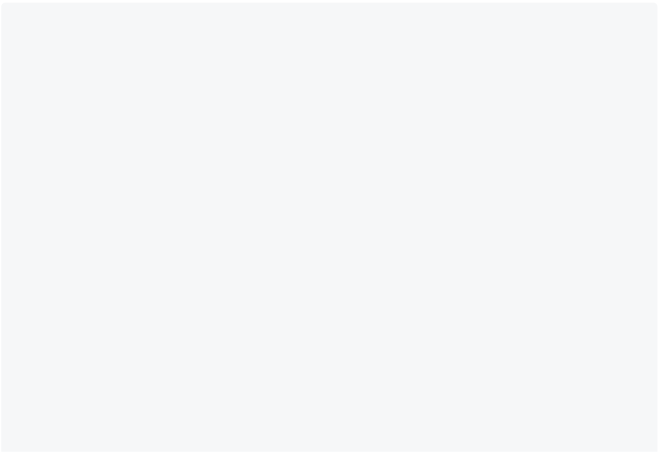
[Run Online](https://dartpad.dev/?id=a8b07d8a0dbeea1f5d15195aa4f7b344)

**Note**: It is generally best to not allow the user to set the value of a field directly. Instead, you should provide a setter method that can validate the value before setting it. This is very important when working on large and complex programs.

**Example 3: Setter In Dart**

In this example, there is a class named **Student**. The class has two private properties **\_name** and **\_classnumber**. We will also create two setters **name** and **classnumber** to update the value of the properties. The **classnumber** setter will only accept a value between 1 and 12. The class also has a method **display()** to display the values of the properties.

class Student {



// Private properties String? \_name;

int? \_classnumber;

// Setter to update the value of name property set name(String name) => this.\_name = name;

// Setter to update the value of classnumber property set classnumber(int classnumber) {

if (classnumber <= 0 || classnumber > 12) {

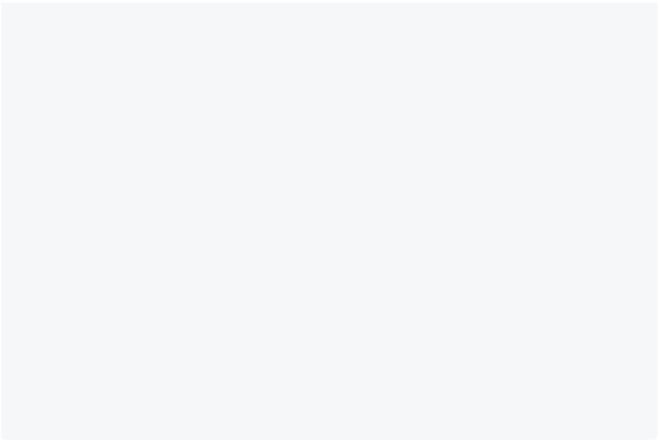
throw ('Classnumber must be between 1 and 12'); }

this.\_classnumber = classnumber;

}

// Method to display the values of the properties void display() {

print("Name: $\_name");



print("Class Number: $\_classnumber");

}

}

void main() {

// Create an object of Student class

Student s = new Student();

// setting values to the object using setter s.name = "John Doe";

s.classnumber = 12;

// Display the values of the object s.display();

// This will generate error //s.setClassNumber(13);

}

[Run Online](https://dartpad.dev/?id=f7f5d71b4d6c39a5bb60130b847ce8de)

**Why Is Setter Important?**

* It is used to set the value of a private property.
  + It is also used for data validation.
    - It gives you better control over the data.

### Inheritance in Dart

In this section, you will learn inheritance in Dart programming and how to define a class that reuses the properties and methods of another class.

**Inheritance In Dart**

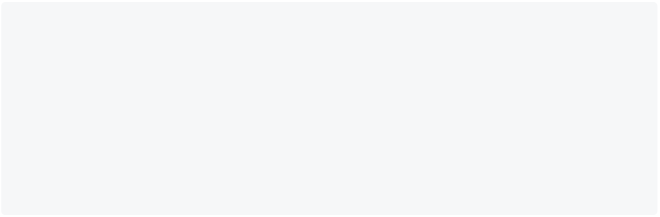
Inheritance is a sharing of behaviour between two classes. It allows you to define a class that extends the functionality of another class. The **extend** keyword is used for inheriting from parent class.

**Note**: Whenever you use inheritance, it always create a **is-a** relation between the parent and child class like **Student is a Person**, **Truck is a Vehicle**, **Cow is a Animal** etc.

Dart supports single inheritance, which means that a class can only inherit from a single class. Dart does not support multiple inheritance which means that a class

cannot inherit from multiple classes. **Syntax**

class ParentClass {



// Parent class code }

class ChildClass extends ParentClass { // Child class code

}

In this syntax, **ParentClass** is the super class and **ChildClass** is the sub class. The **ChildClass** inherits the properties and methods of the **ParentClass**.

**Terminology**

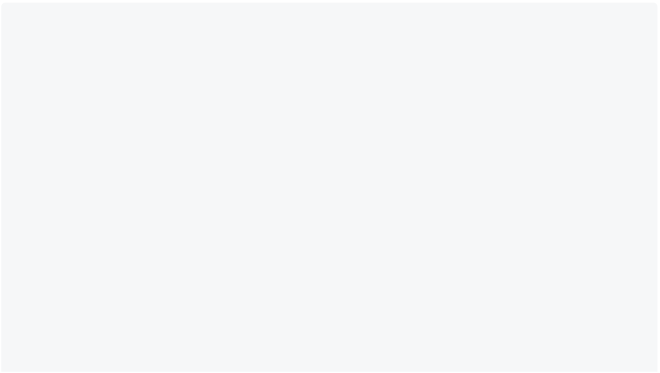
**Parent Class:** The class whose properties and methods are inherited by another class is called parent class. It is also known as base class or super class.

**Child Class:** The class that inherits the properties and methods of another class is called child class. It is also known as derived class or sub class.

**Example 1: Inheritance In Dart**

In this example, we will create a class **Person** and then create a class **Student** that inherits the properties and methods of the **Person** class.

class Person { // Properties String? name; int? age;



// Method

void display() {

print("Name: $name");

print("Age: $age");

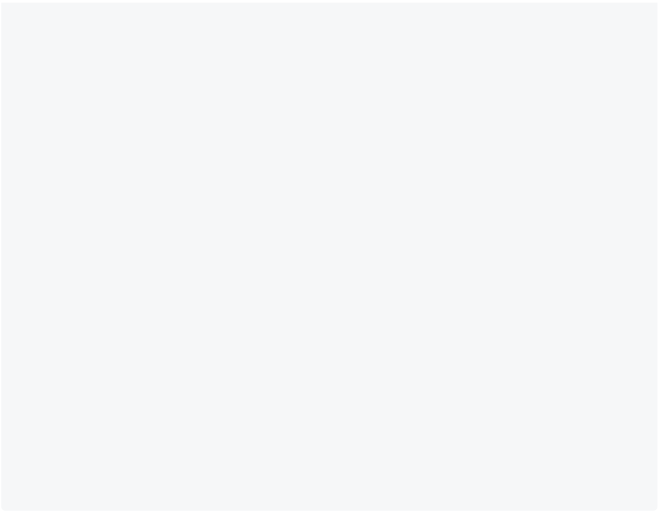
}

}

// Here In student class, we are extending the // properties and methods of the Person class class Student extends Person {

// Fields

String? schoolName;



String? schoolAddress;

// Method

void displaySchoolInfo() {

print("School Name: $schoolName");

print("School Address: $schoolAddress"); }

}

void main() {

// Creating an object of the Student class var student = Student();

student.name = "John";

student.age = 20;

student.schoolName = "ABC School";

student.schoolAddress = "New York";

student.display();

student.displaySchoolInfo();

}

[Run Online](https://dartpad.dev/?id=0b6b2c97c4b1e71e02bf4645461e7069)

**Advantages Of Inheritance In Dart**

* It promotes reusability of the code and reduces redundant code.
  + It helps to design a program in a better way.
    - It makes code simpler, cleaner and saves time and money on maintenance.
      * It facilitates the creation of class libraries.
        + It can be used to enforce standard interface to all children classes.

**Example 2: Inheritance In Dart**

In this example, here is parent class **Car** and child class **Toyota**. The **Toyota** class inherits the properties and methods of the **Car** class.

class Car{

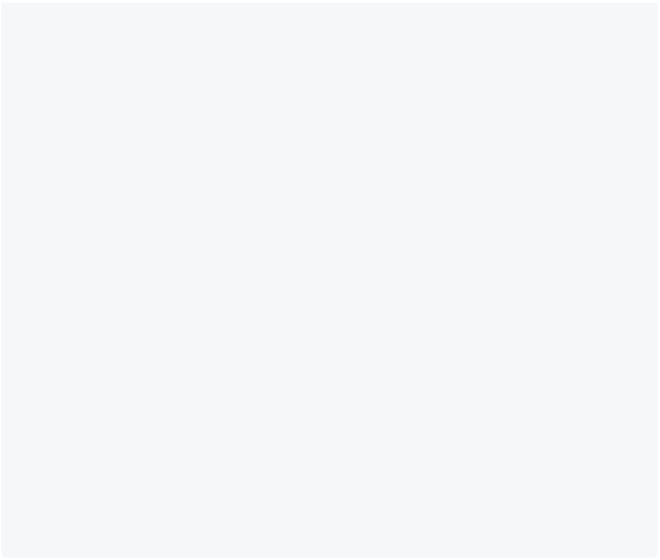


String color; int year;

void start(){

print("Car started");

} }



class Toyota extends Car{ String model;

int price;

void showDetails(){

print("Model: $model"); print("Price: $price"); }

}

void main(){

var toyota = Toyota(); toyota.color = "Red"; toyota.year = 2020;

toyota.model = "Camry"; toyota.price = 20000; toyota.start();

toyota.showDetails();

}

[Run Online](https://dartpad.dev/?id=0f5853514e223cfd1dba559f6b827200)

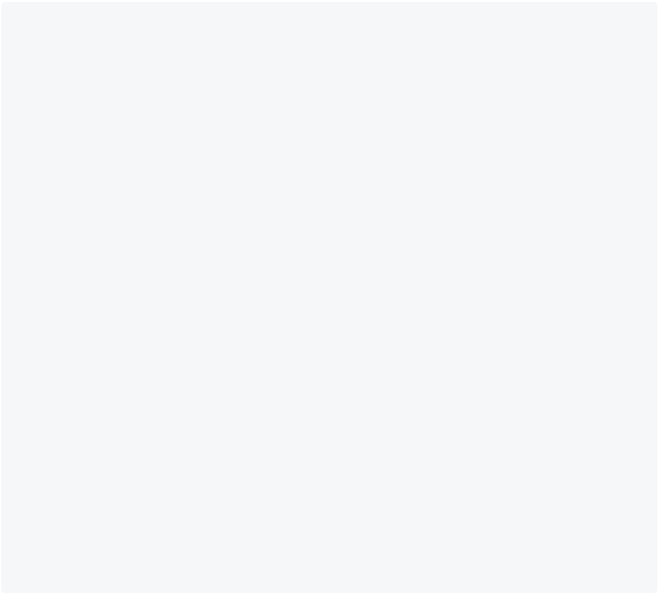
**Types Of Inheritance In Dart**

1. **Single Inheritance** - In this type of inheritance, a class can inherit from only one class. In Dart, we can only extend one class at a time.
2. **Multilevel Inheritance** - In this type of inheritance, a class can inherit from another class and that class can also inherit from another class. In Dart, we can extend a class from another class which is already extended from another class.
3. **Hierarchical Inheritance** - In this type of inheritance, parent class is inherited by multiple subclasses. For example, the **Car** class can be inherited by the **Toyota** class and **Honda** class.
4. **Multiple Inheritance** - In this type of inheritance, a class can inherit from multiple classes. **Dart does not support multiple inheritance.** For e.g. **Class Toyota extends Car, Vehicle {}** is not allowed in Dart.

**Example 3: Single Inheritance In Dart**

In this example below, there is super class named **Car** with two properties **name** and **price**. There is sub class named **Tesla** which inherits the properties of the super class. The sub class has a method **display** to display the values of the properties.

class Car {



// Properties String? name; double? price; }

class Tesla extends Car {

// Method to display the values of the properties void display() {

print("Name: ${name}");

print("Price: ${price}");

}

}

void main() {

// Create an object of Tesla class Tesla t = new Tesla();

// setting values to the object

t.name = "Tesla Model 3";

t.price = 50000.00;

// Display the values of the object t.display();

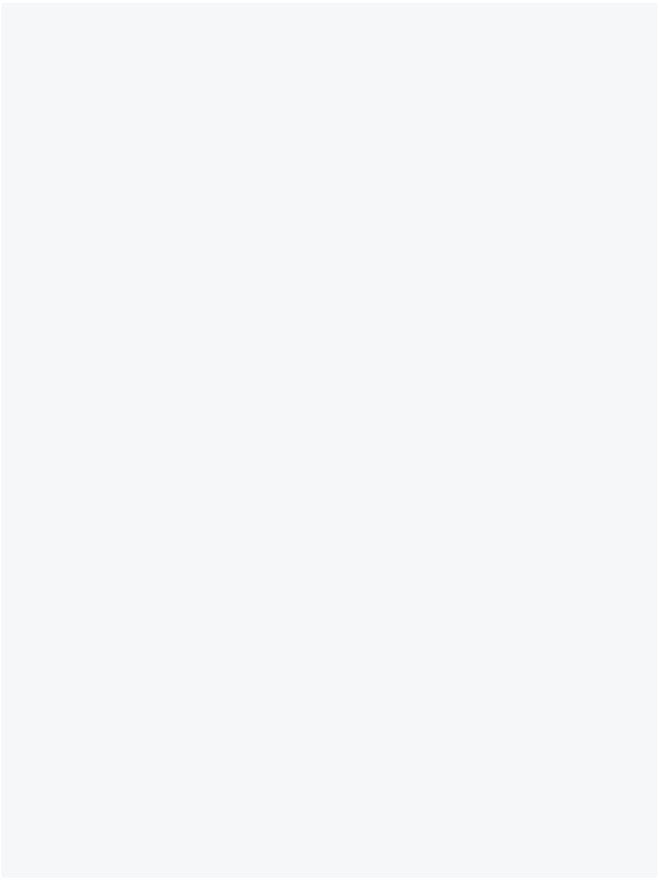
}

[Run Online](https://dartpad.dev/?id=e50d6b08e158f2d5f49d27448b6186e8)

**Example 4: Multilevel Inheritance In Dart**

In this example below, there is super class named **Car** with two properties **name** and **price**. There is sub class named **Tesla** which inherits the properties of the super class. The sub class has a method **display** to display the values of the properties. There is another sub class named **Model3** which inherits the properties of the sub class **Tesla**. The sub class has a property **color** and a method **display** to display the values of the properties.

class Car {



// Properties

String? name;

double? price; }

class Tesla extends Car {

// Method to display the values of the properties void display() {

print("Name: ${name}");

print("Price: ${price}");

}

}

class Model3 extends Tesla {

// Properties String? color;

// Method to display the values of the properties void display() {

super.display();

print("Color: ${color}");

}

}

void main() {

// Create an object of Model3 class Model3 m = new Model3();

// setting values to the object m.name = "Tesla Model 3";

m.price = 50000.00;

m.color = "Red";

// Display the values of the object

m.display();

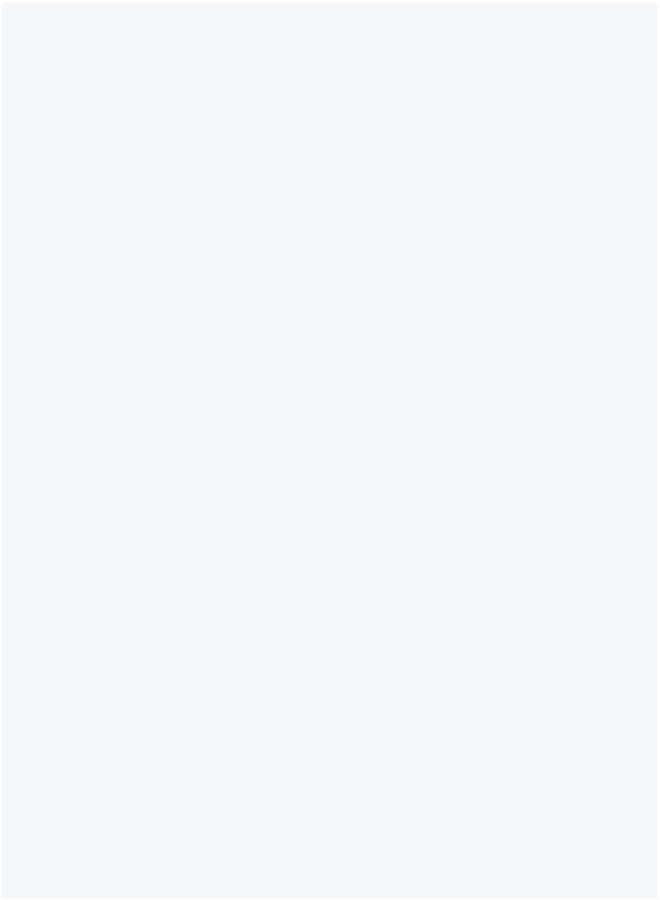
}

[Run Online](https://dartpad.dev/?id=a171dbd36ab17e6ae3c353889fd80bd1)

**Note:** Here super keyword is used to call the method of the parent class. **Example 5: Multilevel Inheritance In Dart**

In this example below, there is class named **Person** with two properties **name** and **age**. There is sub class named **Doctor** with properties **listofdegrees** and **hospitalname**. There is another subclass named **Specialist** with property **specialization**. The sub class has a method **display** to display the values of the properties.

class Person { // Properties String? name; int? age;



}

class Doctor extends Person { // Properties

List<String>? listofdegrees; String? hospitalname;

// Method to display the values of the properties void display() {

print("Name: ${name}");

print("Age: ${age}");

print("List of Degrees: ${listofdegrees}");

print("Hospital Name: ${hospitalname}");

}

}

class Specialist extends Doctor { // Properties

String? specialization;

// Method to display the values of the properties void display() {

super.display();

print("Specialization: ${specialization}");

}

}

void main() {

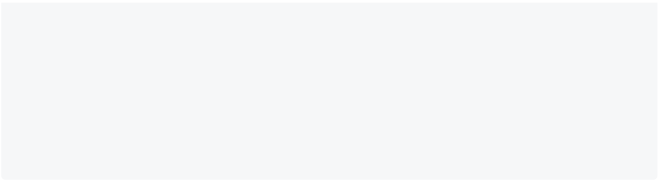
// Create an object of Specialist class Specialist s = new Specialist();

// setting values to the object

s.name = "John";

s.age = 30;

s.listofdegrees = ["MBBS", "MD"]; s.hospitalname = "ABC Hospital";



s.specialization = "Cardiologist"; // Display the values of the object s.display();

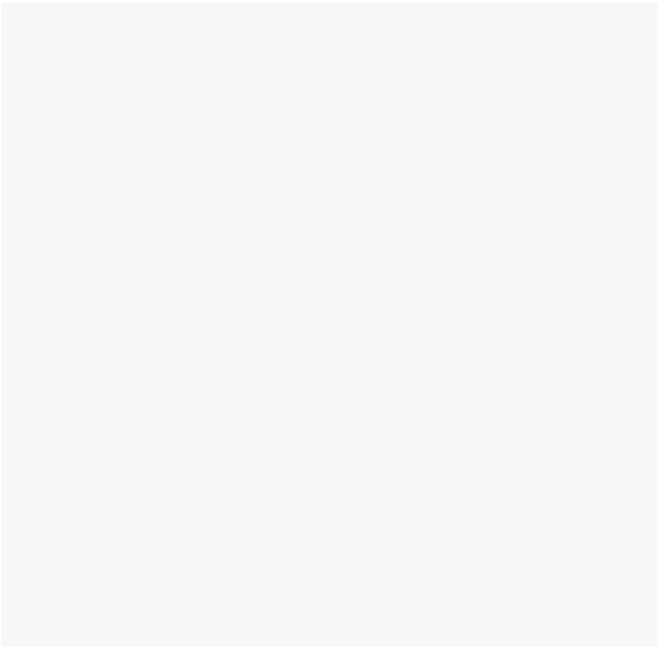
}

[Run Online](https://dartpad.dev/?id=932de854b9eba891c6a7e725c484263f)

**Example 6: Hierarchical Inheritance In Dart**

In this example below, there is class named **Shape** with two properties **diameter1** and **diameter2**. There is sub class named **Rectangle** with method **area** to calculate the area of the rectangle. There is another subclass named **Triangle** with method **area** to calculate the area of the triangle.

class Shape {



// Properties

double? diameter1; double? diameter2; }

class Rectangle extends Shape {

// Method to calculate the area of the rectangle double area() {

return diameter1! \* diameter2!;

}

}

class Triangle extends Shape {

// Method to calculate the area of the triangle double area() {

return 0.5 \* diameter1! \* diameter2!;

}

}

void main() {

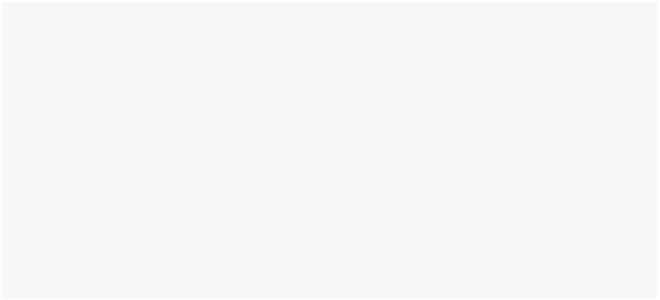
// Create an object of Rectangle class Rectangle r = new Rectangle();

// setting values to the object

r.diameter1 = 10.0;

r.diameter2 = 20.0;

// Display the area of the rectangle



print("Area of the rectangle: ${r.area()}");

// Create an object of Triangle class

Triangle t = new Triangle();

// setting values to the object

t.diameter1 = 10.0;

t.diameter2 = 20.0;

// Display the area of the triangle

print("Area of the triangle: ${t.area()}"); }

[Run Online](https://dartpad.dev/?id=1e6b9c60a85997a758bcba36b2147483) **Key Points**

* Inheritance is used to reuse the code.
  + Inheritance is a concept which is achieved by using the **extends** keyword.
    - Properties and methods of the super class can be accessed by the sub class.
      * Class **Dog** extends class **Animal**{} means Dog is sub class and Animal is super class.
        + The sub class can have its own properties and methods.

**Why Dart Does Not Support Multiple Inheritance?**

Dart does not support multiple inheritance because it can lead to ambiguity. For example, if class **Apple** inherits class **Fruit** and class **Vegetable**, then there may be two methods with the same name **eat**. If the method is called, then which method should be called? This is the reason why Dart does not support multiple inheritance.

**What’s problem Of Copy Paste Instead Of Inheritance?**

If you copy the code from one class to another class, then you will have to maintain the code in both the classes. If you make any changes in one class, then you will have to make the same changes in the other class. This can lead to errors and bugs in the code.

**Is Inheritance Finished If I Learned Extending Class?**

No, there is a lot more to learn about inheritance. You need to learn about **Constructor Inheritance**, **Method Overriding**, **Abstract Class**, **Interface** and **Mixin**

etc. You will learn about these concepts in the next chapters. Inheritance Of Constructor in Dart

**Introduction**

In this section, you will learn about inheritance of constructor in Dart programming language with the help of examples. Before learning about inheritance of constructor in Dart, you should have a basic understanding of the [constructor] and [inheritance] in Dart.

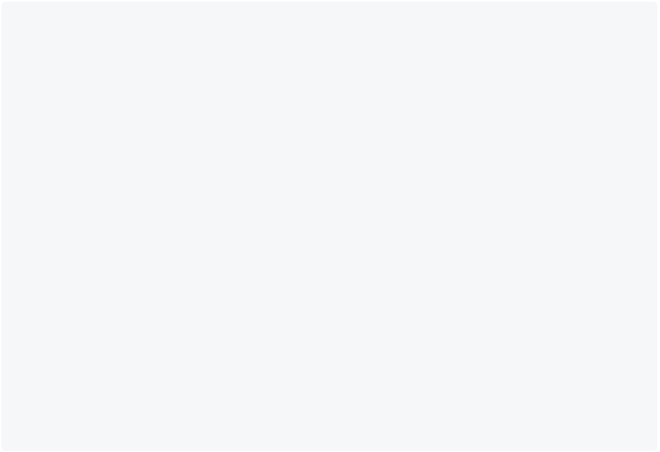
**What Is Inheritance Of Constructor In Dart?**

Inheritance of constructor in Dart is a process of inheriting the constructor of the parent class to the child class. It is a way of reusing the code of the parent class.

**Example 1: Inheritance Of Constructor In Dart**

In this example below, there is class named **Laptop** with a constructor. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor.

class Laptop {



// Constructor

Laptop() {

print("Laptop constructor"); }

}

class MacBook extends Laptop {

// Constructor

MacBook() {

print("MacBook constructor"); }

}

void main() {

var macbook = MacBook(); }

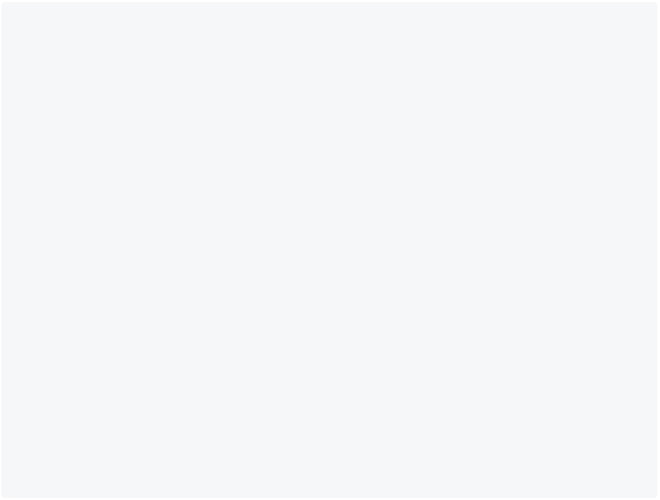
[Run Online](https://dartpad.dev/?id=14f1157aa08e7ac7e44a155062c67206)

**Note**: The constructor of the parent class is called first and then the constructor of the child class is called.

**Example 2: Inheritance Of Constructor With Parameters In Dart**

In this example below, there is class named **Laptop** with a constructor with parameters. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with parameters.

class Laptop {



// Constructor

Laptop(String name, String color) { print("Laptop constructor");

print("Name: $name");

print("Color: $color");

}

}

class MacBook extends Laptop {

// Constructor

MacBook(String name, String color) : super(name, color) { print("MacBook constructor");

}

}

void main() {

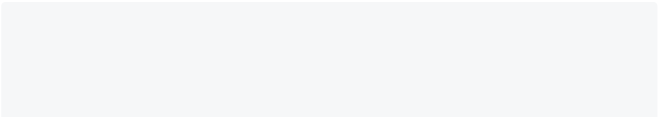
var macbook = MacBook("MacBook Pro", "Silver"); }

[Run Online](https://dartpad.dev/?id=e98f4ba85e104e763da376222f9e60b8)

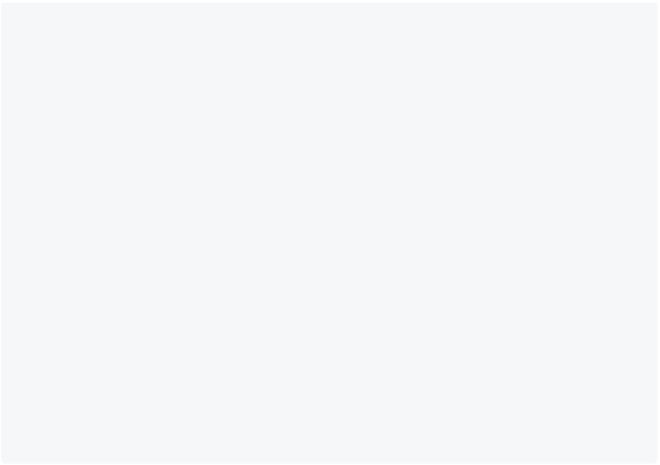
**Example 3: Inheritance Of Constructor**

In this example below, there is class named **Person** with properties **name** and **age**. There is another class named **Student** which extends the **Person** class. The **Student** class has additional property **rollNumber**. Lets see how to create a constructor for the **Student** class.

class Person { String name; int age;



// Constructor



Person(this.name, this.age); }

class Student extends Person { int rollNumber;

// Constructor

Student(String name, int age, this.rollNumber) : super(name, age); }

void main() {

var student = Student("John", 20, 1);

print("Student name: ${student.name}");

print("Student age: ${student.age}");

print("Student roll number: ${student.rollNumber}");

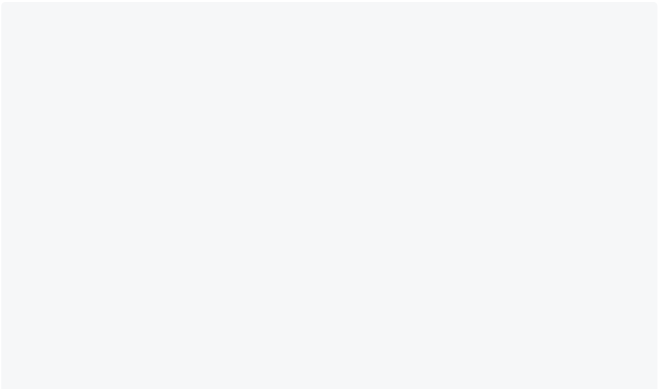
}

[Run Online](https://dartpad.dev/?id=78ec6a73cfc2802f98b447d630f3e4f9)

**Example 4: Inheritance Of Constructor With Named Parameters In Dart**

In this example below, there is class named **Laptop** with a constructor with named parameters. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with named parameters.

class Laptop {



// Constructor

Laptop({String name, String color}) { print("Laptop constructor");

print("Name: $name");

print("Color: $color");

}

}

class MacBook extends Laptop {

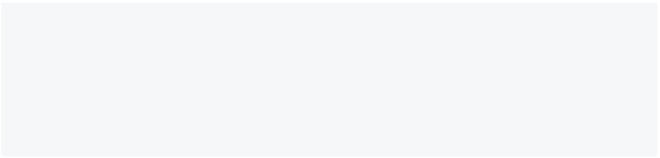
// Constructor

MacBook({String name, String color}) : super(name: name, color: color) {

print("MacBook constructor");

}

}



void main() {

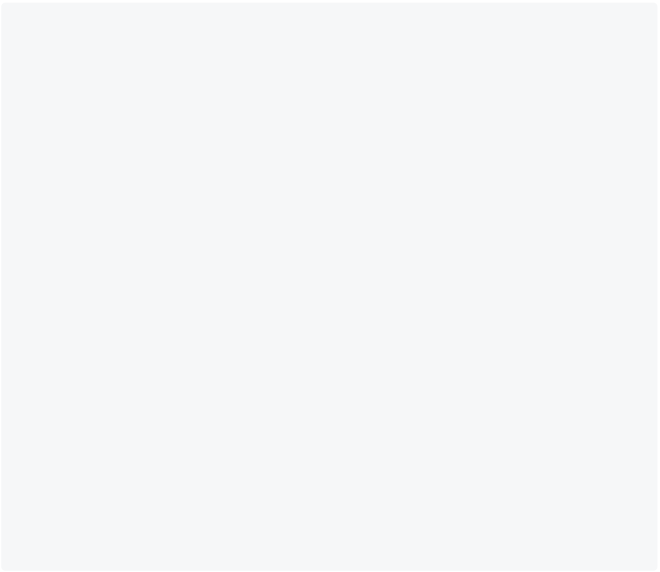
var macbook = MacBook(name: "MacBook Pro", color: "Silver"); }

[Run Online](https://dartpad.dev/?id=248394f6a14c17f77c24f2d948fd89bf)

**Example 5: Calling Named Constructor Of Parent Class In Dart**

In this example below, there is class named **Laptop** with one default constructor and one named constructor. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with named parameters. You can call the named constructor of the parent class using the **super** keyword.

class Laptop {



// Default Constructor

Laptop() {

print("Laptop constructor"); }

// Named Constructor

Laptop.named() {

print("Laptop named constructor"); }

}

class MacBook extends Laptop {

// Constructor

MacBook() : super.named() {

print("MacBook constructor"); }

}

void main() {

var macbook = MacBook(); }

[Run Online](https://dartpad.dev/?id=9c9d85ce409c3b128ba1c9c34ace7672)

### Super in Dart

In this section, you will learn about Super in Dart programming language with the help of examples. Before learning about Super in Dart, you should have a basic understanding of the [constructor](https://dart-tutorial.com/object-oriented-programming/constructor-in-dart/) and [inheritance](https://dart-tutorial.com/object-oriented-programming/inheritance-in-dart/) in Dart.

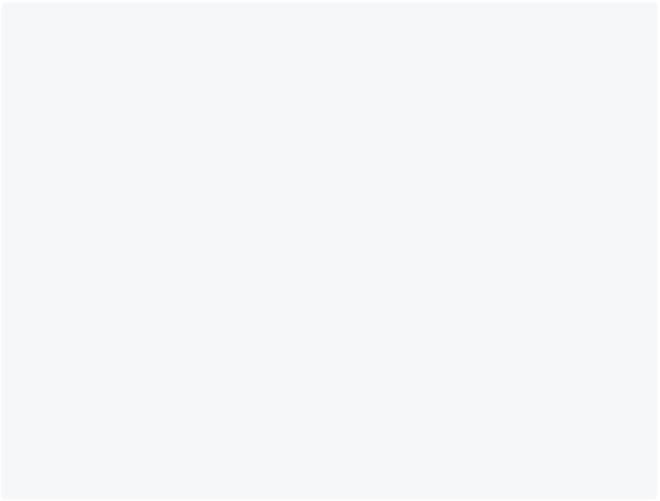
**What Is Super In Dart?**

Super is used to refer to the parent class. It is used to call the parent class’s properties and methods.

**Example 1: Super In Dart**

In this example below, the **show()** method of the **MacBook** class calls the **show()** method of the parent class using the **super** keyword.

class Laptop {



// Method

void show() {

print("Laptop show method"); }

}

class MacBook extends Laptop {

void show() {

super.show(); // Calling the show method of the parent class print("MacBook show method");

}

}

void main() {

// Creating an object of the MacBook class MacBook macbook = MacBook();

macbook.show();

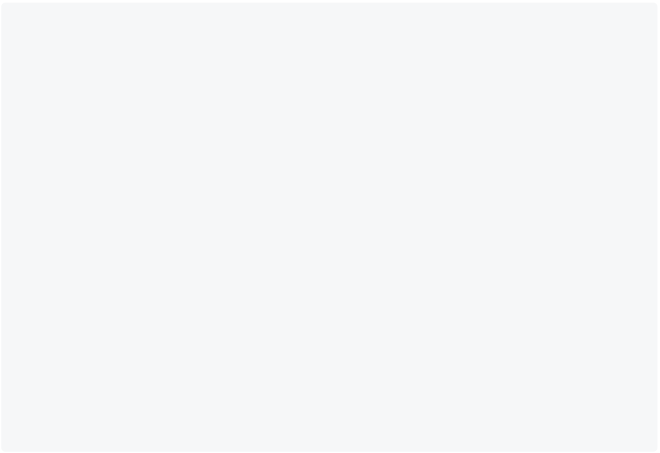
}

[Run Online](https://dartpad.dev/?id=742d67a7b0be20144dabd8d93c4d2e63)

**Example 2: Accessing Super Properties In Dart**

In this example below, the **display()** method of the **Tesla** class calls the **noOfSeats** property of the parent class using the **super** keyword.

class Car {



int noOfSeats = 4; }

class Tesla extends Car { int noOfSeats = 6;

void display() {

print("No of seats in Tesla: $noOfSeats");

print("No of seats in Car: ${super.noOfSeats}"); }

}

void main() {

var tesla = Tesla(); tesla.display();

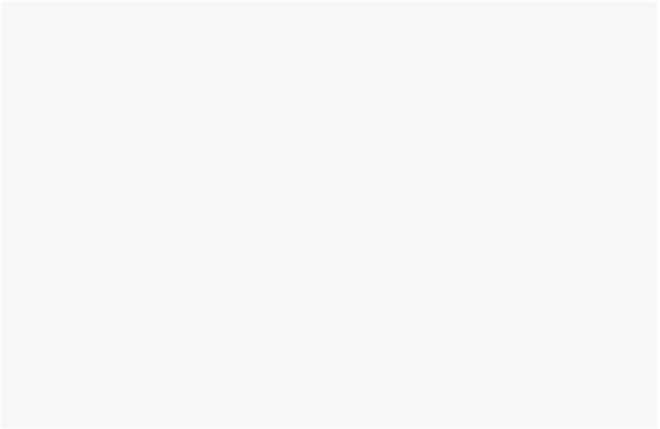
}

[Run Online](https://dartpad.dev/?id=0d3d5f67c3e4153204ce327ac5168d25)

**Example 3: Super With Constructor In Dart**

In this example below, the **Manager** class constructor calls the **Employee** class constructor using the **super** keyword.

class Employee {



// Constructor

Employee(String name, double salary) { print("Employee constructor");

print("Name: $name");

print("Salary: $salary");

}

}

class Manager extends Employee {

// Constructor

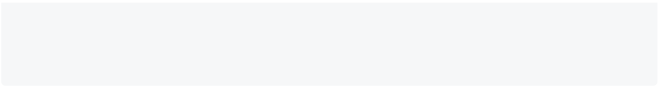
Manager(String name, double salary) : super(name, salary) { print("Manager constructor");

}

}

void main() {

Manager manager = Manager("John", 25000.0); }

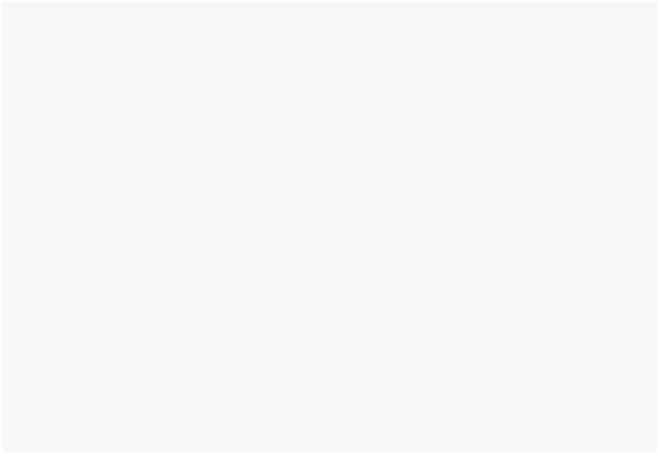


[Run Online](https://dartpad.dev/?id=745bcf7b26059bdac42918f6d15ac063)

**Example 4: Super With Named Constructor In Dart**

In this example below, the **Manager** class named constructor calls the **Employee** class named constructor using the **super** keyword.

class Employee {



// Named constructor

Employee.manager() {

print("Employee named constructor"); }

}

class Manager extends Employee {

// Named constructor

Manager.manager() : super.manager() { print("Manager named constructor"); }

}

void main() {

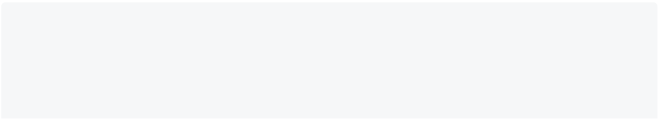
Manager manager = Manager.manager(); }

[Run Online](https://dartpad.dev/?id=1c29f2aeccfe3ee27665b48db34965e9)

**Example 5: Super With Multilevel Inheritance In Dart**

In this example below, the **MacBookPro** class method **display** calls the **display** method of the parent class **MacBook** using the **super** keyword. The **MacBook** class method **display** calls the **display** method of the parent class **Laptop** using the **super** keyword.

class Laptop {

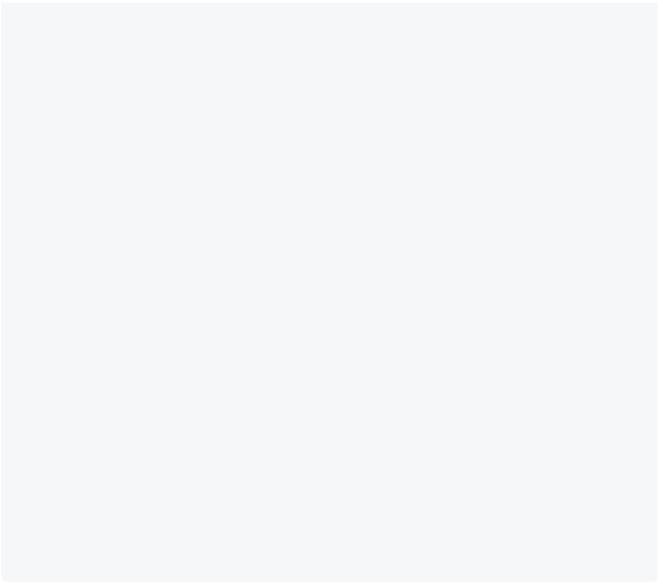


// Method

void display() {

print("Laptop display");

} }



class MacBook extends Laptop { // Method

void display() {

print("MacBook display"); super.display();

}

}

class MacBookPro extends MacBook { // Method

void display() {

print("MacBookPro display"); super.display();

}

}

void main() {

var macbookpro = MacBookPro(); macbookpro.display();

}

[Run Online](https://dartpad.dev/?id=fb99e56566b600eaaea0cf9419b3dd3a)

**Key Points To Remember**

* The **super** keyword is used to access the parent class members.
  + The **super** keyword is used to call the method of the parent class.

### Polymorphism in Dart

**Introduction**

In this section, you will learn about polymorphism in Dart programming language with the help of examples. Before learning about polymorphism in Dart, you should have a basic understanding of the [inheritance](https://dart-tutorial.com/object-oriented-programming/inheritance-in-dart/) in Dart.

**Polymorphism In Dart**

Poly means **many** and morph means **forms**. Polymorphism is the ability of an object to take on many forms. As humans, we have the ability to take on many forms. We can be a student, a teacher, a parent, a friend, and so on. Similarly, in object-oriented programming, polymorphism is the ability of an object to take on many forms.

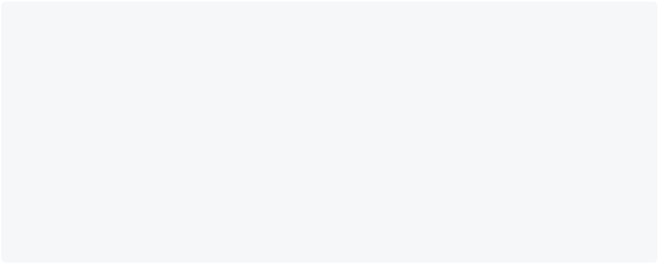
**Note**: In the real world, polymorphism is updating or modifying the feature, function, or implementation that already exists in the parent class.

**Polymorphism By Method Overriding**

Method overriding is a technique in which you can create a method in the child class that has the same name as the method in the parent class. The method in the child class overrides the method in the parent class.

**Syntax**

class ParentClass{



void functionName(){

}

}

class ChildClass extends ParentClass{ @override

void functionName(){

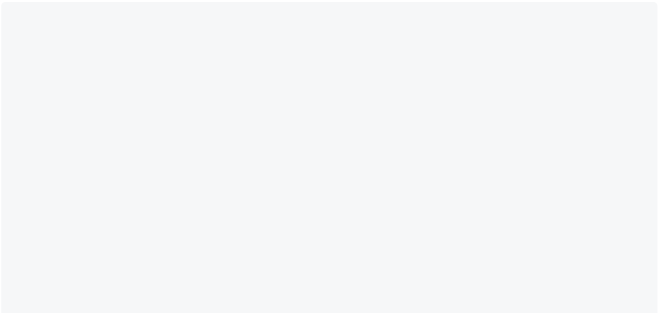
}

}

**Example 1: Polymorphism By Method Overriding In Dart**

In this example below, there is a class named **Animal** with a method named **eat()**. The **eat()** method is overridden in the child class named **Dog**.

class Animal {



void eat() {

print("Animal is eating"); }

}

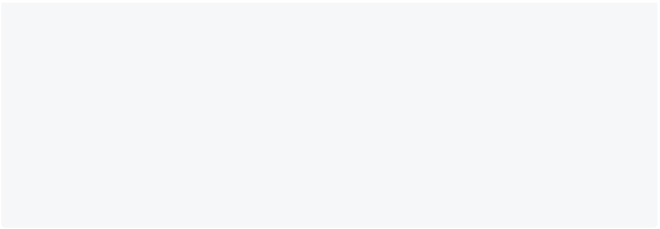
class Dog extends Animal { @override

void eat() {

print("Dog is eating"); }

}

void main() {



Animal animal = Animal(); animal.eat();

Dog dog = Dog(); dog.eat();

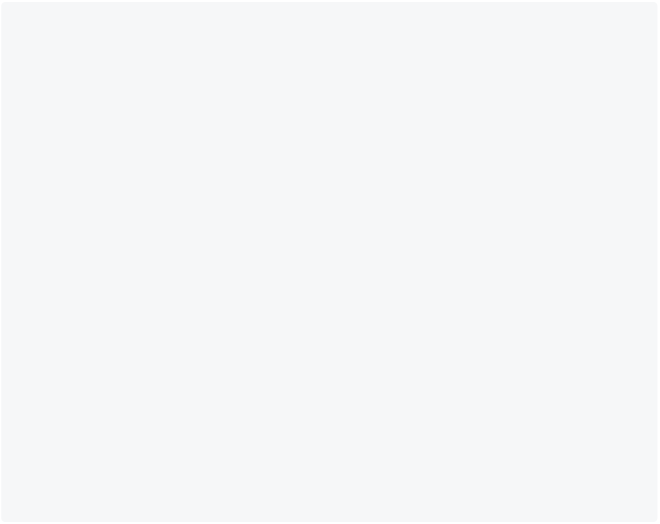
}

[Run Online](https://dartpad.dev/?id=917d2851674d9e0009d70194e3612c74)

**Example 2: Polymorphism By Method Overriding In Dart**

In this example below, there is a class named **Vehicle** with a method named **run()**. The **run()** method is overridden in the child class named **Bus**.

class Vehicle {



void run() {

print("Vehicle is running"); }

}

class Bus extends Vehicle { @override

void run() {

print("Bus is running"); }

}

void main() {

Vehicle vehicle = Vehicle(); vehicle.run();

Bus bus = Bus(); bus.run();

}

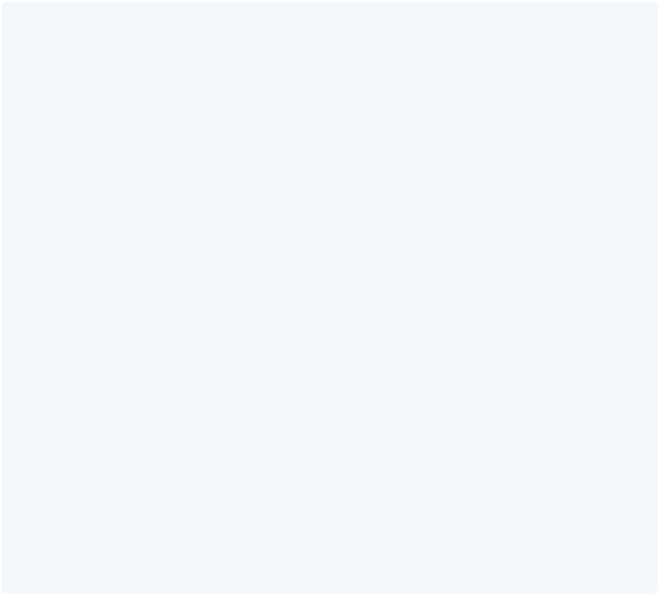
[Run Online](https://dartpad.dev/?id=23815134f26eb17adfcb192e36f4688e)

**Note**: If you don’t write **@override**, the program still runs. But, it is a good practice to write **@override**.

**Example 3: Polymorphism By Method Overriding In Dart**

In this example below, there is a class named **Car** with a method named **power()**. The **power()** method is overridden in two child classes named **Honda** and **Tesla**.

class Car{



void power(){

print("It runs on petrol."); }

}

class Honda extends Car{

}

class Tesla extends Car{

@override

void power(){

print("It runs on electricity."); }

}

void main(){

Honda honda=Honda(); Tesla tesla=Tesla();

honda.power(); tesla.power(); }

[Run Online](https://dartpad.dev/?id=4483f45207fdc7625789afb3a15568eb)

**Example 4: Polymorphism By Method Overriding In Dart**

In this example below, there is a class named **Employee** with a method named **salary()**. The **salary()** method is overridden in two child classes named **Manager** and **Developer**.

class Employee{

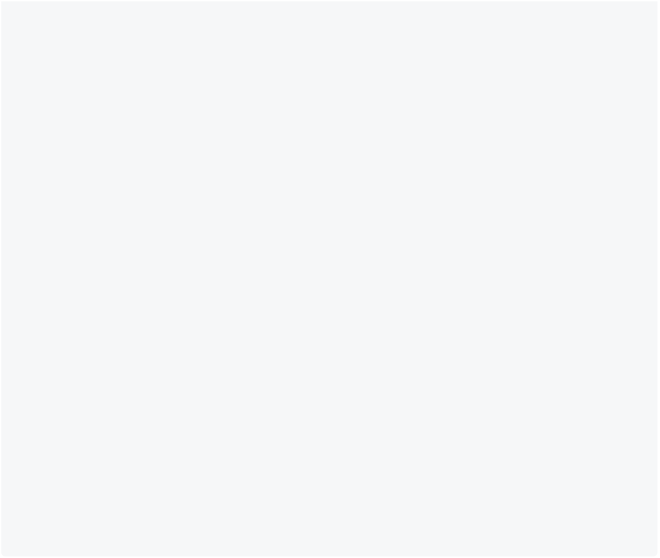


void salary(){

print("Employee salary is \$1000."); }

}

class Manager extends Employee{



@override

void salary(){

print("Manager salary is \$2000."); }

}

class Developer extends Employee{

@override

void salary(){

print("Developer salary is \$3000."); }

}

void main(){

Manager manager=Manager();

Developer developer=Developer();

manager.salary(); developer.salary(); }

[Run Online](https://dartpad.dev/?id=b14fa1232ccb4cbc2b2e5a766e263860)

**Advantage Of Polymorphism In Dart**

* Subclasses can override the behavior of the parent class.
  + It allows us to write code that is more flexible and reusable.

### Static in Dart

In this section, you will learn about **dart static** to share the same variable or method across all instances of a class.

**Static In Dart**

If you want to define a variable or method that is shared by all instances of a class, you can use the **static** keyword. Static members are accessed using the class name. It is used for **memory management**.

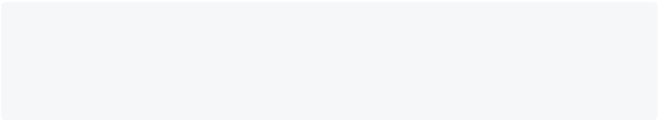
**Dart Static Variable**

A static variable is a variable that is shared by all instances of a class. It is declared using the static keyword. It is initialized only once when the class is loaded. It is used to store the **class-level data**.

**How To Declare A Static Variable In Dart**

To declare a static variable in Dart, you must use the static keyword before the variable name.

class ClassName {



static dataType variableName; }

**How To Initialize A Static Variable In Dart**

To initialize a static variable simply assign a value to it.

class ClassName {



static dataType variableName = value; // for e.g

// static int num = 10;

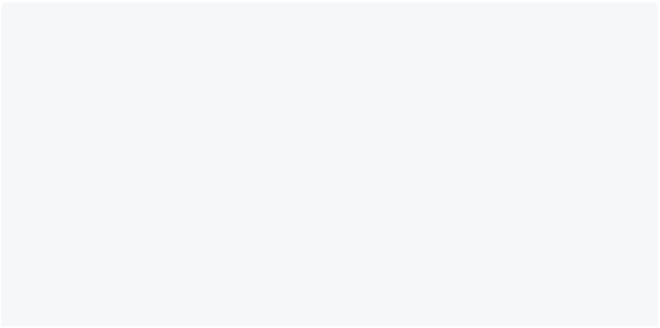
// static String name = "Dart";

}

**How To Access A Static Variable In Dart**

You need to use the **ClassName.variableName** to access a static variable in Dart.

class ClassName {



static dataType variableName = value;

// Accessing the static variable inside same class void display() {

print(variableName);

}

}

void main() {

// Accessing static variable outside the class dataType value =ClassName.variableName;

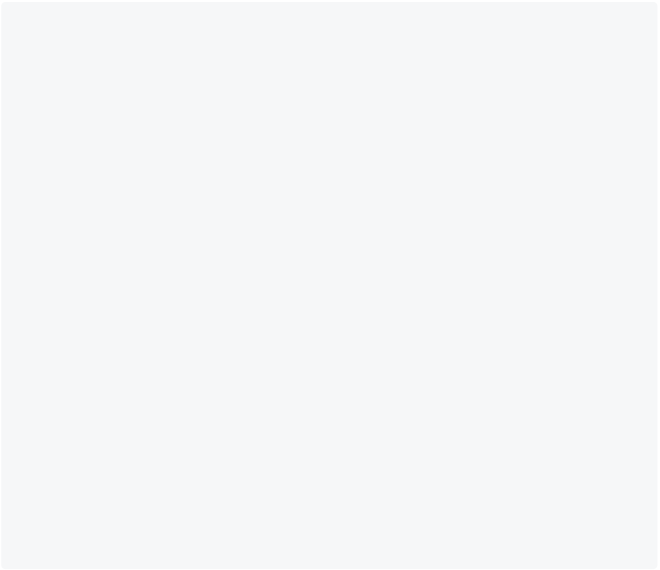
}



**Example 1: Static Variable In Dart**

In this example below, there is a class named **Employee**. The class has a static variable **count** to count the number of employees.

class Employee {



// Static variable

static int count = 0;

// Constructor

Employee() {

count++;

}

// Method to display the value of count void totalEmployee() {

print("Total Employee: $count");

}

}

void main() {

// Creating objects of Employee class Employee e1 = new Employee();

e1.totalEmployee();

Employee e2 = new Employee();

e2.totalEmployee();

Employee e3 = new Employee();

e3.totalEmployee();

}

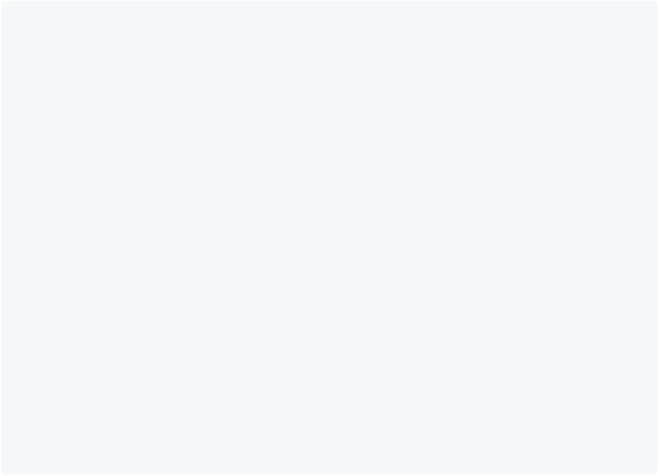
[Run Online](https://dartpad.dev/?id=3e805fdf420430f0508d90dace1b322d)

**Note:** While creating the objects of the class, the static variable **count** is incremented by 1. The **totalEmployee()** method displays the value of the static variable **count**.

**Example 2: Static Variable In Dart**

In this example below, there is a class named **Student**. The class has a static variable **schoolName** to store the name of the school. If every student belongs to the same school, then it is better to use a static variable.

class Student {



int id;

String name;

static String schoolName = "ABC School";

Student(this.id, this.name);

void display() {

print("Id: ${this.id}");

print("Name: ${this.name}");

print("School Name: ${Student.schoolName}");

} }

void main() {

Student s1 = new Student(1, "John"); s1.display();

Student s2 = new Student(2, "Smith"); s2.display();

}

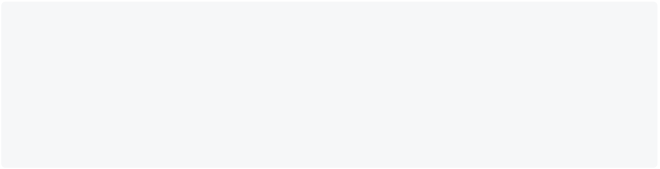
[Run Online](https://dartpad.dev/?id=9f0bc3c76d819faacad8c65cbeb19c51)

**Dart Static Method**

A static method is shared by all instances of a class. It is declared using the static keyword. You can access a static method without creating an object of the class.

**Syntax**

class ClassName{



static returnType methodName(){ //statements

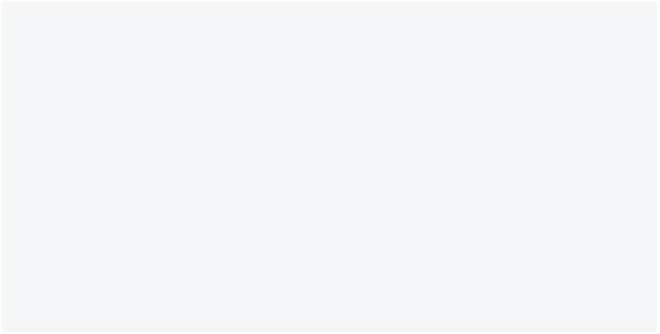
}

}

**Example 3: Static Method In Dart**

In this example, we will create a static method **calculateInterest()** which calculates the simple interest. You can call **SimpleInterest.calculateInterest()** anytime without creating an instance of the class.

class SimpleInterest {



static double calculateInterest(double principal, double rate, double time) {

return (principal \* rate \* time) / 100;

}

}

void main() {

print(

"The simple interest is ${SimpleInterest.calculateInterest(1000, 2, 2)}");

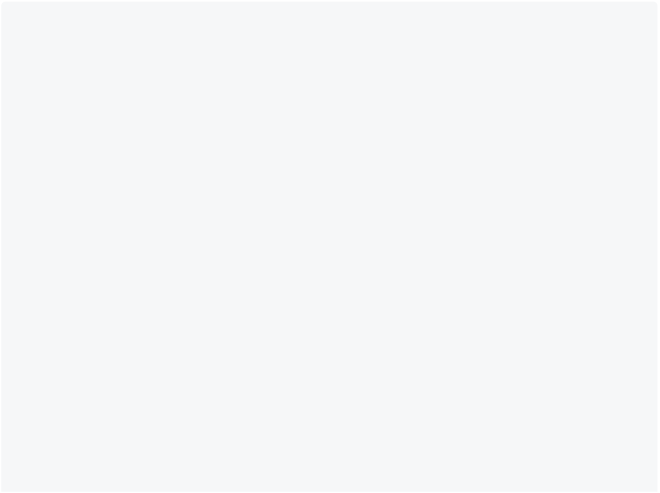
}

[Run Online](https://dartpad.dev/?id=9a6efb56670612a9afd516f5bdce6303)

**Example 4: Static Method In Dart**

In this example below, there is static method **generateRandomPassword()** which generates a random password. You can call **PasswordGenerator.generateRandomPassword()** anytime without creating an instance of the class.

import 'dart:math';



class PasswordGenerator {

static String generateRandomPassword() {

List<String> allalphabets = 'abcdefghijklmnopqrstuvwxyz'.split(''); List<int> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

List<String> specialCharacters = ["@", "#", "%", "&", "\*"];

List<String> password = [];

for (int i = 0; i < 5; i++) {

password.add(allalphabets[Random().nextInt(allalphabets.length)]);

password.add(numbers[Random().nextInt(numbers.length)].toString()); password

.add(specialCharacters[Random().nextInt(specialCharacters.length)]); }

return password.join();

}

}

void main() {



print(PasswordGenerator.generateRandomPassword()); }

[Run Online](https://dartpad.dev/?id=c399202516155336cacbd27f6bd55470)

**Note**: You don’t need to create an instance of a class to call a static method. **Key Points To Remember**

* Static members are accessed using the class name.
  + All instances of a class share static members.

### Enum in Dart

An enum is a special type that represents a fixed number of constant values. An enum is declared using the keyword **enum** followed by the enum’s name.

**Syntax**

enum enumName { constantName1, constantName2, constantName3, ...

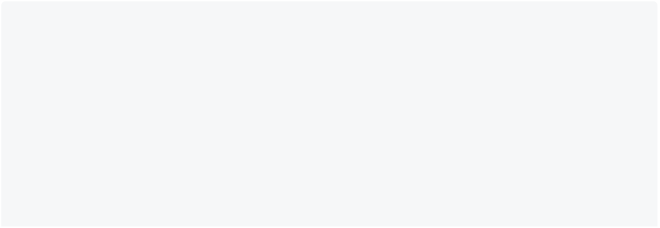


constantNameN }

**Example 1: Enum In Dart**

In this example below, there is enum type named **days**. It contains seven constants days. The **days** enum type is used in the **main()** function.

enum days { Sunday,

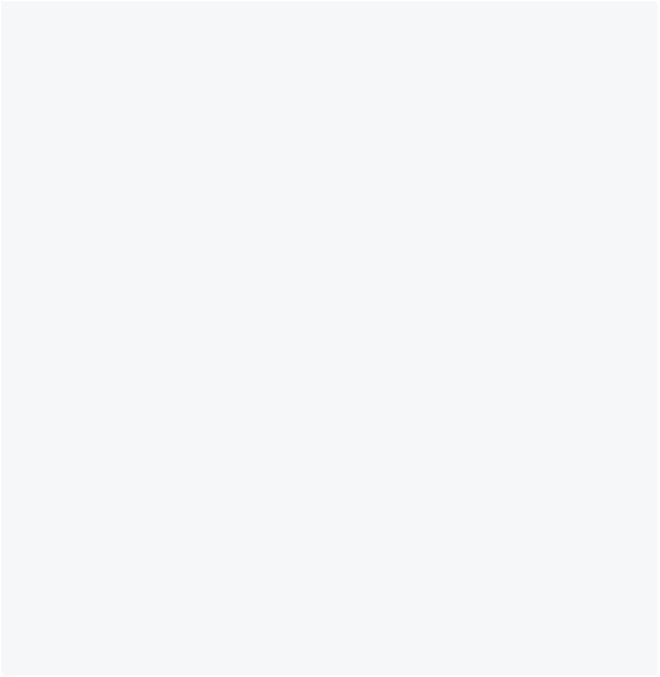


Monday,

Tuesday, Wednesday, Thrusday, Friday,

Saturday }

void main() {



var today = days.Friday;

switch (today) {

case days.Sunday:

print("Today is Sunday."); break;

case days.Monday:

print("Today is Monday.");

break;

case days.Tuesday:

print("Today is Tuesday."); break;

case days.Wednesday:

print("Today is Wednesday."); break;

case days.Thursday:

print("Today is Thursday."); break;

case days.Friday:

print("Today is Friday.");

break;

case days.Saturday:

print("Today is Saturday."); break;

}

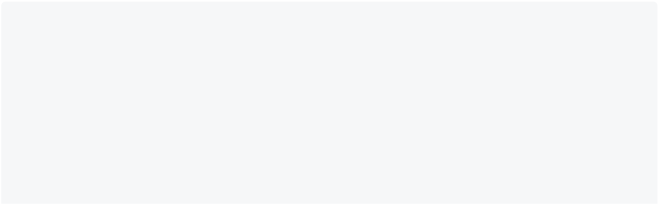
}

[Run Online](https://dartpad.dev/?id=b4108e8df504dfe276e17529cd3ab5b4)

**Example 2: Enum In Dart**

In this example, there is an enum type named **Gender**. It contains three constants **Male**, **Female**, and **Other**. The **Gender** enum type is used in the **Person** class.

enum Gender { Male, Female, Other }

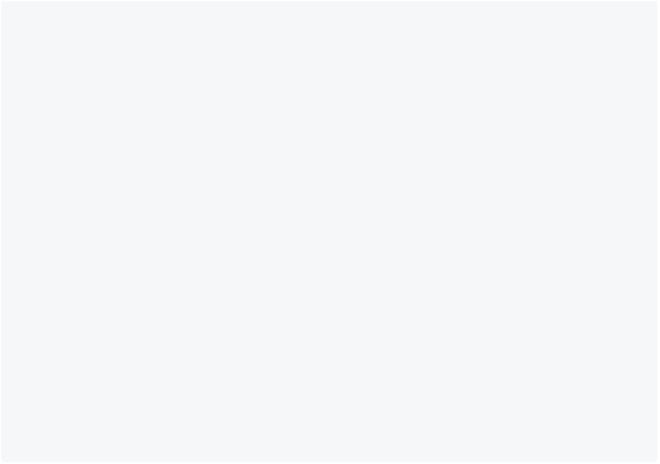


class Person {

// Properties

String? firstName; String? lastName; Gender? gender;

// Constructor



Person(this.firstName, this.lastName, this.gender);

// display() method

void display() {

print("First Name: $firstName"); print("Last Name: $lastName"); print("Gender: $gender");

}

}

void main() {

Person p1 = Person("John", "Doe", Gender.Male); p1.display();

Person p2 = Person("Menuka", "Sharma", Gender.Female); p2.display();

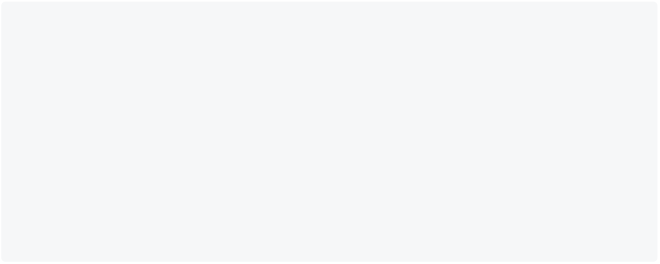
}

[Run Online](https://dartpad.dev/?id=c08137bd12f9af46e1cafd5c37a055c1)

**How to Print All Enum Values**

In this example, there is enum type named **Days**. It contain 7 days. The for loop iterates through all the enum values.

enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }



void main() {

// Days.values: It returns all the values of the enum. for (Days day in Days.values) {

print(day);

}

}

[Run Online](https://dartpad.dev/?id=a5cf142ccbfb103dd38f5fb9b315be8c)

**Advantages Of Enum In Dart**

It is used to define a set of named constants.



* Makes your code more readable and maintainable.
  + It makes the code more reusable and makes it easier for developers.

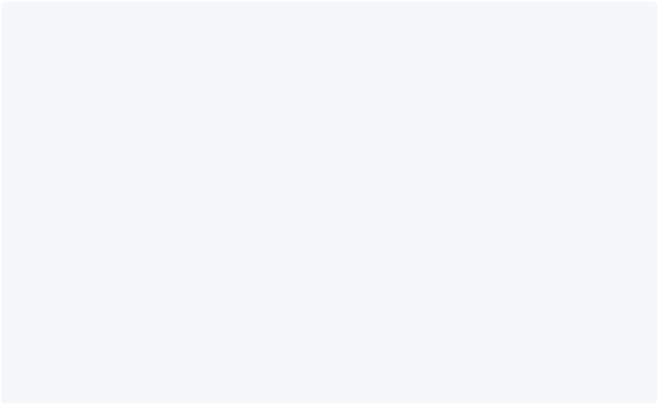
**Characteristics Of Enum**

* It must contain at least one constant value.
  + Enums are declared outside the class.
    - Used to store a large number of constant values.

**Enhanced Enum In Dart**

In dart, you can declare enums with members. For example, for your accounting software you can store company types like **Sole Proprietorship**, **Partnership**, **Corporation**, and **Limited Liability Company**. You can declare an enum with members as shown below.

enum CompanyType {



soleProprietorship("Sole Proprietorship"),

partnership("Partnership"),

corporation("Corporation"),

limitedLiabilityCompany("Limited Liability Company");

// Members

final String text;

const CompanyType(this.text); }

void main() {

CompanyType soleProprietorship = CompanyType.soleProprietorship; print(soleProprietorship.text);

}

[Run Online](https://dartpad.dev/?id=0a0d58c39324391b6d60dfbebf7c3e92)

### Abstract Class

**Introduction**

Previously you learned how to define a class. These classes are **concrete classes**. You can create an object of concrete classes, but you cannot create an object of abstract classes.

**Abstract Class**

Abstract classes are classes that cannot be initialized. It is used to define the behavior of a class that can be inherited by other classes. An abstract class is declared using the keyword **abstract**.

**Syntax**

abstract class ClassName { //Body of abstract class



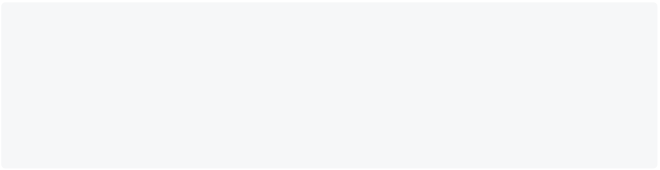
method1(); method2(); }

**Abstract Method**

An abstract method is a method that is declared without an implementation. It is declared with a semicolon (;) instead of a method body.

**Syntax**

abstract class ClassName { //Body of abstract class method1();



method2();

}

**Why We Need Abstract Class**

Subclasses of an abstract class must implement all the abstract methods of the abstract class. It is used to achieve abstraction in the Dart programming language.

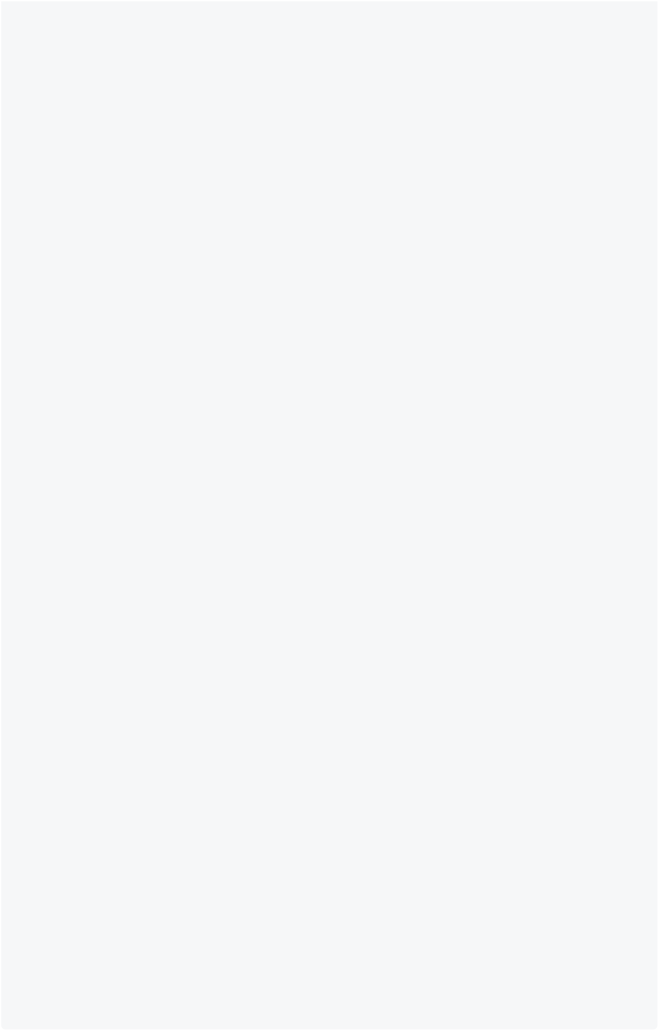
**Example 1: Abstract Class In Dart**

In this example below, there is an abstract class **Vehicle** with two abstract methods **start()** and **stop()**. The subclasses **Car** and **Bike** implement the abstract methods and override them to print the message.

abstract class Vehicle { // Abstract method



void start();



// Abstract method void stop();

}

class Car extends Vehicle {

// Implementation of start() @override

void start() {

print('Car started');

}

// Implementation of stop() @override

void stop() {

print('Car stopped');

}

}

class Bike extends Vehicle { // Implementation of start() @override

void start() {

print('Bike started');

}

// Implementation of stop() @override

void stop() {

print('Bike stopped');

}

}

void main() {

Car car = Car(); car.start();

car.stop();

Bike bike = Bike(); bike.start();

bike.stop();

}

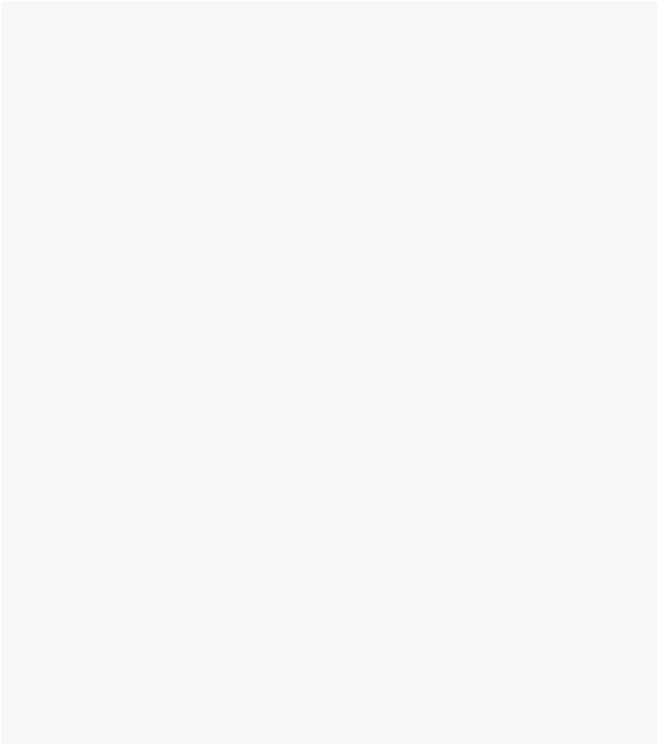
[Run Online](https://dartpad.dev/?id=625b7c5437804f2b5f9a063c2c14cc15)

**Note**: The abstract class is used to define the behavior of a class that can be inherited by other classes. You can define an abstract method inside an abstract class.

**Example 2: Abstract Class In Dart**

In this example below, there is an abstract class **Shape** with one abstract method **area()** and two subclasses **Rectangle** and **Triangle**. The subclasses implement the **area()** method and override it to calculate the area of the rectangle and triangle, respectively.

abstract class Shape {



int dim1, dim2;

// Constructor

Shape(this.dim1, this.dim2); // Abstract method

void area();

}

class Rectangle extends Shape {

// Constructor

Rectangle(int dim1, int dim2) : super(dim1, dim2);

// Implementation of area()

@override

void area() {

print('The area of the rectangle is ${dim1 \* dim2}'); }

}

class Triangle extends Shape {

// Constructor

Triangle(int dim1, int dim2) : super(dim1, dim2);

// Implementation of area()

@override

void area() {

print('The area of the triangle is ${0.5 \* dim1 \* dim2}'); }

}

void main() {



Rectangle rectangle = Rectangle(10, 20); rectangle.area();

Triangle triangle = Triangle(10, 20); triangle.area();

}

[Run Online](https://dartpad.dev/?id=0d8bd998f91eba0c492d6f1d824220f9)

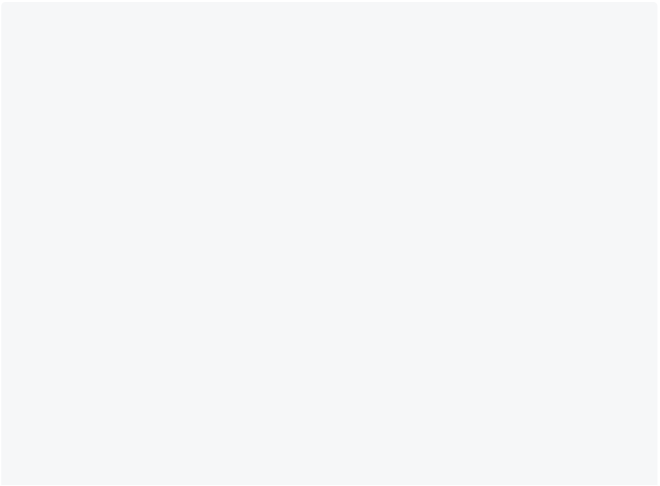
**Constructor In Abstract Class**

You can’t create an object of an abstract class. However, you can define a constructor in an abstract class. The constructor of an abstract class is called when an object of a subclass is created.

**Example 3: Constructor In Abstract Class**

In this example below, there is an abstract class **Bank** with a constructor which takes two parameters **name** and **rate**. There is an abstract method **interest()**. The subclasses **SBI** and **ICICI** implement the abstract method and override it to print the interest rate.

abstract class Bank { String name;



double rate;

// Constructor

Bank(this.name, this.rate);

// Abstract method void interest();

//Non-Abstract method: It have an implementation void display() {

print('Bank Name: $name');

}

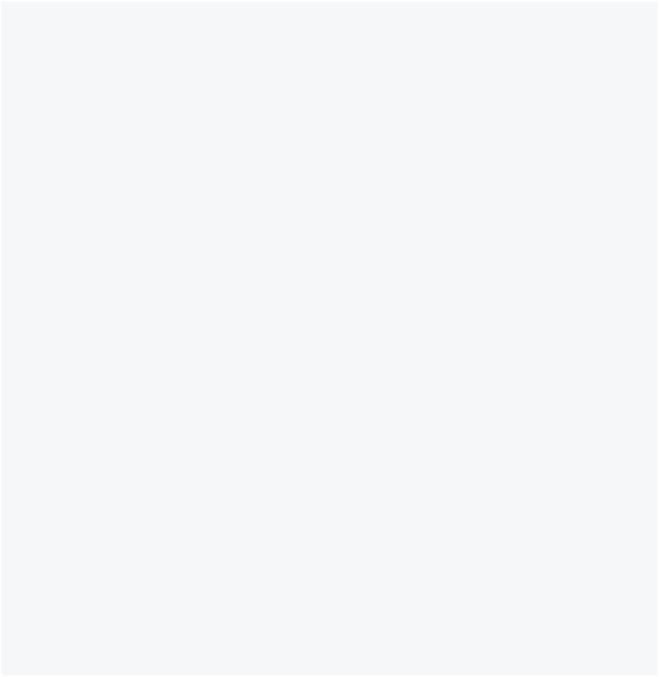
}

class SBI extends Bank {

// Constructor

SBI(String name, double rate) : super(name, rate);

// Implementation of interest()



@override

void interest() {

print('The rate of interest of SBI is $rate'); }

}

class ICICI extends Bank {

// Constructor

ICICI(String name, double rate) : super(name, rate);

// Implementation of interest()

@override

void interest() {

print('The rate of interest of ICICI is $rate'); }

}

void main() {

SBI sbi = SBI('SBI', 8.4);

ICICI icici = ICICI('ICICI', 7.3);

sbi.interest(); icici.interest(); icici.display(); }

[Run Online](https://dartpad.dev/?id=8f87bdababbea5cf84516a0a0e259a52)

**Key Points To Remember**

* You can’t create an object of an abstract class.
  + It can have both abstract and non-abstract methods.
    - It is used to define the behavior of a class that other classes can inherit.
      * Abstract method only has a signature and no implementation.

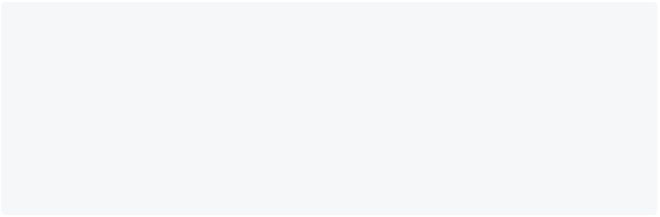
### Interface in Dart

In Dart, every class is **implicit interface**. **Interface In Dart**

**An interface defines a syntax that a class must follow**. It is a contract that defines the capabilities of a class. It is used to achieve abstraction in the Dart programming language. When you implement an interface, you must implement all the properties and methods defined in the interface. Keyword **implements** is used to implement an interface.

**Syntax Of Interface In Dart**

class InterfaceName { // code



}

class ClassName implements InterfaceName { // code

}

**Declaring Interface In Dart**

In dart there is no keyword **interface** but you can use **class** or **abstract class** to declare an interface. All classes implicitly define an interface. Mostly **abstract class** is used to declare an interface.

// creating an interface using abstract class abstract class Person {



canWalk();

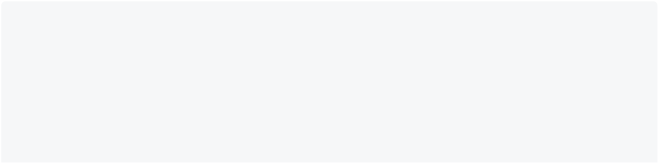
canRun();

}

**Implementing Interface In Dart**

You must use the **implements** keyword to implement an interface. The class that implements an interface must implement all the methods and properties of the interface.

class Student implements Person { // implementation of canWalk() @override

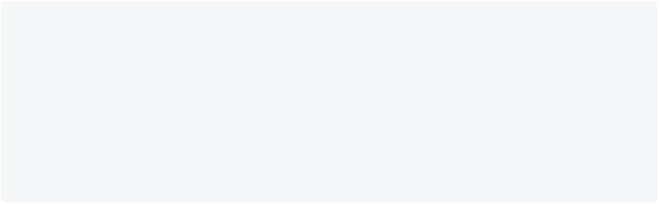


canWalk() {

print('Student can walk');

}

// implementation of canRun() @override



canRun() {

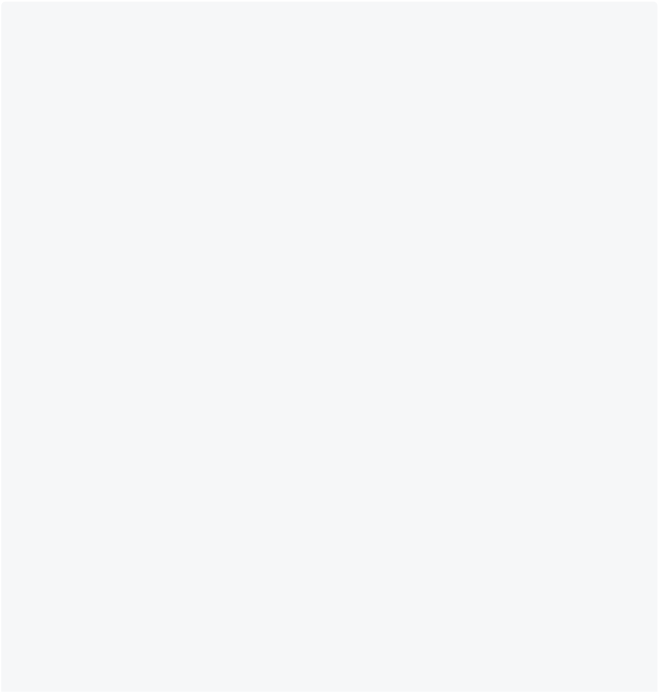
print('Student can run'); }

}

**Example 1: Interface In Dart**

In this example below, there is an interface **Laptop** with two methods **turnOn()** and **turnOff()**. The class **MacBook** implements the interface and overrides the methods to print the message.

// creating an interface using concrete class class Laptop {



// method

turnOn() {

print('Laptop turned on');

}

// method

turnOff() {

print('Laptop turned off');

}

}

class MacBook implements Laptop { // implementation of turnOn() @override

turnOn() {

print('MacBook turned on'); }

// implementation of turnOff() @override

turnOff() {

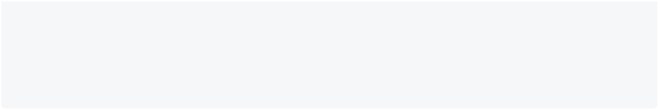
print('MacBook turned off'); }

}

void main() {

var macBook = MacBook();

macBook.turnOn(); macBook.turnOff(); }



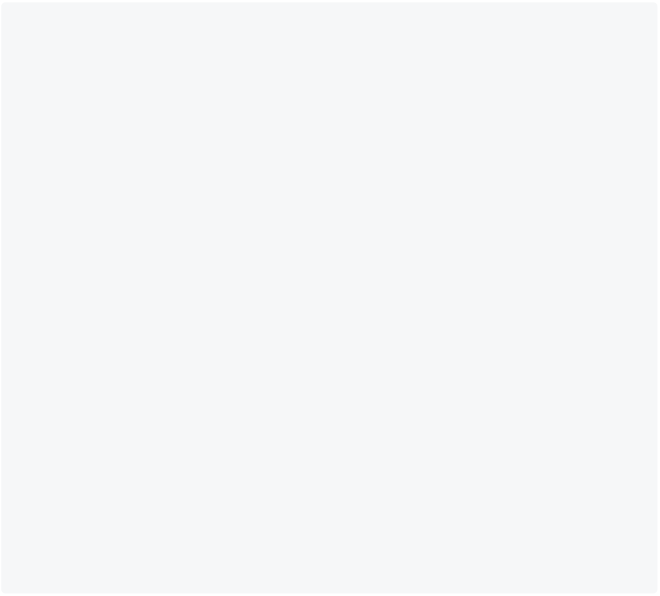
[Run Online](https://dartpad.dev/?id=4a00cb7c6147171e03bd97fa539a1f7e)

**Note:** Most of the time, **abstract class** is used instead of **concrete class** to declare an interface.

**Example 2: Interface In Dart**

In this example below, there is an abstract class named **Vehicle**. The **Vehicle** class has two abstract methods **start()** and **stop()**. The **Car** class implements the **Vehicle** interface. The **Car** class has to implement the **start()** and **stop()** methods.

// abstract class as interface abstract class Vehicle {



void start();

void stop();

}

// implements interface

class Car implements Vehicle { @override

void start() {

print('Car started');

}

@override

void stop() {

print('Car stopped'); }

}

void main() {

var car = Car(); car.start();

car.stop();

}

[Run Online](https://dartpad.dev/?id=2760c84c6f6abc39f40f0b2e421f420f)

**Multiple Inheritance In Dart**

**Multiple inheritance** means a class can inherit from more than one class. In dart, you can’t inherit from more than one class. But you can implement multiple interfaces in a class.

**Syntax For Implementing Multiple Interfaces In Dart**

class ClassName implements Interface1, Interface2, Interface3 { // code

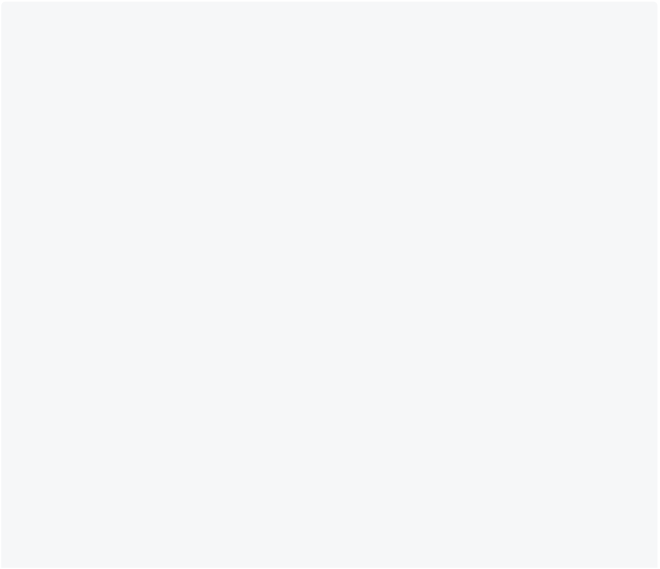


}

**Example 3: Interface In Dart With Multiple Interfaces**

In this example below, two abstract classes are named **Area** and **Perimeter**. The **Area** class has an abstract method **area()** and the **Perimeter** class has an abstract method **perimeter()**. The **Shape** class implements both the **Area** and **Perimeter** classes. The **Shape** class has to implement the **area()** and **perimeter()** methods.

// abstract class as interface



abstract class Area {

void area();

}

// abstract class as interface

abstract class Perimeter {

void perimeter();

}

// implements multiple interfaces

class Rectangle implements Area, Perimeter { // properties

int length, breadth;

// constructor

Rectangle(this.length, this.breadth);

// implementation of area()

@override

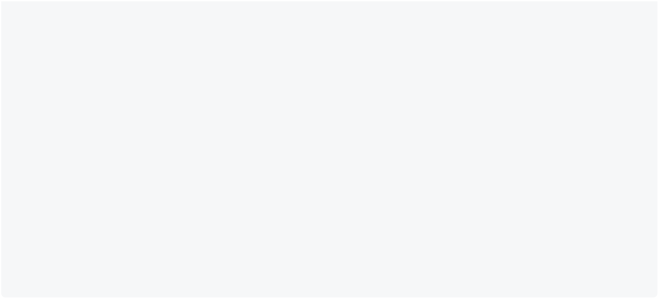
void area() {

print('The area of the rectangle is ${length \* breadth}'); }

// implementation of perimeter()

@override

void perimeter() {



print('The perimeter of the rectangle is ${2 \* (length + breadth)}');

}

}

void main() {

Rectangle rectangle = Rectangle(10, 20); rectangle.area();

rectangle.perimeter();

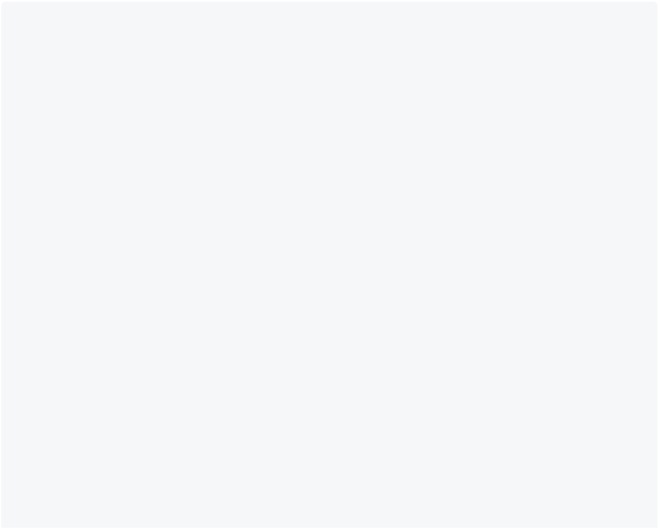
}

[Run Online](https://dartpad.dev/?id=5db8c2a04bc2f3325d6745dc26512d7d)

**Example 4: Interface In Dart**

In this example below, there is an abstract class named **Person**. The **Person** class has one property **name** and two abstract methods **run** and **walk**. The **Student** class implements the **Person** interface. The **Student** class has to implement the **run** and **walk** methods.

// abstract class as interface abstract class Person {



// properties

String? name;

// abstract method

void run();

void walk();

}

class Student implements Person { // properties

String? name;

// implementation of run()

@override

void run() {

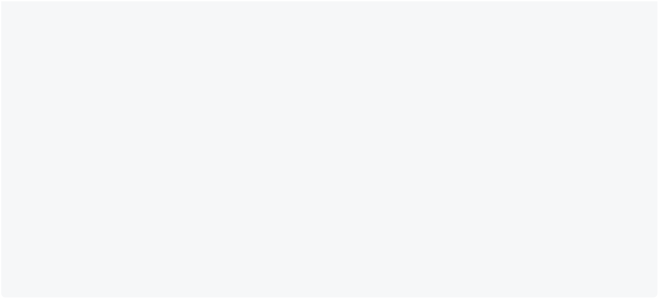
print('Student is running'); }

// implementation of walk()

@override

void walk() {

print('Student is walking'); }



}

void main() {

var student = Student(); student.name = 'John'; print(student.name);

student.run();

student.walk();

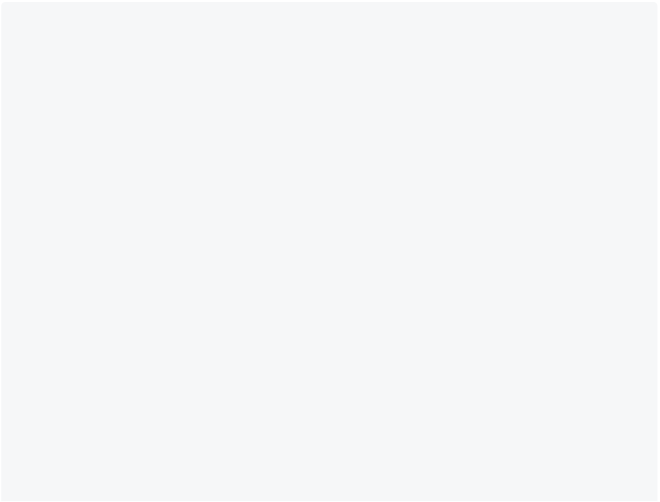
}

[Run Online](https://dartpad.dev/?id=d6a74af393338dc0823a37c2d18ddd58)

**Example 5: Interface In Dart**

In this example below, there is abstract class named **CalculateTotal** and **CalculateAverage**. The **CalculateTotal** class has an abstract method **total()** and the **CalculateAverage** class has an abstract method **average()**. The **Student** class implements both the **CalculateTotal** and **CalculateAverage** classes. The **Student** class has to implement the **total()** and **average()** methods.

// abstract class as interface



abstract class CalculateTotal {

int total();

}

// abstract class as interface

abstract class CalculateAverage {

double average();

}

// implements multiple interfaces

class Student implements CalculateTotal, CalculateAverage { // properties

int marks1, marks2, marks3;

// constructor

Student(this.marks1, this.marks2, this.marks3);

// implementation of average()

@override

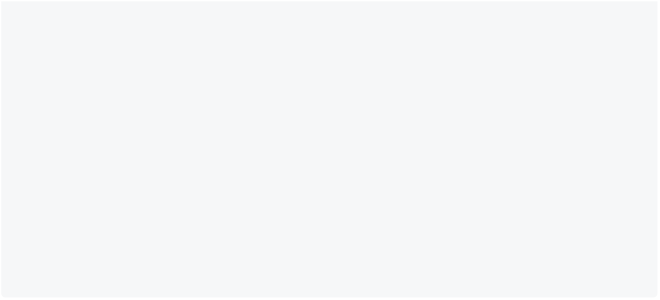
double average() {

return total() / 3;

}

// implementation of total()

@override



int total() {

return marks1 + marks2 + marks3; }

}

void main() {

Student student = Student(90, 80, 70);

print('Total marks: ${student.total()}');

print('Average marks: ${student.average()}'); }

[Run Online](https://dartpad.dev/?id=0bc646d1fa0fb8d7c21d7976cf7bbc82)

**Difference Between Extends & Implements**

* extends: Used to inherit a class in another class.

implements: Used to inherit a class as an interface in another class.

* extends: Gives complete method definition to sub-class.

implements: Gives abstract method definition to sub-class.

* + extends: Only one class can be extended.

implements: Multiple classes can be implemented.

* + - extends: It is optional to override the methods. implements: Concrete class must override the methods of an interface.



* + - * extends: Constructors of the superclass is called before the sub-class constructor. implements: Constructors of the superclass is not called before the sub- class constructor.



* + - * + extends: The super keyword is used to access the members of the superclass. implements: Interface members can’t be accessed using the super keyword.



extends: Sub-class need not to override the fields of the superclass. implements: Subclass must override the fields of the interface.



**Key Points To Remember**

* An interface is a contract that defines the capabilities of a class.
  + Dart has no keyword interface, but you can use class or abstract class to declare an interface.
* Use abstract class to declare an interface.
  + A class can extend only one class but can implement multiple interfaces.
    - Using the interface, you can achieve multiple inheritance in Dart.
      * It is used to achieve abstraction.

### Mixin in Dart

In this section, you will learn about **dart mixins** to reuse the code in multiple classes.

**Mixin In Dart**

Mixins are a way of reusing the code in multiple classes. Mixins are declared using the keyword **mixin** followed by the mixin name. Three keywords are used while working with mixins: **mixin**, **with**, and **on**. It is possible to use multiple mixins in a class.

**Note:** The **with** keyword is used to apply the mixin to the class. It promotes DRY(Don’t Repeat Yourself) principle.

**Rules For Mixin**

* **Mixin** can’t be instantiated. You can’t create object of mixin.
  + Use the **mixin** to share the code between multiple classes.
* **Mixin** has no constructor and cannot be extended.
  + - It is possible to use multiple **mixins** in a class.

**Syntax**

mixin Mixin1{ // code



}

mixin Mixin2{ // code

}

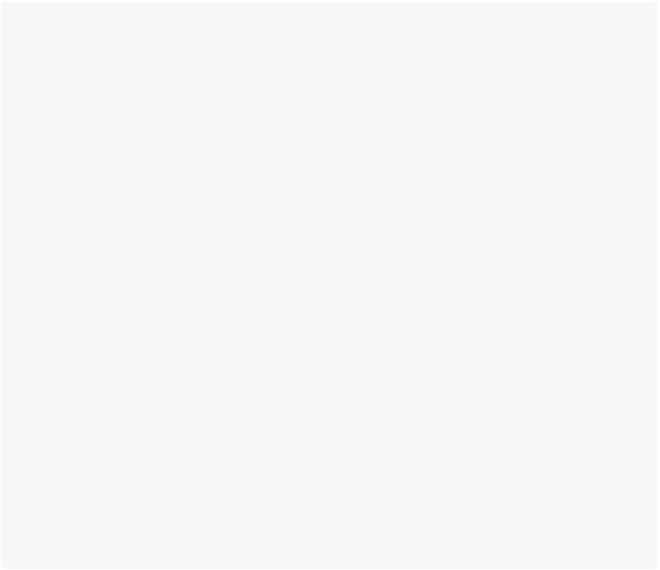
class ClassName with Mixin1, Mixin2{ // code

}

**Example 1: Mixin In Dart**

In this example below, there are two mixins named **ElectricVariant** and **PetrolVariant**. The **ElectricVariant** mixin has a method **electricVariant()** and the **PetrolVariant** mixin has a method **petrolVariant()**. The **Car** class uses both the **ElectricVariant** and **PetrolVariant** mixins.

mixin ElectricVariant {



void electricVariant() {

print('This is an electric variant'); }

}

mixin PetrolVariant {

void petrolVariant() {

print('This is a petrol variant');

}

}

// with is used to apply the mixin to the class

class Car with ElectricVariant, PetrolVariant {

// here we have access of electricVariant() and petrolVariant()

methods }

void main() {

var car = Car();

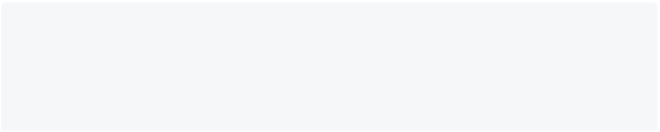
car.electricVariant(); car.petrolVariant(); }

[Run Online](https://dartpad.dev/?id=0eb836a9478580ce08e65b291260dcf8)

**Example 2: Mixin In Dart**

In this example below, there are two mixins named **CanFly** and **CanWalk**. The **CanFly** mixin has a method **fly()** and the **CanWalk** mixin has a method **walk()**. The **Bird** class uses both the **CanFly** and **CanWalk** mixins. The **Human** class uses the **CanWalk** mixin.

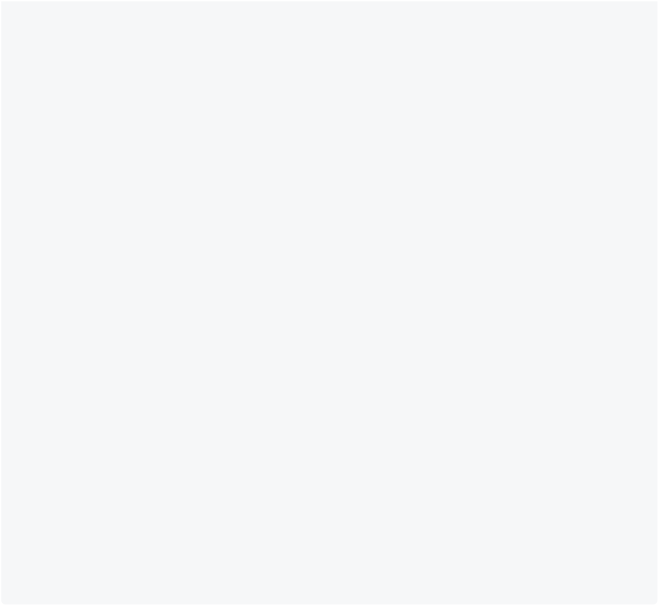
mixin CanFly {



void fly() {

print('I can fly'); }

}



mixin CanWalk {

void walk() {

print('I can walk'); }

}

class Bird with CanFly, CanWalk { }

class Human with CanWalk {

}

void main() {

var bird = Bird(); bird.fly();

bird.walk();

var human = Human(); human.walk();

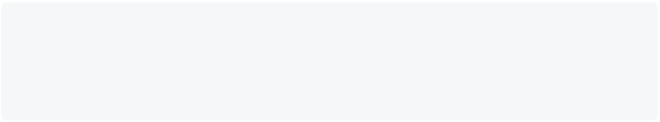
}

[Run Online](https://dartpad.dev/?id=c71ee27c88f652f26eade3dd155c41d1) **On Keyword**

Sometimes, you want to use a mixin only with a specific class. In this case, you can use the **on** keyword.

**Syntax Of On Keyword**

mixin Mixin1 on Class1{ // code

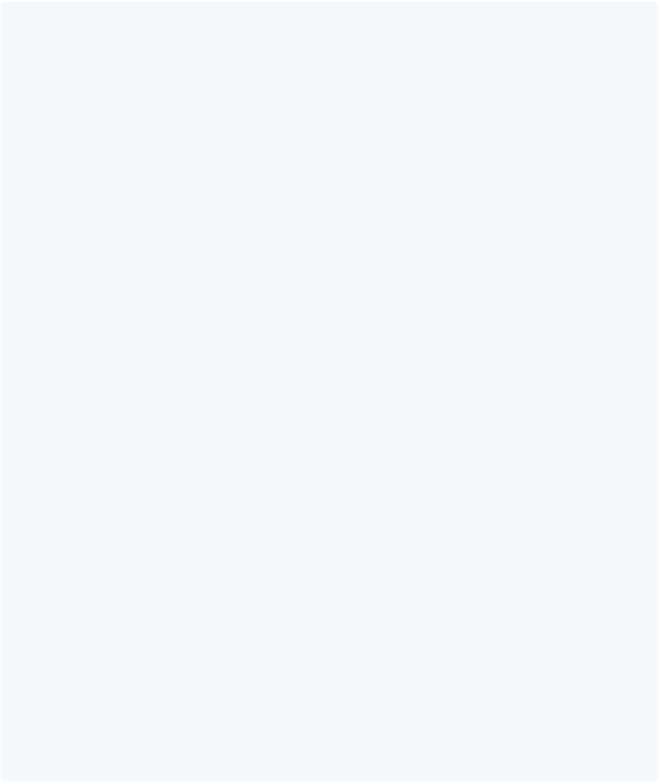


}

**Example 3: On Keyword In Mixin In Dart**

In this example below, there is abstract class named **Animal** with properties **name** and **speed**. The **Animal** class has an abstract method **run()**. The **CanRun** mixin is only used by class that extends **Animal**. The **Dog** class extends the **Animal** class and uses the **CanRun** mixin. The **Bird** class cannot use the **CanRun** mixin because it does not extend the **Animal** class.

abstract class Animal { // properties



String name;

double speed;

// constructor

Animal(this.name, this.speed);

// abstract method void run();

}

// mixin CanRun is only used by class that extends Animal mixin CanRun on Animal {

// implementation of abstract method

@override

void run() => print('$name is Running at speed $speed'); }

class Dog extends Animal with CanRun {

// constructor

Dog(String name, double speed) : super(name, speed); }

void main() {

var dog = Dog('My Dog', 25); dog.run();

}

// Not Possible

// class Bird with Animal { }

[Run Online](https://dartpad.dev/?id=7a5a09ffd6ffb11beb50d3c443672142)

**What Is Allowed For Mixin**

* You can add properties and static variables.
  + You can add regular, abstract, and static methods.
* You can use one or more mixins in a class.

**What Is Not Allowed For Mixin**

* You can’t define a constructor.
  + You can’t extend a mixin.
    - You can’t create an object of mixin.

### Factory Constructor in Dart

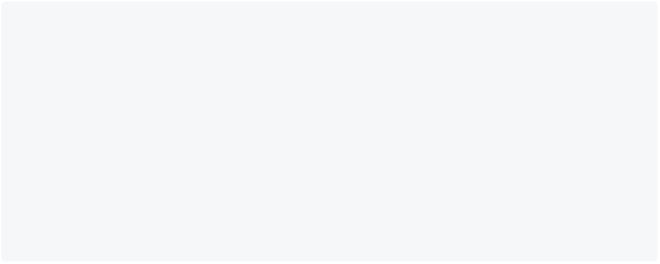
In this section, you will learn about factory constructors with examples. **Factory Constructor In Dart**

All of the constructors that you have learned until now are **generative constructors**. Dart also provides a special type of constructor called a **factory constructor**.

A **factory constructor** gives more flexibility to create an object. Generative constructors only create an instance of the class. But, the factory constructor can return an instance of the **class or even subclass**. It is also used to return the **cached instance** of the class.

**Syntax**

class ClassName {



factory ClassName() {

// TODO: return ClassName instance }

factory ClassName.namedConstructor() { // TODO: return ClassName instance }

}

**Rules For Factory Constructors**

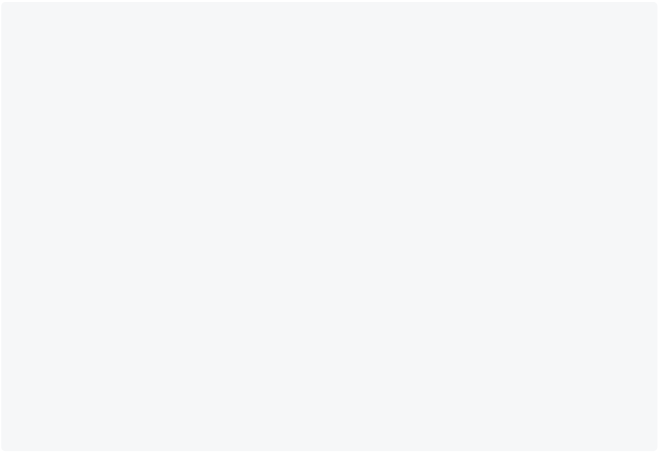
* Factory constructor must return an instance of the **class** or **sub-class**.
  + You can’t use **this** keyword inside factory constructor.
    - It can be **named** or **unnamed** and called like normal constructor.
  + It can’t access **instance members** of the class.

**Example 1: Without Factory Constructor**

In this example below, there is a class named **Area** with final properties **length** and **breadth**, and **area**. When you pass the **length** and **breadth** to the constructor, it calculates the **area** and stores it in the **area** property.

**Note**: An initializer list allows you to assign properties to a new instance variable before the constructor body runs, but after creation.

class Area {



final int length; final int breadth; final int area;

// Initializer list

const Area(this.length, this.breadth) : area = length \* breadth; }

void main() {

Area area = Area(10, 20);

print("Area is: ${area.area}");

// notice that here is a negative value Area area2 = Area(-10, 20);

print("Area is: ${area2.area}");

}

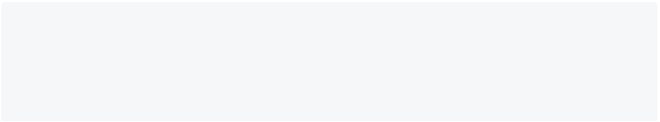
[Run Online](https://dartpad.dev/?id=8a40348e1eff165cdec252e659ead4a0)

Here **area2** object has a negative value. This is because we are not validating the input. Let’s create a factory constructor to validate the input.

**Example 2: With Factory Constructor**

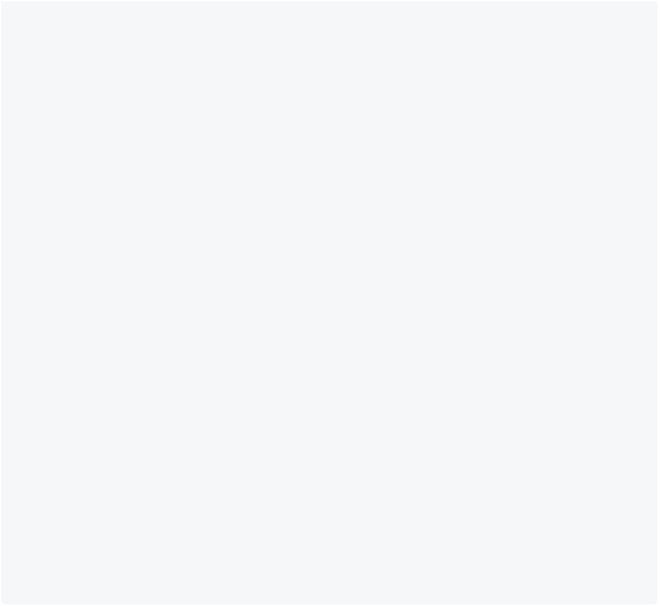
In this example below, **factory constructor** is used to validate the input. If the input is valid, it will return a new class instance. If the input is invalid, then it will throw an exception.

class Area {



final int length; final int breadth; final int area;

// private constructor



const Area.\_internal(this.length, this.breadth) : area = length \* breadth;

// Factory constructor

factory Area(int length, int breadth) {

if (length < 0 || breadth < 0) {

throw Exception("Length and breadth must be positive"); }

// redirect to private constructor

return Area.\_internal(length, breadth);

}

}

void main() {

// This works

Area area = Area(10, 20);

print("Area is: ${area.area}");

// notice that here is negative value Area area2 = Area(-10, 20);

print("Area is: ${area2.area}");

}

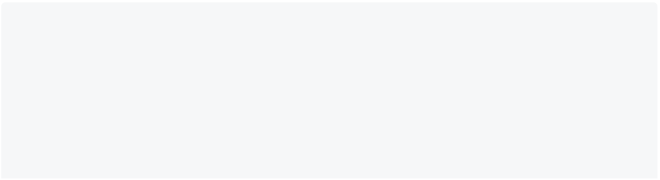
[Run Online](https://dartpad.dev/?id=8ce217ca71fab3d93f48334f47f971e2)

**Note**: With a factory constructor, you can initialize a final variable using logic that can’t be handled in the initializer list.

**Example 3: Factory Constructor In Dart**

In this example below, there is a class named **Person** with two properties, **firstName** and **lastName**, and two constructors, a **normal constructor** and a **factory constructor**. The factory constructor creates a Person object from a [**Map**](https://dart-tutorial.com/collections/map-in-dart/).

class Person {

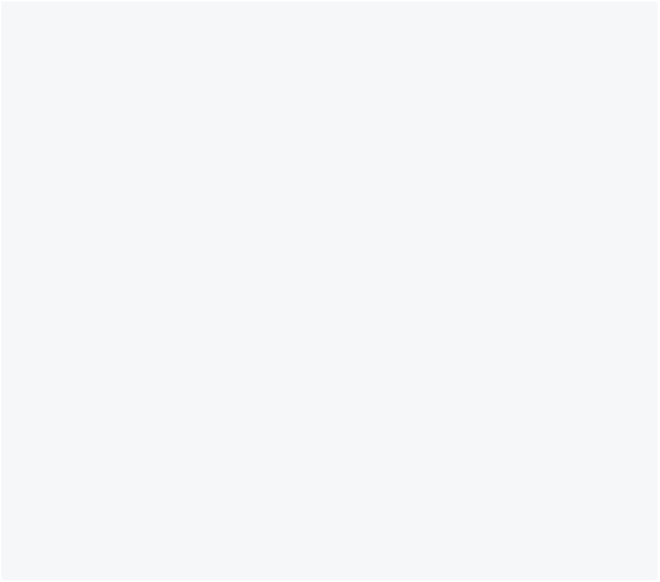


String firstName; String lastName;

// constructor

Person(this.firstName, this.lastName);

// factory constructor Person.fromMap



factory Person.fromMap(Map<String, Object> map) { final firstName = map['firstName'] as String; final lastName = map['lastName'] as String;

return Person(firstName, lastName);

}

}

void main() {

// create a person object

final person = Person('John', 'Doe');

// create a person object from map

final person2 = Person.fromMap({'firstName': 'Harry', 'lastName': 'Potter'});

// print first and last name

print("From normal constructor: ${person.firstName} ${person.lastName}");

print("From factory constructor: ${person2.firstName} ${person2.lastName}");

}

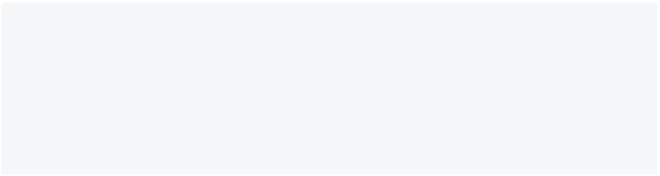
In the main method, two objects are created, one using the **generative/normal constructor** and the other using the **factory constructor**.

[Run Online](https://dartpad.dev/?id=7578ec0bc4d7b3506d6970d8dd5b55c6)

**Example 4: Factory Constructor In Dart**

In this example below, there is [**enum**](https://dart-tutorial.com/object-oriented-programming/enum-in-dart/) **ShapeType** with two values: **circle** and **rectangle**. There is an [**interface**](https://dart-tutorial.com/object-oriented-programming/interface-in-dart/) **Shape** with a factory constructor that creates objects of type Shape, either Circle or Rectangle. The **main** method instantiates two objects, one of each type, and calls the **draw()** method on each.

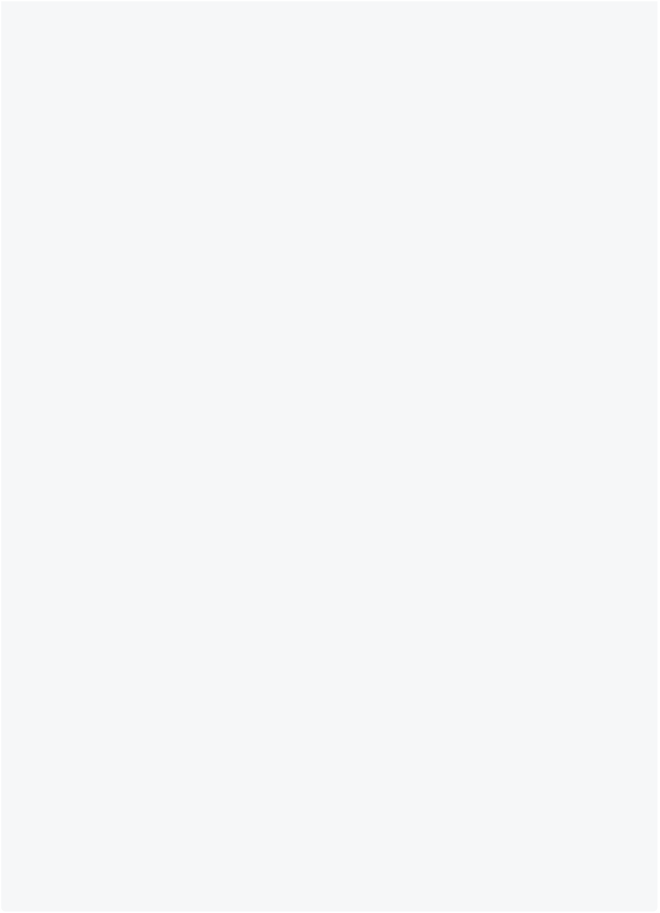
// enum ShapeType



enum ShapeType { circle, rectangle }

// abstract class Shape abstract class Shape { // factory constructor

factory Shape(ShapeType type) { switch (type) {



case ShapeType.circle:

return Circle();

case ShapeType.rectangle:

return Rectangle();

default:

throw 'Invalid shape type'; }

}

// method

void draw();

}

class Circle implements Shape { // implement draw method

@override

void draw() {

print('Drawing circle');

}

}

class Rectangle implements Shape { // implement draw method

@override

void draw() {

print('Drawing rectangle');

}

}

void main() {

// create Shape object

Shape shape = Shape(ShapeType.circle);

Shape shape2 = Shape(ShapeType.rectangle); shape.draw();

shape2.draw();

}

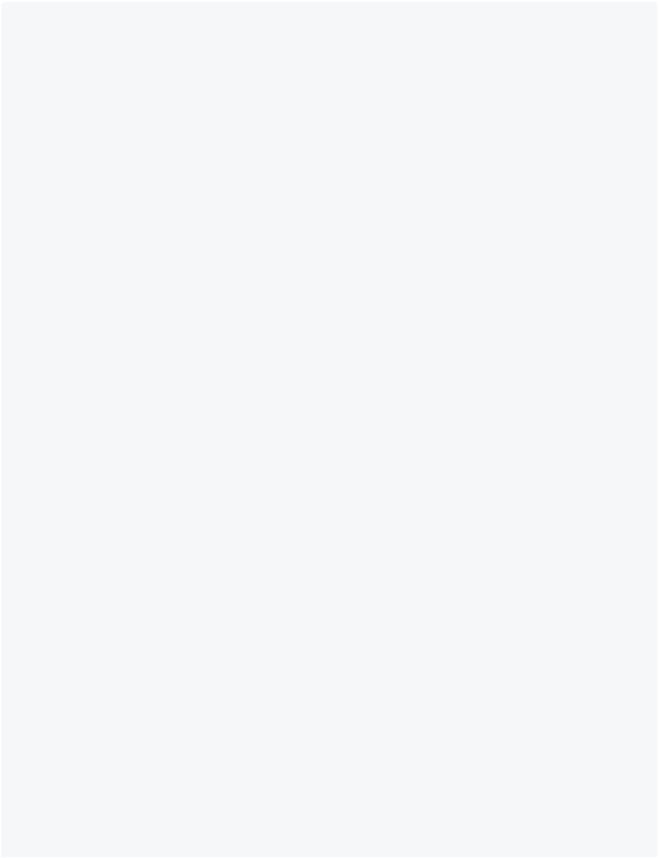
[Run Online](https://dartpad.dev/?id=0af4314bea6399fffaae2e8e41fe32d8)

**Note**: Here it is possible to make **List** which contains both **Circle** and **Rectangle** objects in it.

**Example 5: Factory Constructor In Dart**

In this example below, there is class **Person** with a final field **name**. It also has a private constructor and a static **\_cache** field. The class also has a **factory constructor** that checks if the **\_cache** field contains a key that matches the name parameter. If it does, it returns the Person object associated with that key. Otherwise, it creates a new **Person** object, adds it to the **\_cache**, and returns it.

class Person {



// final fields

final String name;

// private constructor

Person.\_internal(this.name);

// static \_cache field

static final Map<String, Person> \_cache = <String, Person>{};

// factory constructor

factory Person(String name) {

if (\_cache.containsKey(name)) {

return \_cache[name]!;

} else {

final person = Person.\_internal(name); \_cache[name] = person;

return person;

}

}

}

void main() {

final person1 = Person('John'); final person2 = Person('Harry'); final person3 = Person('John');

// hashcode of person1 and person3 are same

print("Person1 name is : ${person1.name} with hashcode ${person1.hashCode}");

print("Person2 name is : ${person2.name} with hashcode ${person2.hashCode}");

print("Person3 name is : ${person3.name} with hashcode ${person3.hashCode}");

}



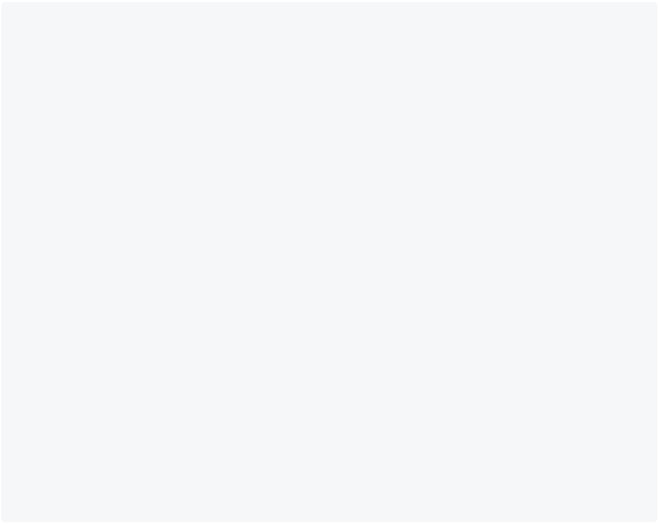
[Run Online](https://dartpad.dev/?id=8afe9651a6aa776a2f8b6ae7d0739be4) **Singleton In Dart**

Singletons are a common design pattern in object-oriented programming. A singleton class can have only one instance and provides a global point of access to it. You can create a singleton in Dart by defining a **factory constructor** that always returns the same instance. It is mostly useful when you want to create a single instance of a class and use it throughout the application like **database connection app**.

**Example 6: Singleton Using Factory Constructor**

This code creates a **Singleton** class that can only be instantiated once, and provides a factory constructor to get the instance of the class. The main method creates two objects of the Singleton class, and prints the hashcode of the objects to verify that **they are same**.

// Singleton using dart factory



class Singleton {

// static variable

static final Singleton \_instance = Singleton.\_internal();

// factory constructor factory Singleton() { return \_instance;

}

// private constructor Singleton.\_internal(); }

void main() {

Singleton obj1 = Singleton(); Singleton obj2 = Singleton(); print(obj1.hashCode);

print(obj2.hashCode);

}

[Run Online](https://dartpad.dev/?id=19b5aac734f3894f46955404fa99356c)

You can see that both objects have the same hashcode. This is because both objects are pointing to the same instance.

**Note**: Here Singleton.\_internal() is a private constructor so that it can not be called from outside the library. The factory constructor is used to return the same instance of the class.

**Key Points**

Here **It** means **factory constructor**

* It uses the **factory** keyword to define a factory constructor.
  + It returns an instance of the same class or sub-class.
    - It is used to implement factory design patterns. [Return sub-class instance based on input parameter as shown in example 4]
      * It is used to implement singleton design patterns. [Return the same instance every time]
        + It is used to initialize a final variable using logic that can’t be handled in the initializer list.

## Null Safety in Dart

**Null safety** is a feature in the Dart programming language that helps developers to avoid null errors. This feature is called **Sound Null Safety** in dart. This allows developers to catch null errors at edit time.

### Advantage Of Null Safety

* Write safe code.
  + Reduce the chances of application crashes.
    - Easy to find and fix bugs in code.

**Note**: Null safety avoids null errors, runtime bugs, vulnerabilities, and system crashes which are difficult to find and fix.

**Example 1: Using Null In Variables**

In the example below, the variable **age** is a **int** type. If you pass a null value to this variable, it will give an error instantly.

int age = null; // give error }

[Run Online](https://dartpad.dev/?id=01113313e1f416914a18fd6c84bb60bd)

**Problem With Null**

Programmers do have a lot of difficulties while handling null values. They forget that there are **null** values, so the program breaks. In real world **null** mostly acts as **time bomb** for programmers, which is ready to break the program.

**Note**: Common cause of errors in programming generally comes from not correctly handling null values.

**Non-Nullable By Default**

In Dart, variables and fields are non-nullable by default, which means that they cannot have a value **null** unless you explicitly allow it.

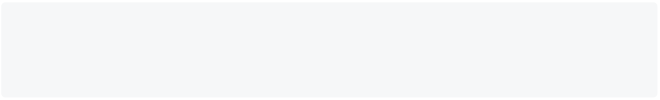
int productid = 20; // non-nullable int productid = null; // give error



**How To Declare Null Value**

With dart **sound null Safety**, you cannot provide a null value by **default**. If you are 100% sure to use it, then you can use **?** operator after the type declaration.

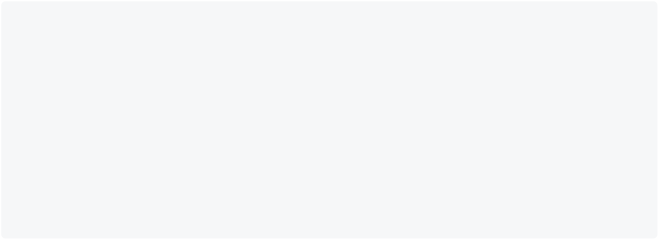
// Declaring a nullable variable by using ? String? name;



This declares a variable **name**, which can be null or a string. **How To Assign Values To Nullable Variables**

You can assign a value to nullable variables just like any other variable. However, you can also assign null to them.

void main(){



// Declaring a nullable variable by using ? String? name;

// Assigning John to name

name = "John";

// Assigning null to name

name = null;

}

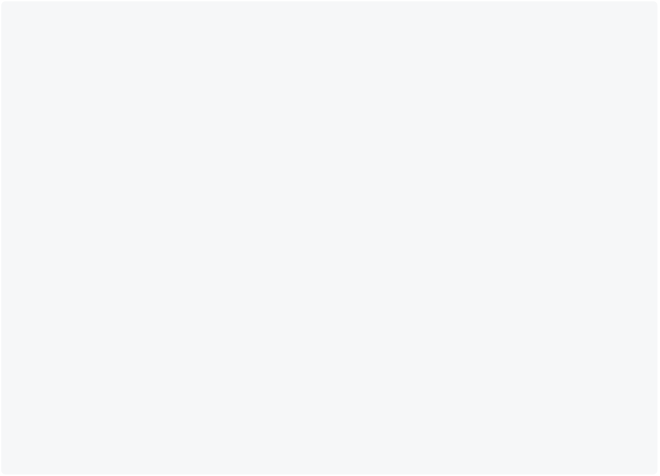
[Run Online](https://dartpad.dev/?id=613c72fcac9f86143ed001aa59d8b94d)

**How To Use Nullable Variables**

You can use nullable variables in many ways. Some of them are shown below:

* You can use **if** statement to check whether the variable is null or not.
  + You can use **!** operator, which returns null if the variable is null.
    - You can use **??** operator to assign a default value if the variable is null.

void main(){



// Declaring a nullable variable by using ? String? name;

// Assigning John to name

name = "John";

// Assigning null to name

name = null;

// Checking if name is null using if statement if(name == null){

print("Name is null");

}

// Using ?? operator to assign a default value String name1 = name ?? "Stranger";

print(name1);

// Using ! operator to return null if name is null String name2 = name!;

print(name2);

}

[Run Online](https://dartpad.dev/?id=9b62520244c3c83e82adf8c709482a4b)

**Example 2: Define List Of Nullable Items**

You can also store null in list values. In this example, the **items** is a list of nullable integers. It can contain null values as well as integers.

void main() {



// list of nullable ints

List<int?> items = [1, 2, null, 4]; print(items);

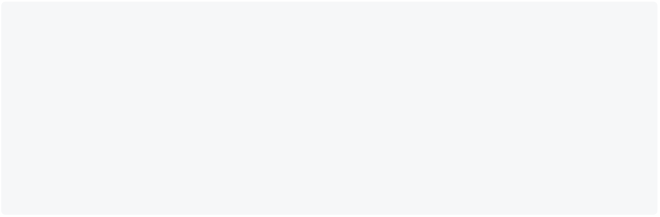
}

[Run Online](https://dartpad.dev/?id=524986d35682cc80987e6bfc5c6e17cd)

**Example 3: Null Safety In Dart Functions**

In this example, the function **printAddress** has a parameter **address** which is a **String** type. If you pass a **null** value to this function, it will give a edit-time error.

void printAddress(String address) { print(address);



}

void main() {

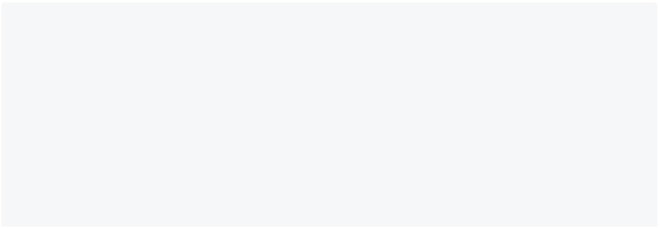
printAddress(null); // give error }

[Run Online](https://dartpad.dev/?id=4747b48f331357beb5d5669e002aedb8)

**Example 4: Define Function With Nullable Parameter**

If you are 100% sure, then you can use **?** for the type declaration. In this example, the function **printAddress** has a parameter **address**, which is a **String?** type. You can pass both null and string values to this function.

// address is a nullable string void printAddress(String? address) { print(address);



}

void main() {

// Passing null to printAddress

printAddress(null); // Works

}

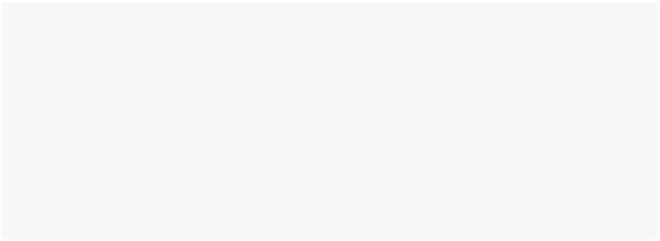


[Run Online](https://dartpad.dev/?id=816ecbb250bfabc9b8da65323bc2f8a5)

**Example 5: Null Safety In Dart Class**

In the example, the class **Person** has a parameter **name**, which is a **String** type. If you pass a null value to this class, it will give a compile-time error.

class Person {



String name;

Person(this.name); }

void main() {

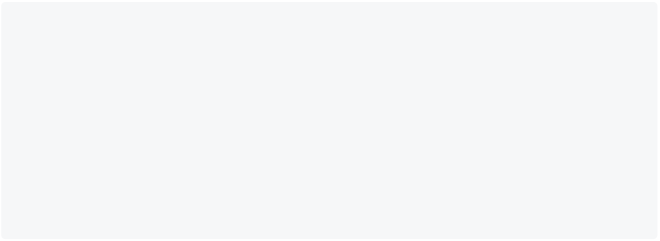
Person person = Person(null); // give error }

[Run Online](https://dartpad.dev/?id=2f2e47ffcb6c083b9b46b6838b149d8b)

**Example 6: Define Null To Class Property**

In this example, the class **Person** has a parameter **name**, which is a **String?** type. You can pass both null and string values to this class. To define a nullable property in a class, you can use the **?** operator after the type.

class Person {



String? name;

Person(this.name); }

void main() {

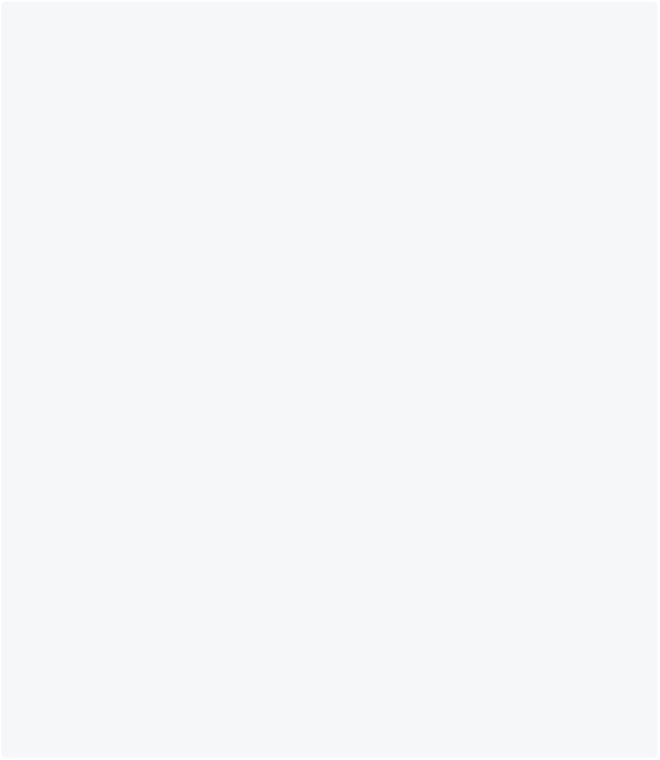
Person person = Person(null); // Works }

[Run Online](https://dartpad.dev/?id=7c4a7e828bfb274e20b12d93e81f0bb5)

**Example 7: Working With Nullable Class Properties**

In the example below, the **Profile** class has two nullable properties: **name** and **bio**. The **printProfile** method prints the name and bio of the profile. If the name or bio is **null**, it prints a default value instead.

class Profile { String? name; String? bio;



Profile(this.name, this.bio);

void printProfile() {

print("Name: ${name ?? "Unknown"}");

print("Bio: ${bio ?? "None provided"}"); }

}

void main() {

// Create a profile with a name and bio

Profile profile1 = Profile("John", "Software engineer and avid reader");

profile1.printProfile();

// Create a profile with only a name

Profile profile2 = Profile("Jane", null); profile2.printProfile();

// Create a profile with only a bio

Profile profile3 = Profile(null, "Loves to travel and try new foods"); profile3.printProfile();

// Create a profile with no name or bio Profile profile4 = Profile(null, null); profile4.printProfile();

}

[Run Online](https://dartpad.dev/?id=6ce7d4d09cc92a9276e7b5b34f777e5c)

**Important Point In Dart Null Safety**

* Null means no value.
  + Common error in programming is caused due to null.
    - Dart 2.12 introduced **sound null Safety** to solve null problems.
* Non-nullable type is confirmed never to be **null**.

**Note**: Sometimes you heard word like **NNBD**. It is **Non-Nullable By Default**, which means you can’t assign null to a variable by default.

### Type Promotion in Dart

**Type promotion in dart** means that dart automatically converts a value of one type to another type. Dart does this when it knows that the value is of a specific type.

**How Type Promotion Works In Dart?**

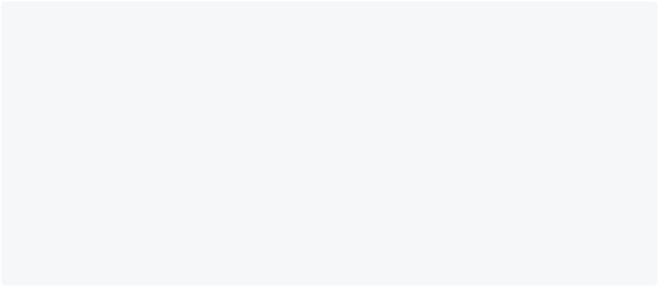
Types Promotion in Dart works in the following ways:

* Promoting from **general types** to **specific subtypes**.
  + Promoting from **nullable types** to **non-nullable types**.

**Example 1: Promoting From General Types To Specific Subtypes**

In this example, the variable **name** is declared as an **Object**. The **Object** class doesn’t have a **.length** property. Variable **name** gets promoted from **Object** to **String** so that you can access the **.length** property of the String class.

void main(){



Object name = "Pratik";

// print(name.length) will not work because Dart doesn't know that name is a String

if(name is String) {

// name promoted from Object to String

print("The length of name is ${name.length}"); }

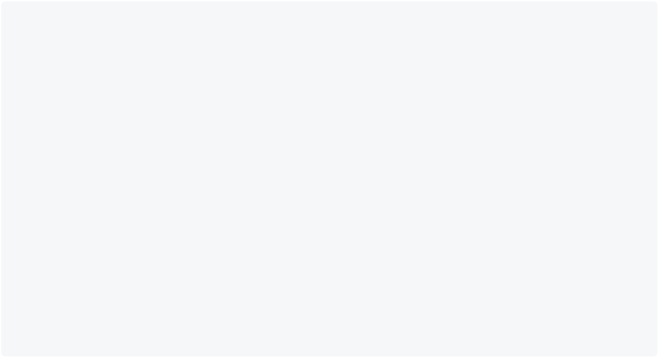
}

[Run Online](https://dartpad.dev/?id=fe8a778897707521b4c305e1da4526eb)

**Example 2: Type Promotion In Dart**

In this example, the variable **result** is declared as a **String**. In both **if** and **else** blocks, the variable **result** is assigned a value of type **String**. Therefore, the variable **result** is automatically promoted to a non-nullable type **String**.

void main(){



// result is a String

String result;

// result is promoted to a non-nullable type String if(DateTime.now().hour < 12) {

result = "Good Morning";

} else {

result = "Good Afternoon";

}

// display the result

print("Result is $result");

print("Length of result is ${result.length}");

}

[Run Online](https://dartpad.dev/?id=4b71f2b7e238bae98743f54014f7878f)

**Example 3: Type Promotion With Nullable To Non-Nullable Type**

In Dart, you can also throw an exception if the variable is null. In this example, method **printLength**, takes a **String** type parameter. If the parameter is null, then it will throw an exception.

// method to print the length of the text void printLength(String? text){



if(text == null) {

throw Exception("The text is null"); }

print("Length of text is ${text.length}"); }

// main method

void main() {

printLength("Hello");

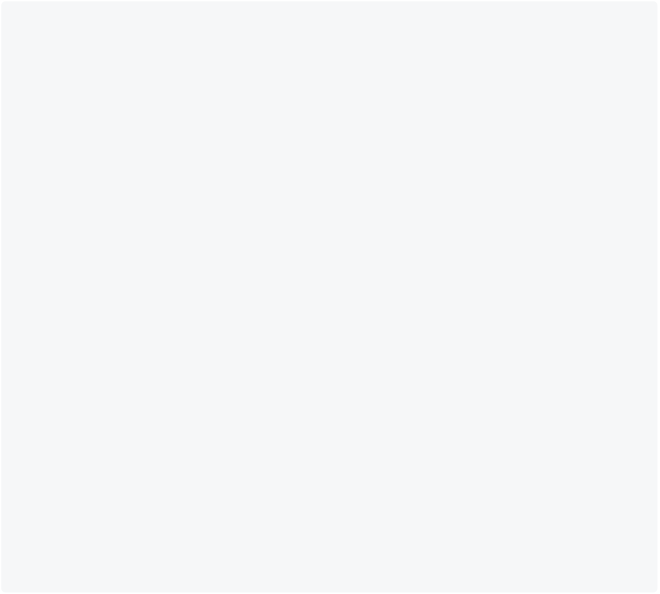
}

[Run Online](https://dartpad.dev/?id=1f8ddc8b76885a6476ae309be9de0518)

**Example 4: Type Promotion With Nullable Type To Non-Nullable Type**

In this example, the variable **value** contains a value of type **String** or **null**. The variable **value** is promoted to a non-nullable type **String** in the **if** block. If the variable **value** is null, then the **else** block is executed.

// importing dart:math library



import 'dart:math';

// creating a class DataProvider

class DataProvider{

// creating a method stringorNull

String? get stringorNull => Random().nextBool() ? "Hello" : null;

// creating a method myMethod

void myMethod(){

String? value = stringorNull;

// checking if value String or not

if(value is String){

print("The length of value is ${value.length}"); }else{

print("The value is not string.");

}

}

}

// main method

void main() {

DataProvider().myMethod(); }

[Run Online](https://dartpad.dev/?id=98dc5342dc26eb91e783065c27f15c86)

**Note:** The output of the above example is random. It can be either **The length of value is 5** or **The value is not string.**

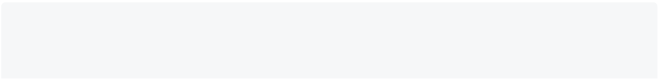
### Late Keyword in Dart

In dart, **late** keyword is used to declare a variable or field that will be initialized at a later time. It is used to declare a **non-nullable** variable that is not initialized at the time of declaration.

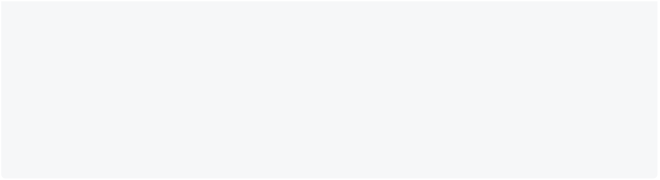
**Example 1: Late Keyword In Dart**

In this example, **name** variable is declared as a **late** variable. The **name** variable is initialized in the **main** method.

// late variable late String name;



void main() {



// assigning value to late variable name = "John";

print(name);

}

[Run Online](https://dartpad.dev/?id=0b62efdc3696031bd48ef6c28bec9e11)

When you put **late** infront of a variable declearation, you tell Dart the following:

* Don’t assign that variable a value yet.
  + You will assign value later.
    - You will make sure the variable has a value before you use it.

**Note**: The **late** keyword is contract between you and Dart. You are telling Dart that you will assign a value to the variable before you use it. If you don’t assign a value to the variable before you use it, Dart will throw an error.

**Example 2: Late Keyword In Dart**

In this example, there is **Person** class with a **name** field. The **name** field is declared as a late variable.

class Person {



// late variable late String name;

void greet() {

print("Hello $name"); }

}

void main() {

Person person = Person();

// late variable is initialized here person.name = "John";

person.greet();

}

[Run Online](https://dartpad.dev/?id=1b3d71610d20b76db295aec626788316)

**Usecase of Late Keyword In Dart** Dart late keyword has two use cases:

* **Declaring a non-nullable variable or field** that is not initialized at the point of declaration.
  + **Lazy initialization** of a variable or field.

**What Is Lazy Initialization**

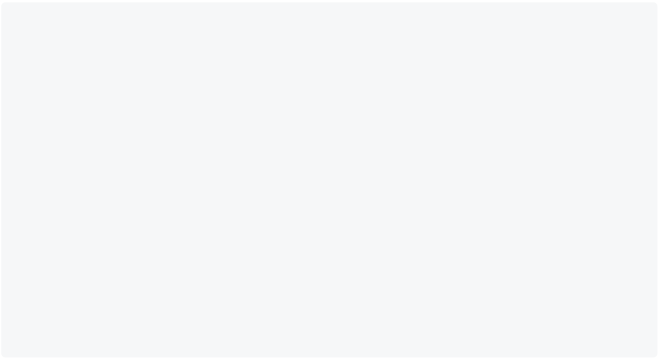
**Lazy initialization** is a design pattern that delays the creation of an object, the calculation of a value, or some other expensive process until the **first time you need it**.

Note: Using **late** means dart doesn’t initialize value right away, it only initializes when you access it for the first time. This is also called **lazy loading**.

**Example 3: Late Keyword In Dart**

In this example, the **provideCountry** function is not called when the **value** variable is declared. The **provideCountry** function is called only when the **value** variable is used. **Lazy initialization** is used to avoid unnecessary computation.

// function



String provideCountry() {

print("Function is called"); return "USA";

}

void main() {

print("Starting");

// late variable

late String value = provideCountry(); print("End");

print(value);

}

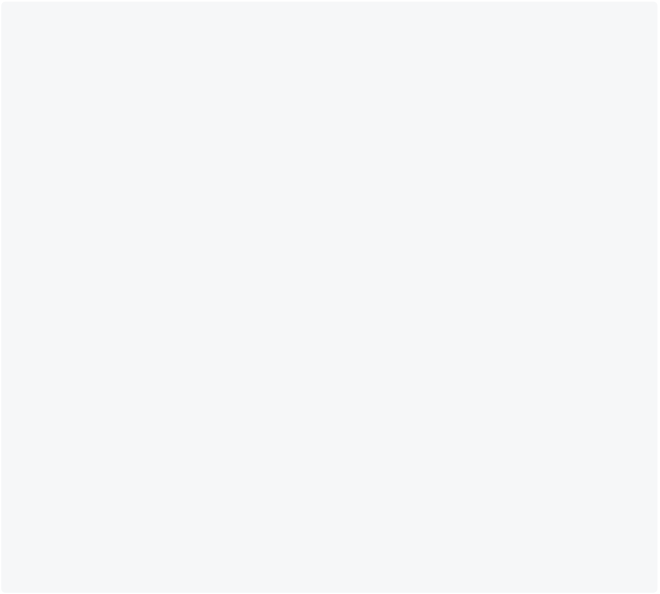
If you remove the **late** keyword from the **value** variable, the **provideCountry** function will be called when the **value** variable is declared.

[Run Online](https://dartpad.dev/?id=0bff775f0130bfa491c1e68eb0399152)

**Example 4: Late Keyword In Class**

In this example, the **heavyComputation** function is called when the **description** variable is used. If you remove the **late** keyword from the **description** variable, the **heavyComputation** function will be called when the **Person** class is instantiated.

// Person class



class Person {

final int age;

final String name;

late String description = heavyComputation();

// constructor

Person(this.age, this.name) {

print("Constructor is called");

}

// method

String heavyComputation() {

print("heavyComputation is called"); return "Heavy Computation";

}

}

void main() {

// object of Person class

Person person = Person(10, "John"); print(person.name);

print(person.description);

}

[Run Online](https://dartpad.dev/?id=675a3320b36f06e57c50b74fc239c86a)

**Example 5: Late Keyword In Class**

In this example, the **\_getFullName** function is called when the **fullName** variable is used. The **firstName** and **lastName** variables are initialized when the **fullName** variable is used.

class Person {

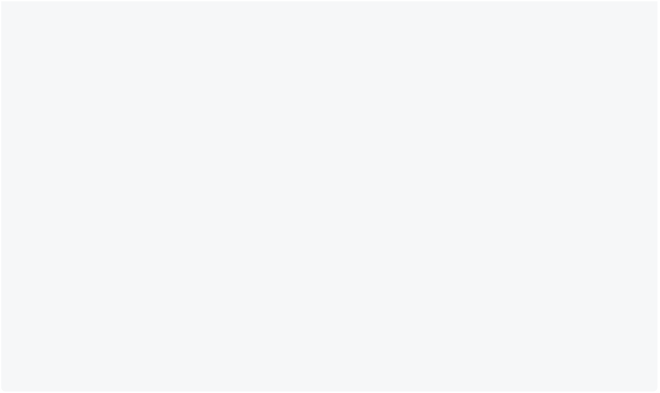


// declaring late variables

late String fullName = \_getFullName();

late String firstName = fullName.split(" ").first; late String lastName = fullName.split(" ").last;

// method



String \_getFullName() {

print("\_getFullName is called");

return "John Doe";

}

}

// main method

void main() {

print("Start");

Person person = Person();

print("First Name: ${person.firstName}"); print("Last Name: ${person.lastName}"); print("Full Name: ${person.fullName}");

print("End"); }

[Run Online](https://dartpad.dev/?id=ed92ec433b14216848b5cb9c3bb8249f)

**Note**: If you remove the **late** keyword from the **fullName** variable, the **\_getFullName** function will be called when the **Person** class is instantiated.

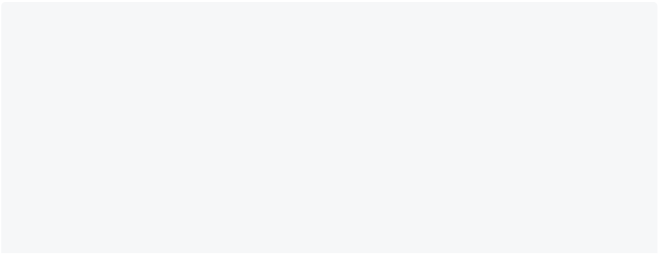
**Late Final Keyword In Dart**

If you want to assign a value to a variable only once, you can use the **late final** keyword. This is useful when you want to initialize a variable only once.

**Example 6: Late Final Keyword In Dart**

In this example, there is class **Student** with a **name** field. The **name** field is declared as a **late final** variable. The **name** field is initialized in the **Student** constructor. The **name** field is assigned a value only once. If you try to assign a value to the **name** field again, you will get an error.

// Student class



class Student {

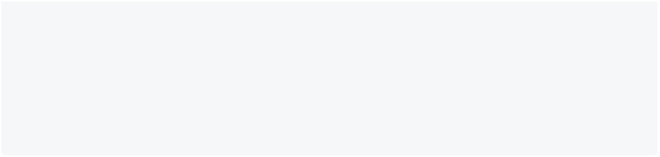
// late final variable late final String name;

// constructor

Student(this.name); }

void main() {

// object of Student class



Student student = Student("John"); print(student.name);

student.name = "Doe"; // Error

}

[Run Online](https://dartpad.dev/?id=13ce7925a45402dcfa1c65789ecaeb84)

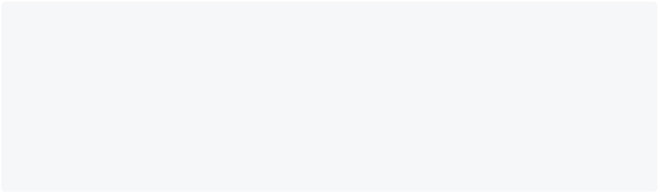
### Null Safety Exercise

Practice these exercises to master **dart null safety**. To practice these exercises, click on **Run Online** button and solve the problem.

**Exercise 1: Null Safety In Dart**

In variable name **age**, assign a **null** value to it using **?**.

// Try to assign a null value to age variable using ? void main() {



int age;

age = null;

print("Age is $age");

}

[Run Online](https://dartpad.dev/?id=0c90ae582f56343e7ba38e247680e786)

**Exercise 2: Nullable Type Parameter For Generics**

Try using **?** to make the type parameter of **List** nullable.

// Try to make the type parameter of List nullable void main() {



List<int> items = [1, 2, null, 4];

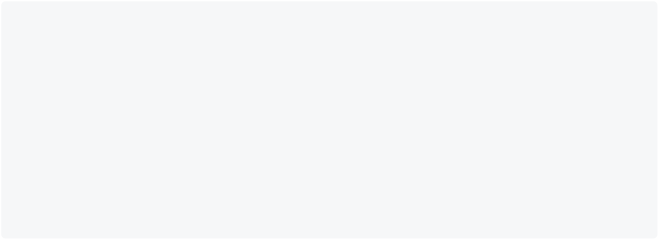
print(items);

}

[Run Online](https://dartpad.dev/?id=f4eb46a46a06bec61c776def77b42158)

**Exercise 3: Null Assertion Operator (!)**

Try using null assertion operator **!** to print null if the variable is null. // Try to use null assertion operator(!) to print null if the variable is null



void main() {

String? name;

name = null;

String name1 = name;

print(name1);

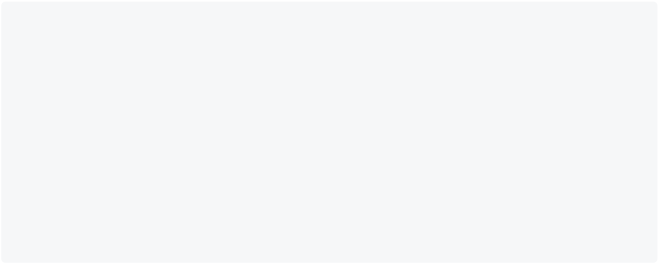
}

[Run Online](https://dartpad.dev/?id=ed43a091ebd5b1b081960e889085d94f)

**Exercise 4: Null Assertion Operator (!) For Generics**

Try using null assertion operator **!** to print null if the variable is null.

// Try to use null assertion operator(!) to print null if the variable is null



void main() {

List<int?> items = [1, 2, null, 4];

int firstItem = items.first;

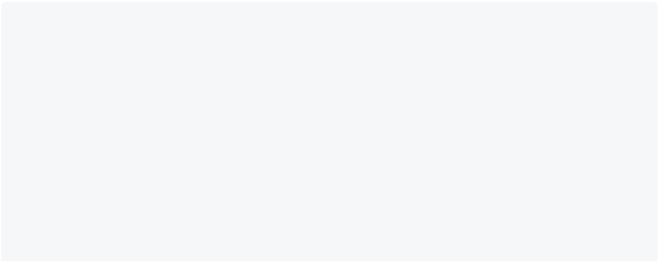
print(firstItem); }

[Run Online](https://dartpad.dev/?id=797bfbffe79c101e870dd1011a57ff3f)

**Exercise 5: Null Assertion Operator (!) For Generics**

Try using null assertion operator **!** to print null if the variable is null.

// Try to use null assertion operator(!) to print null if the variable is null



int? returnNullButSometimesNot() {

return -5;

}

void main() {

int result = returnNullButSometimesNot().abs(); print(result);

}



[Run Online](https://dartpad.dev/?id=960859806c11a03375d303b2bc5f6d06)

**\*\*Exercise 6: Null Assertion Operator (!) \*\***

Try using null assertion operator **!** to print the length of the String or return null if the variable is null.

// Try to use null assertion operator(!) to print the length of the String or return null if the variable is null



int findLength(String? name) {

// add null assertion operator here

return name.length;

}

void main() {

int? length = findLength("Hello");

print("The length of the string is $length"); }

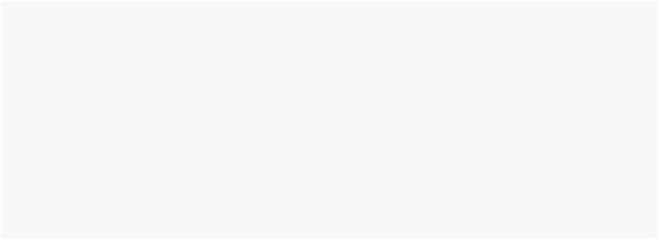
[Run Online](https://dartpad.dev/?id=0cee7a67081a4742670f77c110d9b46a)

**Exercise 7: Null Coalescing Operator (??)**

If you want to assign a default value to a variable if it is null, you can use null coalescing operator **??**.

Try using null coalescing operator **??** to assign a default value to **Stranger** if it is null.

// Try to use null coalescing operator(??) to assign a default value to Stranger if it is null



void main() {

String? name;

name = null;

String name1 = name;

print(name1);

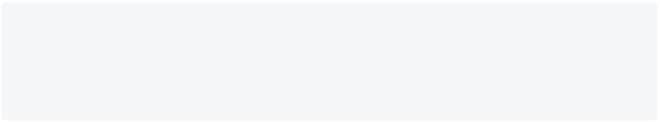
}

[Run Online](https://dartpad.dev/?id=5919b02a8d1c776a8db70d0030dae861)

**Exercise 8: Type Promotion**

Solve the error using type promotion:

// Try to solve the error using type promotion Object name = "Mark";



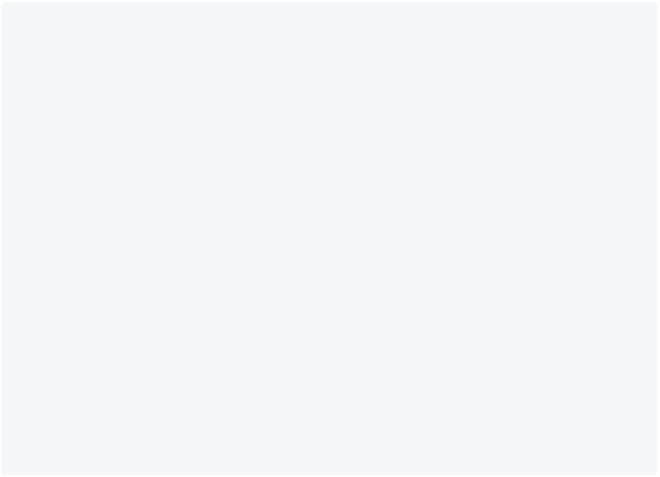
print("The length of name is ${name.length}");

[Run Online](https://dartpad.dev/?id=1408a935c649634b7638350dc78d58f6)

**Exercise 9: Type Promotion**

Solve the error using type promotion:

// Try to solve the error using type promotion



import 'dart:math';

class DataProvider{

String? get stringorNull => Random().nextBool() ? "Hello" : null;

void myMethod(){

if(stringorNull is String){

print("The length of value is ${stringorNull.length}"); }else{

print("The value is not string.");

}

} }

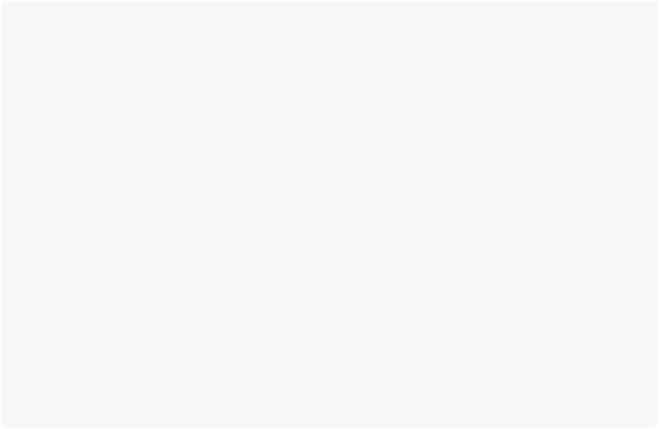
void main() {

DataProvider().myMethod(); }

[Run Online](https://dartpad.dev/?id=9a6874ffda0009702d0614c38944a7f6)

**Exercise 10: Late Keyword**

Try using **late** keyword to solve the error: // Try to solve the error using late keyword class Person{



String \_name;

void setName(String name){ \_name = name;

}

String get name => \_name; }

void main() {

Person person = Person(); person.setName("Mark"); print(person.name);

}

[Run Online](https://dartpad.dev/?id=661faf25a1796ad7d40aa9ffdc5b519e)

## Asynchronous Programming

**Asynchronous Programming** is a way of writing code that allows a program to do multiple tasks at the same time. Time consuming operations like fetching data from the internet, writing to a database, reading from a file, and downloading a file can be performed without blocking the main thread of execution.

### Synchronous Programming

In Synchronous programming, the program is executed line by line, one at a time. Synchronous operation means a task that needs to be solved before proceeding to the next one.

**Example Of Synchronous Programming**

void main() {



print("First Operation");

print("Second Big Operation"); print("Third Operation");

print("Last Operation");

}

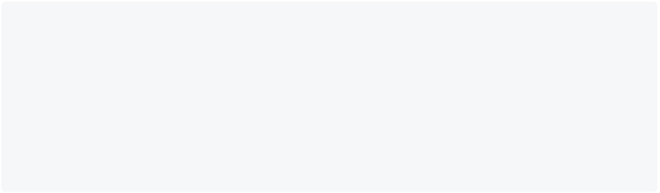
Here in this example, you can see that it will print line by line. Let’s suppose **Second Big Operation** takes 3 seconds to load then **Third Operation** and **Last Operation** need to wait for 3 seconds. To solve this issue asynchronous programming is here.

### Asynchronous Programming

In Asynchronous programming, program execution continues to the next line without waiting to complete other work. It simply means, **Don’t wait**. It represents the task that doesn’t need to solve before proceeding to the next one.

**Note**: Asynchronous Programming improves the responsiveness of the program. **Example Of Asynchronous Programming**

void main() {



print("First Operation");

Future.delayed(Duration(seconds:3),()=>print('Second Big Operation')); print("Third Operation");

print("Last Operation");

}

Here in this example, you can see that it will print **Second Big Operation** at last. It is taking 3 seconds to load and **Third Operation** and **Last Operation** don’t need to wait for 3 seconds. This is the problem solved by Asynchronous Programming. A Future represents a value that is not yet available, you will learn about Future in the next section.

### Why We Need Asynchronous

* To Fetch Data From Internet,
  + To Write Something to Database,
    - To execute a long-time consuming task,
      * To Read Data From File, and
        + To Download File etc.

Such **asynchronous operations** usually take a long time to complete, so it usually provide results in the form of a [**Future**]. If the result has multiple parts, then it provides as a [**Stream**]. You will learn about Future and Stream in the next section. **Note**: To Perform asynchronous operations in dart you can use the **Future** class and the **async** and **await** keywords. We will learn Future, Async, and Await later in this guide.

**Important Terms**

* **Synchronous** operation blocks other operations from running until it completes.
  + **Synchronous** function only perform a synchronous operation.
    - **Asynchronous** operation allows other operations to run before it completes.
      * **Asynchronous** function performs at least one asynchronous operation and can also perform synchronous operations.

### Future In Dart

In dart, the Future represents a value or error that is not yet available. It is used to represent a potential value, or error, that will be available at some time in the future.

**How To Create Future In Dart**

You can create a future in dart by using **Future** class. Here the function will return

**Future<String>** after 5 seconds.

// function that returns a future



Future<String> getUserName() async {

return Future.delayed(Duration(seconds: 2), () => 'Mark'); }

You can also create a future by using **Future.value()** method. Here the function will return **Future<String>** immediately.

// function that returns a future Future<String> getUserName() {



return Future.value('Mark');

}

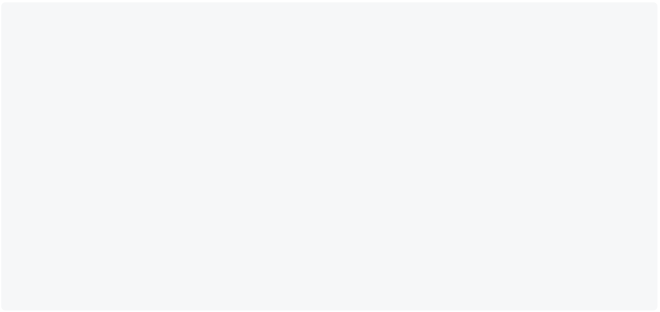
**How To Use Future In Dart**

You can use future in dart by using **then()** method. Here the function will return



**Future<String>** after 5 seconds.

// function that returns a future



Future<String> getUserName() async {

return Future.delayed(Duration(seconds: 2), () => 'Mark'); }

// main function

void main() {

print("Start");

getUserName().then((value) => print(value)); print("End");

}

**More About Future**

**Future** represents the result of an asynchronous operation and can have 2 states. **State Of Future**

* **Uncompleted**
  + **Completed**

**Uncompleted**

When you call an asynchronous function, it returns to an uncompleted future. It means the future is waiting for the function asynchronous operation to finish or to throw an error.

**Completed**

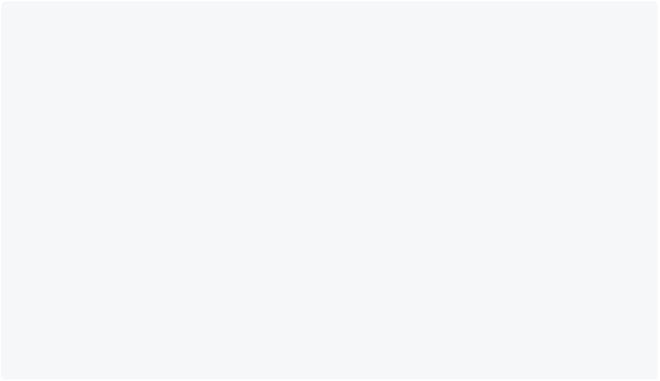
It can be completed with value or completed with error. **Future<int>** produces an int value, and **Future<String>** produces a String value. If the future doesn’t produce any value, then the type of future is **Future<void>**.

**Note**: If the asynchronous operation performed by the function fails due to any reason, the future completes with an error.

**Example 2: Future In Dart**

In this example below, we are creating a function **middleFunction()** that returns a future. The function will return **Future<String>** after 5 seconds.

void main() {



print("Start"); getData();

print("End"); }

void getData() async{

String data = await middleFunction(); print(data);

}

Future<String> middleFunction(){

return Future.delayed(Duration(seconds:5), ()=> "Hello"); }

**Note**: In the above example, First, it prints **Start**, secondly it prints **End**, and after 5 seconds **Hello** will be printed.

### Async and Await In Dart

**Async/await** is a feature in Dart that allows us to write asynchronous code that looks and behaves like synchronous code, making it easier to read.

When a function is marked **async**, it signifies that it will carry out some work that could take some time and will return a Future object that wraps the result of that work.

The **await** keyword, on the other hand, allows you to delay the execution of an async function until the awaited Future has finished. This enables us to create code that appears to be synchronous but is actually asynchronous.

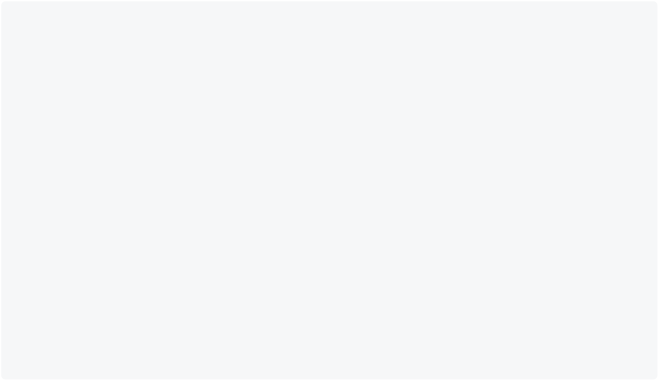
The **async** and **await** keywords both provide a declarative way to define an asynchronous function and use their results. You can use the **async** keyword before a function body to make it asynchronous. You can use the **await** keyword to get the completed result of an asynchronous expression.

**Important Concept**

* To define an Asynchronous function, add async before the function body.
  + The await keyword work only in the async function.

**Example 1: Synchronous Function**

void main() {



print("Start"); getData();

print("End"); }

void getData() {

String data = middleFunction(); print(data);

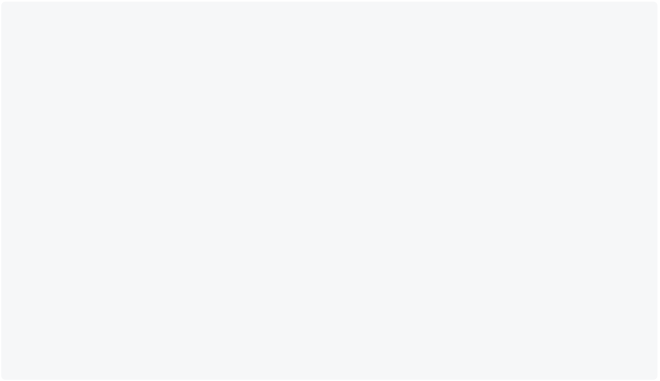
}

Future<String> middleFunction(){

return Future.delayed(Duration(seconds:5), ()=> "Hello"); }

**Example 2: Asynchronous function**

void main() {



print("Start"); getData();

print("End"); }

void getData() async{

String data = await middleFunction(); print(data);

}

Future<String> middleFunction(){

return Future.delayed(Duration(seconds:5), ()=> "Hello"); }

In the above example, async handles the states of the program where any part of the program can be executed.async always comes with await because await holds the part of the program until the rest of the program executed.

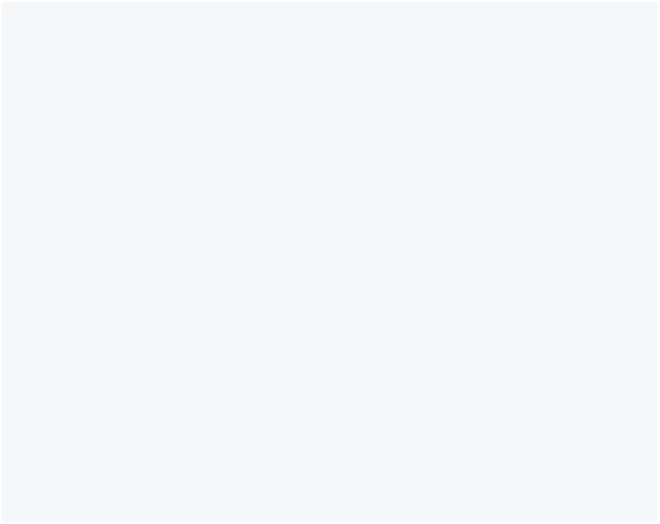


**Handling Errors**

You can handle errors in the dart async function by using try-catch. You can write try-catch code the same way you write synchronous code.

**Example 3: Handling Errors**

main() {



print("Start"); getData();

print("End"); }

void getData() async{

try{

String data = await middleFunction(); print(data);

}catch(err){

print("Some error $err");

}

}

Future<String> middleFunction(){

return Future.delayed(Duration(seconds:5), ()=> "Hello"); }

In the above example, try-catch handles the exception that could come after the program is executed.

Note: We cannot perform an asynchronous operation from a synchronous function.

**Important Terms**

* **async** The async keyword can be used before a function’s body to indicate that a function is asynchronous.
  + **async function** Functions marked with the async keyword are known as async functions.
    - **await** The completed output of an asynchronous expression can be retrieved with the await keyword. Only async functions can use the await keyword.

### Streams In Dart

A stream is a sequence of asynchronous events representing multiple values that will arrive in the future. Stream class deals with sequences of events instead of single events. Stream has one or more listeners, and all listeners will receive the same value.

For example, A stream is like a pipe that emits events, you put a value on the one end, and if there’s a listener on the other end that listener will receive that value. These events can be values of any type, errors or a “done” event to signal the end of the stream.

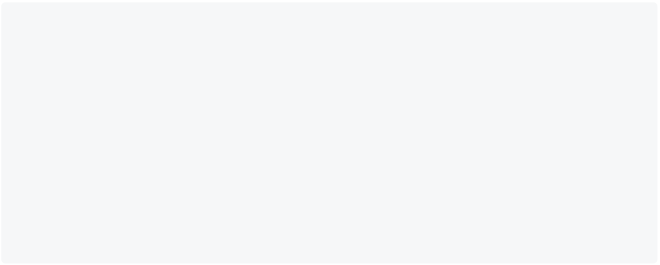
|  |  |  |
| --- | --- | --- |
|  | **Single Value** | **Zero or more values** |
| Sync | int | Iterator |
| Async | Future | Stream |

**How To Create Stream In Dart**

You can create a stream in dart by using **Stream** class. Here the function will return

**Stream<String>** after 5 seconds.

// function that returns a stream Stream<String> getUserName() async\* {



await Future.delayed(Duration(seconds: 1)); yield 'Mark';

await Future.delayed(Duration(seconds: 1)); yield 'John';

await Future.delayed(Duration(seconds: 1)); yield 'Smith';

}

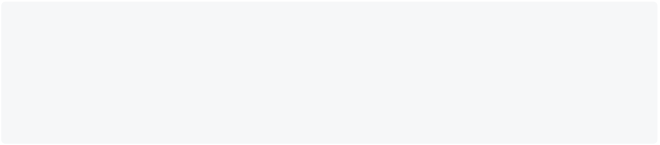
Info

**Note**: Here **yield** returns the value from the stream. To use **yield** you have to use

**async\***.

You can also create a stream by using **Stream.fromIterable()** method. Here the function will return **Stream<String>** immediately.

// function that returns a stream



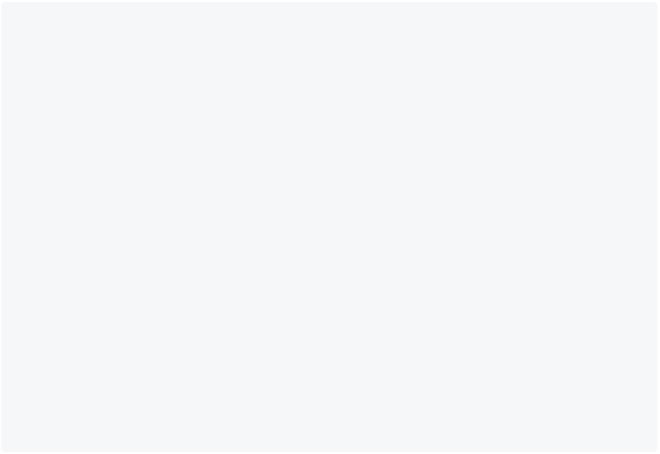
Stream<String> getUserName() {

return Stream.fromIterable(['Mark', 'John', 'Smith']); }

**How To Use Stream In Dart**

You can use stream in dart by using **await for** loop.

// function that returns a stream Stream<String> getUserName() async\* {



await Future.delayed(Duration(seconds: 1)); yield 'Mark';

await Future.delayed(Duration(seconds: 1)); yield 'John';

await Future.delayed(Duration(seconds: 1)); yield 'Smith';

}

// main function

void main() async {

// you can use await for loop to get the value from stream await for (String name in getUserName()) {

print(name);

}

}

**Future vs Stream**

* Future: Future represents the value or error that is supposed to be available in the Future.

Stream: Stream is a way by which we receive a sequence of events.

* + Future: A Future can provide only a single result over time.

Stream: Stream can provide zero or more values.

* + - Future: You can use FutureBuilder to view and interact with data.

Stream: You can use StreamBuilder to view and interact with data.

* + - * Future: It can’t listen to a variable change.

Stream: But Stream can listen to a variable change.

* + - * + Future: Syntax: Future <data\_type> class\_name Stream: Syntax: Stream <data\_type> class\_name



**Types Of Stream**

There are two types of streams:

1. Single Subscription streams
2. Broadcast streams

**Single Subscription Stream**

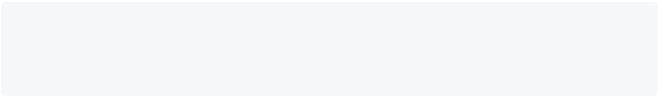
By default, Streams are set up for a single subscription. They hold onto the values until someone subscribes and can only be listened to once. You will get an exception if you try to listen more than once. Any event’s value should not be missed and must be in the correct order. Inside the stream controller, there is only one stream, and only one subscriber can use that stream.

**Broadcast Stream**

This is the stream that is set up for multiple subscriptions. They hold onto the values until subscribers can only listen many times. You can use the broadcast stream if you want more objects to listen to the stream. It can be used for mouse events in a browser. Inside the stream controller, many streams can be used by many subscribers. E.g., You can start watching videos on such a stream at any time, and more than one subscriber can watch the video simultaneously. Similarly, you can watch again after canceling a previous subscription.

**Syntax**

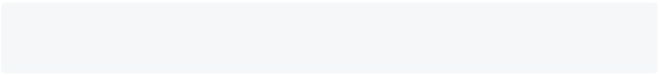
StreamController<data\_type> controller = StreamController<data\_type>.broadcast();



**How Streams Are Created**

You can create a stream in many ways. Let’s create a StreamController first.

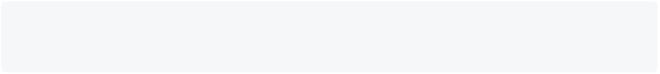
StreamController<data\_type> controller = StreamController<data\_type>();



Now we can access this controller through the stream property.



Stream stream = controller.stream;



**How To Subscribe A Stream**

After getting access from the stream you subscribe to the stream by calling a

listen() method.

stream.listen((value) {



print("Value from controller: $value"); });

**How To Add Value To The Stream**

We can add the stream by calling the add() method. Let’s add some value to the stream.



When we call the above function, we’ll get the output as: Value from controller: 3

**How To Manage The Stream**

To manage the stream, listen() method is used.

StreamSubscription<int> streamSubscription = stream.listen((value){ print("Value from controller: $value");

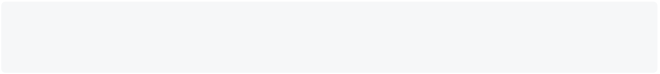


});

**How To Cancel A Stream**

You can cancel a stream by using the cancel() method.

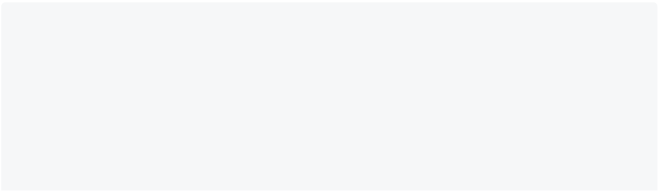
streamSubscription.cancel();



**Types Of Classes In Stream**

Four major classes in Dart’s async libraries are used to manage streams. **Stream:** It represents an asynchronous stream of data. For E.g:

final controller = StreamController<String>();



final subscription = controller.stream.listen((String data) { print(data);

});

controller.sink.add("Data!");



**EventSink:** It is like a stream that flows in the opposite direction.

**StreamController:** It simplifies stream management, automatically creating a stream and sink and also providing methods for controlling a stream’s behavior.

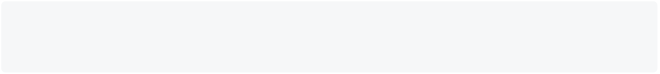
**StreamSubscription:** It saves the references of the subscription and allows them to pause, resume or cancel the flow of data they receive.

**Method Used In Stream**

There are four methods used in the stream: \*listen(): It returns a StreamSubscription object representing the active stream-producing events. The stream subscription allows you to pause, resume the subscription after a pause, and cancel the subscription completely.

**Syntax: listen**

final subscription = myStream.listen()



onError: Stream can provide errors just like a future can; by adding an onError method, you can catch and process an mistakes.



**Syntax: onError**

cancelOnError: This property or method is true by default but can be set to false to keep the subscription going even after an error.



**Syntax: cancelOnError**

onDone: This method can execute some code when the stream is finished sending data, such as when a file has been completely read.

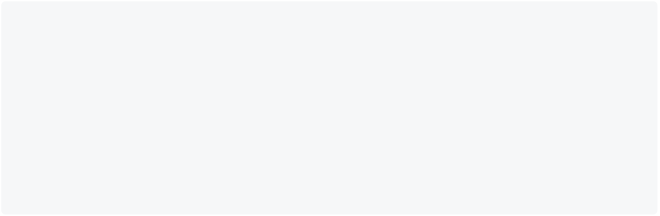


**Syntax: onDone Keywords Used In Stream**

* async\*: It is mainly used in the stream that works like the async in the future.
  + yield: It is used to emit values from a generator, either async or sync. yield returns values from an Iterable or a Stream.
* yield\*: yield\* is used to call its Iterable or Stream function recursively.

**Example Of async**

Future<int> doSomeLongTask() async {



await Future.delayed(const Duration(seconds: 2));

return 21;

}main() async {

int result = await doSomeLongTask();

print(result); // prints '42' after waiting 2 second }

**\_*Example Of async* In Dart\_\***

Stream<int> countForOneMinute() async\* {



for (int i = 1; i <= 5; i++) {

await Future.delayed(const Duration(seconds: 1)); yield i;

}

} main() async {

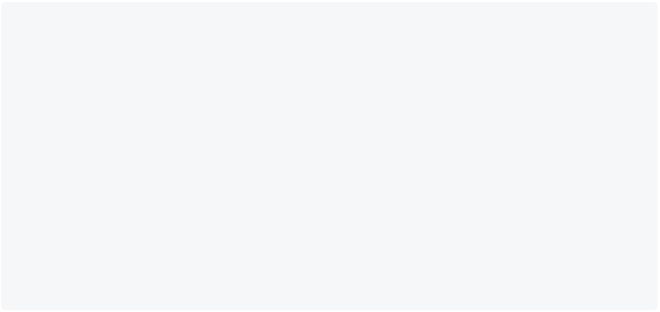
await for (int i in countForOneMinute()) {

print(i); // prints 1 to 5, one integer per second }

}

**\_*Example Of yield* In Dart\_\***

Stream<int> str(int n) async\* {



if (n > 0) {

await Future.delayed(Duration(seconds: 2)); yield n;

yield\* str(n - 2);

}

}

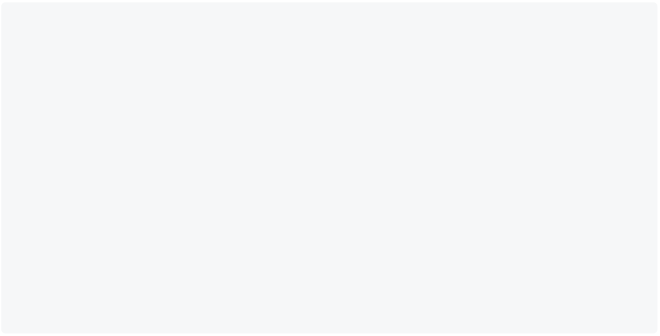
void main() {

str(10).forEach(print); }

In the above example, you have printed only an even number from 10 to 2 using stream. It will print the number after 2 sec.

**Some More Example of Stream Example 1**

import 'dart:async';



void main() {

var controller = StreamController(); controller.stream.listen((event) { print(event);

});

controller.add('Hello');

controller.add(42);

controller.addError('Error!');

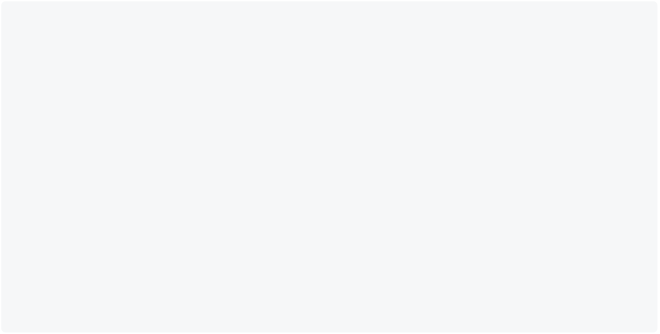
controller.close();

}

In this example, a String, integer and an error are added to the StreamController and then printed using the listen property.

**Example 2**

Stream<int> numberOfStream(int number) async\* { for (int i = 0; i <= number; i++) {



yield i;

}

}

void main(List<String> arguments) {

// Calling the Stream

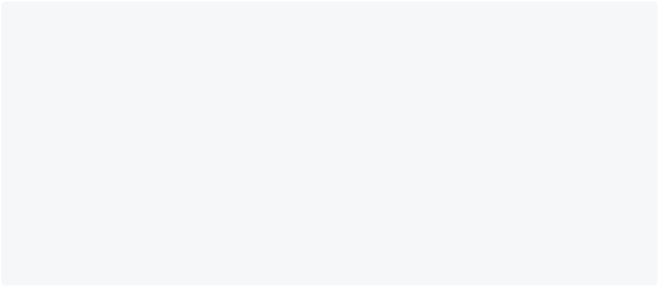
var stream = numberOfStream(6);

// Listening to Stream yielding each number stream.listen((s) => print(s));

}

In the above example, you must print the number from 0 to 6 using stream. **Example 3**

Stream<int> str(int n) async\* {



for (var i = 1; i <= n; i++) {

await Future.delayed(Duration(seconds: 1)); yield i;

}

}

void main() {

str(10).forEach(print); }

In the above example, you must print the number from 1 to 5 using stream. It will print the number after 1 sec.

**\_*async vs async* In Dart\_\***

* async: It gives a Future.
  + - * async\*: It gives a Stream.
  + async: async keyword does some work that might take a long time.
    - * + async*: async* returns a bunch of future values on at a time.
    - async: It gives the result wrapped in future.

async\*: It gives the result wrapped in the stream.

**\_*yield vs yield* In Dart\_\***

* yield: It is a keyword that returns single value to the sequence, but doesn’t stop the generator function.
  + yield\*: It is used for returning recursive generator.

To sum up, Streams are used in Dart to handle asynchronous data flows. They allow us to process data as it becomes available, rather than waiting for it to be fully loaded before processing.

Streams are commonly used in scenarios where data is being continuously updated or where we want to handle events as they occur. For example, we can use streams to monitor user interactions in real-time, or to receive data from a server as it becomes available.

In Dart, we can use the Stream and StreamController classes to create and manage streams. The StreamController class is used to create a stream and add data to it,

while the Stream class is used to listen to the stream and process incoming data.

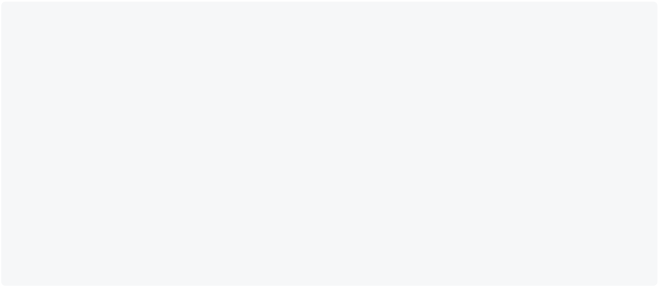
Ultimately, streams are a strong feature in Dart that let us handle asynchronous data flows in a flexible and effective way.

## Final Vs Const

If you do not want to change the value of a variable, then you can use either final or const in dart.

**Example**

void main() {



final finalName = "Final John Doe"; const constName = "Const John Doe";

finalName = "Raj"; // Not Possible constName = "Anu"; // Not Possible

print("Final name is " + finalName); print("Const name is " + constName); }

### Const In Dart

If you need to calculate value at compile-time, it is a good idea to choose const over final. A const variable is a compile-time constant. They must be created from data that can be calculated at compile time. 100+1 is valid const expression but const date = DateTime.now(); is not.



**What Is Compile Time**

When you run code in the dart, it will be compiled into the format that the machine can understand. This time is called compile time. Const value should be known at compile time.

**What Is Run Time**

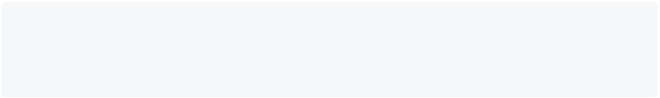
Runtime is the time when your compiled code is started running. It generally occurs after the compile time.

Info

Note: If you use const inside the class, declare it as static const.



const total = 50+50; // Possible



const date = DateTime.now(); // Not Possible

**Advantage Of Constant**

* Improve Performance

### Final In Dart

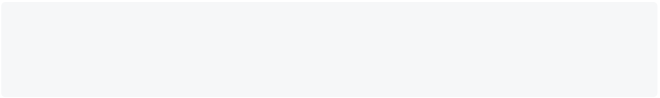
If the value is calculated at runtime, you can choose final for it. For. e.g if you want to calculate date on run time, you can use final date = DateTime.now(); but not

const date = DateTime.now();.

Note: Anything that is unknown at compile time should be final over const.



final date = DateTime.now(); // Possible const date = DateTime.now(); // Not Possible



**When To Use Const**

If you know the value at compile-time, choose const for e.g. const a = 100;.



**When To Use Final**

* If you don’t know the value at compile-time, choose final.



* + If you want a network request that can’t be changed, choose final.



* + - If you want to get some values from the database, choose final.



* + - * If you want to read a local file, choose final.



Note: Final variables will have a value known at runtime. Const variables have a value known at compile time. Instance variable can be final but not const.

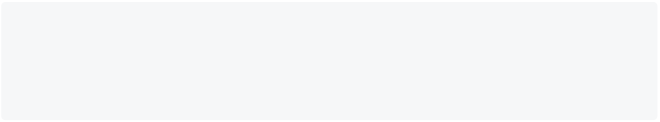
## Datetime In Dart

Date and time are often used in our day-to-day activities. As a programmer you need to know how to find a date and time? How to format date? and how to perform different calculation in date?

**How To Get Date And Time**

Use the following code to get the current date and time in the dart.

void main() {



print(DateTime.now()); }

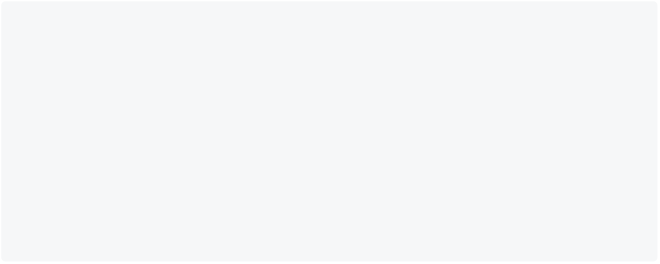
[Run Online](https://dartpad.dev/?id=a5c2708b64671bb0670a5e1717cc7b34)

**Get Year, Month, Day Of Datetime In Dart**

Here is the way to get a year, month, day, hour, minutes, and seconds in Dart. You can convert DateTime to String by using the toString() method.

**Example**

void main() {



DateTime datetime = DateTime.now();

print("Year is " + datetime.year.toString());

print("Month is " + datetime.month.toString());

print("Day is ${datetime.day}"); // If you don't want to use .toString print("Hour is " + datetime.hour.toString());

print("Minutes is " + datetime.minute.toString());

print("Second is " + datetime.second.toString());

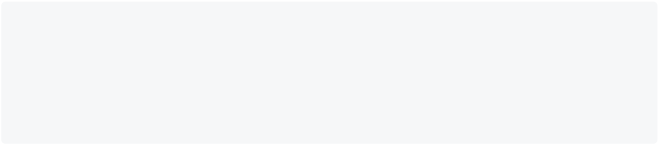
}

[Run Online](https://dartpad.dev/?id=2f3a51f3a9c912e5a9a93126339b7ea4)

**How To Convert Datetime To String In Dart**

Use the following code to convert DateTime to String in the dart.

void main() {



String datetime = DateTime.now().toString(); print(datetime);

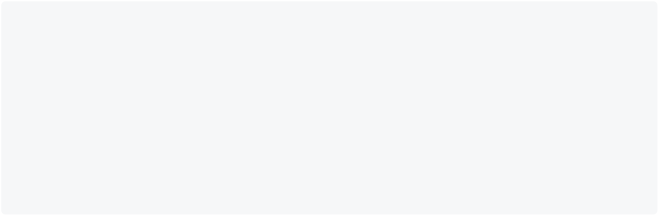
}

[Run Online](https://dartpad.dev/?id=a3a379b6011b2903a2af6cf32d091ca8)

**How To Convert String To DateTime**

You cannot get year, months, or day directly and cannot perform date calculation using a String if that String contains the correct DateTime value. In such a situation, you first need to convert String to DateTime.

void main() {



String myDateInString = "2022-05-01";

DateTime myConvertedDate = DateTime.parse(myDateInString); print("Year is " + myConvertedDate.year.toString());

print("Month is " + myConvertedDate.month.toString());

print("Day is " + myConvertedDate.day.toString());

}

[Run Online](https://dartpad.dev/?id=42b6a76b9a5b44b22ee20e0581dc33e2)

**Methods Supported By Datetime In Dart**

You can use DateTime methods if you want to add days, hours, or minutes to DateTime. Let us suppose you have created a DateTime object named mybirthday.

DateTime mybirthday = DateTime.parse("1997-05-14");

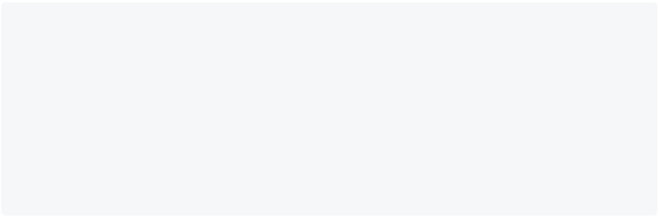
|  |  |
| --- | --- |
| **Method** | **Example** |
| add(Duration) | myBirthday.add(Duration(days: 1)); |
| subtract(Duration) | myBirthday.subtract(Duration(days: 1)); |

Note: You can set a duration to days, hours, minutes, seconds, milliseconds, and microseconds. To understand it more, look at the example below.



**Example: Add Date In Dart**

void main() {



DateTime myBirthday = DateTime.parse("1997-05-14"); myBirthday = myBirthday.add(Duration(days: 1));

print("Year is " + myBirthday.year.toString());

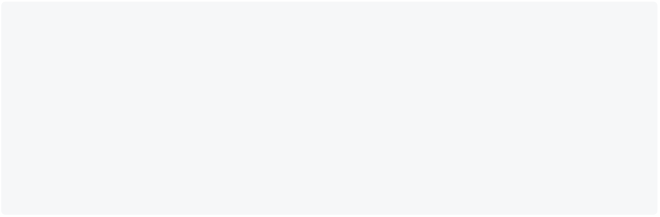
print("Month is " + myBirthday.month.toString()); print("Day is " + myBirthday.day.toString());

}

[Run Online](https://dartpad.dev/?id=c9a67812b6f4193184459b2f837c8299)

**Example: Subtract Date In Dart**

void main() {



DateTime myBirthday = DateTime.parse("1997-05-14"); myBirthday = myBirthday.subtract(Duration(days: 1)); print("Year is " + myBirthday.year.toString());

print("Month is " + myBirthday.month.toString());

print("Day is " + myBirthday.day.toString());

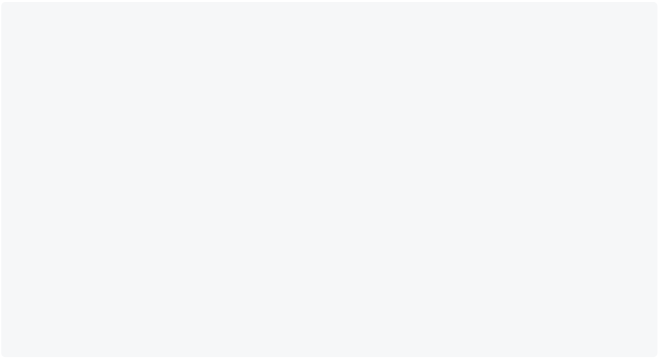
}

[Run Online](https://dartpad.dev/?id=a76f4d8fc70e13ad0cdb59c6abeaeabc)

**Find Difference Between Two Dates In Dart**

Suppose you want to find the difference between two dates in dart. There is a straightforward way.

void main() {



DateTime myBirthday = DateTime.parse("1997-05-14");

DateTime today = DateTime.now();

Duration diff = today.difference(myBirthday);

print("Difference in days: " + diff.inDays.toString());

print("Difference in hours: " + diff.inHours.toString());

print("Difference in minutes: " + diff.inMinutes.toString()); print("Difference in seconds: " + diff.inSeconds.toString()); print("Difference in milliseconds: " + diff.inMilliseconds.toString());

print("Difference in microseconds: " + diff.inMicroseconds.toString());

}

[Run Online](https://dartpad.dev/?id=4b2a6abe1f402387a400eb24dfa01f79)

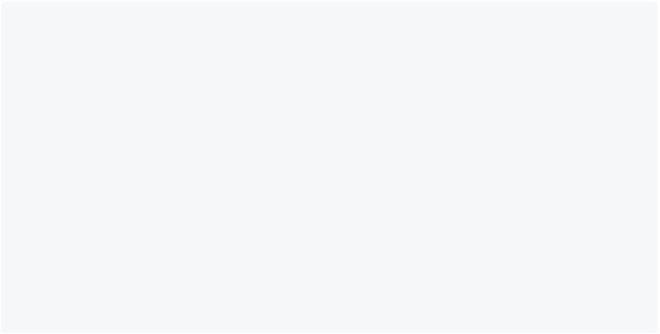
|  |  |
| --- | --- |
| **Name** | **Description** |
| inDays | Convert duration in days. |
| inHours | Convert duration in hours. |
| inMinutes | Convert duration in minutes. |
| inSeconds | Convert duration in seconds. |
| inMilliseconds | Convert duration in milli seconds. |
| **Name** | **Description** |
| inMicroseconds | Convert duration in micro seconds. |

**DateTime Comparision Methods**

If you want to compare two dates, then you can use comparison methods.

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| IsAfter(DateTime) | Returns true or false. bool |
| IsBefore(DateTime) | Returns true or false. bool |
| IsAtTheSameMoment(DateTime) | Returns true or false. bool |

void main() {



DateTime myBirthday = DateTime.parse("1997-05-14"); DateTime today = DateTime.now();

if (myBirthday.isBefore(today)) {

print("My Birthday is before today.");

} else if (myBirthday.isAfter(today)) {

print("My Birthday is after today.");

} else if (myBirthday.isAtSameMomentAs(today)) {

print("My Birthday date and today's date is same."); }

}

[Run Online](https://dartpad.dev/?id=17e19b3ca6ebf2bfc57139c20754c9ab)

# Flutter Tutorial

## Introduction and Setting up the Environment

Flutter is an open-source UI software development kit created by Google. It's used to develop applications for Android, iOS, Linux, Mac, Windows, Google Fuchsia, and the web from a single codebase.

Set up your development environment to work with Flutter and Dart. This will allow you to create and run Dart and Flutter projects on your computer.

By the end of this day, you should be able to create and run a new dart console project.

**Tips**

* Install the latest stable version of Dart and Flutter SDK.



* + - For macOS or Linux, you can download the Flutter SDK from the official website (<https://flutter.dev/docs/get-started/install>). Extract the compressed files and add the flutter tool to your path.
      * For Windows, you can download the Flutter SDK from the official website and run the installer. Add the flutter tool to your path.
  + Configure your IDE or code editor to work with Flutter and Dart.
    - * + If you’re using Android Studio or IntelliJ IDEA, install the Flutter and Dart plugins from the marketplace.

If you’re using Visual Studio Code, install the Flutter and Dart extensions from the marketplace.

If you’re using another IDE or code editor, check the official Flutter documentation for instructions on how to set it up. Once you have installed and configured everything, create a new Flutter project using the following command:

flutter doctor

This will should display the status of your Flutter installation and list any remaining dependencies that are required to complete the setup. If you’re missing any dependencies, follow the instructions provided by the command to install them.

📚 **Resources**

* [Official Flutter Installation Guide](https://flutter.dev/docs/get-started/install)
  + [Dart Tutorial Install Dart on Windows](https://dart-tutorial.com/introduction-and-basics/dart-install/)
    - [Flutter Doctor](https://flutter.dev/docs/reference/flutter-cli#flutter-doctor)
      * [Visual Studio Code Installation](https://code.visualstudio.com/docs/setup/setup-overview)
        + [Flutter and Dart plugins for Visual Studio Code](https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter)

[Installing Android Studio](https://developer.android.com/studio/install)

* + - * + [Installing Xcode](https://developer.apple.com/xcode/resources/)

## Creating a Sample App

**Step 1:** Create a New Flutter Project

1. Open VSCode and press ctrl + shift + p and select the option to create a new Flutter project.
2. Choose a location for your project and provide a name.
3. Wait for the Flutter extension to initialize the project.

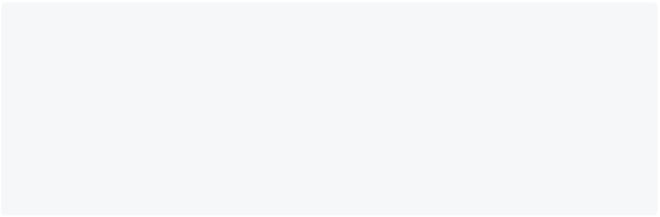
**Step 2:** Running the Flutter App

1. Connect a device or use an emulator.
2. Open the terminal in VSCode.
3. Navigate to the project directory.
4. Run the command: flutter run

This will build and launch your Flutter app on the connected device or emulator. Folder Structure

A Flutter project has the following structure:

my\_flutter\_app/ |-- android/



|-- ios/

|-- lib/

| |-- main.dart |-- test/

|-- pubspec.yaml

* **android/ and ios/:** These folders contain platform-specific configuration files for Android and iOS.
  + **lib/:** This is where you write most of your Dart code. The main.dart file is the entry point of your app.
    - **test/:** Unit tests for your Flutter app go here.
      * **pubspec.yaml:** This file defines the dependencies for your Flutter project and other configurations.

### Widgets in Flutter

In Flutter, everything is a widget. Widgets are the building blocks of a Flutter app, from simple elements like text and buttons to complex layouts. Widgets can be either stateful or stateless.

* **StatelessWidget:** Represents part of the user interface that doesn't change over time.

**StatefulWidget:** Represents part of the user interface that can change dynamically.



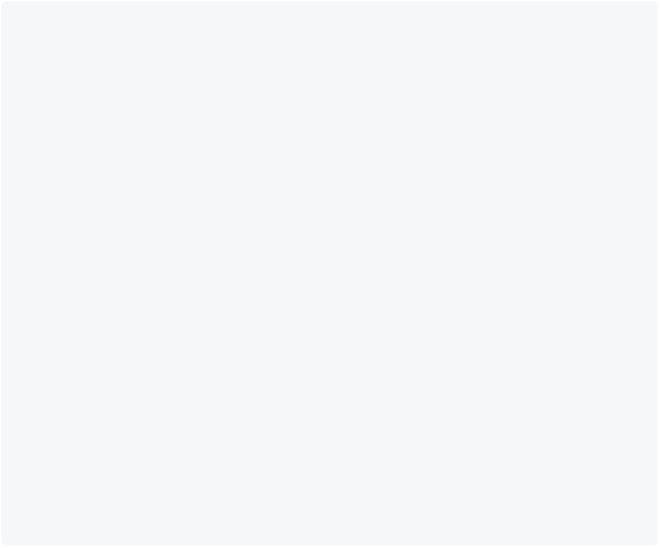
Widgets can be combined to create more complex UIs, and Flutter provides a rich set of pre-built widgets for various purposes.

### Understanding the Sample Code

Open lib/main.dart to see the sample code. A basic Flutter app consists of a

main() function that calls runApp() with the root widget of the application. **Code:**

import 'package:flutter/material.dart';



void main() {

runApp(MyApp()); }

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text('My Flutter App'), ),

body: Center(

child: Text('Hello, Flutter!'),

), ), );

}

}

* **MaterialApp:** A MaterialApp widget is a container for the entire application. It provides basic material design visual elements.
  + **Scaffold:** A Scaffold widget is a basic skeletal structure for a page.
    - **AppBar:** An AppBar widget is a material design app bar that typically contains a title and other widgets.
      * **Center:** A Center widget centers its child widget within its bounds.
* **Text:** A Text widget displays a string of text.
  + The **BuildContext** indicates where the build is taking place.
    - **Keys** are most useful in widgets that build many instances of the same type of widget.

## Types of Widgets in Flutter

In Flutter, everything is a widget. Widgets are the building blocks of a Flutter app, from simple elements like text and buttons to complex layouts. Widgets can be either stateful or stateless.

* **StatelessWidget:** Represents part of the user interface that doesn't change over time.
  + **StatefulWidget:** Represents part of the user interface that can change dynamically.

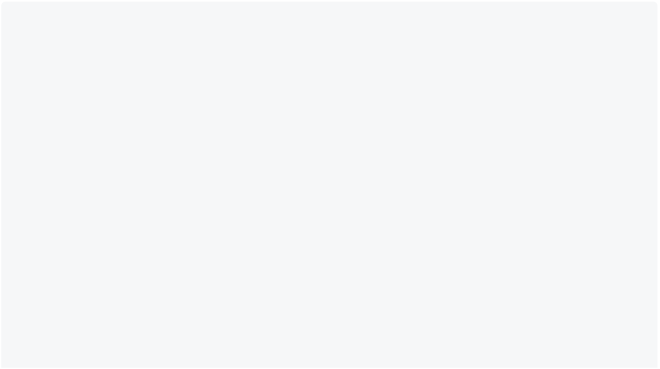
Widgets can be combined to create more complex UIs, and Flutter provides a rich set of pre-built widgets for various purposes.

### Stateless Widgets

A stateless widget is a widget that doesn't store any mutable state. Once created, its properties cannot change. Stateless widgets are used for parts of the user interface that don't change dynamically. They are essentially static and don't have internal state that changes over time.

Example of a stateless widget:

import 'package:flutter/material.dart';



class MyStatelessWidget extends StatelessWidget { final String title;

MyStatelessWidget({required this.title});

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text(title),

),

body: Center(

child: Text('This is a Stateless Widget'), ),



);

}

}

In this example, MyStatelessWidget is a stateless widget. It takes a title as a parameter, and once the widget is built, the title cannot be changed.



### Stateful Widgets

A stateful widget is a widget that can change its state during its lifetime. Stateful widgets are used for dynamic parts of the user interface that can be updated based on user interactions, data changes, etc. They have an associated mutable state object that can be modified.

**Lifecycle Methods**

In Flutter, a StatefulWidget has a lifecycle that consists of various methods that are called at different stages of its existence. Here are the primary lifecycle methods of a StatefulWidget:

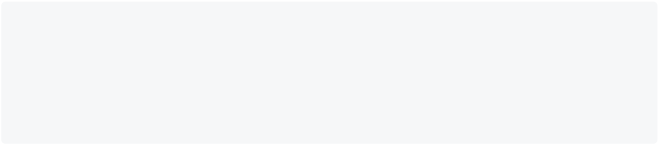
1. **createState (Factory constructor):**

* This method is called when the StatefulWidget is instantiated and needs to create its corresponding mutable state. It returns an instance of the associated State class.



Example:

class MyStatefulWidget extends StatefulWidget {



@override

\_MyStatefulWidgetState createState() => \_MyStatefulWidgetState(); }

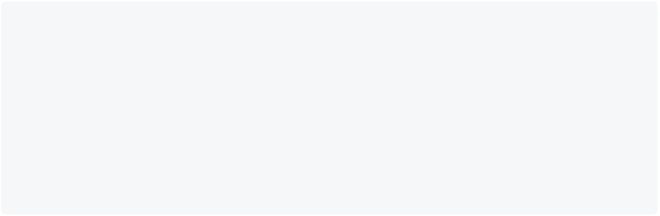
1. **initState:**

This method is called when the associated State object is created. It's typically used for one-time initialization tasks that need to be performed when the widget is inserted into the widget tree. It is called before the build method.



Example:

class \_MyStatefulWidgetState extends State<MyStatefulWidget> { @override



void initState() {

super.initState();

// Initialization tasks

}

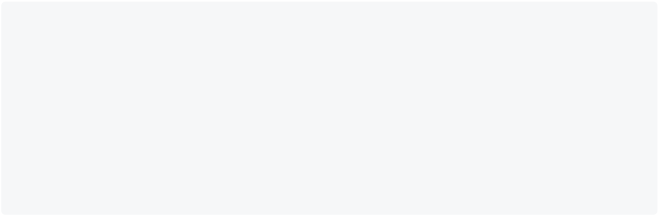
}

1. **didChangeDependencies:**

* This method is called whenever the widget's dependencies change. It's often used for tasks that rely on the widget's context or theme.

Example:

class \_MyStatefulWidgetState extends State<MyStatefulWidget> { @override



void didChangeDependencies() {

super.didChangeDependencies();

// Handle dependency changes

}

}

1. **build:**

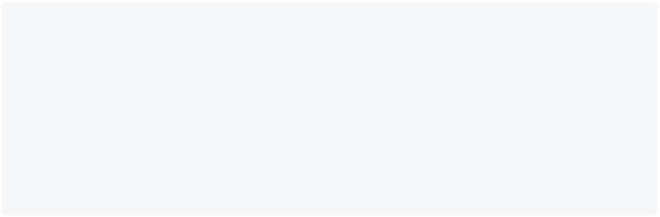


This is the required method that builds the widget tree for the StatefulWidget. It's called whenever the widget needs to be rebuilt, such as when it's inserted into the tree, its state changes, or when a parent widget rebuilds.



Example:

class \_MyStatefulWidgetState extends State<MyStatefulWidget> { @override



Widget build(BuildContext context) {

// Build the widget tree

return Container();

}

}

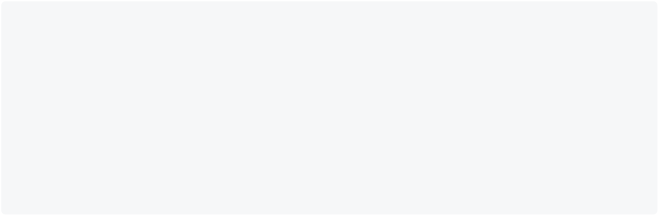
1. **didUpdateWidget:**

This method is called whenever the widget is updated, i.e., when its parent rebuilds and provides a new configuration.



Example:

class \_MyStatefulWidgetState extends State<MyStatefulWidget> { @override



void didUpdateWidget(MyStatefulWidget oldWidget) {

super.didUpdateWidget(oldWidget);

// Handle widget updates

}

}

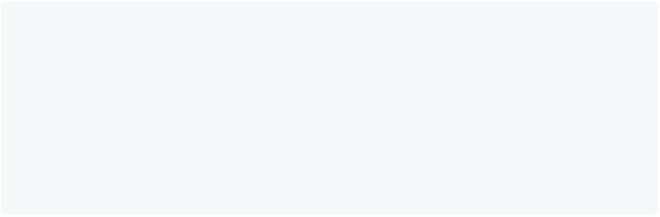
1. **dispose:**

This method is called when the widget is removed from the tree. It's used for cleanup tasks, such as closing streams or releasing resources.



Example:

class \_MyStatefulWidgetState extends State<MyStatefulWidget> { @override



void dispose() {

// Cleanup tasks

super.dispose();

}

}

These methods collectively form the lifecycle of a StatefulWidget. **Example of a stateful widget:**

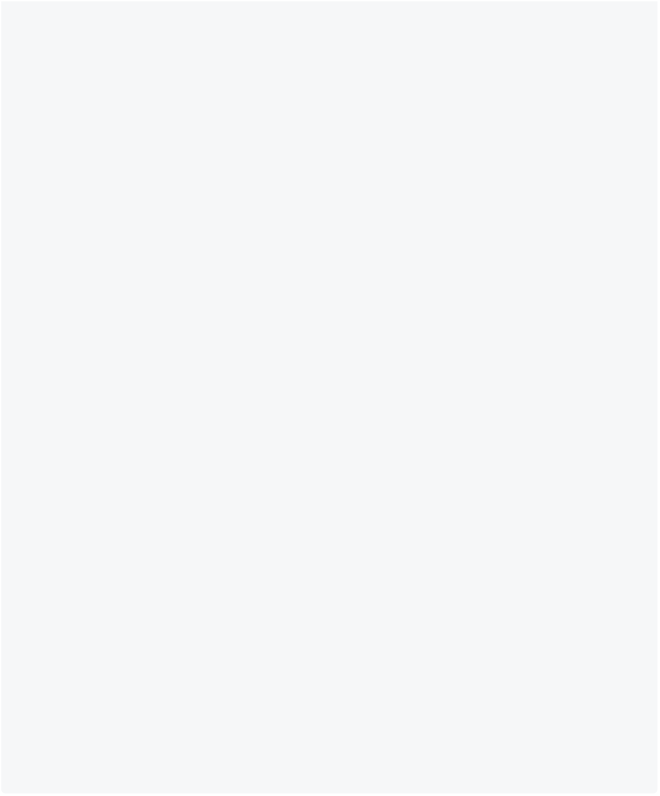
import 'package:flutter/material.dart';



class MyStatefulWidget extends StatefulWidget { final String title;

MyStatefulWidget({required this.title}); @override

State<MyStatefulWidget> createState() => \_MyStatefulWidgetState(); }



class \_MyStatefulWidgetState extends State<MyStatefulWidget> { int counter = 0;

void incrementCounter() { setState(() {

counter++;

});

}

@override

Widget build(BuildContext context) { return Scaffold(

appBar: AppBar(

title: Text(widget.title),

),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center, children: [

Text('Counter: $counter'),

ElevatedButton(

onPressed: incrementCounter,

child: Text('Increment'),

),

],

),

),

);

}

}

In this example, MyStatefulWidget is a stateful widget. It has an associated

\_MyStatefulWidgetState class that holds the mutable state (counter in this case). The incrementCounter method is used to update the state, and setState is called to trigger a rebuild of the widget.

You might wonder why StatefulWidgetand Stateare separate objects. In Flutter, these two types of objects have different life cycles. Widgetsare temporary objects, used to construct a presentation of the application in its current

state. Stateobjects, on the other hand, are persistent between calls to build(), allowing them to remember information.

**Key Points:**

* Stateless widgets are immutable and don't have mutable state.
  + Stateful widgets have mutable state and can be updated over time.
    - Stateful widgets have a corresponding state class that extends State.



* + - * setState is used to notify Flutter to rebuild the widget when the state

changes.

List of Widgets

**Note** Remember that to view all the properties of a widget press ctrl + space

bar in vs code after creating the widget.



Spacer: Search in GPT

SliverAppBar: Search in GPT

NestedScrollView: Search in GPT

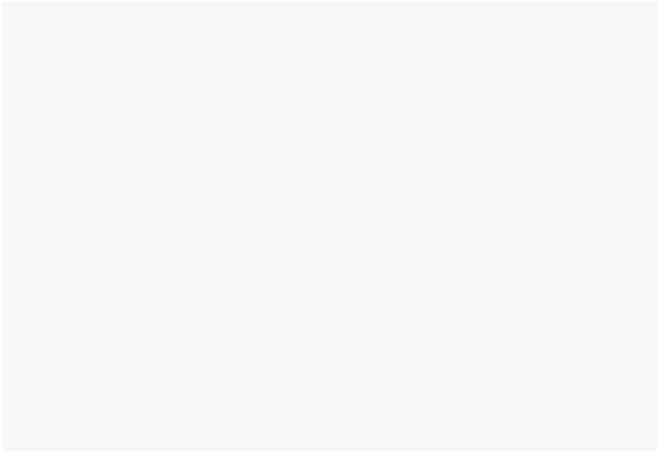
ClipRRect: Search in GPT

To make images rounded

**Container :**

A Container is a box model that can contain other widgets.

Container(



width: 200,

height: 100,

decoration: BoxDecoration(

color: Colors.blue, // Background color border: Border.all(

color: Colors.black, // Border color width: 2.0, // Border width

),

borderRadius: BorderRadius.circular(10.0), // Border radius boxShadow: [ BoxShadow(

color: Colors.grey, // Shadow color

blurRadius: 5.0, // Blur radius

offset: Offset(3, 3), // Shadow offset

),

],

),

child: Center(child: Text('Styled Container')), )

decoration Property:

1. **color**: Sets the background color of the container.



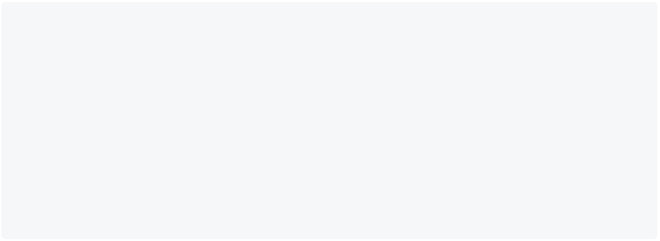
1. **border**: Defines the border properties, including color and width.
2. **borderRadius**: Specifies the radius of the container's corners for rounded edges.
3. **boxShadow**: Adds a shadow to the container, including color, blur radius, and

offset. **Alignment**

In Flutter, Alignment is used to position widgets within a container. It represents a point within a rectangle, and you can use it to specify where a widget should be placed.

Example:

Container(



width: 200,

height: 200,

color: Colors.blue,

child: Align(

alignment: Alignment(0.5, 0.5), child: Text('Aligned Text'),

),

)

In Flutter, the Alignment class is often used to represent a point within a rectangle. The Alignment class takes two parameters, x and y, both ranging from -1.0 to 1.0, where (0,0) represents the center of the rectangle.



Here's a breakdown of how the Alignment class works:

* x: Positive values move the point right, and negative values move it left.



* + y: Positive values move the point down, and negative values move it up.



For example:

* Alignment(0.0, 0.0): Center of the rectangle.
  + Alignment(1.0, 1.0): Bottom-right corner.
    - Alignment(-1.0, -1.0): Top-left corner.

In this example, Alignment(0.5, 0.5) centers the child widget within the parent container.

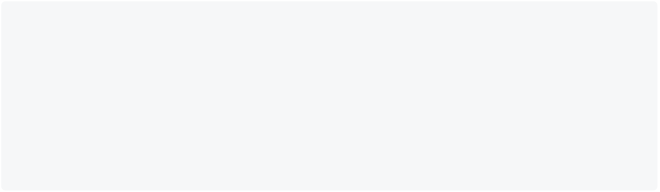
**Padding**

The Padding widget adds space around a child widget. It's useful for creating margins and providing visual separation between elements.



Example:

Container(



color: Colors.green,

child: Padding(

padding: EdgeInsets.all(16.0), child: Text('Padded Text'),

),

)

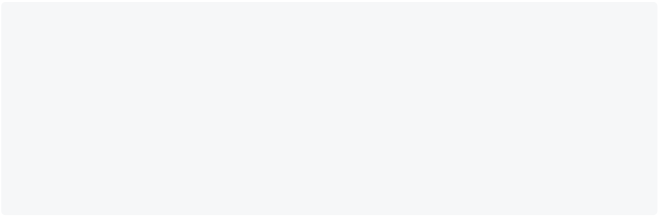
Here, EdgeInsets.all(16.0) adds 16 pixels of padding on all sides of the child widget.

**Opacity**

The Opacity widget is used to control the transparency of a child widget. It makes the child widget partially or fully transparent.

Example:

Container(



color: Colors.red,

child: Opacity(

opacity: 0.5,

child: Text('Transparent Text'), ),

)

In this case, opacity: 0.5 makes the child widget 50% transparent.

**Row :**



A Row widget arranges its children in a horizontal line.



Row(



mainAxisAlignment: MainAxisAlignment.spaceBetween, crossAxisAlignment: CrossAxisAlignment.center, mainAxisSize: MainAxisSize.min,

children: [

Text('Hello,'),



Text(' Flutter!'), ],

)

mainAxisAlignment: (In Horizontal direction from Left to Right)

1. **MainAxisAlignment.start**: Aligns children at the start of the main axis.
2. **MainAxisAlignment.center**: Aligns children at the center of the main axis.
3. **MainAxisAlignment.end**: Aligns children at the end of the main axis.
4. **MainAxisAlignment.spaceBetween**: Distributes children evenly along the main

axis, with space between them.

1. **MainAxisAlignment.spaceAround**: Distributes children evenly along the main axis with equal space on both ends.
2. **MainAxisAlignment.spaceEvenly**: Distributes children evenly along the main

axis with equal space around them.

crossAxisAlignment: (In Vertical direction from Top to Bottom)

1. **CrossAxisAlignment.start**: Aligns children at the start of the cross axis.
2. **CrossAxisAlignment.center**: Aligns children at the center of the cross axis.
3. **CrossAxisAlignment.end**: Aligns children at the end of the cross axis.
4. **CrossAxisAlignment.stretch**: Stretches children across the cross axis.

mainAxisSize:

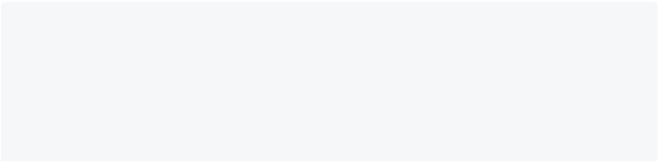
1. **MainAxisSize.min**: The column takes the minimum height necessary to contain its children.
2. **MainAxisSize.max**: The column takes the maximum height available.

**Column :**



A Column widget arranges its children in a vertical line. For Column Widget the mainAxis direction is from top to bottom and for crossAxis it is left to right.

Column(



mainAxisSize: MainAxisSize.min, mainAxisAlignment: MainAxisAlignment.spaceBetween, crossAxisAlignment: CrossAxisAlignment.center,

children: [

Text('Hello,'),

Text(' Flutter!'), ],



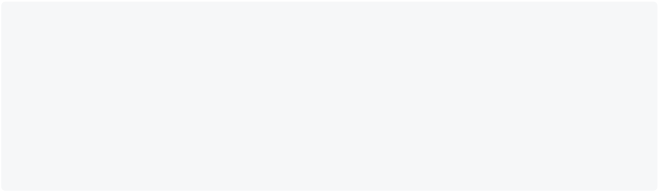
)

**ListView :**



A ListView is a scrollable list of widgets. Additionally, ListView.builder is a specialized constructor that efficiently builds lazy-loaded, scrollable lists.

ListView(



children: [

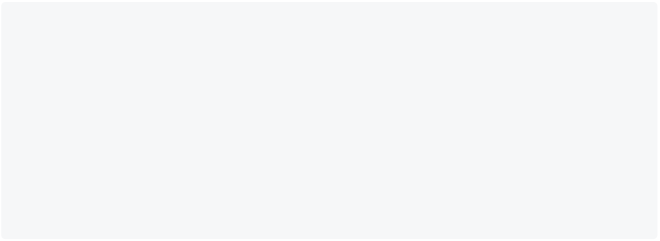
ListTile(title: Text('Item 1')), ListTile(title: Text('Item 2')), ListTile(title: Text('Item 3')), ],

)

**ListView.builder for Efficient List Building:**

When dealing with a large number of items in a list, ListView.builder becomes more efficient as it only creates widgets for the items that are currently in view. This is particularly useful for optimizing performance and memory usage.

ListView.builder(



itemCount: 1000, // Number of items in the list itemBuilder: (BuildContext context, int index) { return ListTile(

title: Text('Item $index'),

);

},

)

In the ListView.builder example:

* itemCount: Specifies the total number of items in the list.
  + itemBuilder: Defines a callback function that generates the widget for each

item at a given index.

The ListView.builder constructor is especially beneficial when dealing with large datasets, as it only renders the widgets that are currently visible on the screen, improving performance.

**ListTile :**



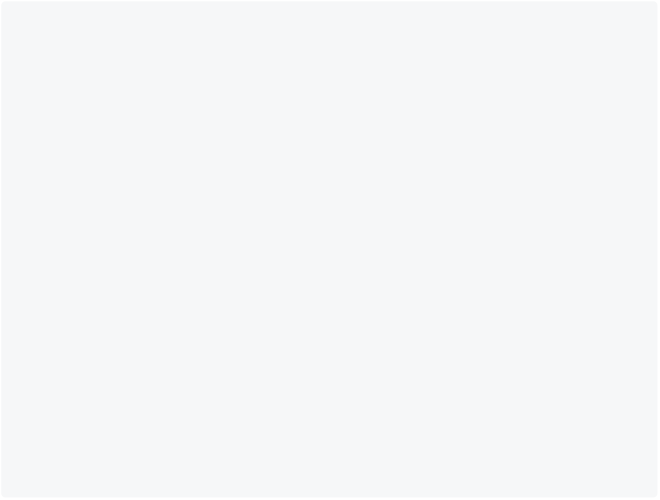
The ListTile widget in Flutter is a versatile component commonly used for creating individual rows in lists, menus, and other UI elements. It provides a consistent and customizable layout for displaying information within a fixed- height container.

**Basic ListTile Example:**



Here's a simple example of using ListTile within a ListView:

ListView(



children: [

ListTile(

leading: Icon(Icons.star),

title: Text('Star Item'),

subtitle: Text('This is a subtitle'), trailing: IconButton(

icon: Icon(Icons.favorite),

onPressed: () {

// Handle favorite button press

},

),

onTap: () {

// Handle tile tap

},

),

// Additional ListTiles can be added here ],

)

**Components of ListTile :**



1. **leading**: The widget to be displayed on the left side of the ListTile, often used for icons or avatars.
2. **title**: Displays the primary text content of the ListTile.



1. **subtitle**: Provides additional descriptive text below the title.
2. **trailing**: Positioned on the right side of the ListTile, commonly used for buttons or icons.
3. **onTap**: Defines the function to be executed when the ListTile is tapped. There are many more properties which you can look into yourself.



**GridView Widget in Flutter**

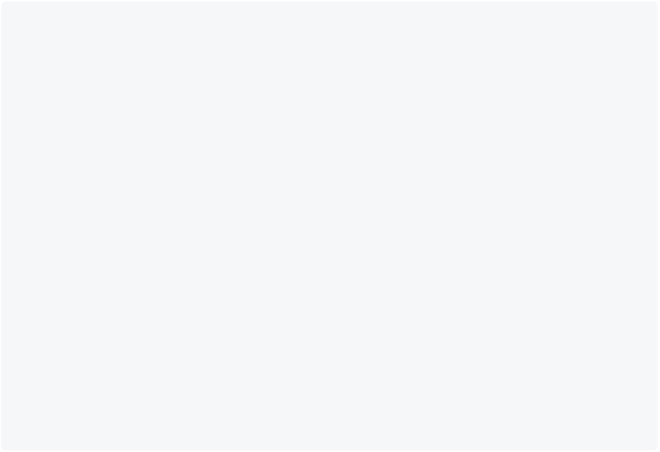
The GridView widget in Flutter is a powerful tool for creating scrollable, two- dimensional grids of widgets. It's commonly used to display collections of items in a grid layout, providing an organized and visually appealing user interface.

**Basic GridView Example:**



Here's a simple example of using the GridView widget:

GridView(



gridDelegate: SliverGridDelegateWithFixedCrossAxisCount( crossAxisCount: 2, // Number of columns in the grid

),

children: [

// Widgets to be displayed in the grid

Container(

color: Colors.blue,

child: Center(child: Text('Item 1')),

),

Container(

color: Colors.green,

child: Center(child: Text('Item 2')),

),

// Additional items can be added here

],

)

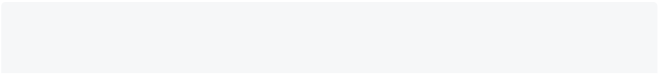
In this example:

* gridDelegate: Defines the layout of the grid, specifying the number of columns, spacing, and other aspects.
  + children: Contains the list of widgets to be displayed in the grid.

**SliverGridDelegate and SliverGridDelegateWithFixedCrossAxisCount :**

The SliverGridDelegate is responsible for defining the layout of the grid. In the example above, we use SliverGridDelegateWithFixedCrossAxisCount, which creates a grid with a fixed number of columns.

gridDelegate: SliverGridDelegateWithFixedCrossAxisCount( crossAxisCount: 2, // Number of columns in the grid

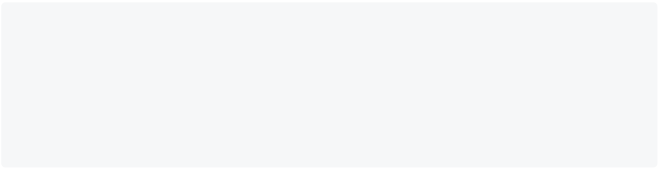


**Additional SliverGridDelegate Options:**

1. **crossAxisSpacing and mainAxisSpacing**: Define the spacing between grid

items.

gridDelegate: SliverGridDelegateWithFixedCrossAxisCount( crossAxisCount: 2,



crossAxisSpacing: 16.0, // Spacing between columns

mainAxisSpacing: 16.0, // Spacing between rows

),

1. **childAspectRatio**: Specifies the ratio of the cross-axis to the main-axis

extent of each grid item.

gridDelegate: SliverGridDelegateWithFixedCrossAxisCount( crossAxisCount: 2,



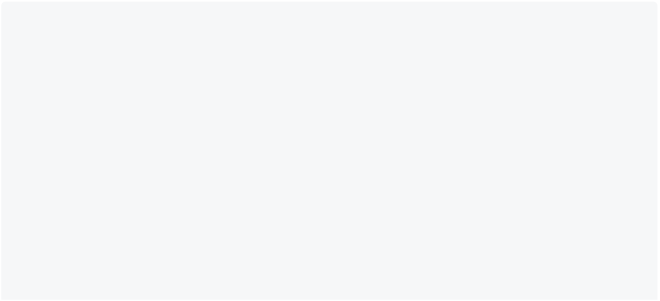
childAspectRatio: 0.75, // Width / Height ratio

),

**Using GridView.builder :**

The GridView.builder constructor is advantageous when dealing with a large number of items, as it only creates widgets for items that are currently in view, optimizing performance.

GridView.builder(



gridDelegate: SliverGridDelegateWithFixedCrossAxisCount( crossAxisCount: 2,

),

itemBuilder: (context, index) {

return Container(

color: Colors.blue,

child: Center(child: Text('Item $index')),

);

},

itemCount: 10, // Total number of items in the grid

In this example:

* itemBuilder: Defines a callback function that returns a widget for each item in the grid.
  + itemCount: Specifies the total number of items in the grid.

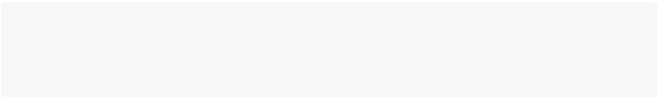
**Advanced GridView Features:**



1. **SliverGridDelegateWithMaxCrossAxisExtent :**

This delegate creates a grid with a maximum cross-axis extent for each item, allowing for more flexibility.

gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(

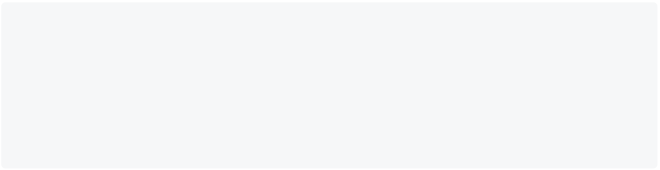


maxCrossAxisExtent: 200.0, // Maximum width/height of each item ),

1. **Scrollable GridView:**

Enclose the GridView widget within a SingleChildScrollView to make the entire grid scrollable.

SingleChildScrollView(



child: GridView(

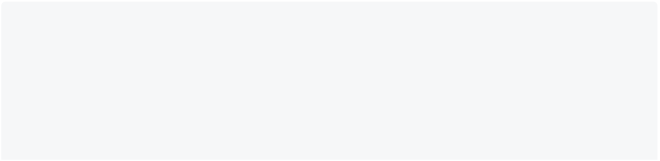
// ... other properties ),

)

1. **Infinite Scroll:**

Implementing infinite scroll can be achieved by listening to the onScroll event and loading more items as needed.

controller.addListener(() {



if (controller.position.pixels == controller.position.maxScrollExtent) {

// Load more items

}

**SingleChildScrollView :**

* SingleChildScrollView + Column = ListView
  + The SingleChildScrollView widget in Flutter is a versatile tool for making any single child widget scrollable. It's useful when:

1. **Limited Screen Space:**

* Your content exceeds the available screen size in one direction (vertically or horizontally).
  + You want to ensure users can access all content by scrolling.

1. **Shrink-wrapping:**

* You want the scrollable area to adjust its size based on its content's size.
  + This is common in dialogs,pop-up menus,or flexible layouts.

1. **Controlled Scrolling:**

* You need to control the scroll behavior (e.g.,enabling/disabling,snapping to positions).
  + Use its properties like scrollDirection, reverse,and controllerfor customization.

**Key Properties:**

* **child:**The single widget you want to make scrollable.
  + **scrollDirection:**Specifies the scrolling direction (horizontal or vertical).
    - **reverse:**If true,scrolling is reversed (opposite direction).
      * **padding:**Adds padding around the content within the scroll view.
        + **physics:**Defines the scroll behavior (e.g.,bouncing,no resistance).

**controller:**Allows controlling the scroll position programmatically.

**Common Use Cases:**

* Lists of items that overflow the screen height.
  + Long text content needing vertical scrolling.
    - Image viewers or carousels requiring horizontal scrolling.

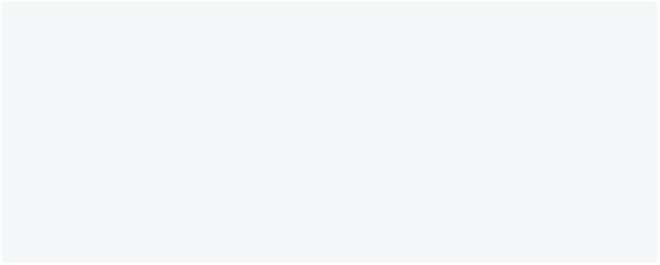
Complex layouts with dynamic content needing adaptable scrolling.



**Remember:**

* While SingleChildScrollViewis easy to use,consider ListViewfor longer lists with efficient rendering.
  + Be mindful of accessibility when implementing scrolling content.
    - Nested scrollable widgets require careful configuration to avoid conflicts.

SingleChildScrollView(



child: Column(

children: [

Text("Scrollable content 1"), Text("Scrollable content 2"), Text("Scrollable content 3"), ],

),

);

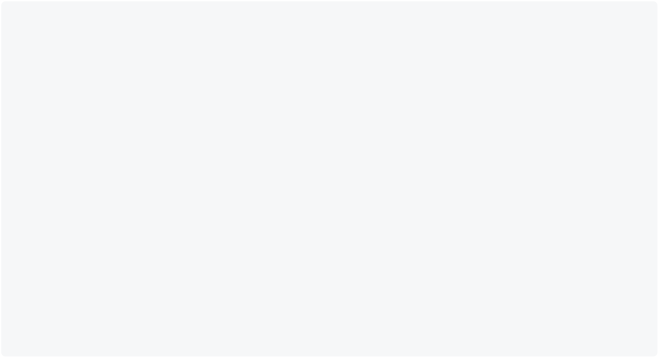
**Stack and Positioned :**



A Stack allows widgets to be overlaid on top of each other. Positioned widgets control the positioning of children within the Stack.



Stack(



children: [

Positioned(

left: 10,

top: 10,

child: Text('Positioned 1'), ),

Positioned(

right: 10,

bottom: 10,

child: Text('Positioned 2'), ),

],

)

**left and top**: Specify the offset of the child widget from the top-left corner



of the Stack.



* **right and bottom**: Specify the offset of the child widget from the bottom- right corner of the Stack.

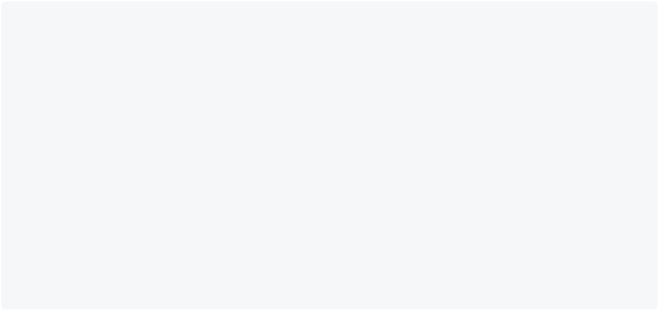


**AppBar :**



An AppBar is a material design app bar that typically contains the title and optional actions.

AppBar(



leading: Icon(Icons.settings),

title: Text('My App'),

actions: [

IconButton(

icon: Icon(Icons.settings),

onPressed: () {

// Handle settings button press

}, ), ],

)

**Properties:**

1. **leading**: Widget placed at the start of the app bar, typically an icon or button. In this example, an icon representing settings is used.



1. **title**: Displays the title of the app in the center of the app bar. Here, the title is set to 'My App'.



1. **actions**: A list of widgets (usually buttons or icons) placed at the end of the app bar. In this example, an IconButton with a settings icon is added, and you can handle its press event.

**Scaffold :**



A Scaffold is a top-level container that holds the structure of the visual interface.

Scaffold(



appBar: AppBar(

title: Text('My App'),

backgroundColor: Colors.green, // Customize app bar color

),

body: Center(

child: YourCustomWidget(), // Replace with your custom content

),



drawer: YourCustomDrawer(), // Replace with your custom drawer content bottomNavigationBar: YourCustomBottomNavigationBar(), // Replace with your custom bottom navigation bar content

floatingActionButton: YourCustomFloatingActionButton(), // Replace with your custom floating action button

)

**Properties:**

1. **appBar**: The AppBar widget contains the title of the app. In this example, the title is set to 'My App'.
2. **body**: The main content of the app is centered using the Center widget. You can replace the Text widget with any other widget or layout.



1. **drawer**: The drawer property is used for a side navigation drawer. In this case, an empty Container is provided. You can replace it with your custom drawer content.
2. **bottomNavigationBar**: The bottomNavigationBar property is used for a bottom navigation bar. An empty Container is added in this example. Replace it with your custom bottom navigation bar.
3. **floatingActionButton**: The floatingActionButton property is used for a floating action button. An empty Container is added in this example. Replace it with your custom floating action button.

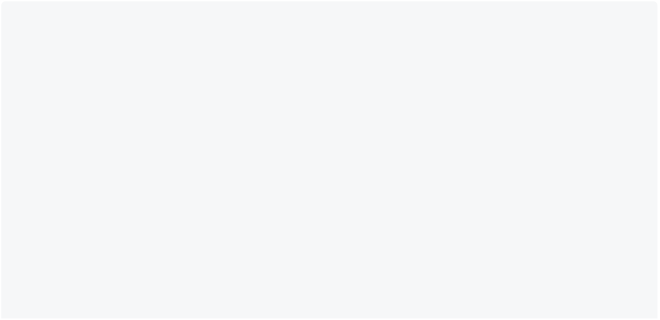
**Text :**



A Text widget displays a paragraph of text.



Text(



'Hello, Flutter!',

style: TextStyle(

fontSize: 20, // Font size

fontWeight: FontWeight.bold, // Font weight

fontStyle: FontStyle.italic, // Font style

color: Colors.blue, // Text color

letterSpacing: 1.5, // Letter spacing

wordSpacing: 2.0, // Word spacing

decoration: TextDecoration.underline, // Text decoration

decorationColor: Colors.red, // Decoration color

decorationStyle: TextDecorationStyle.dotted, // Decoration style

)

**TextStyle Properties:**



1. **fontSize**: Sets the size of the font.
2. **fontWeight**: Specifies the thickness of the font characters.
3. **fontStyle**: Defines the font style as normal, italic, or oblique.
4. **color**: Sets the color of the text.



1. **letterSpacing**: Adjusts the space between letters.
2. **wordSpacing**: Adjusts the space between words.
3. **decoration**: Adds a line decoration (underline, overline, or line through).
4. **decorationColor**: Sets the color of the text decoration.
5. **decorationStyle**: Specifies the style of the text decoration.

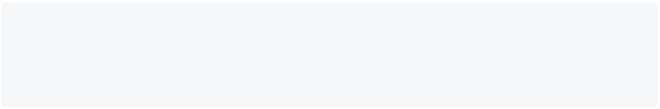
**Image :**



In Flutter, there are several ways to display images, each catering to different use cases.

* Adding Image Assets
  + Create a directory in your project’s root directory to store your image assets.
    - Add your image files (e.g. PNG, JPEG, or GIF files) to this directory.
      * In your app’s pubspec.yaml file, specify the location of the image assets directory and the images you want to use, like so:

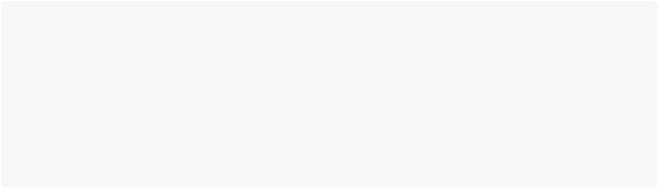
flutter:



assets:

* assets/images/
* Adding Custom Fonts
  + Create a directory in your project’s root directory to store your font files.
    - Add your font files (e.g. TrueType or OpenType font files) to this directory.
      * In your app’s pubspec.yaml file, specify the location of the font files directory and the fonts you want to use, like so:

flutter:



fonts:

* family: MyCustomFont

fonts:

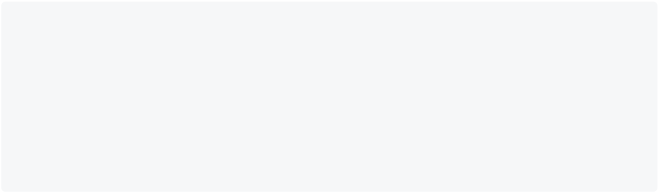
* asset: assets/fonts/my\_custom\_font.ttf

weight: 400

To use a custom font in your app, specify the font family and weight when creating a TextStyle, like so:



Text(



'Hello, World!',

style: TextStyle(

fontFamily: 'MyCustomFont', fontWeight: FontWeight.w400, ),

)

Here's a brief description of some common Image methods:

1. **Image.asset:**

The Image.asset widget is used to display images that are bundled with your app as assets.



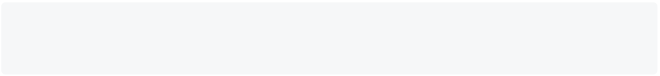
Image.asset('assets/your\_image.png')



1. **Image.network:**

* The Image.network widget is used to load and display images from a URL.

Image.network('https://example.com/your\_image.png')



1. **Image.file:**

The Image.file widget is used to display images from a local file system.



Image.file(File('path/to/your\_image.png'))



1. **Image.memory:**

* The Image.memory widget is used to display images from raw bytes in memory, such as from network responses or other sources.

Image.memory(Uint8List.fromList(yourRawImageData))

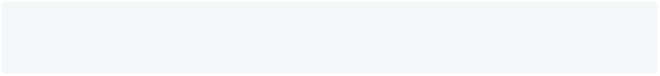


1. **Image constructor:**

The basic Image constructor can be used to create images from various sources, such as Image.file, Image.network, etc.



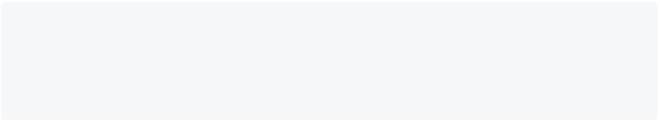
Image(image: AssetImage('assets/your\_image.png'))



1. **FadeInImage:**

* The FadeInImage widget is used to show a placeholder while loading an image from the network. It smoothly transitions from the placeholder to the actual image.

FadeInImage(



placeholder: AssetImage('assets/placeholder.png'),

image: NetworkImage('https://example.com/your\_image.png'), )

**Icon :**



An Icon widget displays a graphic symbol representing a command.



Icon(Icons.star)



**Different Types of Buttons:**

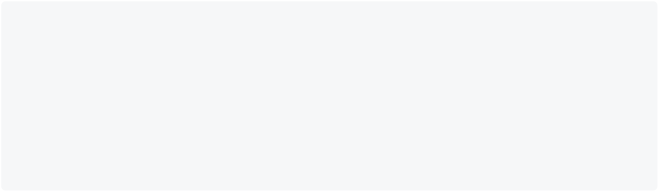
Flutter provides various button widgets, including ElevatedButton, TextButton, and OutlinedButtonetc.

1. **ElevatedButton:**

The ElevatedButton is a material design raised button. It has a shadow and is typically used for the primary action in an application.



ElevatedButton(



onPressed: () {

// Your button action

},

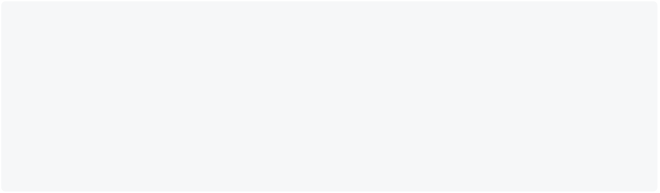
child: Text('Elevated Button'), )

1. **TextButton:**

The TextButton is a material design flat button. It's typically used for less prominent actions or in conjunction with other buttons.



TextButton(



onPressed: () {

// Your button action

},

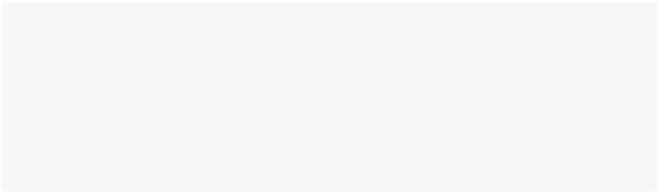
child: Text('Text Button'), )

1. **OutlinedButton:**

The OutlinedButton is a material design outlined button. It has a border and is used for actions that are less prominent than the primary action.



OutlinedButton(



onPressed: () {

// Your button action

},

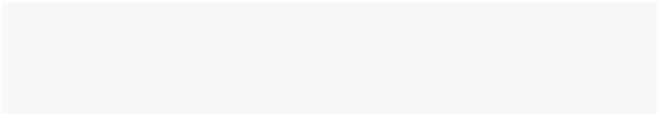
child: Text('Outlined Button'), )

1. **IconButton:**

The IconButton is a button that consists of an icon. It is commonly used for actions in app bars, dialogs, or other contexts where space is limited.



IconButton(



onPressed: () {

// Your button action },

icon: Icon(Icons.add), )

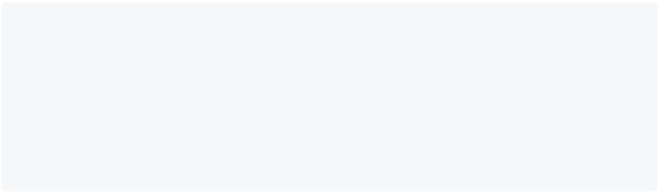


1. **FloatingActionButton:**

The FloatingActionButton is a circular button typically used for a promoted action. It's often placed in the bottom-right corner of the screen.



FloatingActionButton(



onPressed: () {

// Your button action },

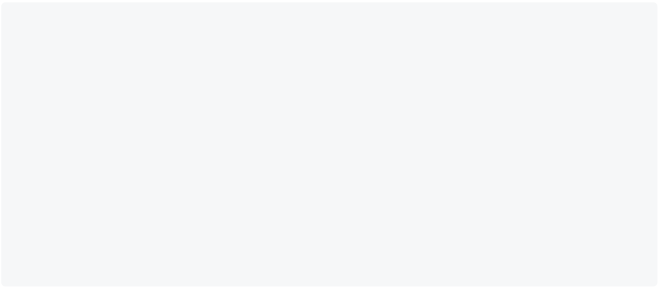
child: Icon(Icons.add), )

1. **DropdownButton:**

The DropdownButton is used to create a dropdown menu with a list of items. It allows users to select one option from a list.



DropdownButton<String>(



items: ['Option 1', 'Option 2', 'Option 3']

.map((String value) => DropdownMenuItem<String>( value: value,

child: Text(value),

))

.toList(),

onChanged: (String? newValue) {

// Handle dropdown selection

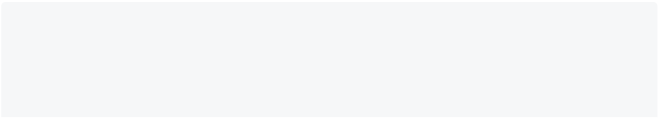
},

)

**TextField :**

A TextField widget allows the user to enter text.

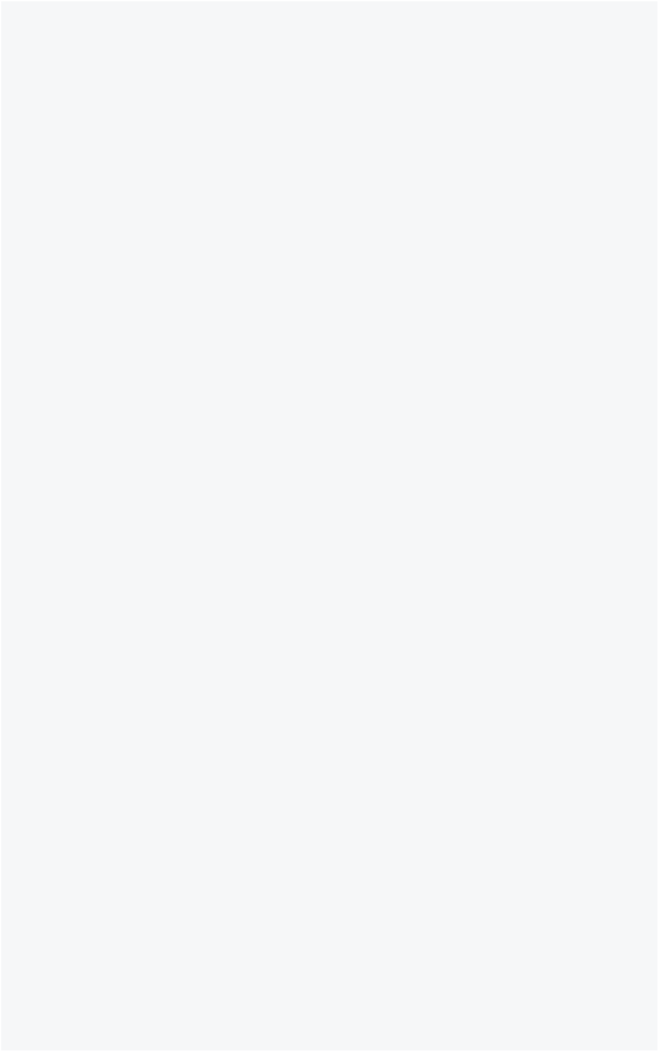
class MyTextFieldExample extends StatefulWidget {



@override

\_MyTextFieldExampleState createState() => \_MyTextFieldExampleState(); }

class \_MyTextFieldExampleState extends State<MyTextFieldExample> { TextEditingController \_textController = TextEditingController();



@override

Widget build(BuildContext context) {

return TextField(

controller: \_textController,

onChanged: (String value) {

// Handle text input changes

},

onSubmitted: (String value) {

// Handle when the user submits the text },

keyboardType: TextInputType.text,

decoration: InputDecoration(

labelText: 'Enter text',

hintText: 'Type something here',

prefixIcon: Icon(Icons.text\_fields),

suffixIcon: IconButton(

icon: Icon(Icons.clear),

onPressed: () {

// Clear the text when the clear icon is pressed \_textController.clear();

},

),

border: OutlineInputBorder(),

focusedBorder: OutlineInputBorder(

borderSide: BorderSide(color: Colors.blue),

),

errorText: \_validateText ? 'Please enter valid text' : null, ),

style: TextStyle(

fontSize: 16.0,

color: Colors.black,

),

maxLines: 1,

maxLength: 50,

cursorColor: Colors.blue,

textAlign: TextAlign.start,

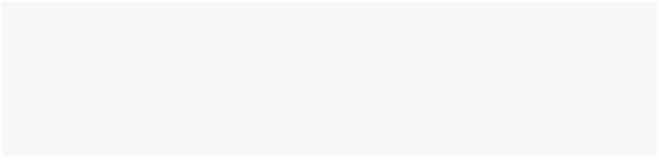
obscureText: false,

autocorrect: true,

autofocus: false,

enabled: true,

enableInteractiveSelection: true,



textCapitalization: TextCapitalization.sentences, );

}

}

**Properties:**

1. **controller:**

A TextEditingController object that allows you to control the text being displayed and entered in the TextField. It provides methods like clear(), text, etc.



1. **onSubmitted:**

A callback function that is called when the user submits the text (e.g., pressing the enter key on the keyboard).



1. **keyboardType:**

Specifies the type of keyboard to display (e.g., TextInputType.text, TextInputType.emailAddress, etc.).



1. **maxLines:**

* The maximum number of lines to display for a multiline TextField.

1. **maxLength:**

The maximum number of characters allowed in the TextField.



1. **obscureText:**

Set to true if the text should be obscured (e.g., for password input).



1. **autocorrect:**

Set to true to enable autocorrection of the entered text.



1. **textAlign:**

Specifies the horizontal alignment of the text (e.g., TextAlign.start, TextAlign.center, etc.).



1. **enabled:**

Set to false to disable the TextField.



1. **errorText:**

Displays an error message below the TextField when non-null.



1. **autofocus:**

Set to true to automatically focus the TextField when the widget is built.



1. **textCapitalization:**

Specifies how the text should be capitalized (e.g., TextCapitalization.sentences, TextCapitalization.words, etc.).



**TextFormField**

The TextFormField widget is a specialized version of TextField that integrates with the Form widget.

**Properties:**

* **controller:** A controller for an editable text field.
  + **decoration:** An InputDecoration object that configures the appearance of the text field.
    - **keyboardType:** The type of keyboard to use for editing the text.
      * **validator:** A callback that validates the input.

Example:

TextFormField(



controller: TextEditingController(),

decoration: InputDecoration(labelText: 'Enter your email'), keyboardType: TextInputType.emailAddress,

validator: (value) {

if (value.isEmpty) {

return 'Please enter your email';

}

return null;

},

)

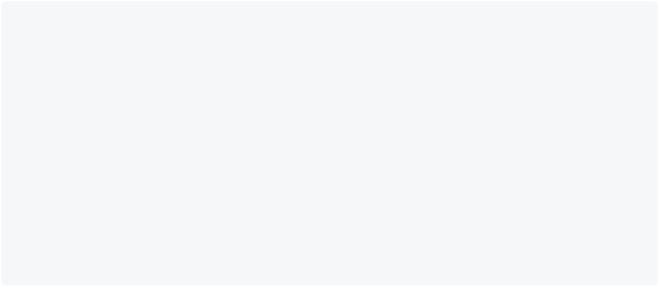
**Checkbox**

The Checkbox widget allows users to toggle between two states. **Properties:**

* **value:** The current state of the checkbox.
  + **onChanged:** Called when the user toggles the checkbox.

Example:

bool isChecked = false;



Checkbox(

value: isChecked,

onChanged: (value) {

// Handle checkbox state change setState(() {

isChecked = value;

});

},

)

**Radio**

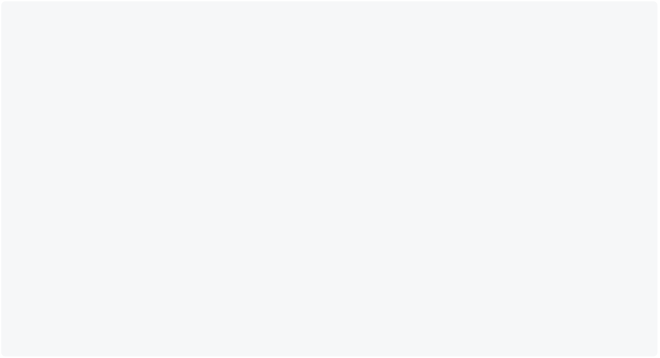
The Radio widget allows users to select a single option from a group. **Properties:**



* **value:** The current value of the radio button.
  + **groupValue:** The selected value of the entire radio group.
    - **onChanged:** Called when the user selects the radio button.

Example:

enum Gender { male, female }



Gender selectedGender = Gender.male;

Radio(

value: Gender.male,

groupValue: selectedGender,

onChanged: (value) {

// Handle radio button selection setState(() {

selectedGender = value;

});

},

)

**DropdownButton**

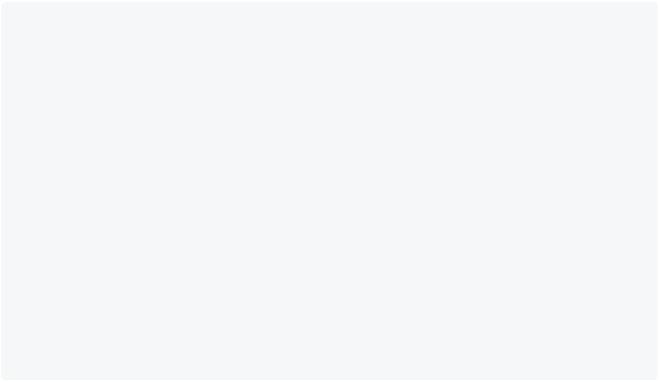
The DropdownButton widget displays a dropdown menu with a list of items.

**Properties:**

* **items:** The list of dropdown menu items.
  + **value:** The current selected value.
    - **onChanged:** Called when the user selects an item.

Example:

DropdownButton<String>(



items: ['Option 1', 'Option 2', 'Option 3']

.map((String value) => DropdownMenuItem<String>( value: value,

child: Text(value),

))

.toList(),

value: selectedValue,

onChanged: (value) {

// Handle dropdown selection

setState(() {

selectedValue = value;

});

},

)

**Switch**

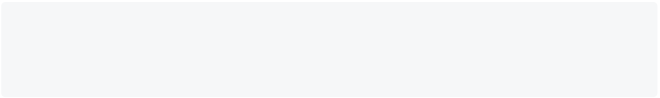
The Switch widget allows users to toggle between two states, similar to a checkbox.

**Properties:**

* **value:** The current state of the switch.
  + **onChanged:** Called when the user toggles the switch.

Example:

bool isSwitched = false; Switch( value: isSwitched, onChanged: (value) { // Handle switch state change



setState(() { isSwitched = value; }); }, )

**Slider**

The Slider widget allows users to select a value from a range by sliding a thumb along a track.



**Properties:**

* **value:** The current value selected on the slider.
  + **onChanged:** Called when the user drags the slider thumb.
    - **min:** The minimum value of the slider.
      * **max:** The maximum value of the slider.

Example:

double sliderValue = 50.0;



Slider(

value: sliderValue,

onChanged: (value) {

// Handle slider value change setState(() {

sliderValue = value;

});

},

min: 0,

max: 100,

)

**Date Picker**

The showDatePicker function displays a date picker dialog. **Properties:**

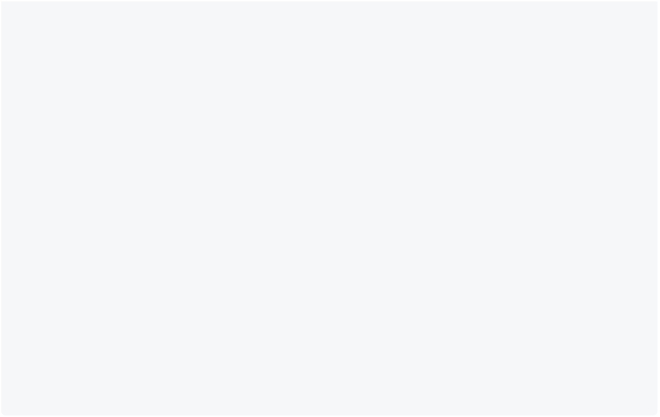
* **context:** The build context.
  + **initialDate:** The initial selected date.
    - **firstDate:** The earliest selectable date.
      * **lastDate:** The latest selectable date.

Example:

DateTime selectedDate = DateTime.now(); ElevatedButton(



onPressed: () async {



final DateTime pickedDate = await showDatePicker( context: context,

initialDate: selectedDate,

firstDate: DateTime(2000),

lastDate: DateTime(2101),

);

if (pickedDate != null && pickedDate != selectedDate) { setState(() {

selectedDate = pickedDate;

});

}

},

child: Text('Select Date'),

)

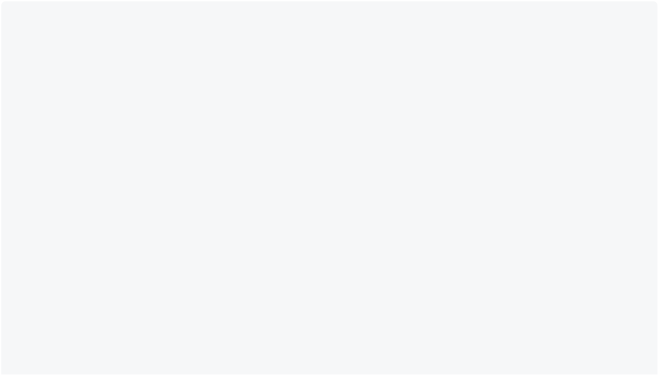
**Time Picker**

The showTimePicker function displays a time picker dialog. **Properties:**

* **context:** The build context.
  + **initialTime:** The initial selected time.

Example:

TimeOfDay selectedTime = TimeOfDay.now();



ElevatedButton(

onPressed: () async {

final TimeOfDay pickedTime = await showTimePicker( context: context,

initialTime: selectedTime,

);

if (pickedTime != null && pickedTime != selectedTime) { setState(() {

selectedTime = pickedTime;

});

}

},

child: Text('Select Time'), )



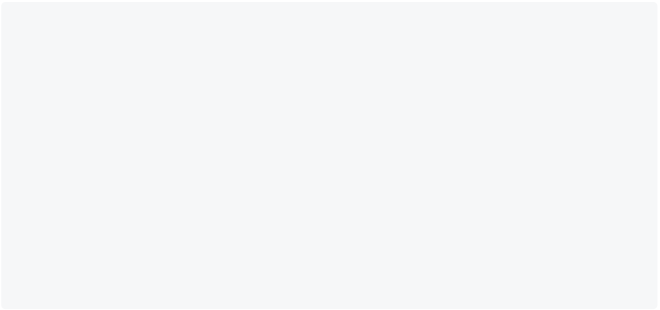
**Autocomplete**

The Autocomplete widget provides suggestions as users type. **Properties:**

* **optionsBuilder:** A callback that returns a list of suggestions.
  + **onSelected:** Called when a suggestion is selected.

Example:

Autocomplete<String>(



optionsBuilder: (TextEditingValue textEditingValue) {

return ['Apple', 'Banana', 'Cherry', 'Date', 'Fig']

.where((String option) =>

option.contains(textEditingValue.text.toLowerCase())) .toList();

},

onSelected: (String selected) {

// Handle the selected option

},

)

**Stepper**

The Stepper widget displays a sequence of steps for the user to progress through. **Properties:**

* **steps:** A list of Step objects.
  + **currentStep:** The index of the current step.

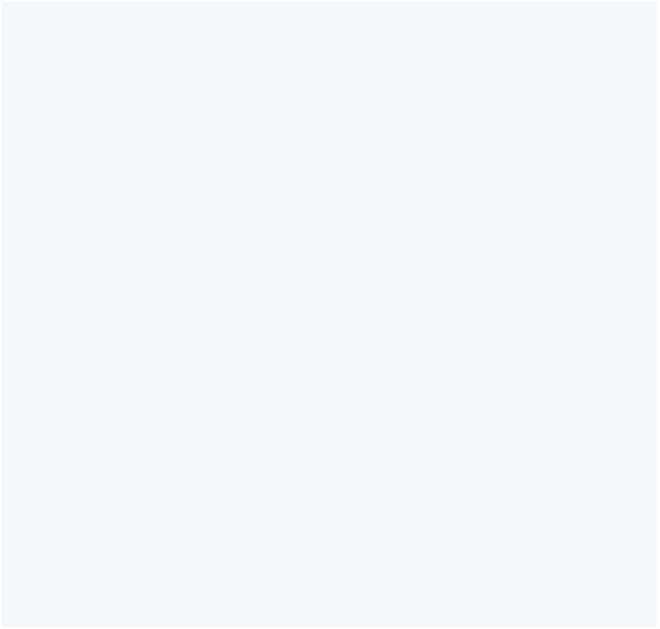
Example:

int currentStep = 0;



Stepper( steps: [ Step(

title: Text('Step 1'),



content: Text('Description for Step 1'), ),

Step(

title: Text('Step 2'),

content: Text('Description for Step 2'), ),

Step(

title: Text('Step 3'),

content: Text('Description for Step 3'),

),

],

currentStep: currentStep,

onStepContinue: () {

// Handle continue button pressed

setState(() {

currentStep < 2 ? currentStep += 1 : null;

});

},

onStepCancel: () {

// Handle cancel button pressed

setState(() {

currentStep > 0 ? currentStep -= 1 : null; });

},

)

**Form :**

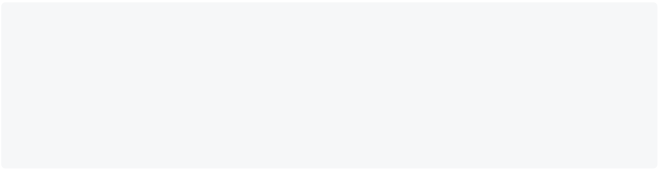


Forms are essential elements for collecting user input in Flutter applications. Here's an overview of building forms in Flutter:

**Creating a Form:**

* The Formwidget acts as a container for your form fields.
  + Use a GlobalKeyto uniquely identify the form and enable validation later.

final \_formKey = GlobalKey<FormState>();



Form(

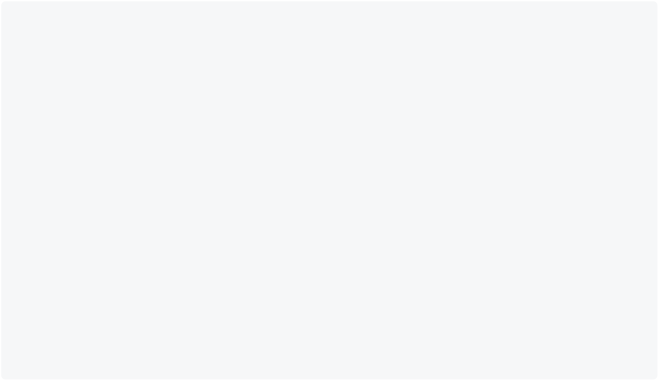
key: \_formKey,

// ...your form fields here )

**Adding Form Fields:**

* Wrap each input field with a FormFieldwidget to manage its state and validation.
  + Use different field types like TextFormFieldfor text input, Checkboxfor boolean choices,etc.

TextFormField(



validator: (value) => value!.isEmpty ? 'Field cannot be empty' : null, decoration: InputDecoration(

labelText: 'Name',

),

),

Checkbox(

value: \_isChecked,

onChanged: (bool? value) { setState(() {

\_isChecked = value!; });

},

),

**Validation:**

* You can define validation logic for each field using the validatorproperty of the FormField.
  + Use the Form.validate()method to check if all fields are valid before submitting the form.

if (\_formKey.currentState!.validate()) { // Submit the form

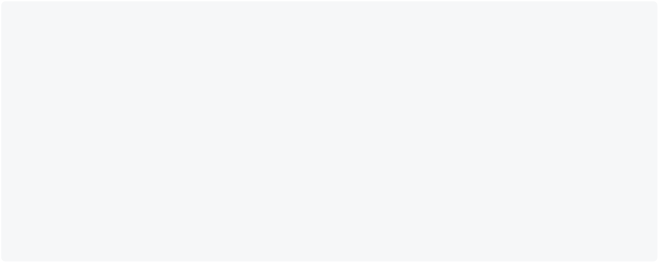


}

**Submitting the Form:**

* Use a button or similar widget to trigger form submission.
  + In the button's onPressed handler,access the form state using \_formKey.currentStateand handle form data.

ElevatedButton(



onPressed: () {

if (\_formKey.currentState!.validate()) {

// Get data from form fields

final name = \_formKey.currentState!.fields['name']!.value!; // Process or send data

}

},

child: Text('Submit'),

),

**Additional Tips:**

* Handle focus and errors for a better user experience.
  + Consider pre-filling fields with existing data.
    - Explore packages like form\_validatoror bloc\_formfor advanced validation and form management.

Remember, this is a basic overview. Flutter's built-in widgets and packages offer extensive options to customize and enhance your forms.

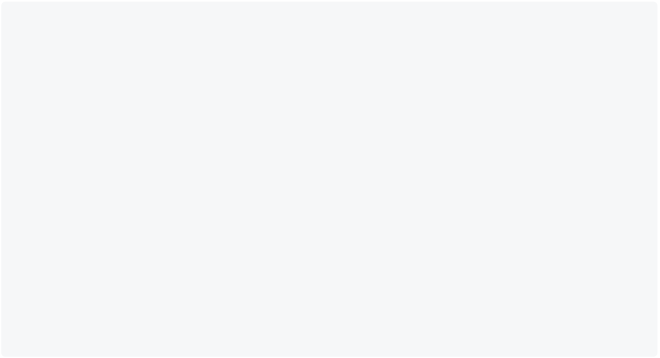
**Card :**



A Card widget is a material design card.



Card(



elevation: 4.0,

margin: EdgeInsets.all(16.0),

color: Colors.white,

shape: RoundedRectangleBorder(

borderRadius: BorderRadius.circular(10.0), side: BorderSide(

color: Colors.blue,

width: 2.0,

),

),

child: Container()

)

**Properties Explained:**

1. **elevation:**

The elevation property defines the shadow of the Card to give it a lifted appearance. Higher values create a more pronounced shadow.



1. **margin:**



The margin property sets the margin around the Card, controlling the spacing between the Card and surrounding widgets.



1. **color:**



The color property sets the background color of the Card.



1. **shape:**



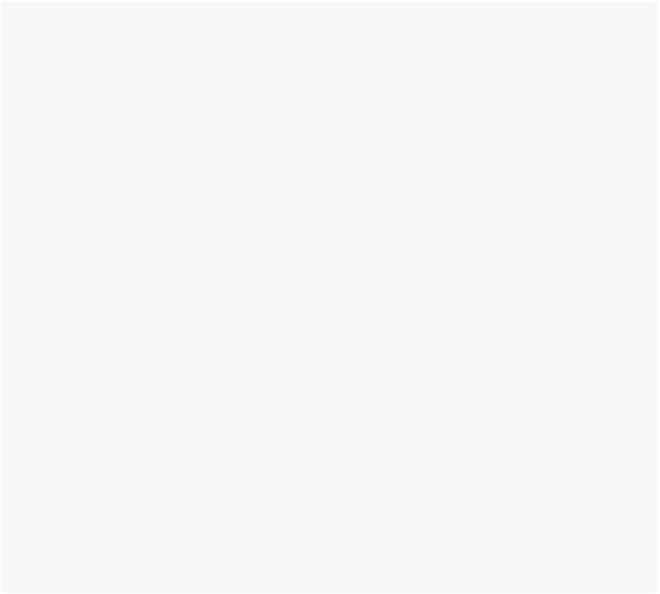
* The shape property allows you to define the shape of the Card. In this example, it is set to a rounded rectangle with a circular border and a blue side border.



**AlertDialog :**

An AlertDialog displays an alert dialog to the user.

ElevatedButton(



onPressed: () {

showDialog(

context: context, builder

* (BuildContext context) {

return AlertDialog(

title: Text('Alert Dialog'),

content: Text('This is an alert message.'),

actions: [

TextButton(

onPressed: () {

Navigator.of(context).pop(); // Close the dialog

},

child: Text('OK'), ),

],

);

},

);

},

child: Text('Show Alert'),

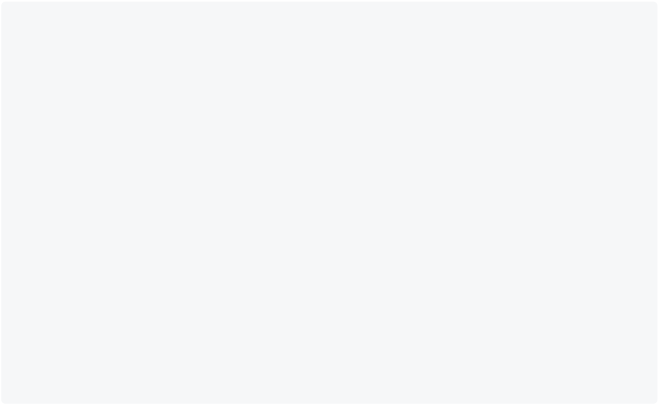
)

**BottomSheet :**



A BottomSheet displays a sheet from the bottom of the screen.

ElevatedButton(



onPressed: () {

showModalBottomSheet(

context: context,

builder: (BuildContext context) {

return Container(

height: 200,

child: Center(

child: Text('Bottom Sheet Content'), ),

);

},

);

},

child: Text('Show Bottom Sheet'),

)

**Drawer :**



A Drawer widget creates a material design drawer.

Scaffold(



appBar: AppBar(

title: Text('My App'),

),

drawer: Drawer(

child: ListView(

padding: EdgeInsets.zero,

children: [

DrawerHeader(

child: Text('Drawer Header'), decoration: BoxDecoration(

color: Colors.blue,

),

),

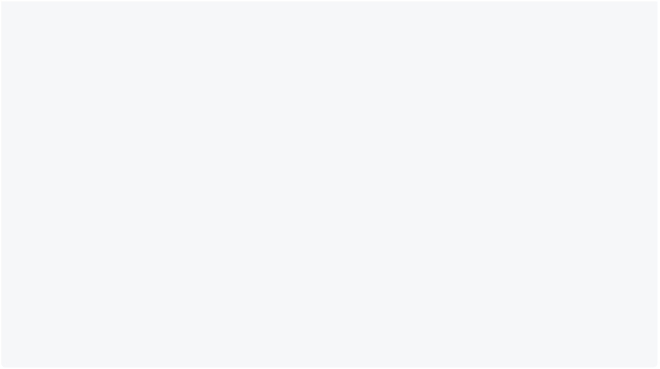
ListTile(

title: Text('Item 1'),

onTap: () {

// Handle item 1 tap

},



),

ListTile(

title: Text('Item 2'),

onTap: () {

// Handle item 2 tap

},

),

],

),

),

body: Center(

child: Text('Hello, Flutter!'), ),

)

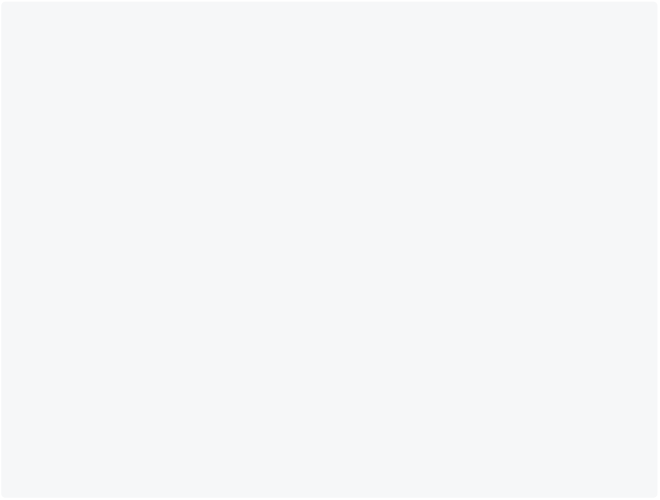
**TabBar and TabView :**



A TabBar displays a horizontal row of tabs, and TabView displays the corresponding tab views.



DefaultTabController(



length: 2,

child: Scaffold(

appBar: AppBar(

title: Text('Tabs Example'),

bottom: TabBar(

tabs: [

Tab(icon: Icon(Icons.tab)), Tab(icon: Icon(Icons.tab)), ],

),

),

body: TabBarView(

children: [

Center(child: Text('Tab 1')), Center(child: Text('Tab 2')),

], ), ),

)

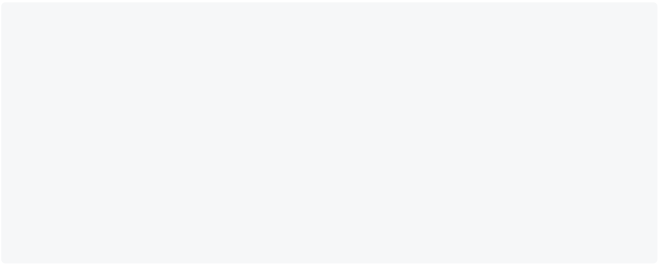
**Expanded and Flexible :**



Expanded and Flexible are used to control how a widget flexes within a Column or Row.



Column(



children: [

Expanded(

child: Container(color: Colors.red), ),

Expanded(

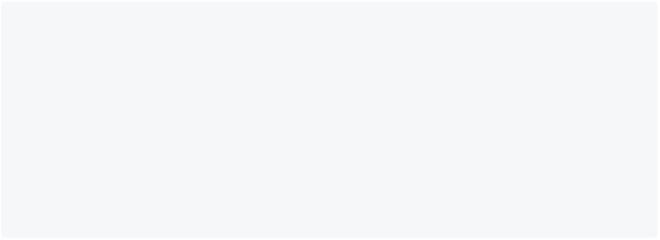
child: Container(color: Colors.blue),

), ], )

**GestureDetector :**

A GestureDetector allows you to capture gestures such as taps and swipes.

GestureDetector(



onTap: () {

// Handle tap

},

child: Container(

color: Colors.green,

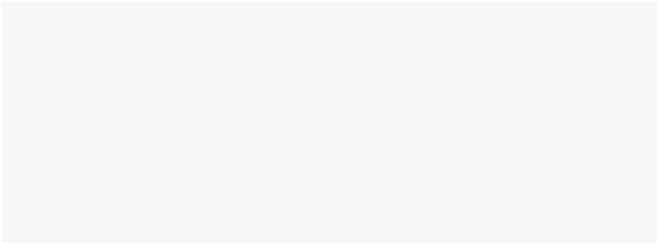
child: Center(child: Text('Tap me!')), ),

)

**FutureBuilder :**

A FutureBuilder is used to build a widget tree based on the latest snapshot of an asynchronous computation.

Future fetchData()async{



//fetch all data

}

@override

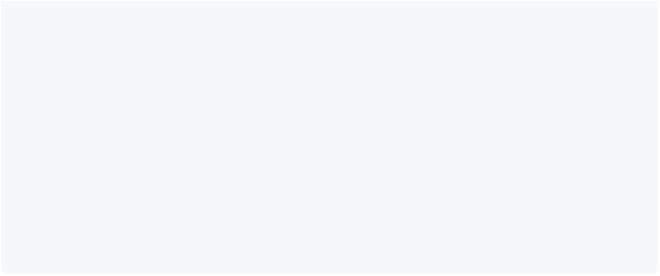
Widget build(BuildContext context){

return FutureBuilder<String>(

future: fetchData(), // async function that produces a future builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.waiting)

{



return CircularProgressIndicator();

} else if (snapshot.hasError) {

return Text('Error: ${snapshot.error}'); } else {

return Text('Data: ${snapshot.data}'); }

}, )

}

**StreamBuilder :**

A **Stream** is a collection of **Futures**.

A StreamBuilder is similar to FutureBuilder but for asynchronous streams.

StreamBuilder<int>(



stream: countStream(),

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.waiting) { return CircularProgressIndicator();

} else if (snapshot.hasError) {

return Text('Error: ${snapshot.error}');

} else {

return Text('Count: ${snapshot.data}');

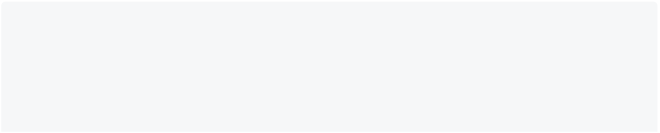
} }, )

**InkWell :**



* **Purpose:**Adds a splash effect and click functionality to any widget wrapped within it.This is used for creating interactive elements like buttons and links.
  + **Properties:**You can customize the splash color,highlight color,shape,and tap callback function.

InkWell(



onTap: () {

print("Tapped!");

},

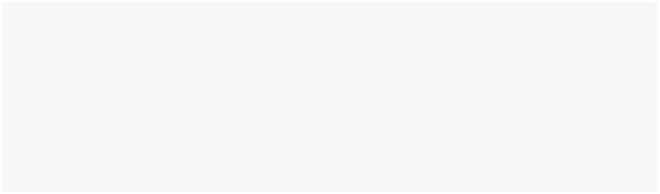
child: Text("Click Me"),

**Chip :**



* **Purpose:**Represents a small,self-contained piece of information with a close button.Used for things like tags,filters,or recently selected items.
  + **Properties:**You can customize the text,avatar,shape,color,deletion icon,and tap callback function.

Chip(



label: Text("Flutter"),

avatar: Icon(Icons.flutter), onDeleted: () {

// Handle deletion here

},

),

### Responsive Layouts in Flutter

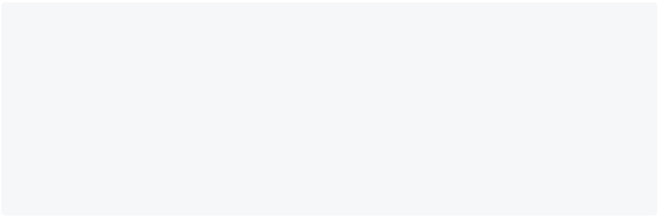
**MediaQuery:**

* Provides device specifics like size, orientation, etc.



* Example:

MediaQueryData mediaQuery = MediaQuery.of(context); double screenWidth = mediaQuery.size.width;



if (screenWidth > 600) {

// Use a wider layout for larger screens

} else {

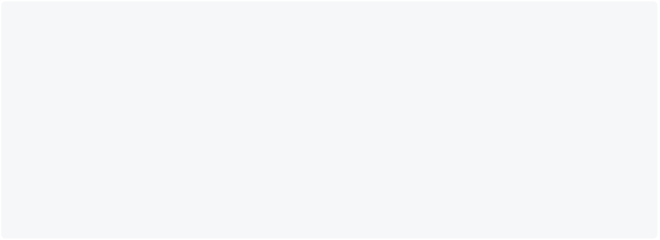
// Use a narrower layout for smaller screens }

**LayoutBuilder:**

* Rebuilds when layout constraints change, providing access to those constraints.



LayoutBuilder(



builder: (context, constraints) {

if (constraints.maxWidth > 600) {

// Use a wider layout for larger screens

} else {

// Use a narrower layout for smaller screens }

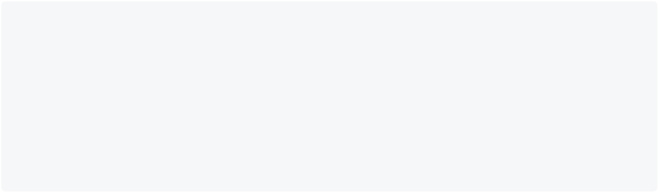
},

);

**Wrap:**

Arranges children in a wrap-around fashion.

Wrap(



children: [

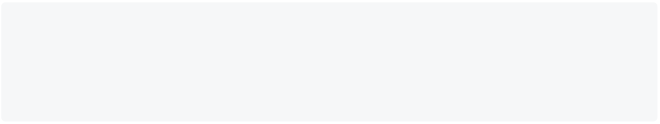
Text("Text 1"), Text("Text 2"), Text("Text 3"), ],

);

**AspectRatio:**

Maintains a specific aspect ratio for its child.

AspectRatio(



aspectRatio: 16 / 9,

child: Image.network("your\_image\_url"), );

**Flexible:**

Allows child to flex within a row or column based on a flex factor.

Row(



children: [

Flexible(flex: 2, child: Text("Wider section")),

Flexible(flex: 1, child: Text("Narrower section")), ],

);

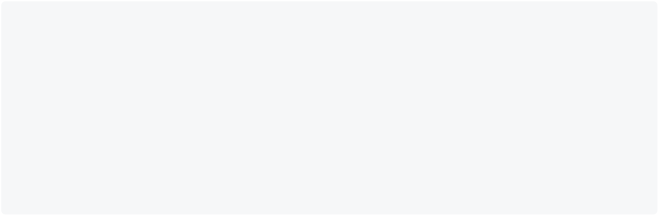
**Expanded:**

Use Expandedwhen you want the child to fill any remaining space in its parent,regardless of its preferred size.



|  |  |  |
| --- | --- | --- |
| **Feature** | **Flexible** | **Expanded** |
| Purpose | Flexible size based on factor | Fill remaining space |
| Respects child size | Yes | No |
| Fills remaining space | Proportionally | Equally (with other Expanded |

Row(



children: [

Text("Fixed text"),

Expanded(child: ElevatedButton(onPressed: () {}, child: Text("Fill rest"))),

],

);

**Flexible:**

* **Purpose:**Allows its child to flex within a row or column based on a**flex factor**.
  + **Behavior:**
    - Respects its child's preferred size if possible.
      * Takes up remaining space in its parent**proportionally**to its flex factor compared to other Flexiblewidgets within the same row/column.
        + Does not force its child to fit its own size.

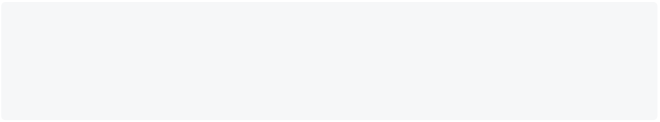
**Expanded:**

* **Purpose:**Forces its child to **fill the remaining available space**in its parent.
  + **Behavior:**
    - Ignores the preferred size of its child.
      * Expands its child to fill the remaining space,potentially causing resizing or overflow if the child has fixed dimensions.
        + If multiple Expandedwidgets are used in the same row/column,they share the remaining space**equally**.

**FractionallySizedBox:**

Allocates a specific fraction of its parent's size to its child.

FractionallySizedBox(



widthFactor: 0.5,

child: Text("Half width content"), );

**IV. Additional Techniques:**

* **Media Queries:**Define different styles or layouts for specific screen sizes/orientations.
  + **Breakpoints:**Set specific points where layout changes significantly for different device groups.
    - **State Management:**Use solutions like Provider or BLoC to share layout information and adapt UI dynamically.

### Creating Custom Widgets

**Step 1: Create a New Dart File**

Create a new Dart file in your Flutter project to define your custom widget. Let's name it custom\_widget.dart.

**Step 2: Import Flutter Material Library**

In your custom\_widget.dart file, import the Flutter Material library. This library provides essential widgets for building material design applications.

import 'package:flutter/material.dart';



**Step 3: Define Your Custom Widget Class**

Create a class that extends StatelessWidget or StatefulWidget based on whether your widget needs to hold mutable state. For this example, we'll create a simple stateless widget named CustomWidget.

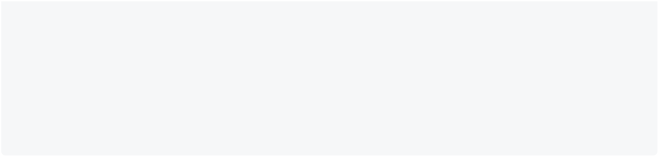
class CustomWidget extends StatelessWidget { @override



Widget build(BuildContext context) {

// Widget's UI goes here

return Container(



// Your widget's content

child: Text('Hello, Custom Widget!'), );

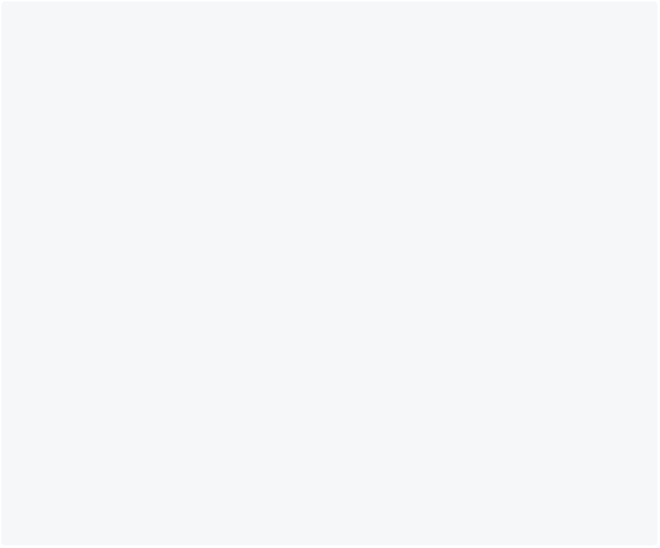
}

}

**Step 4: Using the Custom Widget**

Now that you've defined your custom widget, you can use it in any part of your app. Import your custom\_widget.dart file and add CustomWidget() wherever you need it.

import 'package:flutter/material.dart';



import 'custom\_widget.dart'; // Import your custom widget

void main() {

runApp(MyApp()); }

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text('Custom Widget Example'), ),

body: Center(

child: CustomWidget(), // Use your custom widget here ),

),

);

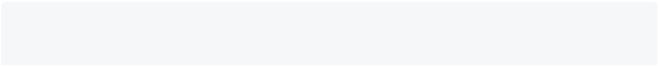
}

}

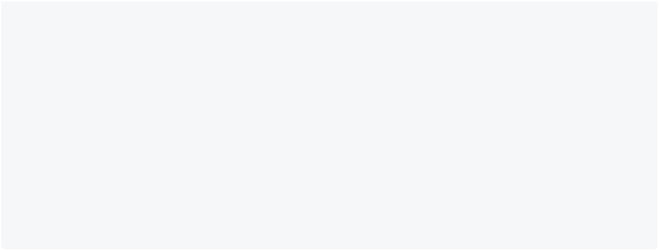
**Step 5: Customize Your Widget**

You can add parameters to your custom widget for customization. For example, let's allow users to customize the displayed text:

class CustomWidget extends StatelessWidget { final String customText;



CustomWidget({required this.customText});



@override

Widget build(BuildContext context) { return Container(

child: Text(customText),

);

}

}

Now, when using your CustomWidget, you can provide different text values:

CustomWidget(customText: 'Welcome to My App'),



Congratulations! You've successfully created a custom widget in Flutter.

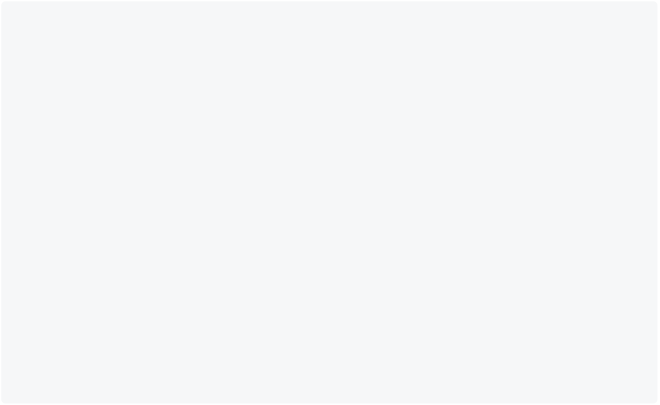
## Custom Themes and Animations

### Using Custom Theme

**Step 1: Create a Custom Theme**

Define a ThemeData instance in a separate file or in your main.dart file. Customize it according to your app's design requirements.

// themes.dart



import 'package:flutter/material.dart';

final ThemeData myCustomTheme = ThemeData(

primarySwatch: Colors.blue,

accentColor: Colors.green,

fontFamily: 'Roboto',

textTheme: TextTheme(

headline1: TextStyle(fontSize: 36.0, fontWeight: FontWeight.bold), bodyText1: TextStyle(fontSize: 16.0, color: Colors.black87),

// Add more custom text styles as needed

),

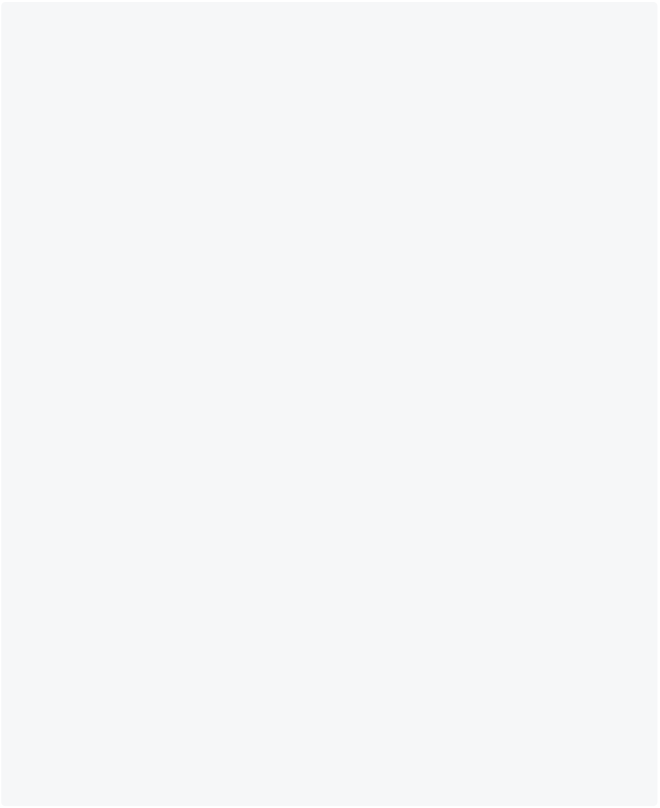
// Add other theme properties

);

**Step 2: Import the Custom Theme**

Import your custom theme in your main.dart file.

// main.dart



import 'package:flutter/material.dart';

import 'themes.dart'; // Import your custom theme

void main() {

runApp(

MaterialApp(

title: 'My Flutter App',

theme: myCustomTheme, // Apply the custom theme home: MyHomePage(),

),

);

}

class MyHomePage extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('My App Home'),

),

body: Center(

child: Text(

'Hello, Flutter!',

style: Theme.of(context).textTheme.headline1, // Use the custom text style

),

),

);

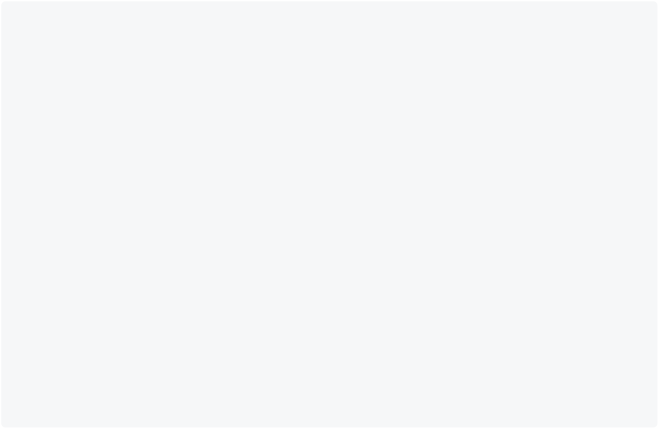
}

}

**Step 3: Customize Theme Throughout Your App**

Now, use the custom theme properties wherever needed in your app.

// AnyWidget.dart



import 'package:flutter/material.dart';

import 'themes.dart'; // Import your custom theme

class AnyWidget extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Container(

color: Theme.of(context).primaryColor,

child: Text(

'Custom Theme Widget',

style: Theme.of(context).textTheme.bodyText1, ),

);

}

}

### Using Animation

**Step 1: Import Flutter Packages**

Start by importing the necessary Flutter packages.

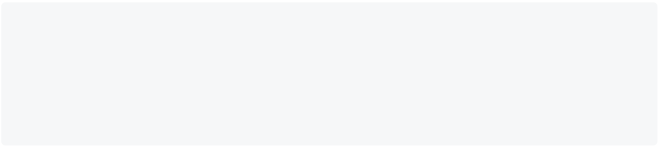
import 'package:flutter/material.dart';



**Step 2: Create a StatefulWidget**

Create a StatefulWidget that will contain the animated elements.

class FadeTransitionExample extends StatefulWidget { @override



\_FadeTransitionExampleState createState() => \_FadeTransitionExampleState();

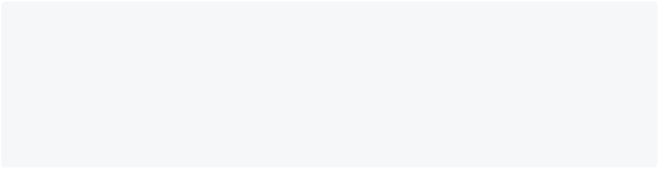
}

**Step 3: Create State Class**

Within the StatefulWidget, create a State class that extends

SingleTickerProviderStateMixin.

class \_FadeTransitionExampleState extends State<FadeTransitionExample> with SingleTickerProviderStateMixin {



late AnimationController \_controller;

late Animation<double> \_animation;

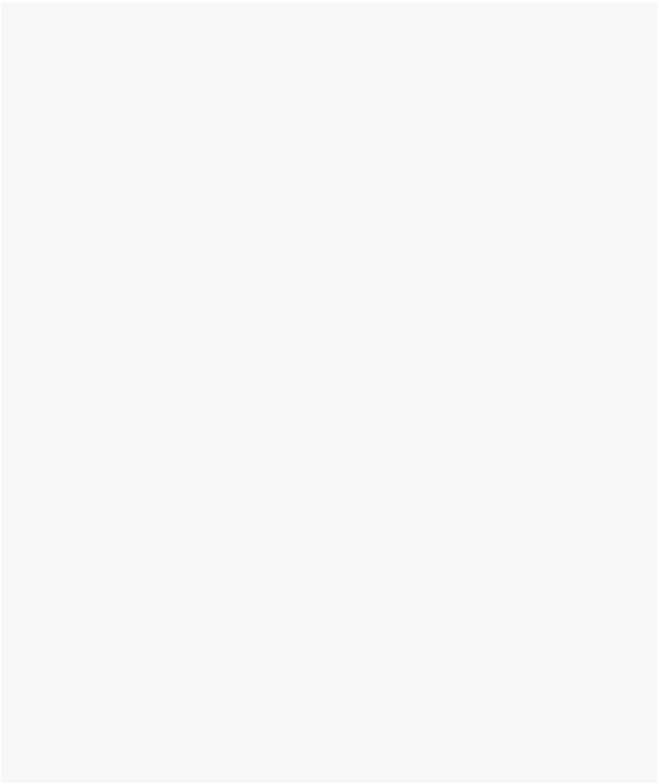
bool \_isFirstWidget = true;

}

**Step 4: Initialize Animation Controllers**

Initialize Animation Controllers within the State class.

@override



void initState() {

super.initState();

\_controller = AnimationController( duration: Duration(seconds: 2), vsync: this,

);

// Define the animation

\_animation = Tween<double>( begin: 0.0,

end: 1.0,

).animate(

CurvedAnimation(

parent: \_controller,

curve: Curves.easeInOut, ),

);

// Add a listener to switch between widgets when the animation completes

\_animation.addStatusListener((status) {

if (status == AnimationStatus.completed) {

setState(() {

\_isFirstWidget = !\_isFirstWidget;

\_controller.reverse();

});

}

});

// Start the animation

\_controller.forward(); }



* In Flutter, the vsync (Vertical Sync) parameter is used in conjunction with animation controllers to synchronize animations with the vertical refresh rate of the device's display. The vsync parameter is typically set to this when the widget's state class extends TickerProviderStateMixin.



* + Here's what vsync: this means and why it's used:

1. **AnimationController and TickerProviderStateMixin:**

* AnimationController is a class that manages animations over time.
  + TickerProviderStateMixin is a mixin that provides a vsync property required by the AnimationController. This mixin is commonly used with StatefulWidget to manage animations in the widget's lifecycle.



1. **TickerProvider:**

* In the context of animations, a "ticker" is a callback that fires each frame of the animation.
  + TickerProvider is an interface that provides a Ticker for animations. It's implemented by TickerProviderStateMixin.

1. **vsync: this:**
   * vsync specifies the object that will be used as the TickerProvider for the animation.



* + - When vsync is set to this, it means the current state object (which extends TickerProviderStateMixin) is used as the TickerProvider.



* + - * This allows the animation controller to be synchronized with the state's lifecycle and, consequently, the frame rate of the device.
        + In summary, setting vsync: this means that the animation controller will use the current state object as the TickerProvider, ensuring that the animations are synchronized with the device's display refresh rate. This synchronization helps in optimizing performance and avoiding unnecessary computations when the widget is not visible or active.
* In Flutter animations, a Tween is a class that defines a range of values over which an animation should interpolate.



**Tween Class:**



The Tween class is part of the Flutter animation framework (dart:ui package).



* It defines a range between a begin value and an end value.



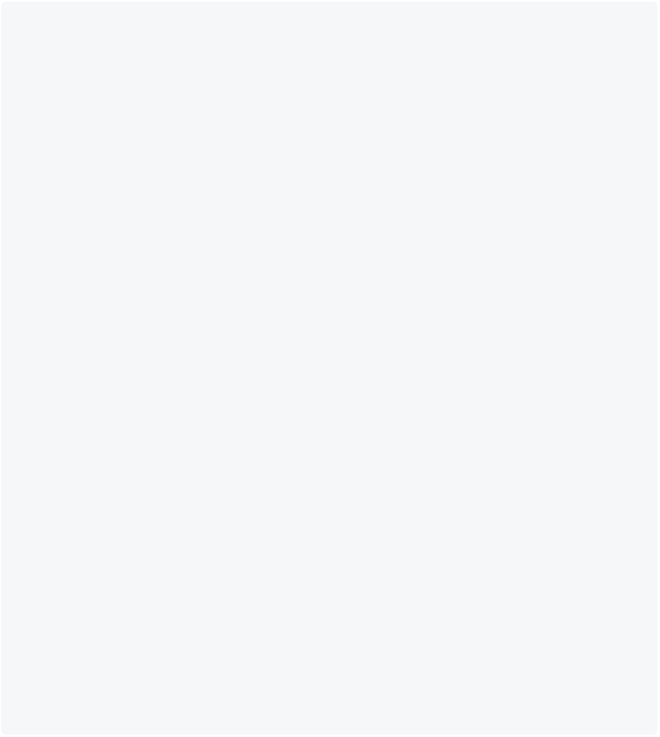
* + **Use with Animation Controllers:**
    - You typically use a Tween in conjunction with an AnimationController. The Tween defines the range of values, and the AnimationController manages how those values are animated over time.



**Step 5: Build Widget Using AnimatedBuilder**

Use the AnimatedBuilder widget to build the widget tree with the fading transition.

@override



Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('Flutter Fading Transition Example'), ),

body: Center(

child: AnimatedBuilder(

animation: \_animation,

builder: (context, child) {

return Stack(

alignment: Alignment.center,

children: [

Opacity(

opacity: \_isFirstWidget ? 1.0 - \_animation.value : \_animation.value,

child: WidgetA(),

),

Opacity(

opacity: \_isFirstWidget ? \_animation.value : 1.0 - \_animation.value,

child: WidgetB(),

),

],

);

},

),

),

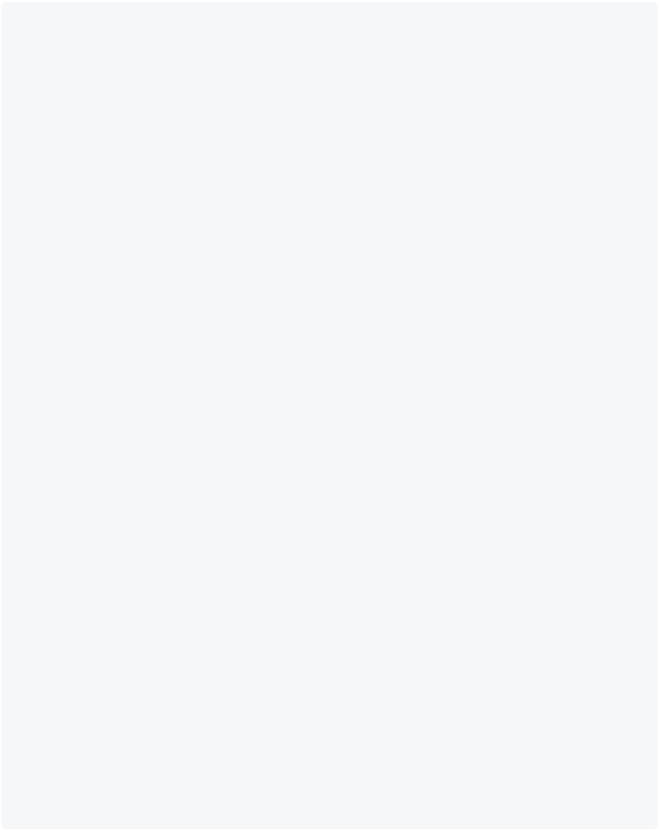
);

}

**Step 6: Create Widgets for Transition**

Create two widgets (WidgetA and WidgetB) to transition between.

class WidgetA extends StatelessWidget {



@override

Widget build(BuildContext context) {

return Container(

width: 200,

height: 200,

color: Colors.blue,

child: Center(

child: Text(

'Widget A',

style: TextStyle(color: Colors.white), ),

),

);

}

}

class WidgetB extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Container(

width: 200,

height: 200,

color: Colors.green,

child: Center(

child: Text(

'Widget B',

style: TextStyle(color: Colors.white), ),

),

);

}

}

**Step 7: Dispose Animation Controllers**

Dispose of the Animation Controllers to free up resources when the widget is disposed.

@override



void dispose() {

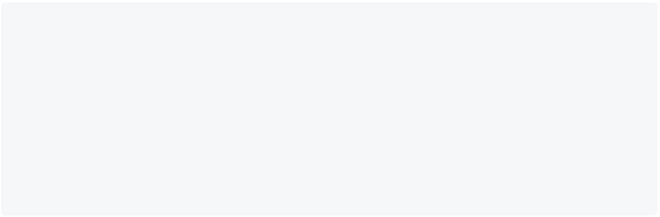
\_controller.dispose(); super.dispose();

}

**Step 8: Run the App**

Finally, run your app. The two widgets will smoothly transition between each other with a fading effect.

void main() {



runApp(

MaterialApp(

home: FadeTransitionExample(), ),

);

}

## Flutter Navigation – How to Add Stack, Tab, and Drawer Navigators to Your Apps

There are three types of navigation that are common to all apps – **stack**, **tab**, and **drawer** navigation. Flutter supports all three types.

### Types of Navigation



There are three main types of navigation that you might use in your apps. Again, they are:

1. Stack Navigation
2. Tab Navigation
3. Drawer Navigation

Let's understand how each one works. **Stack Navigation**

It helps you navigate between pages or screens by stacking new pages on top of existing ones.

When you move to a new screen, the current screen is pushed onto the navigation stack, and when you return, the top screen is popped off the stack.

This navigation type is commonly used for hierarchical and linear flows within an app.

**Tab Navigation**

Tabs are a staple of mobile app navigation, allowing users to quickly switch between different sections or views without losing their current context.

Flutter makes it easy to implement tabbed navigation with its built-in widgets, such as **TabBar** and **TabBarView**.

**Drawer Navigation**

The Drawer Navigation pattern, also known as the "hamburger menu" or "side menu," is a popular navigation style in mobile apps. It consists of a hidden panel that slides out from the side of the screen, revealing a menu with various navigation options.

### How to Build the Stack Navigation



**Navigation and Routing - Flutter Tutorials**

In Flutter, the screen and pages are called a **route**. In android, it is called **Activity**, and in iOS, it is similar to **ViewController**.

In an app, you may need to move from different pages. Flutter provides the routing class MaterialPageRoute, and two methods Navigator.push()

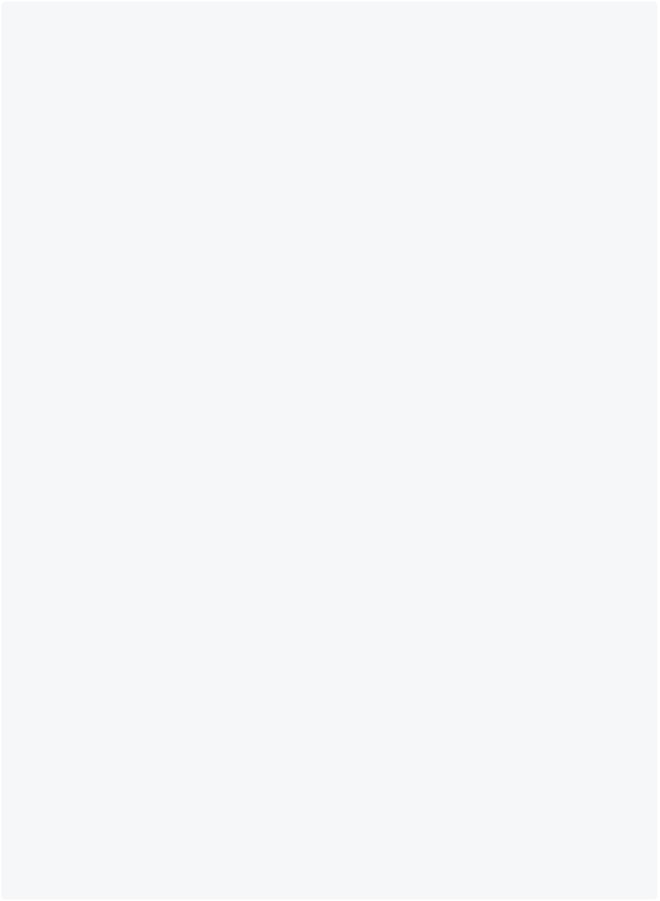
and Navigator.pop() to handle navigations.

**Navigation With Named Routes:**



To navigate between different named routes, you need to create those route classes and index them into **MaterialApp()** widget. For example, create two routes like below:

class HomePage extends StatelessWidget{



@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text("Home Page"),

),

body: Center(

child: RaisedButton(

child: Text("Click on Me"), //click me button

onPressed: (){

Navigator.pushNamed(context, "/secondscreen"); }

)

),

);

}

}

class SecondScreen extends StatelessWidget{

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text("Second Page"),

),

body: Center(

child: RaisedButton(

child: Text("Go Back"), //go back button onPressed: (){

Navigator.pop(context);

}

)

),

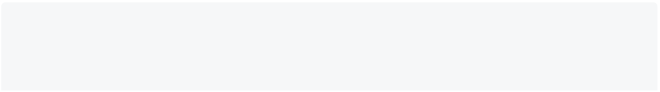
);

}

}

Now index them into MaterialApp() widget.

MaterialApp(



initialRoute: '/', routes: {

'/':(context)=>HomePage(),

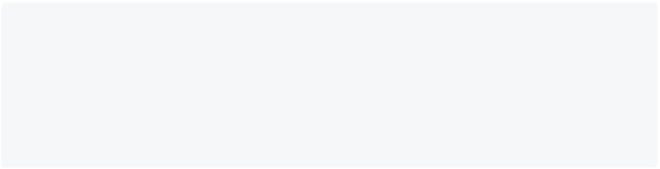


'/secondscreen':(context)=>SecondScreen(), },

);

Put **Navigator.pushNamed()** on '**click on me**' button like below to go to "**secondscreen**" route.

RaisedButton(



child: Text("Click on Me"), //click me button

onPressed: (){

Navigator.pushNamed(context, "/secondscreen"); }

)

Now put **Navigator.pop()** on "**Go Back**" button like below to dismiss "**secondscreen**" route and go back to the home screen.

RaisedButton(



child: Text("Go Back"), //go back button onPressed: (){

Navigator.pop(context);

}

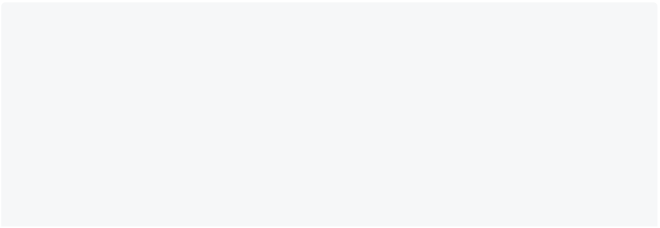
)

**Navigation Without Named Routes:**



To navigate between different pages without named routes, you need to put **Navigator.push()** method instead of **Navigator.pushNamed()** and pass **MaterialPageRoute** class on route parameter.

RaisedButton(



child: Text("Click on Me"), //click me button

onPressed: (){

Navigator.push(context, MaterialPageRoute(builder: (context){ return SecondScreen();

})

);

} )



Note: If you are navigating without named routes, you are not required to mention routes list on **MaterialApp** widget.

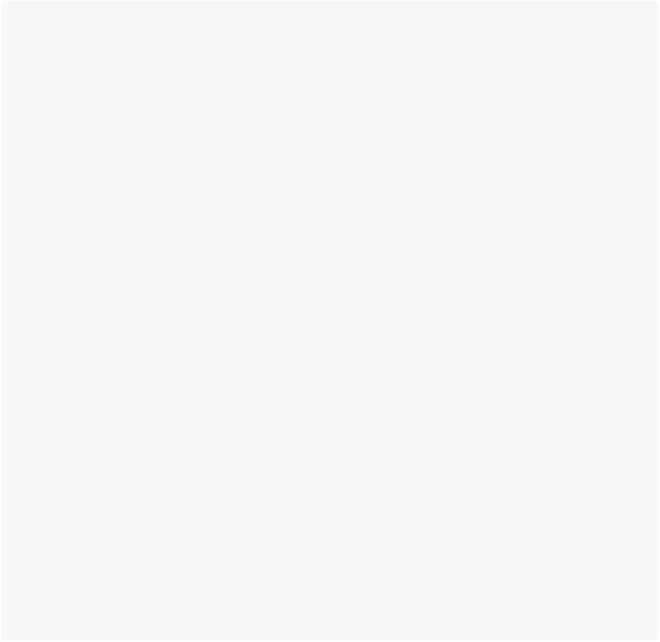
**Passing Data to Forwarding Page:**



While navigating from one page to another, you may need to pass data to navigating page. See the example below to pass data from one page to another.

**Step 1:** Create a page class with a constructor like below:

class SecondScreen extends StatelessWidget{



String word; int val;

SecondScreen({this.word, this.val});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text("Second Page"),

),

body: Center(

child: RaisedButton(

child: Text("Word:$word, Value:$val, - Go Back"), //go back button

// you need to put like widget.word on stateful widgets. onPressed: (){

Navigator.pop(context);

}

)

),

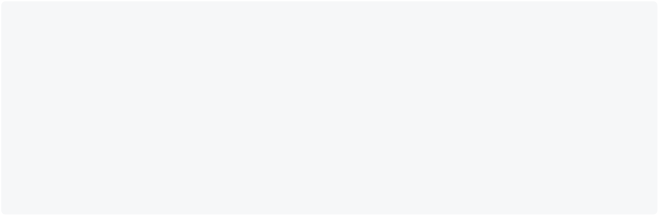
);

}

}

**Step 2:** Now put the command on "**Click on Me**" button like below:

RaisedButton(



child: Text("Click on Me"), //click me button

onPressed: (){

Navigator.push(context, MaterialPageRoute(builder: (context){ return SecondScreen(word: "Hello", val:4);

})

);

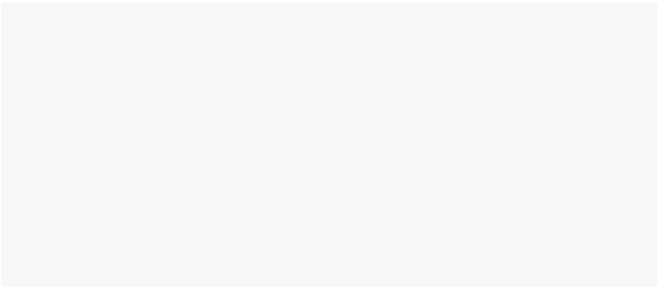
}

**Passing data while going back page:**



To pass data to the back page, you need to put **Navigator.push()** method on "**Click on Me**" or any back buttons like below:

RaisedButton(



child: Text("Click on Me"), //click me button

onPressed: () async {

var backdata = await Navigator.push(context, MaterialPageRoute(builder: (context){

return SecondScreen(word: "Hello", val:4); })

);

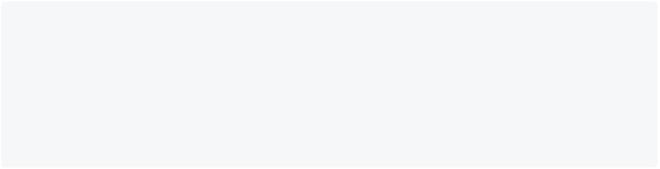
print(backdata);

}

)

On the back button, put **Navigator.pop()** method like below:

RaisedButton(



child: Text("Go Back"), //go back button onPressed: (){

Navigator.pop(context, "returntext"); }

)

### How to Build the Tab Navigation

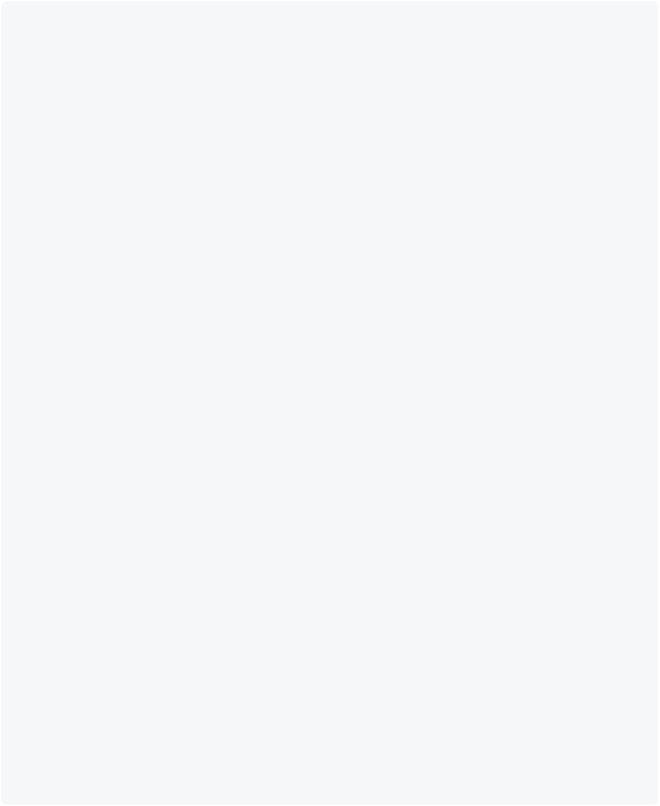


Let's begin with building the tab navigator. Let's assume the tab will be on the home page (ideally that's where it would be).

Create a new file named tab.dart in the lib/ directory. Add the following code:



import 'package:flutter/material.dart'; import './tabs/tab1.dart';



import './tabs/tab2.dart';

import './tabs/tab3.dart';

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

@override

Widget build(BuildContext context) {

return DefaultTabController(

length: 3,

child: Scaffold(

appBar: AppBar(

title: const Text("Home"),

bottom: const TabBar(

tabs: [

Tab(icon: Icon(Icons.phone\_android)), Tab(icon: Icon(Icons.tablet\_android)), Tab(icon: Icon(Icons.laptop\_windows)),

],

),

),

body: const TabBarView( children: <Widget>[ Tab1(),

Tab2(),

Tab3(),

],

)),

);

}

}

**Tab Navigation in Flutter**

In the above code, we're creating a class named HomePage. In the build method, we return the DefaultTabController widget, which is basically a tab view. We

define that we need 3 tabs in the length property.



At the bottom of the appBar property we have defined icons for each tab (Phone, Tablet, and Computer icons). Below that we define the body property with a



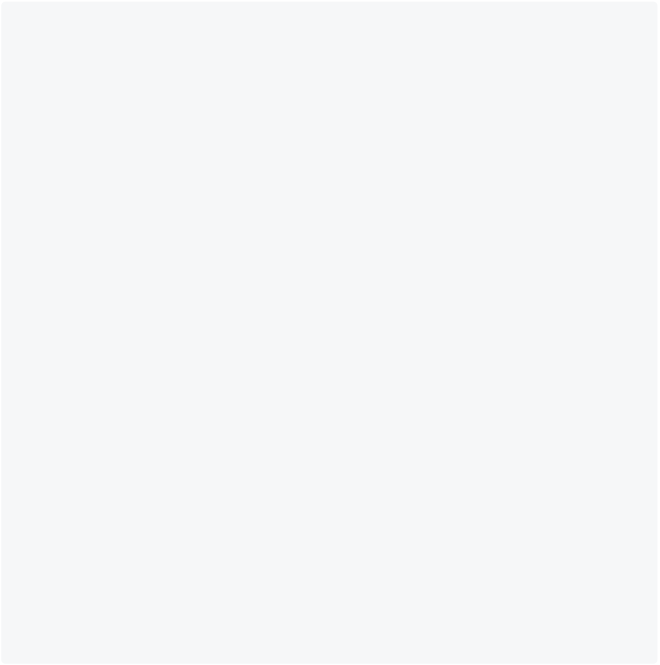
TabBarView rendering all the tabs inside it.

Create a new folder named tabs inside the lib/ directory and create three files named tab1.dart, tab2.dart, and tab3.dart.



Copy the below content into the tab1.dart file:

import 'package:flutter/material.dart';



class Tab1 extends StatelessWidget { const Tab1({super.key});

@override

Widget build(BuildContext context) {

return SizedBox(

width: double.infinity,

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

const Text("Mobiles"),

Padding(

padding: const EdgeInsets.only(top: 16.0),

child: ElevatedButton(

onPressed: () {

Navigator.of(context).pushNamed("/secret");

},

child: const Text('Disclose Secret'), ),

),

],

),

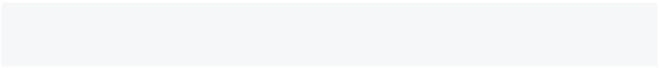
);

}

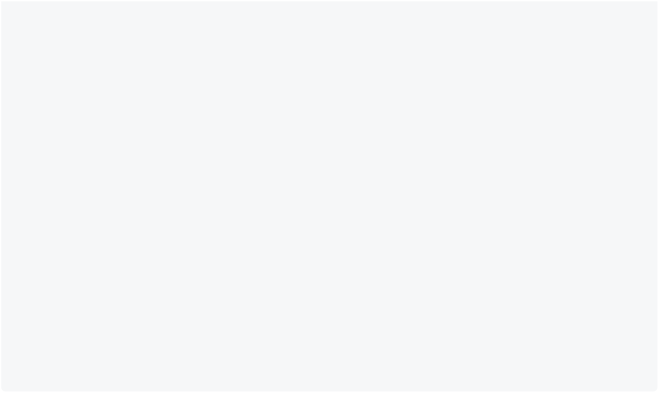
}

Copy the below content into the tab2.dart file:

import 'package:flutter/material.dart';



class Tab2 extends StatelessWidget { const Tab2({super.key});



@override

Widget build(BuildContext context) {

return SizedBox(

width: double.infinity,

child: Column(

mainAxisAlignment: MainAxisAlignment.center, children: const <Widget>[

Text("Tablets"),

],

),

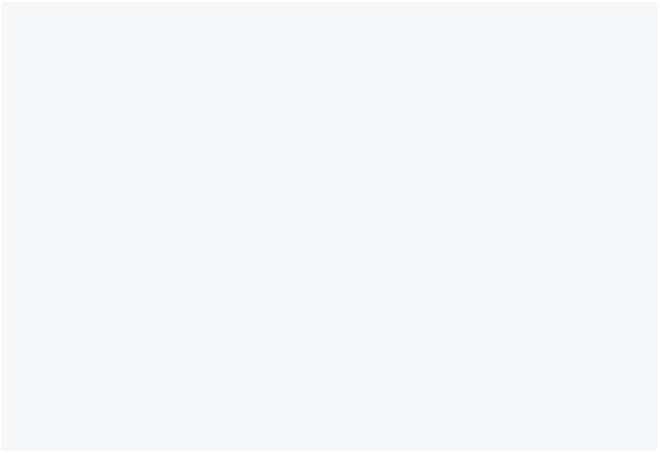
);

}

}

Copy the below code into the tab3.dart file:

import 'package:flutter/material.dart';



class Tab3 extends StatelessWidget { const Tab3({super.key});

@override

Widget build(BuildContext context) {

return SizedBox(

width: double.infinity,

child: Column(

mainAxisAlignment: MainAxisAlignment.center, children: const <Widget>[

Text("Laptops"),

],

),

);

}

}

If you look at the code for all three files, you'll notice everything is the same except that the first tab file (tab1.dart) has an additional button called "Disclose Secret". Pressing that will navigate the user to the /secret route. It won't throw

any error as this route has not been defined yet. The other two files (tab2.dart and tab3.dart) will show only the text.

Add the following line at the top of the main.dart file:

import './tab.dart';



Replace home: const MyHomePage(title: 'Home') with home: const HomePage(), in the build method of the MyApp class.



Save the file and run your app. You should be able to see the tab layout on your screen now.

### How to Build the Drawer Navigation



Our next target is to add the drawer navigation. But before that, we have to create two files:

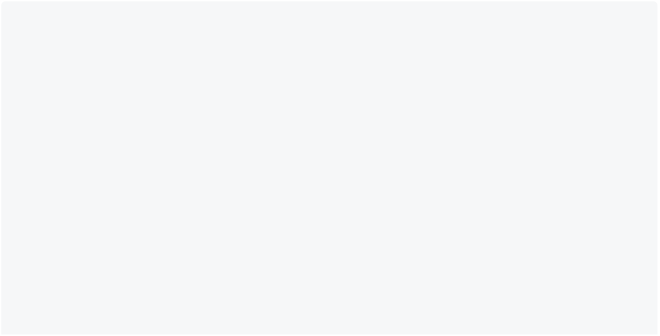
1. drawer.dart: to show the Navigation Drawer
2. about.dart: an option will be provided on the Drawer Navigator to navigate

here

Create the drawer.dart file inside the lib/ directory and not inside the tab/ directory. The tab/ directory is only for tabs and we don't need to touch that further as we're done with the tabs. Copy the below code into the drawer.dart file:



import 'package:flutter/material.dart';



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

navigateTo(String route, BuildContext context) {

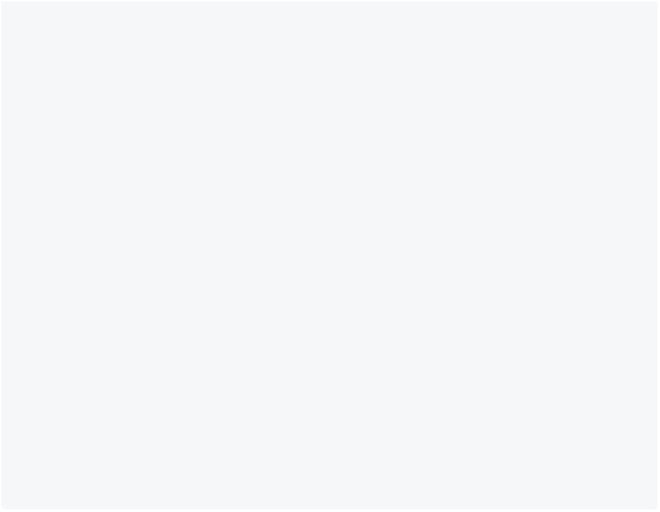
Navigator.of(context).pushReplacementNamed(route); }

@override

Widget build(BuildContext context) { return Drawer(

child: ListView(

padding: const EdgeInsets.all(16.0), children: <Widget>[



ListTile(

leading: const Icon(Icons.home), title: const Text('Home'),

onTap: () {

navigateTo("/home", context); },

),

ListTile(

leading: const Icon(Icons.info), title: const Text('About'),

onTap: () {

navigateTo("/about", context);

}, ), ],

),

);

}

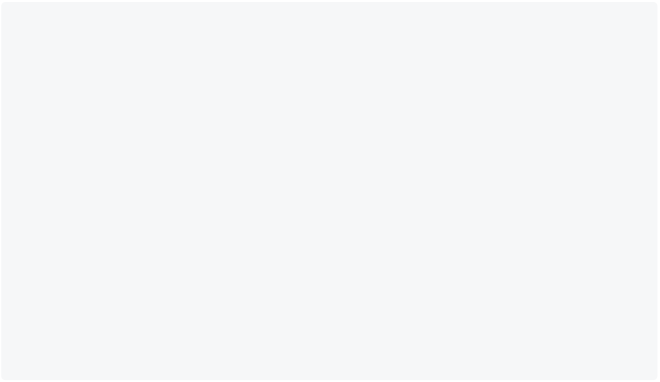
}

In this file, we define the class named MyDrawer. In the build method we render the Drawer widget with Home and About options in the list. Clicking on those options will navigate us to the appropriate routes.



Create an about.dart file in the same directory and copy the below code:

import './drawer.dart';



import 'package:flutter/material.dart';

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

@override

Widget build(BuildContext context) {

return Scaffold(

drawer: const MyDrawer(),

appBar: AppBar(title: const Text("About")), body: const Center(child: Text("About")), );

}

}

In this file, we create a class named About which returns a Scaffold widget containing the drawer which we defined right before this file. The appBar and the



body will show the text "About".



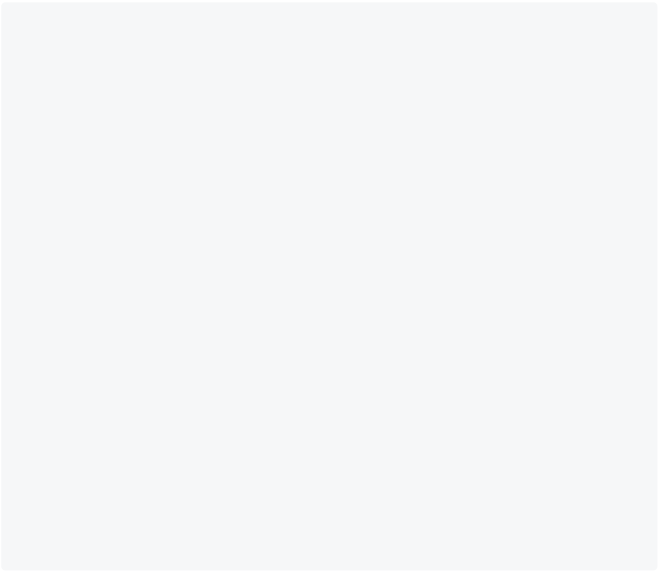
Again, you'll not be able to see these changes immediately in the app. This is because we haven't linked it into the main.dart file.

Before we link them, we have one item in our backlog. Let's finish it and come back to linking them all together.

Create a file named secret.dart in the lib/ directory and copy the below code:



import 'package:flutter/material.dart';



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

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

// backgroundColor: Colors.red,

title: const Text("Secret"),

),

body: SizedBox(

width: double.infinity,

child: Column(

mainAxisAlignment: MainAxisAlignment.center, children: const <Widget>[

Text("Nothing to show"),

],

),

));

}

}

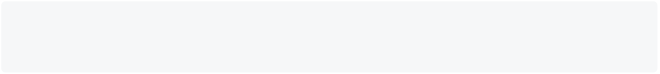
In this file, we have created a class named SecretPage and returned just a Text in the body. Nothing fancy here. It's a super simple Flutter widget.



Our backlog item is also done. This is what you've been waiting for: we're going to define our routes now.

Open the main.dart file and add the following imports at the top of the file:

import './about.dart'; import './secret.dart';



Replace the build method of the MyApp class with the below code:



@override



Widget build(BuildContext context) {

return MaterialApp(

routes: <String, WidgetBuilder>{

"/about": (BuildContext context) => const About(),

"/home": (BuildContext context) => const HomePage(),

"/secret": (BuildContext context) => const SecretPage(), },

initialRoute: "/home",

title: 'Flutter Navigation',

theme: ThemeData(

primarySwatch: Colors.blue,

),

home: const HomePage(),

);

}

Update in main.dart file

In the above code, you can see we're defining the MaterialApp to contain routes. They're defined as key-value pairs, mapping a route with a Widget. We have defined three routes:

* /about – the route for the drawer navigator
  + /home – the route for the tab navigator



* + - /secret – the route for the stack navigator

We have set the initial route to be /home, which has the tab navigator. Run the app and you should be able to see the output on your device.



On pressing the "Disclose Secret" button you'll be taken to the Secret page which we created (ideally it does not have a secret). You should also be able to scroll through the tabs smoothly.

By now, I hope you will have noticed an error here. If not, here's what it is: the back button is shown on the first screen of our app.

"Why would we need to show the back button on the first screen?"

That's an error and we have to resolve it. Press the back button and let's see what happens. Hopefully, you see what I saw. The back button was hidden and we see just the "Home" title in the appBar



But there's an another issue on the same screen. Hopefully you saw that too. If not, don't worry, I'll reveal it right here.

"Can you access the drawer navigator by any means?" No. Right?

But fortunately, the fix for the above two issues is the same. If we fix the second issue, the first issue will automatically be fixed.

That's great. But how do we fix the second issue?

You have to show the drawer navigator button (Hamburger icon) on the top left. This will eventually hide the back button.

Open the tab.dart file and import the drawer file at the top of this file.

import './drawer.dart';



Add the following line inside the Scaffold widget of the build method:



drawer: const MyDrawer(),



And that's it!

**Resources**

* [Flutter Navigation](https://docs.flutter.dev/ui/navigation)

## Dependency Management

Dependency management is an essential aspect of any software development project. It involves managing the external libraries or packages that your project relies on to function correctly. In Dart, you can use a tool called pub and a configuration file called pubspec.yaml to manage dependencies for your project.



By the end of this day, you should have a good understanding of how to use pub and pubspec to manage dart project dependencies. And learn the basics of how to import and use packages in Dart

**Tips**



* pub is a package manager that comes bundled with the Dart SDK, and it allows you to search for and download external packages from the Dart package repository. You can also use it to install, upgrade, and remove packages as needed for your project.



* + The pubspec.yaml file is where you define your project’s dependencies and other metadata about your project, such as its name, version, and description. This file is used by pub to manage your project’s dependencies and ensure that your project has the correct versions of packages installed.



* + - Use the[pub.dev](https://pub.dev/)website to search for Flutter packages.
      * Use the flutter pub add <package-name>command to install packages or you can manually add the package name to the pubspec.yamlfile and run flutter pub getto install it.
        + To import and use packages in your Dart project, you can use the import statement to bring in the package’s functionality into your project. For example, to use the http package, you would add the following line to your Dart file:



import 'package:http/http.dart';



This would allow you to use the http package’s functions and classes in your code.



**Resources**



* [Pub tool](https://dart.dev/tools/pub)
  + [Pubspec format](https://dart.dev/tools/pub/pubspec)
    - [Dart package repository](https://pub.dev/) - a repository of Dart packages
      * [Official Dart documentation on Packages](https://dart.dev/guides/packages)

## Data persistence in Flutter

### Using Shared Preferences

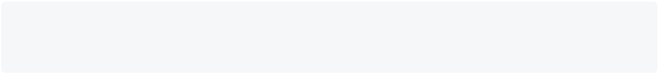
Persistence refers to the ability to store data locally on a device so that it can be accessed later, even when the app is closed or the device is restarted. Shared preferences is one of the simplest approach to persist data locally in a Flutter app. It provides a simple way to store key-value pairs of data locally in a Flutter app. It is simple to use and suitable in various cases where you need to store small amounts of data locally on a device.

**Tips**



* Shared preference is a simple way to store key-value pairs of data locally in a Flutter app.
  + Shared preferences is suitable for storing small amounts of data locally on a device.
    - Adding shared preferences to a Flutter app is easy

dependencies:



shared\_preferences: <latest version>

Import the package in your Dart code



import 'package:shared\_preferences/shared\_preferences.dart';



Create an instance of SharedPreferences



SharedPreferences prefs = await SharedPreferences.getInstance();



* Using shared preferences to store and read data

int counter = prefs.getInt('counter') ?? 0; prefs.setInt('counter', counter + 1);



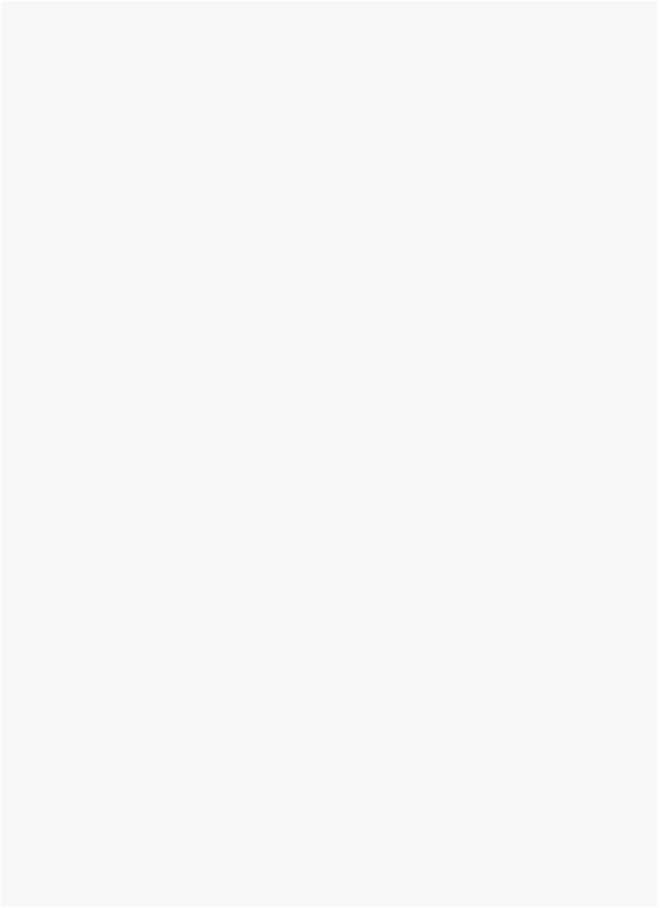
String username = prefs.getString('username') ?? ''; prefs.setString('username', 'John');

bool isDarkModeEnabled = prefs.getBool('isDarkModeEnabled') ?? false; prefs.setBool('isDarkModeEnabled', true);



**Complete Example**

import 'package:flutter/material.dart';



import 'package:shared\_preferences/shared\_preferences.dart';

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

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

@override

Widget build(BuildContext context) {

return const MaterialApp(

title: 'Shared preferences demo',

home: MyHomePage(title: 'Shared preferences demo'), );

}

}

class MyHomePage extends StatefulWidget {

const MyHomePage({super.key, required this.title});

final String title;

@override

State<MyHomePage> createState() => \_MyHomePageState(); }

class \_MyHomePageState extends State<MyHomePage> { int \_counter = 0;

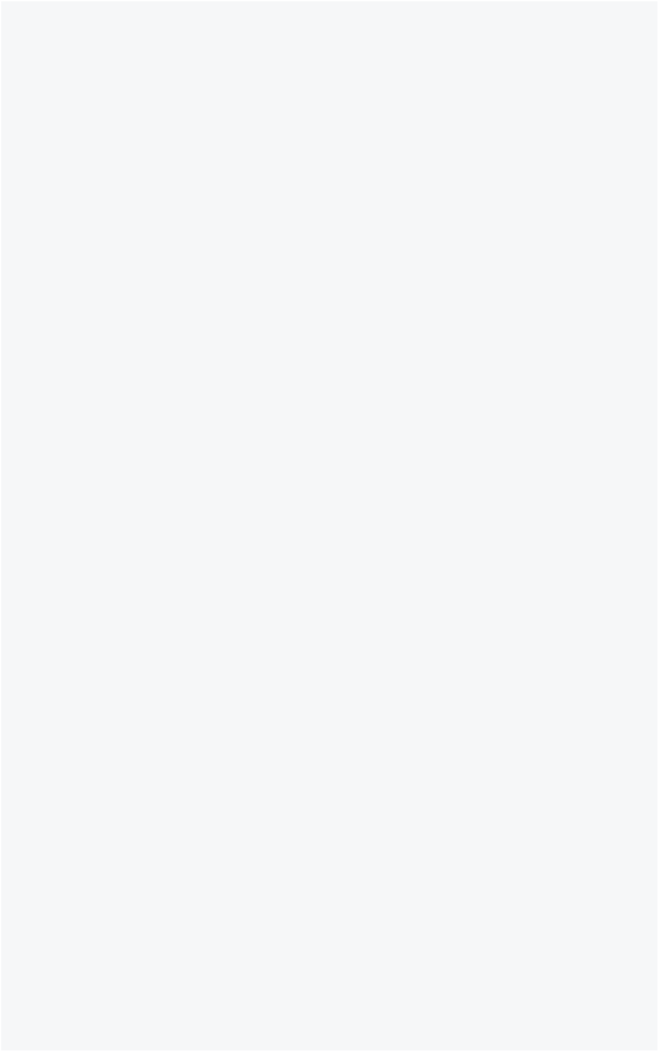
@override

void initState() { super.initState(); \_loadCounter();

}

/// Load the initial counter value from persistent storage on start, /// or fallback to 0 if it doesn't exist.

Future<void> \_loadCounter() async {



final prefs = await SharedPreferences.getInstance(); setState(() {

\_counter = prefs.getInt('counter') ?? 0;

});

}

/// After a click, increment the counter state and

/// asynchronously save it to persistent storage.

Future<void> \_incrementCounter() async {

final prefs = await SharedPreferences.getInstance(); setState(() {

\_counter = (prefs.getInt('counter') ?? 0) + 1;

prefs.setInt('counter', \_counter);

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text(widget.title),

),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

const Text(

'You have pushed the button this many times: ',

),

Text(

'$\_counter',

style: Theme.of(context).textTheme.headlineMedium, ),

],

),

),

floatingActionButton: FloatingActionButton(

onPressed: \_incrementCounter,

tooltip: 'Increment',

child: const Icon(Icons.add),

),

);

} }



**Resources**

* [Flutter official documentation on shared\_preferences](https://flutter.dev/docs/cookbook/persistence/key-value)
  + [Shared preferences package](https://pub.dev/packages/shared_preferences)

### Using SQLite

If you are writing an app that needs to persist and query large amounts of data on the local device, consider using a database instead of a local file or key-value store. In general, databases provide faster inserts, updates, and queries compared to other local persistence solutions.

Flutter apps can make use of the SQLite databases via the [sqflite](https://pub.dev/packages/sqflite) plugin available on pub.dev. This guide demonstrates the basics of using sqflite to insert, read, update, and remove data about various Dogs.

If you are new to SQLite and SQL statements, review the [SQLite Tutorial](http://www.sqlitetutorial.net/) to learn the basics before completing this tutorial.

This tutorial uses the following steps:

1. Add the dependencies.
2. Define the Dog data model.



1. Open the database.
2. Create the dogs table.



1. Insert a Dog into the database.



1. Retrieve the list of dogs.
2. Update a Dog in the database.



1. Delete a Dog from the database.



1. **Add the dependencies**



To work with SQLite databases, import the sqflite and path packages.



The sqflite package provides classes and functions to interact with a SQLite database.

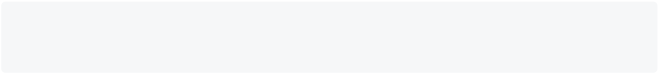


The path package provides functions to define the location for storing the database on disk.



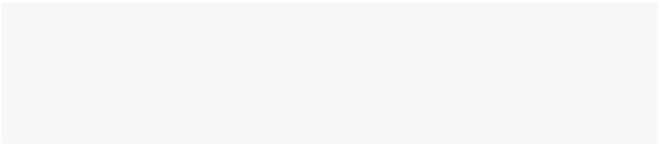
To add the packages as a dependency, run flutter pub add:

$ flutter pub add sqflite path



Make sure to import the packages in the file you’ll be working in.

import 'dart:async';



import 'package:flutter/widgets.dart'; import 'package:path/path.dart'; import 'package:sqflite/sqflite.dart';

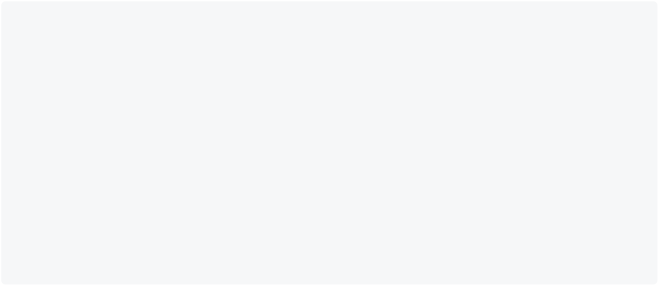
1. **Define the Dog data model**



Before creating the table to store information on Dogs, take a few moments to define the data that needs to be stored. For this example, define a Dog class that contains three pieces of data: A unique id, the name, and the age of each dog.



class Dog {



final int id;

final String name; final int age;

const Dog({

required this.id, required this.name, required this.age, });

}

1. **Open the database**



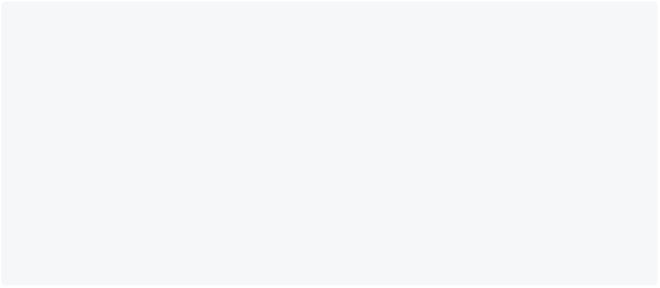
Before reading and writing data to the database, open a connection to the database. This involves two steps:

1. Define the path to the database file using getDatabasesPath() from the sqflite package, combined with the join function from the path package.



1. Open the database with the openDatabase() function from sqflite.

// Avoid errors caused by flutter upgrade.



// Importing 'package:flutter/widgets.dart' is required. WidgetsFlutterBinding.ensureInitialized();

// Open the database and store the reference.

final database = openDatabase(

// Set the path to the database. Note: Using the `join` function from the

// `path` package is best practice to ensure the path is correctly

// constructed for each platform.

join(await getDatabasesPath(), 'doggie\_database.db'),

);

**4. Create the dogs table**



Next, create a table to store information about various Dogs. For this example, create a table called dogs that defines the data that can be stored. Each Dog contains an id, name, and age. Therefore, these are represented as three columns in the dogs table.



1. The id is a Dart int, and is stored as an INTEGER SQLite Datatype. It is also good practice to use an id as the primary key for the table to improve query and update times.



1. The name is a Dart String, and is stored as a TEXT SQLite Datatype.

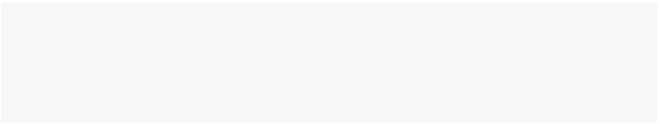


1. The age is also a Dart int, and is stored as an INTEGER Datatype.



For more information about the available Datatypes that can be stored in a SQLite database, see the [official SQLite Datatypes documentation](https://www.sqlite.org/datatype3.html).

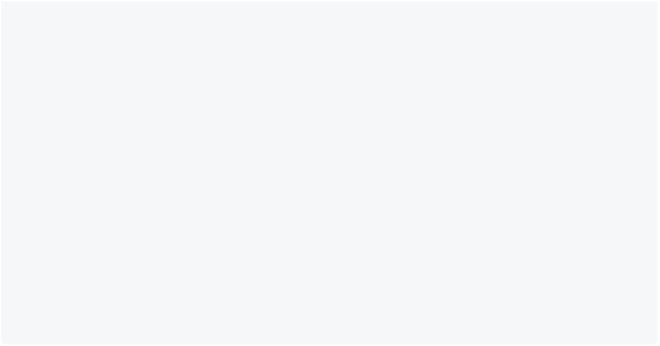
final database = openDatabase(



// Set the path to the database. Note: Using the `join` function from the

// `path` package is best practice to ensure the path is correctly

// constructed for each platform.



join(await getDatabasesPath(), 'doggie\_database.db'),

// When the database is first created, create a table to store dogs. onCreate: (db, version) {

// Run the CREATE TABLE statement on the database.

return db.execute(

'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',

);

},

// Set the version. This executes the onCreate function and provides a // path to perform database upgrades and downgrades.

version: 1,

);

**5. Insert a Dog into the database**



Now that you have a database with a table suitable for storing information about various dogs, it’s time to read and write data.

First, insert a Dog into the dogs table. This involves two steps:



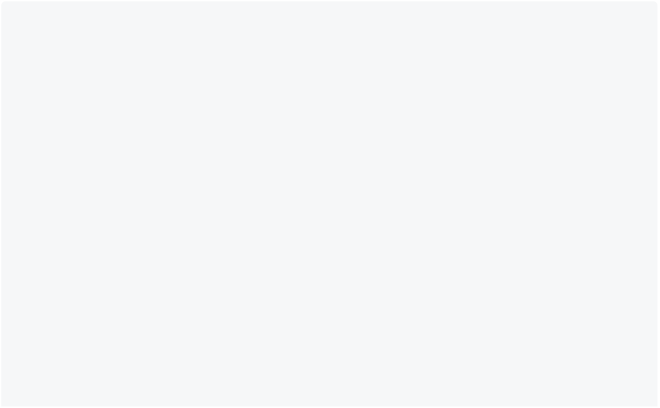
1. Convert the Dog into a Map



1. Use the [insert()](https://pub.dev/documentation/sqflite_common/latest/sqlite_api/DatabaseExecutor/insert.html) method to store the Map in the dogs table.



class Dog {



final int id;

final String name; final int age;

Dog({

required this.id, required this.name, required this.age, });

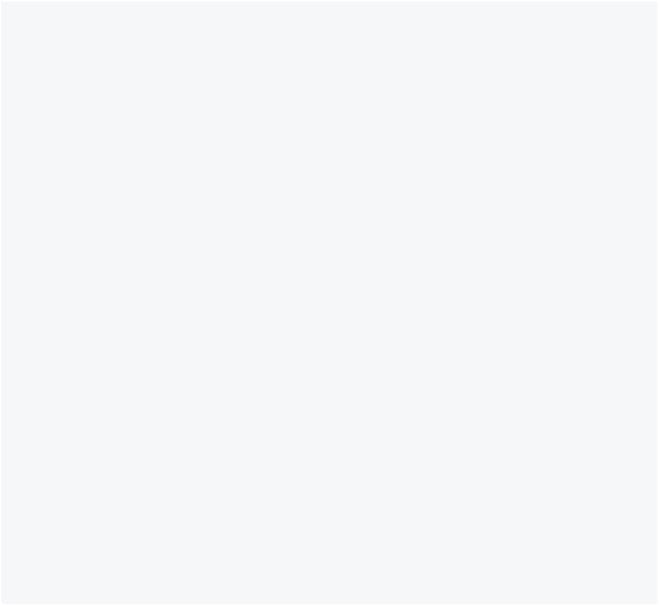
// Convert a Dog into a Map. The keys must correspond to the names of the

// columns in the database.

Map<String, Object?> toMap() {

return {

'id': id,



'name': name, 'age': age, };

}

// Convert a Map into a Dog. The keys must correspond to the names of the

// columns in the database.

factory Dog.fromMap(Map<String, dynamic> map) {

return Dog(

id: map['id'] as int,

name: map['name'] as String,

age: map['age'] as int,

);

}

// Implement toString to make it easier to see information about // each dog when using the print statement.

@override

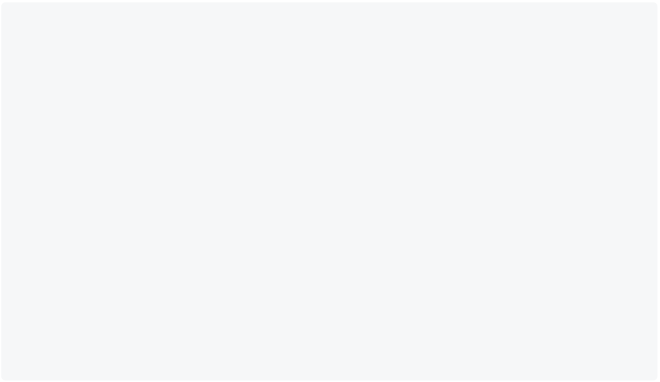
String toString() {

return 'Dog{id: $id, name: $name, age: $age}';

}

}

// Define a function that inserts dogs into the database Future<void> insertDog(Dog dog) async {



// Get a reference to the database.

final db = await database;

// Insert the Dog into the correct table. You might also specify the // `conflictAlgorithm` to use in case the same dog is inserted twice. //

// In this case, replace any previous data.

await db.insert(

'dogs',

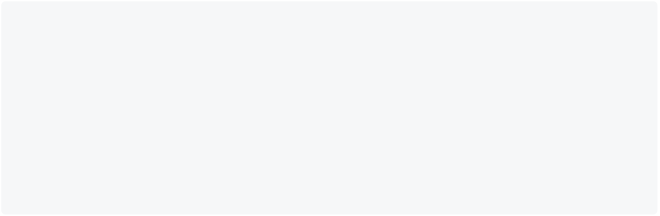
dog.toMap(),

conflictAlgorithm: ConflictAlgorithm.replace,

);

}

// Create a Dog and add it to the dogs table var fido = Dog(



id: 0,

name: 'Fido',

age: 35,

);

await insertDog(fido);

**6. Retrieve the list of Dogs**



Now that a Dog is stored in the database, query the database for a specific dog or a list of all dogs. This involves two steps:

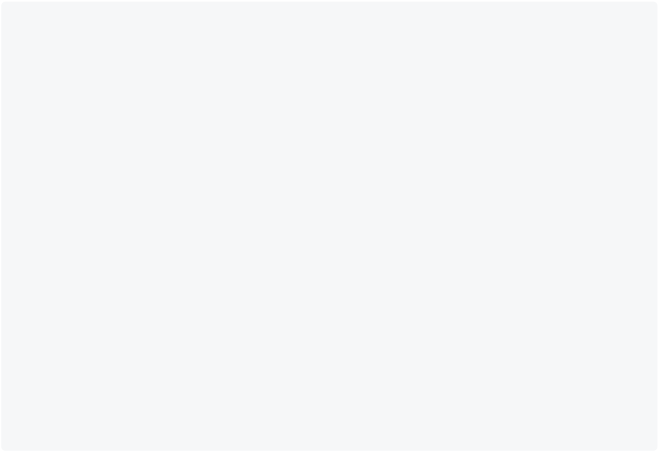


1. Run a query against the dogs table. This returns a List<Map>.



1. Convert the List<Map> into a List<Dog>.

// A method that retrieves all the dogs from the dogs table. Future<List<Dog>> dogs() async {



// Get a reference to the database.

final db = await database;

// Query the table for all the dogs.

final List<Map<String, Object?>> dogMaps = await db.query('dogs');

// Convert the list of each dog's fields into a list of `Dog` objects. return [

for (final {

'id': id as int,

'name': name as String,

'age': age as int,

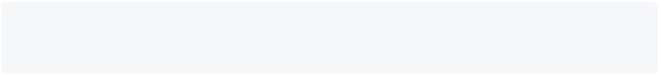
} in dogMaps)

Dog(id: id, name: name, age: age),

];

}

// Now, use the method above to retrieve all the dogs. print(await dogs()); // Prints a list that include Fido.



**7. Update a Dog in the database**



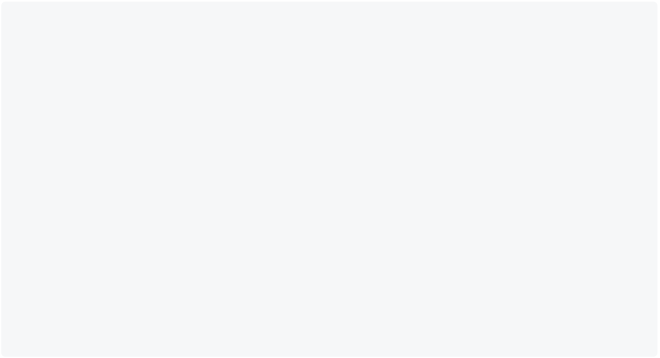
After inserting information into the database, you might want to update that information at a later time. You can do this by using the [update()](https://pub.dev/documentation/sqflite_common/latest/sqlite_api/DatabaseExecutor/update.html) method from the sqflite library.

This involves two steps:

1. Convert the Dog into a Map.
2. Use a where clause to ensure you update the correct Dog.



Future<void> updateDog(Dog dog) async { // Get a reference to the database. final db = await database;



// Update the given Dog.

await db.update(

'dogs',

dog.toMap(),

// Ensure that the Dog has a matching id.

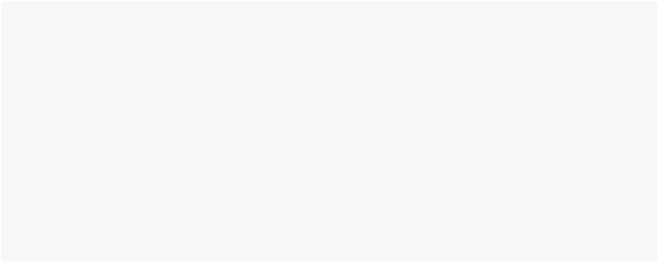
where: 'id = ?',

// Pass the Dog's id as a whereArg to prevent SQL injection. whereArgs: [dog.id],

);

}

// Update Fido's age and save it to the database. fido = Dog(



id: fido.id,

name: fido.name,

age: fido.age + 7,

);

await updateDog(fido);

// Print the updated results.

print(await dogs()); // Prints Fido with age 42.

**8. Delete a Dog from the database**

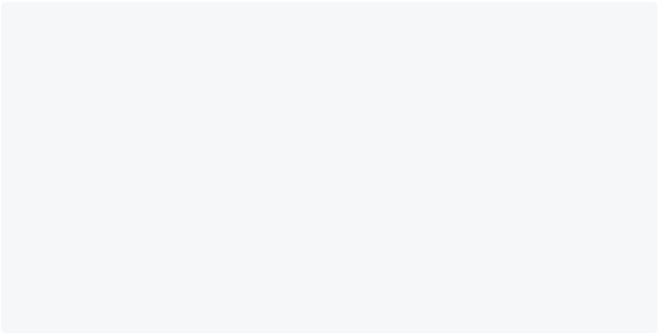


In addition to inserting and updating information about Dogs, you can also remove dogs from the database. To delete data, use the [delete()](https://pub.dev/documentation/sqflite_common/latest/sqlite_api/DatabaseExecutor/delete.html) method from the sqflite library.

In this section, create a function that takes an id and deletes the dog with a matching id from the database. To make this work, you must provide a where clause to limit the records being deleted.



Future<void> deleteDog(int id) async { // Get a reference to the database. final db = await database;



// Remove the Dog from the database.

await db.delete(

'dogs',

// Use a `where` clause to delete a specific dog.

where: 'id = ?',

// Pass the Dog's id as a whereArg to prevent SQL injection. whereArgs: [id],

);

}

**Example**



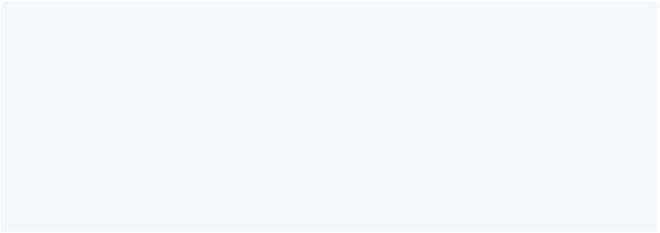
To run the example:

1. Create a new Flutter project.
2. Add the sqflite and path packages to your pubspec.yaml.



1. Paste the following code into a new file called lib/db\_test.dart.
2. Run the code with flutter run lib/db\_test.dart.

import 'dart:async';



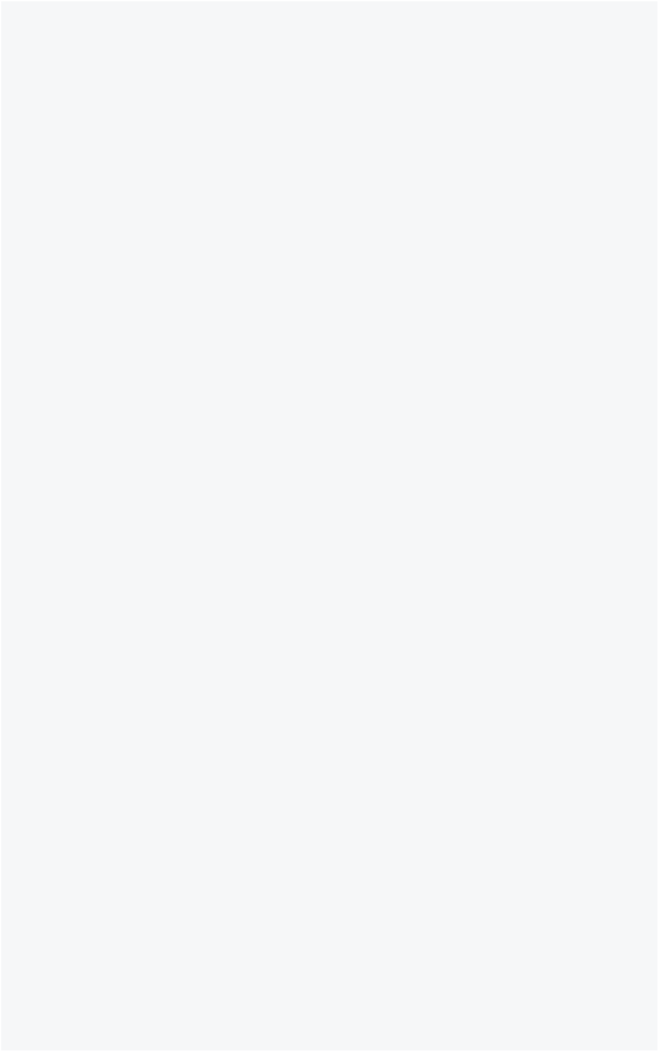
import 'package:flutter/widgets.dart'; import 'package:path/path.dart'; import 'package:sqflite/sqflite.dart';

void main() async {

// Avoid errors caused by flutter upgrade.

// Importing 'package:flutter/widgets.dart' is required.

WidgetsFlutterBinding.ensureInitialized();



// Open the database and store the reference.

final database = openDatabase(

// Set the path to the database. Note: Using the `join` function from the

// `path` package is best practice to ensure the path is correctly // constructed for each platform.

join(await getDatabasesPath(), 'doggie\_database.db'),

// When the database is first created, create a table to store dogs. onCreate: (db, version) {

// Run the CREATE TABLE statement on the database.

return db.execute(

'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',

);

},

// Set the version. This executes the onCreate function and provides a

// path to perform database upgrades and downgrades.

version: 1,

);

// Define a function that inserts dogs into the database Future<void> insertDog(Dog dog) async {

// Get a reference to the database.

final db = await database;

// Insert the Dog into the correct table. You might also specify the // `conflictAlgorithm` to use in case the same dog is inserted twice.

//

// In this case, replace any previous data.

await db.insert(

'dogs',

dog.toMap(),

conflictAlgorithm: ConflictAlgorithm.replace,

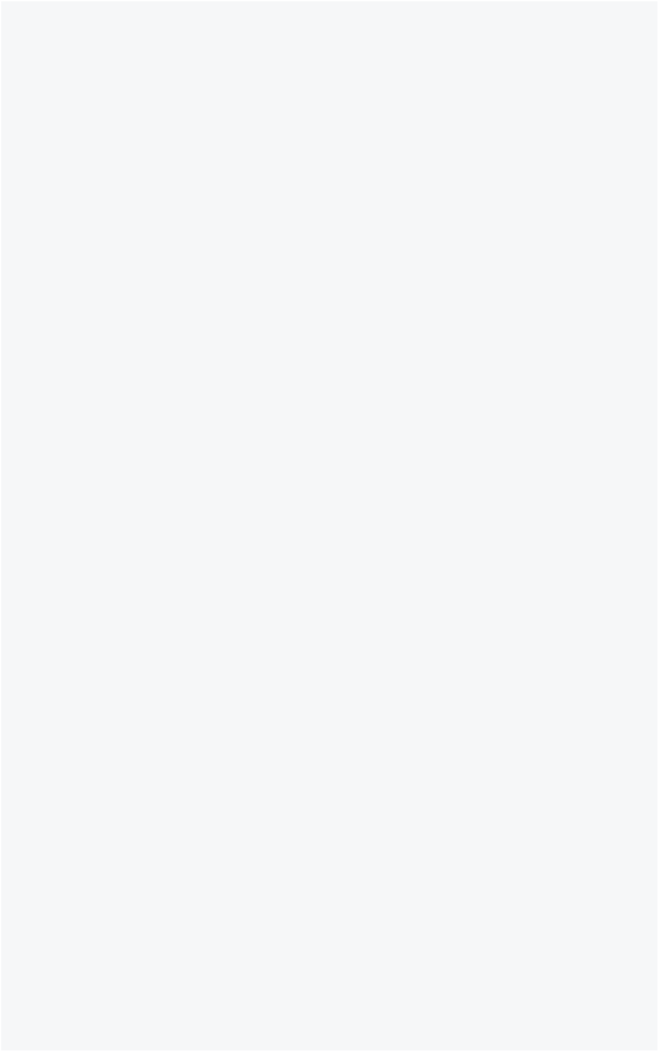
); }

// A method that retrieves all the dogs from the dogs table. Future<List<Dog>> dogs() async {

// Get a reference to the database.

final db = await database;

// Query the table for all the dogs.



final List<Map<String, Object?>> dogMaps = await db.query('dogs');

// Convert the list of each dog's fields into a list of `Dog` objects.

return [

for (final {

'id': id as int,

'name': name as String,

'age': age as int,

} in dogMaps)

Dog(id: id, name: name, age: age),

]; }

Future<void> updateDog(Dog dog) async { // Get a reference to the database. final db = await database;

// Update the given Dog.

await db.update(

'dogs',

dog.toMap(),

// Ensure that the Dog has a matching id.

where: 'id = ?',

// Pass the Dog's id as a whereArg to prevent SQL injection. whereArgs: [dog.id],

);

}

Future<void> deleteDog(int id) async { // Get a reference to the database. final db = await database;

// Remove the Dog from the database.

await db.delete(

'dogs',

// Use a `where` clause to delete a specific dog.

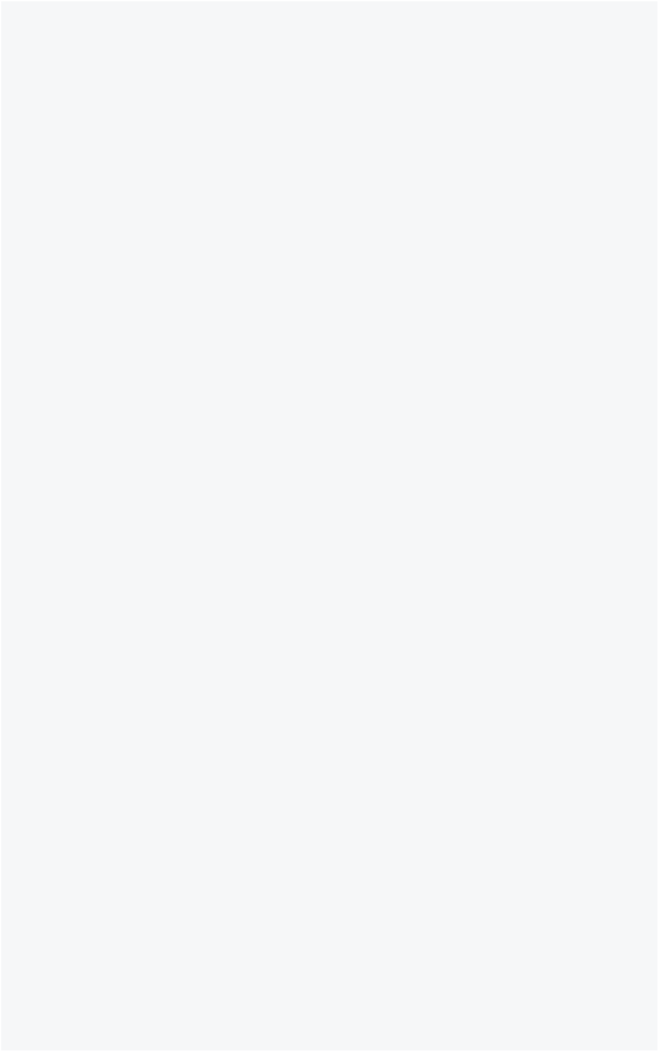
where: 'id = ?',

// Pass the Dog's id as a whereArg to prevent SQL injection. whereArgs: [id],

);

}

// Create a Dog and add it to the dogs table var fido = Dog(



id: 0,

name: 'Fido',

age: 35,

);

await insertDog(fido);

// Now, use the method above to retrieve all the dogs. print(await dogs()); // Prints a list that include Fido.

// Update Fido's age and save it to the database. fido = Dog(

id: fido.id,

name: fido.name,

age: fido.age + 7,

);

await updateDog(fido);

// Print the updated results.

print(await dogs()); // Prints Fido with age 42.

// Delete Fido from the database. await deleteDog(fido.id);

// Print the list of dogs (empty). print(await dogs());

}

class Dog {

final int id;

final String name; final int age;

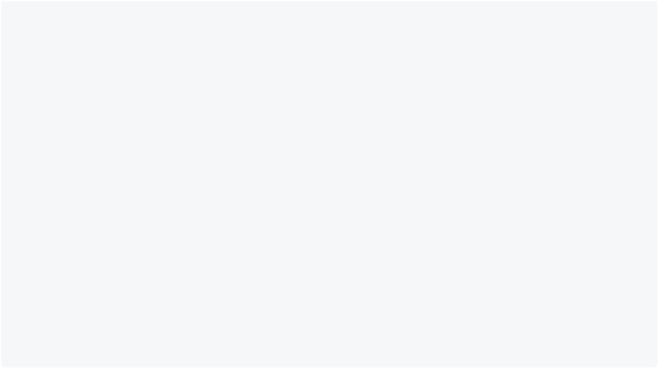
Dog({

required this.id, required this.name, required this.age, });

// Convert a Dog into a Map. The keys must correspond to the names of the

// columns in the database.

Map<String, Object?> toMap() { return {



'id': id,

'name': name,

'age': age,

};

}

// Implement toString to make it easier to see information about // each dog when using the print statement.

@override

String toString() {

return 'Dog{id: $id, name: $name, age: $age}';

}

}

## Networking in Flutter

Networking in Flutter involves making HTTP requests to web APIs to retrieve or send data. The [http](https://pub.com/packages/http) package is a official and popular package in Dart that makes it easy to perform HTTP requests.

**Tips**



### Study HTTP requests and how to make them using the http package

import 'package:http/http.dart' as http;



void main() async {

var response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));

print(response.body);

}

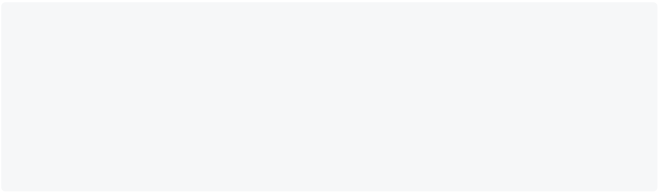
In this example, we import the http package and use the get method to make a GET request to the specified URL. The await keyword is used to wait for the response, which is then printed to the console.



### Learn how to parse JSON data in Dart

Many web APIs return data in JSON format. Dart provides built-in support for parsing JSON data using the dart:convert library.

import 'dart:convert';



void main() {

String jsonData = '{"name": "John", "age": 30}'; Map<String, dynamic> data = jsonDecode(jsonData); print(data['name']);

}

In this example, we have a JSON string representing an object with two properties,

name and age. We use the jsonDecode function to parse the JSON data into a Map object. We can then access the properties of the object using the keys.

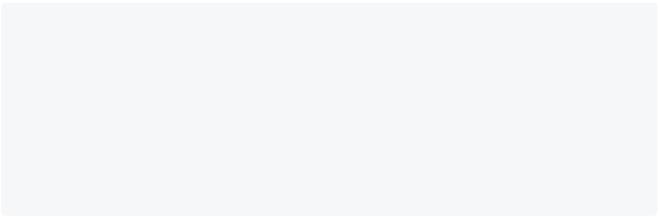


### Use the http package to make GET and POST requests to a web API

To make a POST request using the http package, we can use the post method.



import 'package:http/http.dart' as http;



void main() async {

var url = Uri.parse('https://jsonplaceholder.typicode.com/posts'); var response = await http.post(url, body: {'title': 'foo', 'body': 'bar', 'userId': '1'});

print(response.statusCode);

}

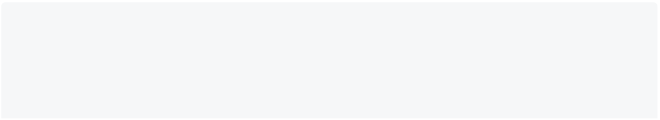
In this example, we make a POST request to the specified URL and pass in a Map object as the body parameter.



### Parse the JSON data returned by the API into Dart objects

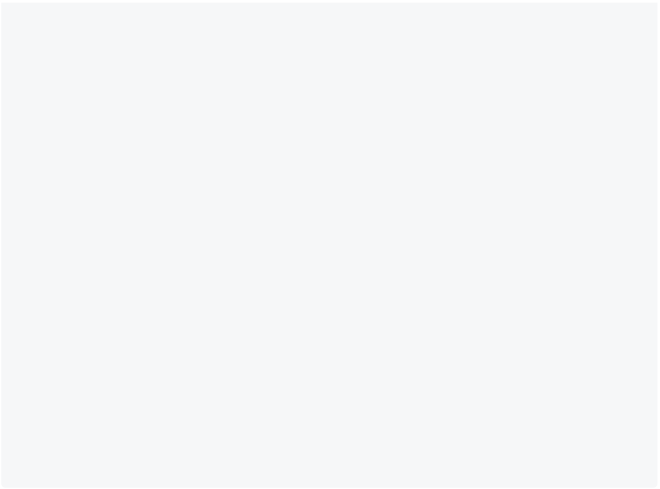
When we receive JSON data from a web API, we often want to convert it into Dart objects for easier manipulation. We can create Dart classes that mirror the structure of the JSON data and then use the jsonDecode function to convert the JSON data into Dart objects.

import 'dart:convert';



class Post { int id;

String title; String body;



Post({this.id, this.title, this.body});

factory Post.fromJson(Map<String, dynamic> json) { return Post(

id: json['id'],

title: json['title'],

body: json['body'],

);

}

}

void main() {

String jsonData = '{"id": 1, "title": "foo", "body": "bar"}'; Map<String, dynamic> data = jsonDecode(jsonData);

Post post = Post.fromJson(data);

print(post.title);

}

In this example, we have a Dart class Post that represents a post object with three properties: id, title, and body. We also have a factory constructor fromJson that takes a JSON object and returns a Post object.



### Resources

* [Beginners Guide to http package](https://www.appwriters.dev/blog/a-beginners-guide-to-the-http-package-in-flutter)
  + [Integrating with REST API in Flutter](https://www.appwriters.dev/blog/integrating-with-rest-api-in-flutter)
    - [Dart HTTP Client package](https://pub.dev/packages/http)
      * [Parsing json in Dart](https://codewithandrea.com/articles/parse-json-dart/)
        + [JSON to Dart converter](https://javiercbk.github.io/json_to_dart/)
      * [Detailed explanation of REST API](https://nerdleveltech.com/a-full-guide-understand-everything-about-apis-with-examples/) - **recommended** if you are new to REST API

[JSONPlaceholder API](https://jsonplaceholder.typicode.com/) - Free placeholder API that you can use to practice

## Using JSON Server

### Guide

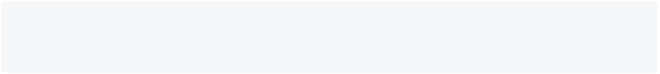
JSON Server is a simple and convenient tool to create a RESTful API using a JSON file as a database.

**Step 1: Install JSON Server**

Make sure you have Node.js installed. If not, download and install it from <https://nodejs.org/>.

Open a terminal and install JSON Server globally:

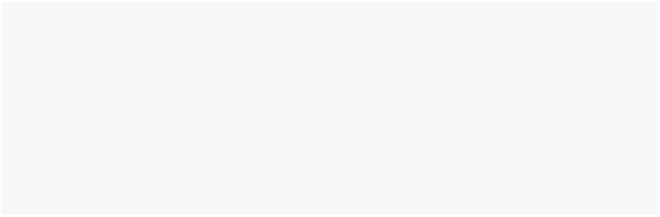
npm install -g json-server



**Step 2: Create a JSON file**

Create a db.json file with some sample data inside your project folder. For example:

{



"posts": [

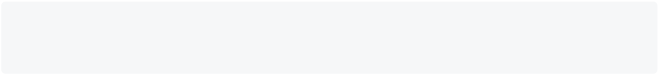
{ "id": 1, "title": "Post 1" }, { "id": 2, "title": "Post 2" }, { "id": 3, "title": "Post 3" } ]

}

**Step 3: Start JSON Server with Host 0.0.0.0**

Run JSON Server with the --host 0.0.0.0 option to allow external access:

json-server --watch db.json --host 0.0.0.0



Your API is now accessible externally. **Step 4: Find Your Local IPv4 Address**

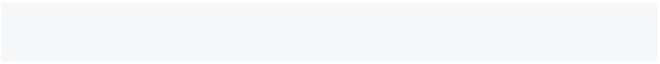
Open a new terminal and run ipconfig (on Windows) or ifconfig (on Linux/Mac). Find your IPv4 address under the network adapter you are connected to.

Example output:

Ethernet adapter Ethernet:



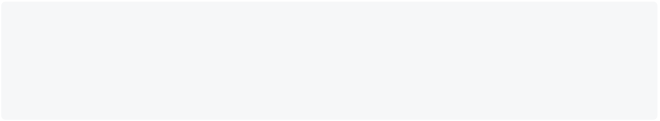
IPv4 Address. . . . . . . . . . . : 192.168.1.2



**Step 5: Update Flutter App to Use Local IPv4 Address**

Update your Flutter app to use the local IPv4 address instead of localhost. For example:

final String baseUrl = 'http://192.168.1.2:3000'; // Replace with your IPv4 address

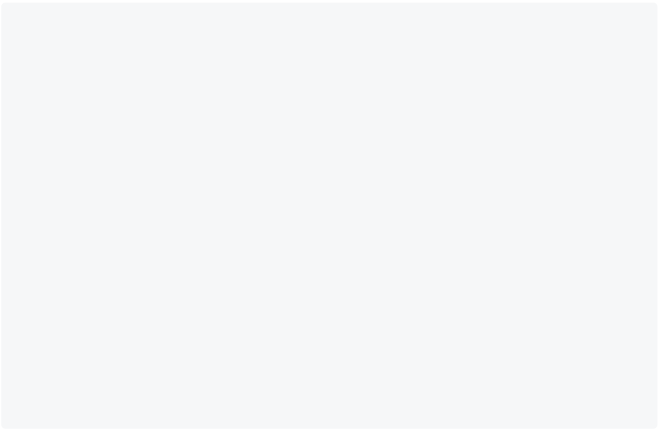


final String postsEndpoint = '/posts';

**Step 6: Use the API in Flutter**

Now, you can make HTTP requests from your Flutter app to the JSON Server API using the updated base URL.

import 'dart:convert';



import 'package:http/http.dart' as http;

final String baseUrl = 'http://192.168.1.2:3000'; // Replace with your IPv4 address

final String postsEndpoint = '/posts';

Future<List<Map<String, dynamic>>> fetchPosts() async {

final response = await http.get(Uri.parse('$baseUrl$postsEndpoint'));

if (response.statusCode == 200) {

return json.decode(response.body).cast<Map<String, dynamic>>(); } else {

throw Exception('Failed to load posts');

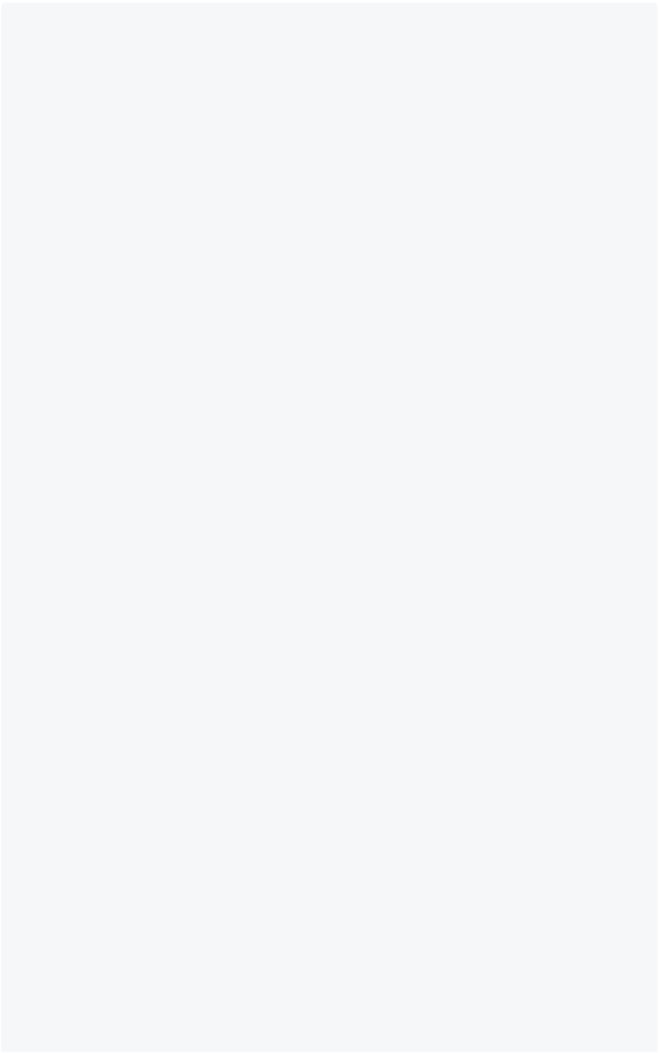
}

}

That's it! You've successfully set up JSON Server with external access and integrated it into your Flutter app using your local IPv4 address

### Complete Example

import 'dart:convert';



import 'package:flutter/material.dart'; import 'package:http/http.dart' as http;

void main() {

runApp(MyApp()); }

class MyApp extends StatelessWidget { @override

Widget build(BuildContext context) { return MaterialApp(

title: 'JSON Server Example',

theme: ThemeData(

primarySwatch: Colors.blue,

),

home: HomePage(),

);

}

}

class Post {

int id;

String title;

Post({this.id, this.title});

factory Post.fromJson(Map<String, dynamic> json) { return Post(

id: json['id'],

title: json['title'],

);

}

}

class HomePage extends StatefulWidget {

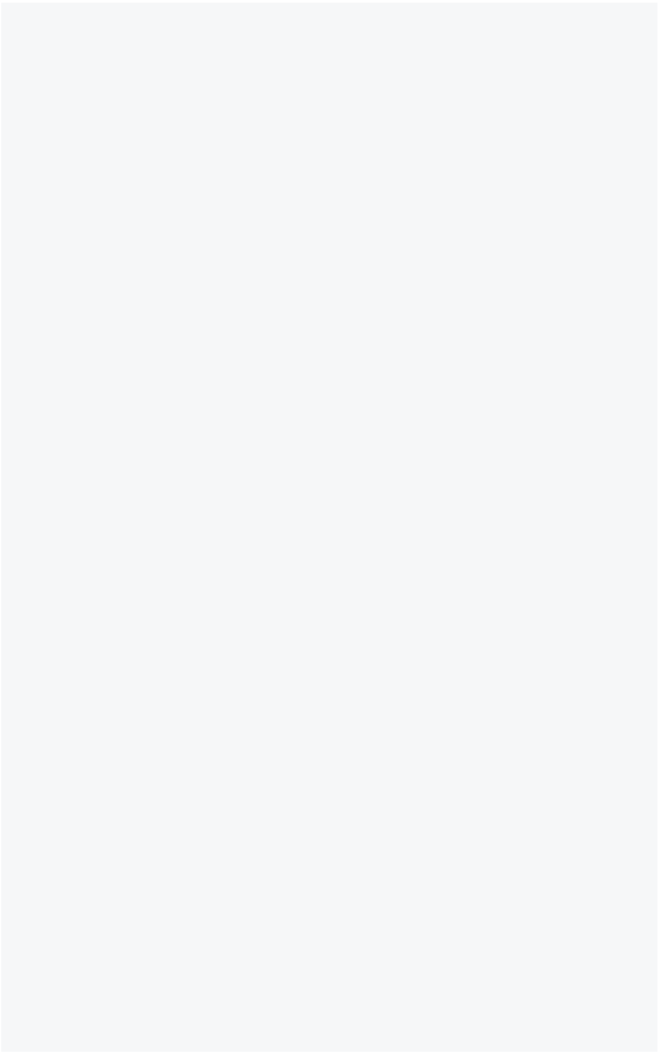
@override

\_HomePageState createState() => \_HomePageState(); }

class \_HomePageState extends State<HomePage> {

final String baseUrl = 'http://192.168.1.2:3000'; // Replace with your IPv4 address

final String postsEndpoint = '/posts';



Future<List<Post>> fetchPosts() async {

final response = await http.get(Uri.parse('$baseUrl$postsEndpoint'));

if (response.statusCode == 200) {

List<dynamic> data = json.decode(response.body);

return data.map((json) => Post.fromJson(json)).toList(); } else {

throw Exception('Failed to load posts');

}

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('JSON Server Example'), ),

body: FutureBuilder(

future: fetchPosts(),

builder: (context, AsyncSnapshot<List<Post>> snapshot) {

if (snapshot.connectionState == ConnectionState.waiting) { return Center(child: CircularProgressIndicator());

} else if (snapshot.hasError) {

return Center(child: Text('Error: ${snapshot.error}')); } else if (!snapshot.hasData || snapshot.data!.isEmpty) { return Center(child: Text('No posts available.'));

} else {

return ListView.builder(

itemCount: snapshot.data!.length,

itemBuilder: (context, index) {

final post = snapshot.data![index];

return ListTile(

title: Text(post.title),

subtitle: Text('ID: ${post.id}'),

);

},

);

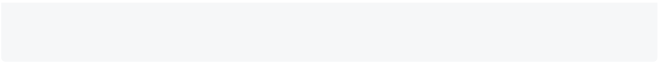
}

},

),

);

} }



## State Management in Flutter

### Using GetX

State management is a complex topic of discussion in Flutter. However, many state management libraries, such as [Provider](https://blog.logrocket.com/quick-guide-provider-flutter-state-management/), are available, which most developers recommend.

But…

Today, we will discuss a simplified state management solution for Flutter application development that does not require context for most of its features, known as GetX.

**What is GetX?**



GetX is not only a state management library, but instead, it is a microframework combined with route management and dependency injection. GetX has three basic principles on which it is built:

1. Performance: focused on minimum consumption of memory and resources
2. Productivity: intuitive and efficient tool combined with simplicity and straightforward syntax that ultimately saves development time
3. Organization: decoupling business logic from view and presentation logic cannot get better than this. You do not need context to navigate between routes, nor do you need stateful widgets

**The three pillars of GetX**



1. State management: GetX has two state managers. One is a simple state manager used with the GetBuilder function, and the other is a reactive state manager used with Getx or Obx. We will be talking about it in detail below



1. Route management: whether navigating between screens, showing SnackBars, popping dialog boxes, or adding bottom sheets without the use of context, GetX has you covered. I will not write details on route management because it is beyond the scope of this article, but indeed a few examples to get an idea of how GetX syntax simplicity works
2. Dependency management: GetX has a simple yet powerful solution for dependency management using controllers. With just a single line of code, it can be accessed from the view without using an inherited widget or context. Typically, you would instantiate a class within a class, but with GetX, you are instantiating with the Get instance, which will be available throughout your application



**Value-added features of GetX**



GetX has some great features out of the box, making it even easier to develop mobile applications in Flutter without any boilerplate code:

1. Internationalization: translations with key-value maps, various language support, using translations with singulars, plurals, and parameters. Changing the application’s locale using only the Get word throughout the app



1. Validation: email and password validations are also covered by GetX. Now you do not need to install a separate validation package
2. Storage: GetX also provides fast and extra light synchronous key-value memory backup of data entirely written in Dart that easily integrates with the GetX core package
3. Themes: switching between light and dark themes is made simple with GetX
4. Responsive view: if you are building an application for different screen sizes, you just need to extend with GetView, and you can quickly develop your UI, which will be responsive for desktop, tablet, phone, and watch



**Let’s get going with GetX state management**



**Step 1: Create a new application**

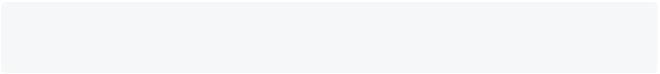
Create a brand new application in your preferred IDE. First, remove all the starter comments by selecting the find and replace option in the **Edit** menu and type this:

\/\/.\*. This will select Flutter’s comments in the starter code, and you can just hit the delete button.

**Step 2: Add required dependencies**

Add these dependencies in your pubspec.yaml file:

get: ^4.6.1 //YAML get\_storage: ^2.0.3 //YAML



Run this command:

flutter pub get //YAML



Before going on to Step 3, let me explain what we are building here. The application is about a store where the user can:

1. change the name of the store
2. add follower names
3. add follower count
4. change the status of the store from open to closed and vice versa
5. add reviews to the store
6. change the theme of the store from light to dark

All of the above will explain state management, dependency management, route management, storage, and themes.

You can read along and [test the application through this link](https://getx-store.web.app/#/).

**Step 3: Update the MaterialApp Widget**



After adding the dependencies, the first thing you need to do is change the

MaterialApp widget to GetMaterialApp in your main.dart file. This gives access to all GetX properties across the application.

**Step 4: Add GetX Controller**

We have already established that GetX separates the UI from the business logic. This is where GetX Controller comes into play.

You can always create more than one controller in your application. The GetX Controller class controls the state of the UI when you wrap an individual widget with its Observer so that it only rebuilds when there is a change in the state of that particular widget.

We are adding a new Dart file to create our controller class, StoreController, which extends GetxController:

class StoreController extends GetxController {}



Next, we add a few variables and initialize them with default values. Normally we would add these variables like this as given below:

final storeName = 'Thick Shake';



But, when using GetX, we have to make the variables observable by adding **obs** at the end of value. Then when the variable changes, other parts of the application that depend on it will be notified about it. So now, our initialized value will look like this:

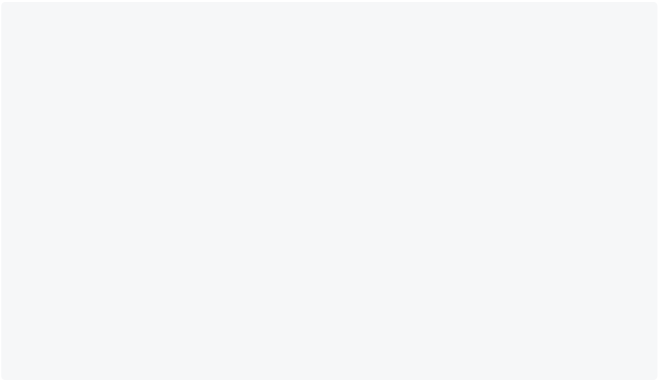


final storeName = 'Thick Shake'.obs;



The rest of the variables are given below:

// String for changing the Store Name



final storeName = 'Thick Shake'.obs;

// int for increasing the Follower count

final followerCount = 0.obs;

// bool for showing the status of the Store open or close final storeStatus = true.obs;

// List for names of Store Followers

final followerList = [].obs;

// Map for Names and their Reviews for the Store

final reviews = <StoreReviews>[].obs;

// text editing controllers

final storeNameEditingController = TextEditingController(); final reviewEditingController = TextEditingController(); final followerController = TextEditingController();

final reviewNameController = TextEditingController();

Next, we create three methods for changing the name, increasing the follower count, and changing the store status:

updateStoreName(String name) { storeName(name);



}

updateFollowerCount() {

followerCount(followerCount.value + 1); }

void storeStatusOpen(bool isOpen) { storeStatus(isOpen);

}

**Step 5: Dependency injection**

In layman’s terms, we add the controller class we just created into our view class. There are three ways to instantiate.



1. Extending the whole view class with GetView and injecting our StoreController with it:



class Home extends GetView<StoreController>{}



1. Instantiating the storeController like this:

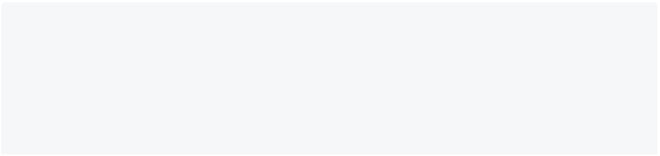
final storeController = Get.put(StoreContoller())



1. For option three, start by creating a new StoreBinding class and implementing Bindings. Inside its default dependencies, you need to lazyPut the StoreController by using Get.lazyPut(). Secondly, you need to add the binding class inside the initialBinding property in GetMaterialWidget.

Lastly, instead of Get.Put as mentioned above, now you can use Get.find and GetX will find your controller for you when you instantiate in any of your classes:

class StoreBinding implements Bindings { // default dependency

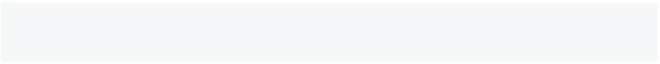


@override

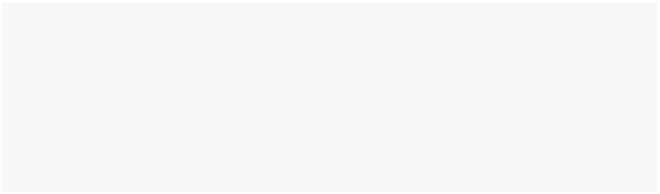
void dependencies() {

Get.lazyPut(() => StoreController();

} }



@override



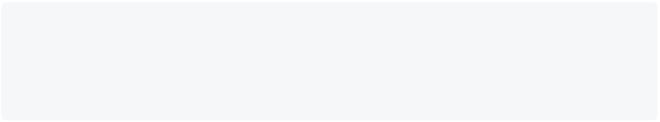
Widget build(BuildContext context) { return GetMaterialApp(

debugShowCheckedModeBanner: false, title: 'GetX Store',

initialBinding: StoreBinding(),

}

class UpdateStoreName extends StatelessWidget {



UpdateStoreName({Key? key}) : super(key: key); //Getx will find your controller.

final storeController = Get.find<StoreController>();

There are a lot of code and Dart files in the project. I am only writing about the three methods that I have mentioned above. The rest of the code will be available on Git. The link will be provided at the end of this article.

**Step 6: Instantiate Controller**

Since we have extended our Home view with GetView and created a binding class to lazyPut our controller inside it, we will now use Get.find to instantiate our controller inside our classes.

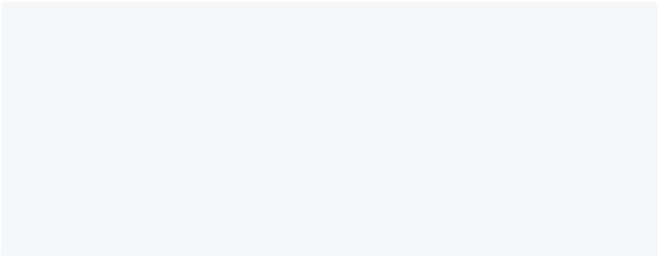


First, we add a new stateless widget, UpdateStoreName. Instantiate our controller class like this:

final storeController = Get.find<StoreController>();



RoundedInput(



hintText: "Store Name",

controller: storeController.storeNameEditingController, ),

const SizedBox(height: 20),

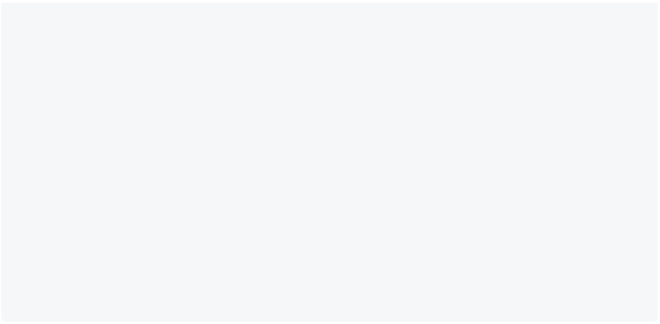
ElevatedButton(

onPressed: () {

storeController.updateStoreName(

storeController.storeNameEditingController.text); Get.snackbar(

'Updated',



'Store name has been updated ton '

'${storeController.storeNameEditingController.text}', snackPosition: SnackPosition.BOTTOM);

},

child: const Padding(

padding: EdgeInsets.all(10.0),

child: Text(

'Update',

style: TextStyle(fontSize: 20.0),

),

),

),

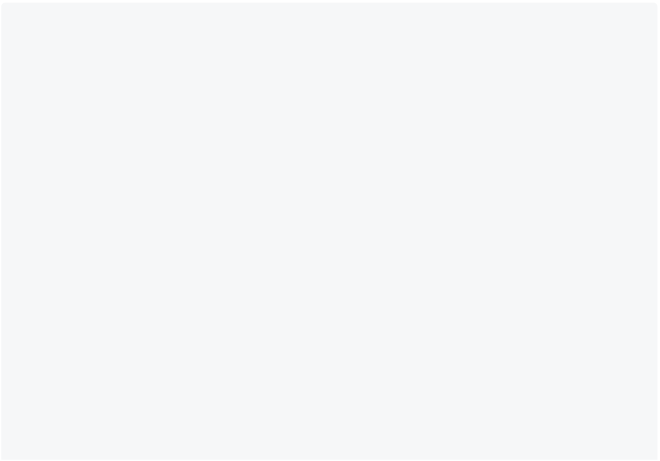
Let me explain the above code: RoundedInput is just a custom TextField, and we are adding a TextEditingController for the TextField using our

storeController. We are also calling the updateStoreName() method in the same way inside the onPressed of ElevatedButton. And then we are showing a

SnackBar as a confirmation that the store name has been updated.

Below is the code for AddFollowerCount and StoreStatus. Again both are stateless widgets, and the method of implementing the storeController and calling our controller is similar:

class AddFollowerCount extends StatelessWidget {



AddFollowerCount({Key? key}) : super(key: key);

final storeController = Get.find<StoreController>();

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text("Add Follower Count")), floatingActionButton: FloatingActionButton(

onPressed: () {storeController.updateFollowerCount(); },

child: const Icon(Icons.add),

),

body: Container(

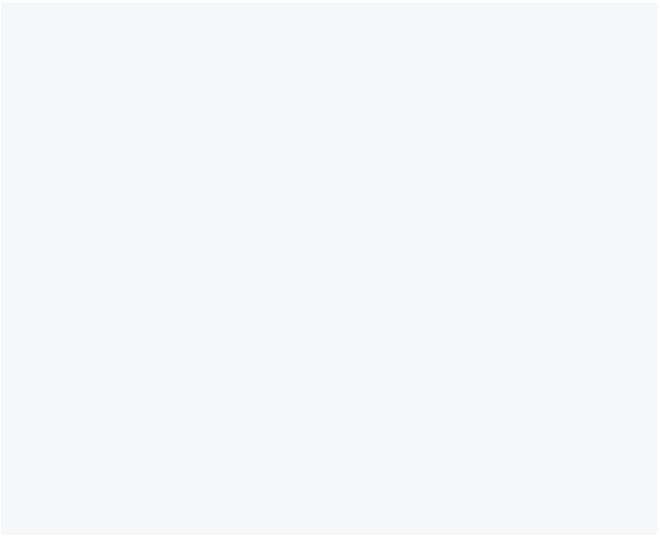
padding: const EdgeInsets.all(24),

child: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [



const Text(

'You have add these many followers to your store', textAlign: TextAlign.center,

style: TextStyle(fontSize: 28),

),

const SizedBox(

height: 40.0,

),

Obx(

() => Text(

storeController.followerCount.value.toString(), style: const TextStyle(fontSize: 48),

),

)

],

),

),

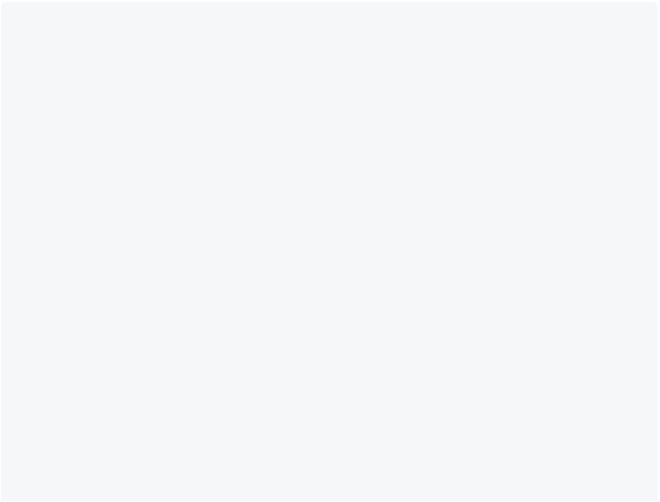
),

);

}

}

class StoreStatus extends StatelessWidget {



StoreStatus({Key? key}) : super(key: key);

//final storeController = Get.put(StoreController()); final storeController = Get.find<StoreController>();

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text("Test Status Toggle")), body: Container(

padding: const EdgeInsets.all(24),

child: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

const Text(

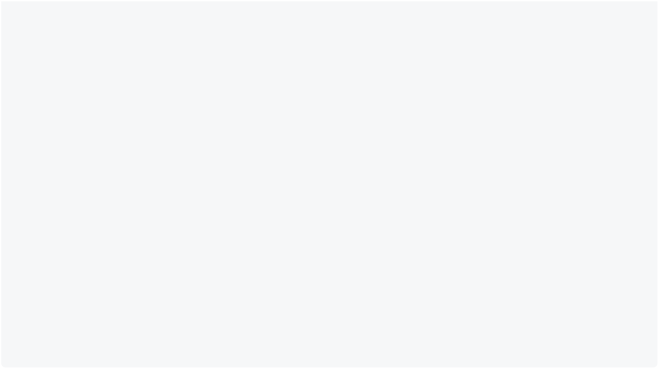
"Is the Store open?",

style: TextStyle(fontSize: 22),

),

const SizedBox(height: 16),

Obx(



() => Switch(

onChanged: (value) => storeController.storeStatus(value),

activeColor: Colors.green,

value: storeController.storeStatus.value, ),

)

],

),

),

),

);

}

}

**Step 7: Obx Widget (Observer)**



Now, let us get to the part where the entered value of our store name, increased count of followers, and store status will be shown using our storeController.

Our Home view is extended with GetView<StoreController>, so we do not need to instantiate our storeController here. Instead, we can just use GetX’s default controller. Please look at the code given below to get a clear picture and understand the difference between Step 6 and Step 7.



You must have noticed that the Text widget inside the Flexible widget is wrapped with an Obx widget where we have also called our controller. Remember how we added (.obs) to our variables? Now, when we want to see the change in that observable variable, we have to wrap the widget with Obx, also known as Observer, similar to what you must have noticed in the above code.



Wrapping the widget with Obx will only rebuild that particular widget and not the whole class when the state changes. This is how simple it is:



class Home extends GetView<StoreController> { Home({Key? key}) : super(key: key);



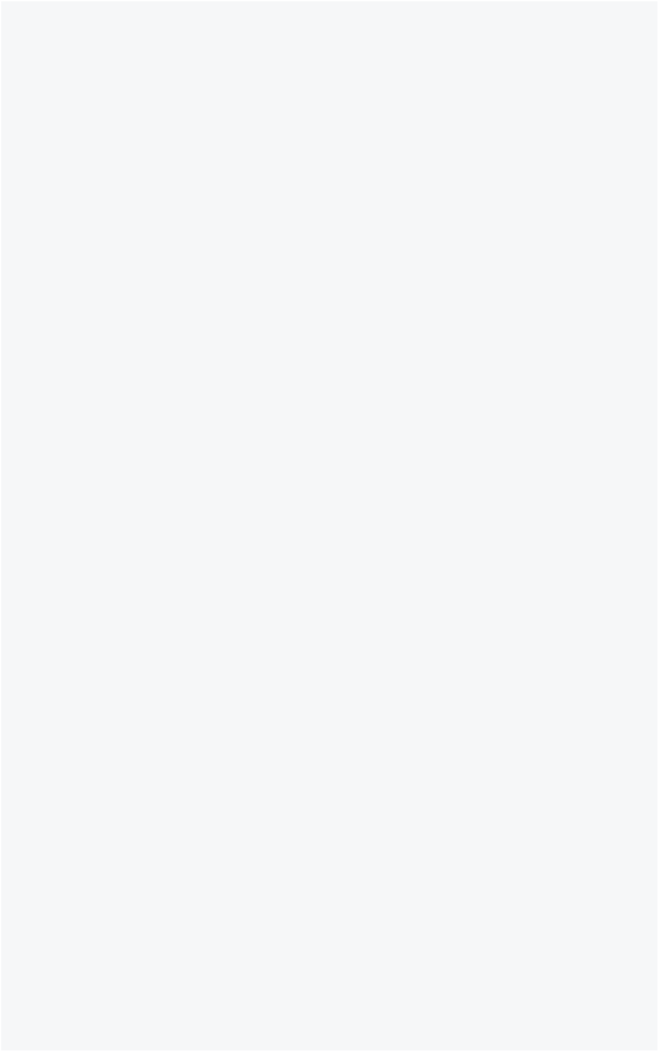
@override

Widget build(BuildContext context) {

return Scaffold(

backgroundColor: AppColors.spaceCadet, appBar: AppBar(

title: const Text("GetX Store"),), drawer: const SideDrawer(),



body: Container(

padding: const EdgeInsets.all(10), child: SingleChildScrollView(

child: Column(

children: [

MainCard(

title: "Store Info",

body: Column(

crossAxisAlignment: CrossAxisAlignment.stretch,

children: [

Row(

mainAxisAlignment: MainAxisAlignment.spaceBetween, children: [

const Flexible(

child: Text('Store Name:',

style: TextStyle(fontSize: 20),),

fit: FlexFit.tight,),

const SizedBox(width: 20.0),

// Wrapped with Obx to observe changes to the

storeName

// variable when called using the StoreController.

Obx(

() => Flexible(

child: Text(

controller.storeName.value.toString(),

style: const TextStyle(

fontSize: 22, fontWeight: FontWeight.bold) ),

fit: FlexFit.tight,

),),

],),

const SizedBox(height: 20.0),

Row(

mainAxisAlignment: MainAxisAlignment.spaceBetween, children: [

const Flexible(

child: Text('Store Followers:',

style: TextStyle(fontSize: 20),),

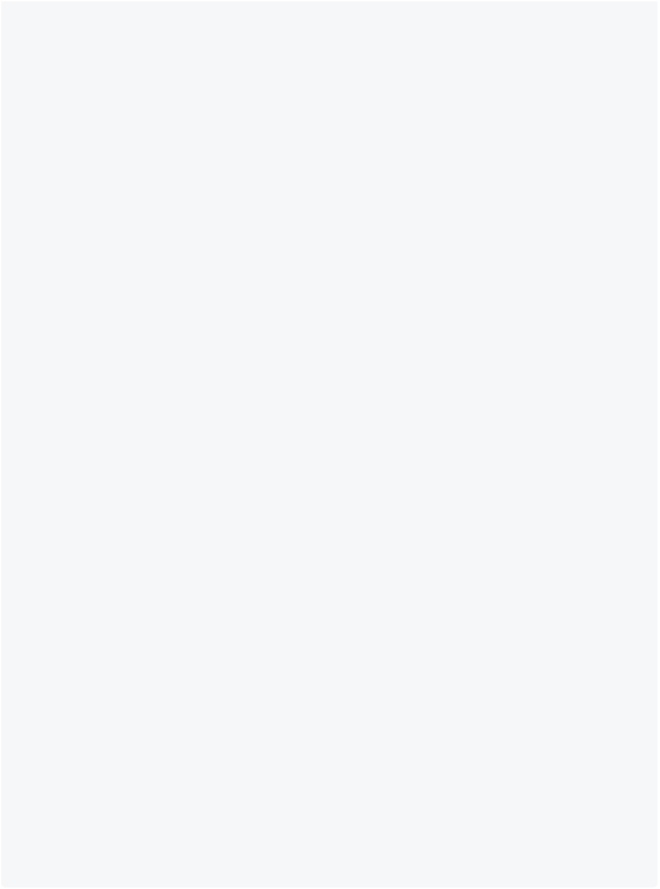
fit: FlexFit.tight, ),

const SizedBox(width: 20.0),

// Wrapped with Obx to observe changes to the

followerCount

// variable when called using the StoreController.



Obx(

() => Flexible(

child: Text(

controller.followerCount.value.toString(), textAlign: TextAlign.start,

style: const TextStyle(

fontSize: 22, fontWeight: FontWeight.bold), ),

fit: FlexFit.tight,),), ],

),

const SizedBox(height: 20.0),

Row(

mainAxisAlignment: MainAxisAlignment.spaceBetween, children: [

const Flexible(

child: Text('Status:',

style: TextStyle(fontSize: 20),),

fit: FlexFit.tight,),

const SizedBox(width: 20.0),

// Wrapped with Obx to observe changes to the

storeStatus

// variable when called using the StoreController. Obx(

() => Flexible(

child: Text(

controller.storeStatus.value ? 'Open' : 'Closed',

textAlign: TextAlign.start,

style: TextStyle(

color: controller.storeStatus.value

* Colors.green.shade700
* Colors.red,

fontSize: 22,

fontWeight: FontWeight.bold),),

fit: FlexFit.tight,

), ), ], ), ], ), ),

I have purposely highlighted the controllersand Obx to understand the difference between a default stateful widget provided by Flutter and using GetX for managing the state of a view or an entire application.

If we were using a stateful widget, we would have to use the setState() method every time we wanted to see changes. We would also have to dispose of

controllersmanually. So instead, we avoid all the boilerplate code and just wrap our widget with Obx, and the rest is taken care of.



If we had to summarize all the above, it could be done in only two steps:

1. Add obs to your variable



1. Wrap your widget with Obx

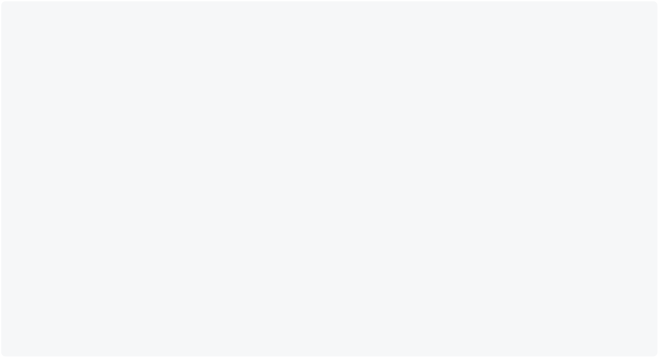


**An alternative method**

Well, that is not the only way to do it. For example, if you make your variables observable, you can also wrap the widget with GetX<StoreController> directly instead of Obx. However, the functionality remains the same. This way, you do not need to instantiate the storeController before it can be called. Please look at the code below:



// Wrapped with GetX<StoreController> to observe changes to the //storeStatus variable when called using the StoreController. GetX<StoreController>(



builder: (sController) => Flexible(

child: Text(

sController.storeStatus.value ? 'Open' : 'Closed',

textAlign: TextAlign.start,

style: TextStyle(

color: sController.storeStatus.value

* Colors.green.shade700
* Colors.red,

fontSize: 22,

fontWeight: FontWeight.bold), ),

fit: FlexFit.tight, ),),

*N.B., I have changed the storeStatus from Obx to GetX<StoreController> and it is using sController from the building function.*



Wrapping the widgets with Obx or GetX is known as reactive state management.



**Simple state management**

Let us see an example for simple state management. First, the advantage of using simple state management is that you do not need to change your MaterialWidget

to GetMaterialWidget. Secondly, you can combine other state management libraries with simple state management.

*N.B., if you do not change your MaterialWidget* *to GetMaterialWidget, you will not be able to use other GetX features such as route management.*

For simple state management:

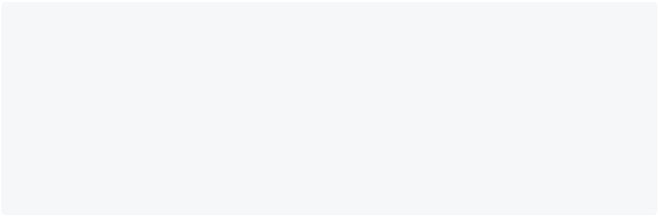
1. you need to use the GetBuilder function
2. you do not need observable variables
3. you have to call the update() function in your method

I have created a new variable in our StoreController. But this time, I have not added (obs) at the end of the variable. It means now it is not observable.



But I still need my view to get updated when the store count increases, so I have to call the update() function inside my newly created method. Check the code below:

// variable is not observable int storeFollowerCount = 0;



void incrementStoreFollowers() {

storeFollowerCount++;

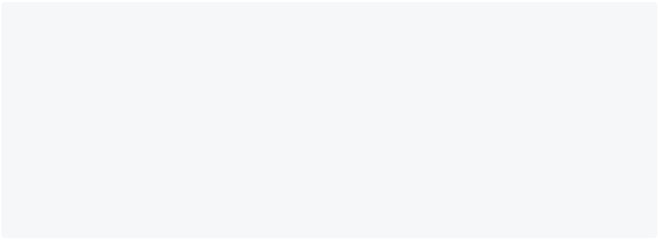
//update function needs to be called update();

}

Now, in our **Home** view I have changed Obx to GetBuilder to the Text widget, which displays the follower count:



GetBuilder<StoreController>(



builder: (newController) => Flexible(

child: Text(

newController.storeFollowerCount.toString(),

textAlign: TextAlign.start,

style: const TextStyle(

fontSize: 22, fontWeight: FontWeight.bold), ),

fit: FlexFit.tight, ),),

Since we are wrapping our follower count with GetBuilder in our **Home** view, we also have to make changes to the AddFollowerCount Dart file.

1. Add this inside the onPressed function in the Fab button:



```dart



storeController.incrementStoreFollowers();

1. Wrap the `Text` widget with `GetBuilder` as well so that it



displays the follower count:

```dart

GetBuilder<StoreController>(

builder: (newController) => Text(

'With GetBuilder: ${newController.storeFollowerCount.toString()}', textAlign: TextAlign.start,

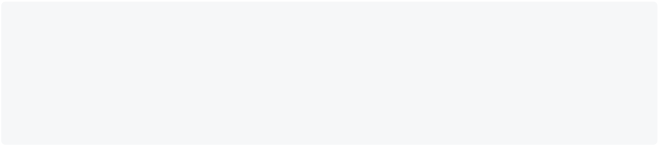
style: const TextStyle(

fontSize: 22, fontWeight: FontWeight.bold), ),),

There is one more difference between using Obx or GetX and using GetBuilder. When using Obx or GetX, you need to add value after calling your method using the StoreController. But when using GetBuilder, you do not need to add a value parameter to it. Please look at the difference below:



// value parameter to be added with Obx or GetX controller.storeName.value.toString(),



// value parameter is not needed with GetBuilder newController.storeFollowerCount.toString(),

That is all for different state managements provided by GetX. Furthermore, as promised, I am writing a little about route management and other features of the GetX package. Hence, a whole new article is needed to write in detail about it all.

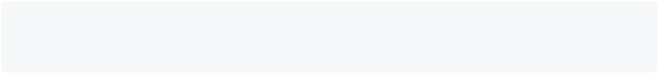
**Other GetX features**



**Route management**

Traditionally, when a user wants to go from one screen to another with a click of a button, code would look like this:

Navigator.push(context,



MaterialPageRoute(builder: (context)=> Home()));

But, with GetX, there are literally just two words:

Get.to(Home());



When you want to navigate back to your previous screen:

Navigator.pop(context);



There is absolutely no need for context when you are using GetX:

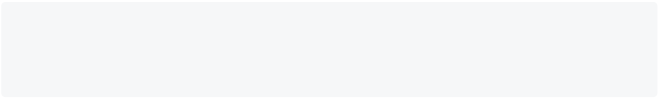
Get.back();



If you have a dialog or a drawer opened and you want to navigate to another screen while closing the drawer or dialog, there are two ways to do this with default Flutter navigation:

1. Close the drawer or dialog and then navigate like this:

Navigator.pop(context);



Navigator.push(context,

MaterialPageRoute(builder: (context)=> SecondScreen()));

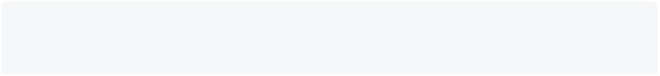
1. If you have named routes generated:

Navigator.popAndPushNamed(context, '/second');



With GetX, it gets a lot simpler to generate named routes and navigate between screens while closing any dialogs or drawers that are open:

// for named routes Get.toNamed('/second'),



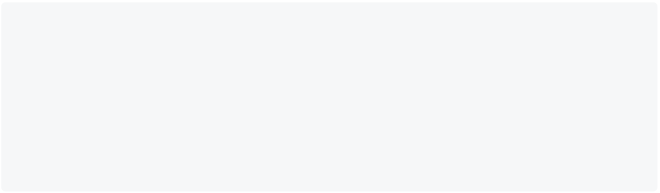
// to close, then navigate to named route Get.offAndToNamed('/second'),



**Value-added features**

1. Snackbars

Get.snackbar(



'title',

'message',

snackPosition: SnackPosition.BOTTOM, colorText: Colors.white, backgroundColor: Colors.black, borderColor: Colors.white);

1. Dialogs

Get.defaultDialog(



radius: 10.0,

contentPadding: const EdgeInsets.all(20.0), title: 'title',

middleText: 'content',

textConfirm: 'Okay',

confirm: OutlinedButton.icon(

onPressed: () => Get.back(),

icon: const Icon(

Icons.check,

color: Colors.blue, ),

label: const Text('Okay',

style: TextStyle(color: Colors.blue),

), ),

cancel: OutlinedButton.icon( onPressed: (){},

icon: Icon(),

label: Text(),),);

1. Bottom sheets

Get.bottomSheet(



Container(

height: 150,

color: AppColors.spaceBlue,

child: Center(



child: Text(

'Count has reached ${obxCount.value.toString()}',

style: const TextStyle(fontSize: 28.0, color: Colors.white), )),

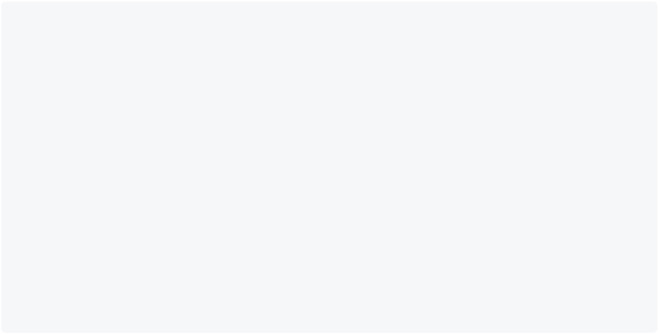
));

Looking at the above code, you can easily understand how simple it is to show and customize snackbars, dialogs, and bottom sheets.

**Switching from light to dark themes and vice versa**

First, I created a ThemeControllersimilar to our StoreController. Inside my controller, I am using the GetStorage function to save the switched theme:

class ThemeController extends GetxController { final \_box = GetStorage();



final \_key = 'isDarkMode';

ThemeMode get theme => \_loadTheme() ? ThemeMode.dark : ThemeMode.light;

bool \_loadTheme() => \_box.read(\_key) ?? false;

void saveTheme(bool isDarkMode) => \_box.write(\_key, isDarkMode); void changeTheme(ThemeData theme) => Get.changeTheme(theme);

void changeThemeMode(ThemeMode themeMode) => Get.changeThemeMode(themeMode);

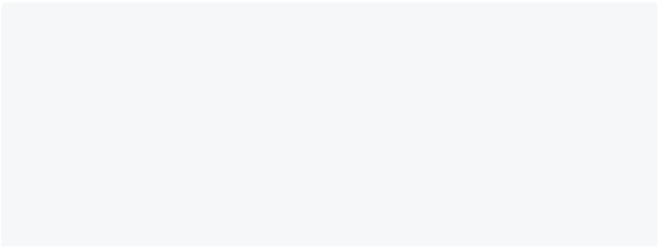
}

Inside the GetMaterialApp widget, I have added properties for theme and



darkTheme as well as initialized themeController and added the same to the themeMode property:

class MyApp extends StatelessWidget {



MyApp({Key? key}) : super(key: key);

final themeController = Get.put(ThemeController());

@override

Widget build(BuildContext context) { return GetMaterialApp(

debugShowCheckedModeBanner: false, title: 'GetX Store',

initialBinding: StoreBinding(), theme: Themes.lightTheme,



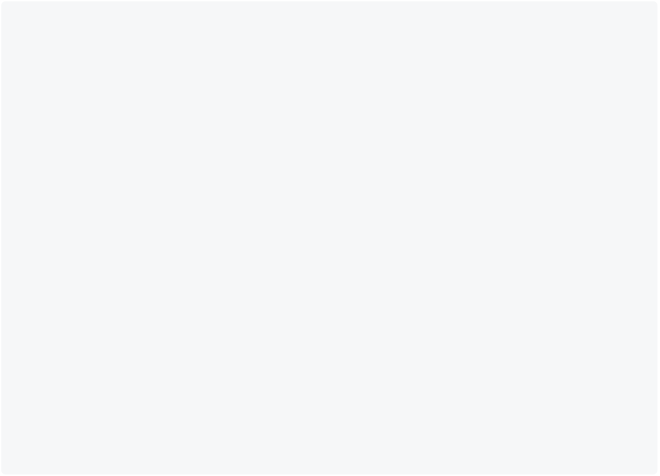
darkTheme: Themes.darkTheme,

themeMode: themeController.theme, }

}

Next, in our **Home** screen in the appBar, I have added an icon that switches the theme between light and dark. Just have a look at the code below:

class Home extends GetView<StoreController> {



Home({Key? key}) : super(key: key);

final themeController = Get.find<ThemeController>();

@override

Widget build(BuildContext context) {

return Scaffold(backgroundColor: AppColors.spaceCadet,

appBar: AppBar(title: const Text("GetX Store"),

actions: [IconButton(

onPressed: () {

if (Get.isDarkMode) {

themeController.changeTheme(Themes.lightTheme); themeController.saveTheme(false);

} else {

themeController.changeTheme(Themes.darkTheme); themeController.saveTheme(true); }},

icon: Get.isDarkMode

* const Icon(Icons.light\_mode\_outlined)
* const Icon(Icons.dark\_mode\_outlined),),], ),

And that’s it. Now you can easily switch between light and dark themes. **Links to the source code on GitHub**

GetX store link: <https://github.com/timelessfusionapps/getx_store>

GetX counter link: <https://github.com/timelessfusionapps/getx_counter>**Links to the web app**

GetX store link: <https://getx-store.web.app/#/>

GetX counter app: <https://getx-counter.web.app/#/>

### Using BLOC

When working on a Flutter app, you might encounter the need to split a large UI component into several smaller ones to improve the readability of the code. With multiple components, it’s crucial to implement effective communication between them. All UI components should be aware of the state of the app at all times. This is called state management.

In Flutter, you can manage the state of your app just by using setState. But while

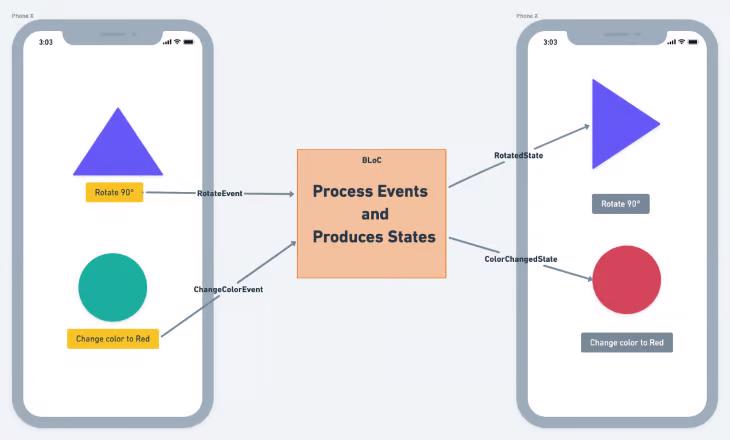
setState can be your best friend, it’s not a good idea to depend on it solely. There are many other factors you should also consider while developing a Flutter app, such as architecture, scalability, readability, complexity, etc. Staying on top of everything requires an effective state management technique.

**What is BLoC?**



Business logic components (BLoC) allow you to separate the business logic from the UI. Writing code in BLoC makes it easier to write and reuse tests.

In simple terms, BLoC accepts a stream of events, processes the data based on events, and produces the output as states. Take the simple example below:



As soon as the **Rotate 90°** button is clicked, the RotateEvent is dispatched to BLoC and the state representing the rotation, i.e. RotatedState, is emitted. The triangle widget rotates itself upon receiving the RotatedState from the BLoC. Similarly, the circle widget changes its color when the **Change color to Red** button is clicked.

Since the BLoC handles the rotation and changing color operation, both operations can be performed on any widget. This facilitates the reusability of the code.

**Advantages of BLoC Pattern:**



* **Excellent Documentation**: BLoC benefits from a rich and well-maintained documentation base. Flutter's official documentation provides comprehensive information, tutorials, and examples that guide developers in effectively implementing BLoC in their applications.
  + **Separation of Concerns**: BLoC enforces a clear separation of concerns by isolating the business logic from the UI layer. This results in code that is more organized, easier to manage, and adaptable to changes.
    - **Testability**: BLoC facilitates efficient testing of application components. Business logic, being separate, can be thoroughly unit tested without the need to consider the UI. This contributes to robust and bug-free code.
      * **State Management**: BLoC excels at managing the application's state. It offers a structured approach to handle various states of the application, making it easier to track and manage how the application behaves in different scenarios.
        + **Community Support**: Being a widely adopted pattern, BLoC has a large and active community. Developers can seek help, share knowledge, and access numerous resources, including libraries and packages, to enhance their BLoC implementation.

**Disadvantages of BLoC Pattern:**



**Steep Learning Curve**: Adopting BLoC requires a learning curve, especially for developers new to the reactive programming paradigm and state management patterns. Understanding streams, sinks, and the principles of reactive programming is crucial.



* **Not Recommended for Simple Applications**: For simple applications with limited business logic, implementing BLoC might introduce unnecessary complexity. It's essential to assess the project requirements and choose an appropriate state management approach.
  + **Boilerplate Code**: Implementing BLoC can involve writing boilerplate code, particularly when done manually. While tools and extensions can mitigate this, there's still a need for careful structuring and organization.

**Important BLoC concepts**



Before we dive in, let’s review some basic BLoC concepts and terms so we’re all on the same page.

**Events**

Events tell BLoC to do something. An event can be fired from anywhere, such as from a UI widget. External events, such as changes in network connectivity, changes in sensor readings, etc., look like this:

class RotateEvent { final double angle;



const RotateEvent(this.angle);

@override

List<Object> get props => [angle]; }

**BLoC**

BLoC is a man in the middle. All the business logic sits inside the BLoC file. It simply accepts events, performs the logic, and outputs the states. Here’s how it looks:

class TransformationBloc



extends Bloc<TransformationEvent, TransformationState> { TransformationBloc() : super(RotatedState(angle: 0);

@override

Stream<TransformationState> mapEventToState( TransformationEvent event) async\* {



if (event is RotateEvent) {

yield RotatedState(angle: event.angle); }

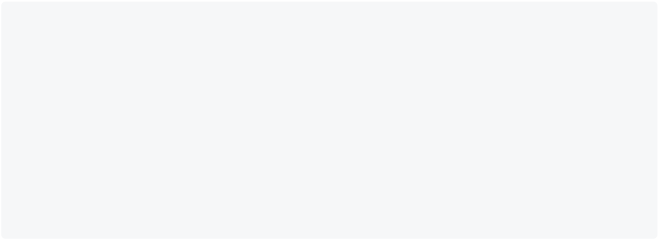
}

}

**States**

States represent the information to be processed by any widget. A widget changes itself based on the state.

class RotatedState { final double angle;



const RotatedState({@required this.angle});

@override

List<Object> get props => [angle]; }

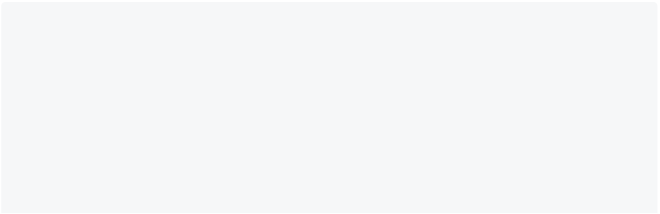
**Cubit**

Cubit is a simpler version of the BLoC pattern. It eliminates the need to write events.

Cubit exposes direct functions, which can result in appropriate states. Writing a Cubit instead of BLoC also reduces boilerplate code, making the code easier to read.

Here’s a simple example:

class TransformCubit extends Cubit<TransformState> { TransformCubit() : super(RotatedState(angle: 0));



void rotate(double angle) {

emit(RotatedState(angle: angle)); }

**The problem with setState**



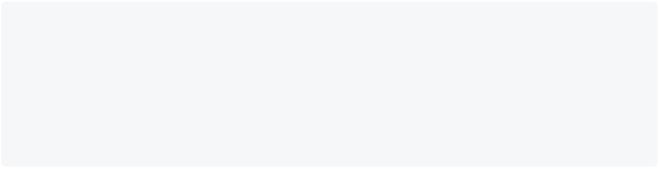
The setState approach to state management in Flutter works well for simple apps with just a few components. But for more complex, real-world Flutter apps with deep widget trees, using setState can lead to the following issues:

* Code duplication — data has to be passed from all widgets to the bottom widget, which makes the code difficult to read
  + Performance degradation due to unnecessary redraws that result from lifting a setState to a parent widget with a deep hierarchy

**How to manage state in Flutter with BLoC** First, add the [BLoC library](https://pub.dev/packages/flutter_bloc):



dependencies:



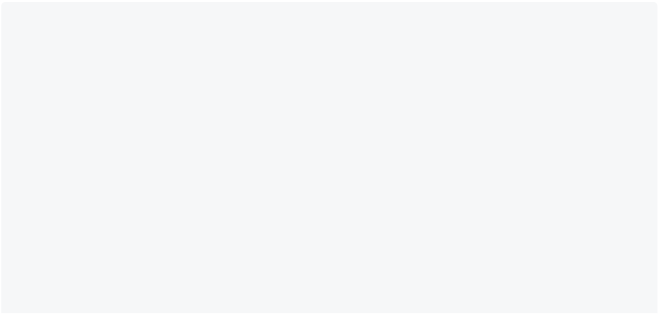
flutter:

sdk: flutter

cupertino\_icons: ^1.0.2 flutter\_bloc: ^7.0.0

Next, create and add a BLoC observer. This helps you determine the sequence of events and states that have occurred, which is great for debugging the app.

void main() {



Bloc.observer = SimpleBlocObserver();

runApp(MyApp());

}

import 'package:flutter\_bloc/flutter\_bloc.dart';

/// Custom [BlocObserver] which observes all bloc and cubit instances. class SimpleBlocObserver extends BlocObserver {

@override

void onEvent(Bloc bloc, Object event) {

super.onEvent(bloc, event);

print(event);

@override

void onTransition(Bloc bloc, Transition transition) { super.onTransition(bloc, transition);

print(transition);

}

@override

void onError(BlocBase bloc, Object error, StackTrace stackTrace) { print(error);

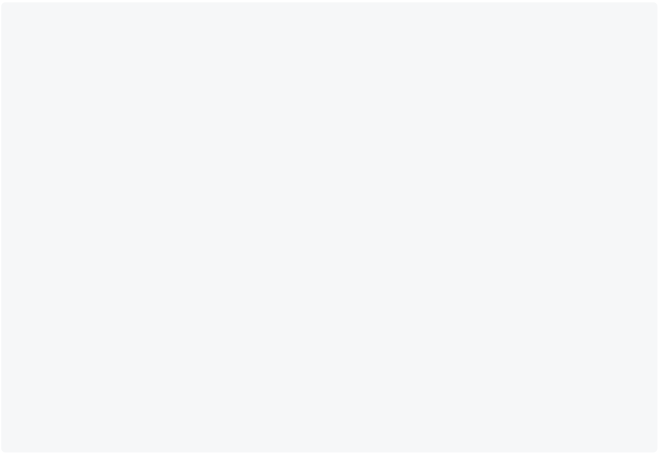
super.onError(bloc, error, stackTrace);

}

}

Create events to add and remove products from the list of cart items:

import 'package:equatable/equatable.dart';



abstract class CartEvent extends Equatable { const CartEvent();

@override

List<Object> get props => []; }

class AddProduct extends CartEvent {

final int productIndex;

const AddProduct(this.productIndex);

@override

List<Object> get props => [productIndex];

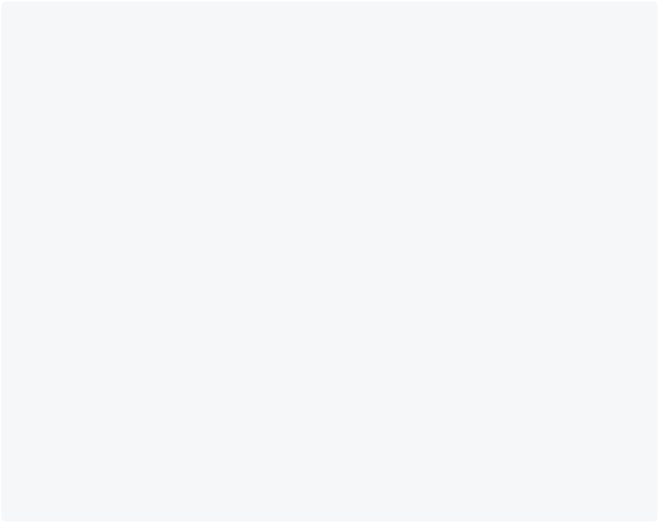
@override

String toString() => 'AddProduct { index: $productIndex }'; }

By extending Equatable, we can easily compare two instances of the state to determine if they are equal. This is crucial for efficient state management, especially when working with complex applications

Now, create states to represent a product being added and removed:

import 'package:flutter/material.dart';



abstract class CartState {

final List<int> cartItem;

const CartState({@required this.cartItem});

@override

List<Object> get props => []; }

class ProductAdded extends CartState {

final List<int> cartItem;

const ProductAdded({@required this.cartItem}) : super(cartItem: cartItem);

@override

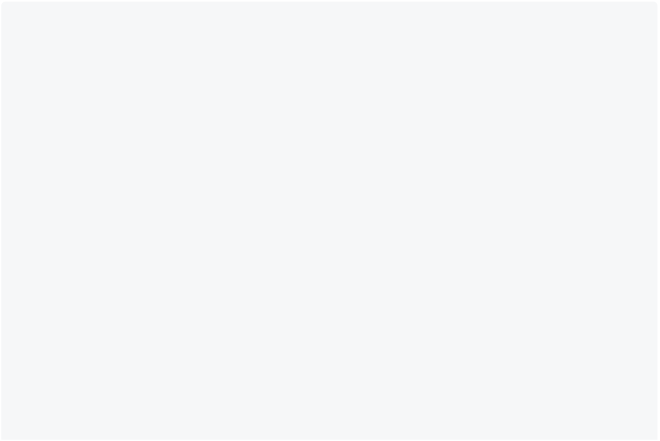
List<Object> get props => [cartItem];

@override

String toString() => 'ProductAdded { todos: $cartItem }'; }

Write business logic to add and remove products into the cartItems and emit the respective state. The actual list of items in the cart is maintained at the BLoC level.

class CartBloc extends Bloc<CartEvent, CartState> { CartBloc() : super(ProductAdded(cartItem: []));



final List<int> \_cartItems = []; List<int> get items => \_cartItems;

@override

Stream<CartState> mapEventToState(CartEvent event) async\* { if (event is AddProduct) {

\_cartItems.add(event.productIndex);

yield ProductAdded(cartItem: \_cartItems);

} else if (event is RemoveProduct) {

\_cartItems.remove(event.productIndex);

yield ProductRemoved(cartItem: \_cartItems);

} }

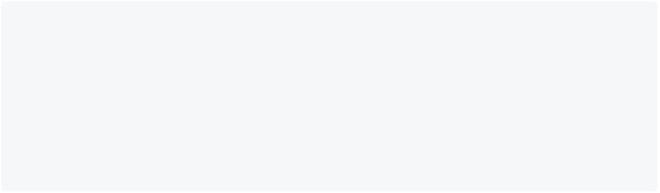
Next, wrap the scaffold widget inside [BlocProvider](https://pub.dev/packages/flutter_bloc#blocprovider).

BlocProvider is a Flutter widget that makes any BLoC available to the entire widget tree below it. In our case, any widget in between Home (top) and



ProductTile (bottom) can have access to the cart, so no need to pass the cart data from the top of the widget tree to the bottom.

BlocProvider(



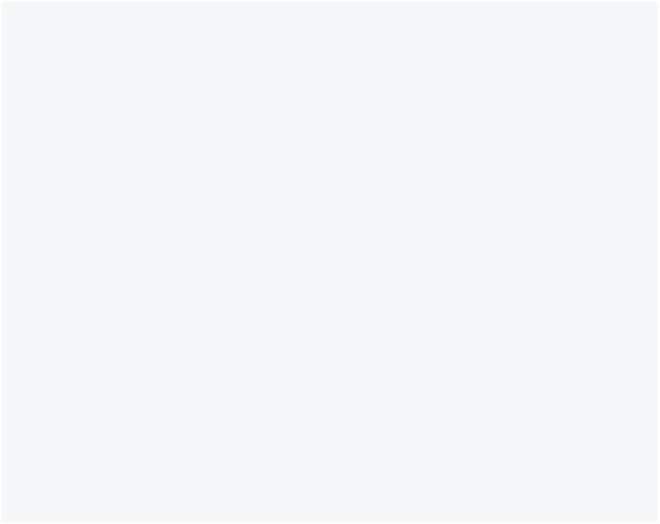
create: (\_) => CartBloc(), child: Scaffold(

appBar: CartCounter(), body: ProductList(),

));

Wrap the cart icon and product list inside the BlocBuilder. BlocBuilder simply rebuilds the widget inside it upon receiving the new states from the BLoC.

// Cart icon



BlocBuilder<CartBloc, CartState>(builder: (\_, cartState) { List<int> cartItem = cartState.cartItem;

return Positioned(

left: 30,

child: Container(

padding: EdgeInsets.all(5),

decoration: BoxDecoration(

borderRadius: BorderRadius.circular(10),

color: Colors.red),

child: Text(

'${cartItem.length}',

style: TextStyle(fontWeight: FontWeight.bold),

),

),

);

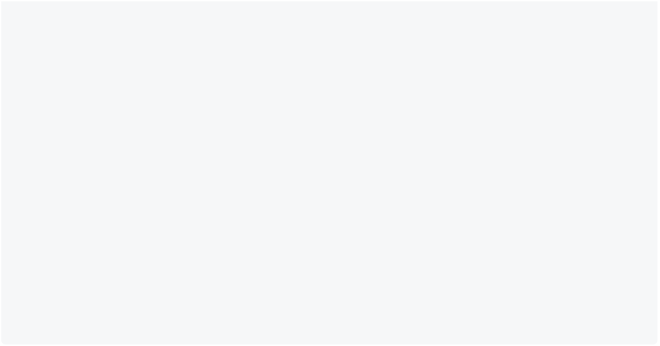
}),

//Product list

BlocBuilder<CartBloc, CartState>(builder: (\_, cartState) { List<int> cart = cartState.cartItem;

return LayoutBuilder(builder: (context, constraints) {

return GridView.builder(



itemCount: 100,

itemBuilder: (context, index) => ProductTile(

itemNo: index,

cart: cart,

),

gridDelegate: SliverGridDelegateWithFixedCrossAxisCount( crossAxisCount: constraints.maxWidth > 700 ? 4 : 1,

childAspectRatio: 5,

),

);

});

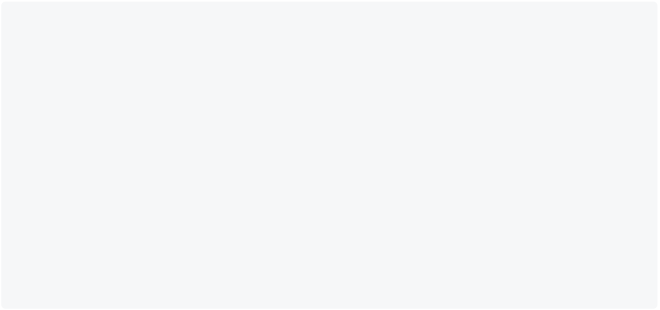
});

Note: The BlocBuilder for CartBloc is added only in two places because we only want these two widgets to rebuild when something happen at CartBloc. This approach of only refreshing widgets that are required significantly reduces the number of unnecessary redraws.

The next step is to shoot events to CartBloc for adding and removing items in the cart. BlocProvider.of<CartBloc>(context) finds the nearest instance of

CartBloc in the widget tree and adds the events to it:

IconButton(



key: Key('icon\_$itemNo'),

icon: cart.contains(itemNo)

* Icon(Icons.shopping\_cart)
* Icon(Icons.shopping\_cart\_outlined),

onPressed: () {

!cart.contains(itemNo)

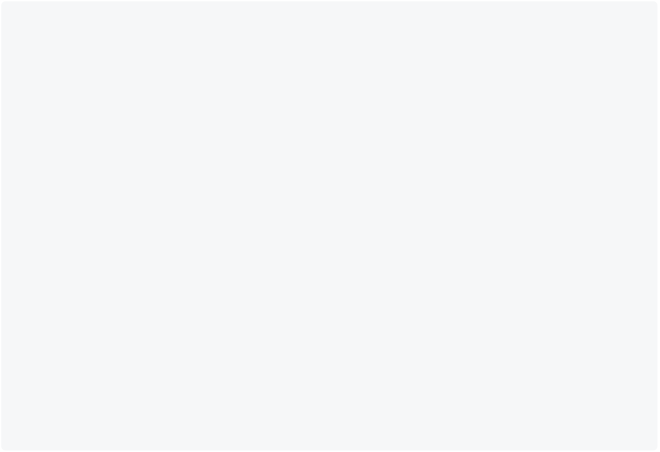
* BlocProvider.of<CartBloc>(context).add(AddProduct(itemNo))
* BlocProvider.of<CartBloc>(context).add(RemoveProduct(itemNo));

}, )

Now replace BlocBuilder with BlocConsumer. BlocConsumer allows us to rebuild the widget and react to the states. It should be only used when you want to rebuild the widget and also perform some action.

For our example, we want to refresh the list and show a snackbar whenever a product is added or removed from the cart:

BlocConsumer<CartBloc, CartState>(



listener: (context, state) {

Scaffold.of(context).showSnackBar(

SnackBar(

content: Text(

state is ProductAdded ? 'Added to cart.' : 'Removed from cart.'),

duration: Duration(seconds: 1),

),

);

},

builder: (\_, cartState) {

List<int> cart = cartState.cartItem;

return LayoutBuilder(builder: (context, constraints) {

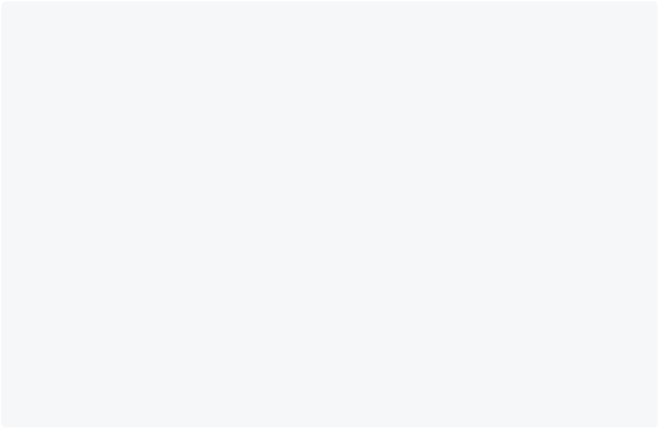
return GridView.builder();

});

});

Optionally, if you want to reduce some boilerplate code and the sequence of the states doesn’t matter to you, try Cubit. Here is what CartCubit would look like:

class CartCubit extends Cubit<CartState> {



CartCubit() : super(ProductAdded(cartItem: []));

final List<int> \_cartItems = []; List<int> get items => \_cartItems;

void add(int productIndex) {

\_cartItems.add(productIndex);

emit (ProductAdded(cartItem: \_cartItems)); }

void remove(int productIndex) {

\_cartItems.remove(productIndex);

emit (ProductRemoved(cartItem: \_cartItems)); }

}

Note: Replace CartBloc with CartCubit throughout the code and fire the events as shown below:

onPressed: () {

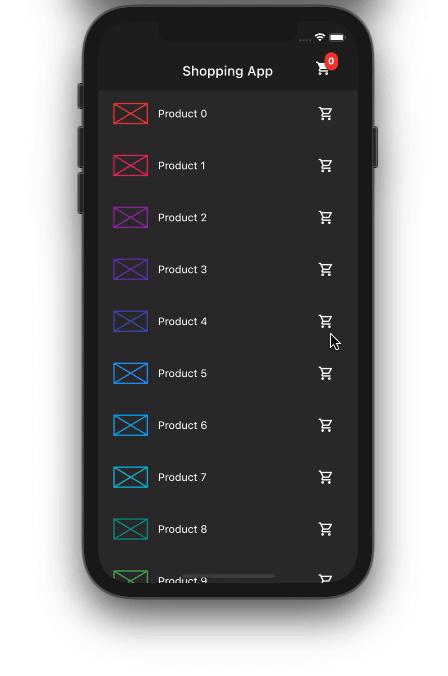


!cart.contains(itemNo)

* BlocProvider.of<CartCubit>(context).add(itemNo)
* BlocProvider.of<CartCubit>(context).remove(itemNo);

},

The output is the same but with improved state management:



**Conclusion**

Having a solid BLoC architecture in place leads to a good separation of concerns. Although using the BLoC pattern requires more code than using setState, it makes the code more readable, scalable, and testable.



**Resources**

* [Flutter Bloc Tutorials](https://bloclibrary.dev/getting-started/)
  + [Flutter Bloc](https://pub.dev/packages/flutter_bloc)
    - [Flutter Bloc Comprehensive Guide](https://www.blup.in/blog/understanding-flutter-bloc-for-state-management-a-comprehensive-guide)

## Using Flutter Plugins

### Take a picture using the camera

Many apps require working with the device’s cameras to take photos and videos. Flutter provides the [camera](https://pub.dev/packages/camera) plugin for this purpose. The camera plugin provides tools to get a list of the available cameras, display a preview coming from a specific camera, and take photos or videos.

This recipe demonstrates how to use the camera plugin to display a preview, take a photo, and display it using the following steps:

1. Add the required dependencies.
2. Get a list of the available cameras.
3. Create and initialize the CameraController.
4. Use a CameraPreview to display the camera’s feed.
5. Take a picture with the CameraController.
6. Display the picture with an Image widget.



1. **Add the required dependencies**



To complete this recipe, you need to add three dependencies to your app:

[camera](https://pub.dev/packages/camera)

Provides tools to work with the cameras on the device.

[path\_provider](https://pub.dev/packages/path_provider)

Finds the correct paths to store images.

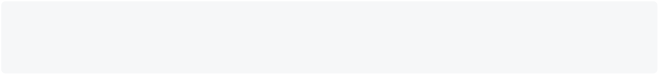
[path](https://pub.dev/packages/path)



Creates paths that work on any platform.

To add the packages as dependencies, run flutter pub add:

$ flutter pub add camera path\_provider path



1. **Get a list of the available cameras**



Next, get a list of available cameras using the camera plugin.

// Ensure that plugin services are initialized so that `availableCameras()`



// can be called before `runApp()` WidgetsFlutterBinding.ensureInitialized();

// Obtain a list of the available cameras on the device. final cameras = await availableCameras();

// Get a specific camera from the list of available cameras. final firstCamera = cameras.first;

1. **Create and initialize the CameraController**



Once you have a camera, use the following steps to create and initialize a

CameraController. This process establishes a connection to the device’s camera that allows you to control the camera and display a preview of the camera’s feed.

1. Create a StatefulWidget with a companion State class.



1. Add a variable to the State class to store the CameraController.

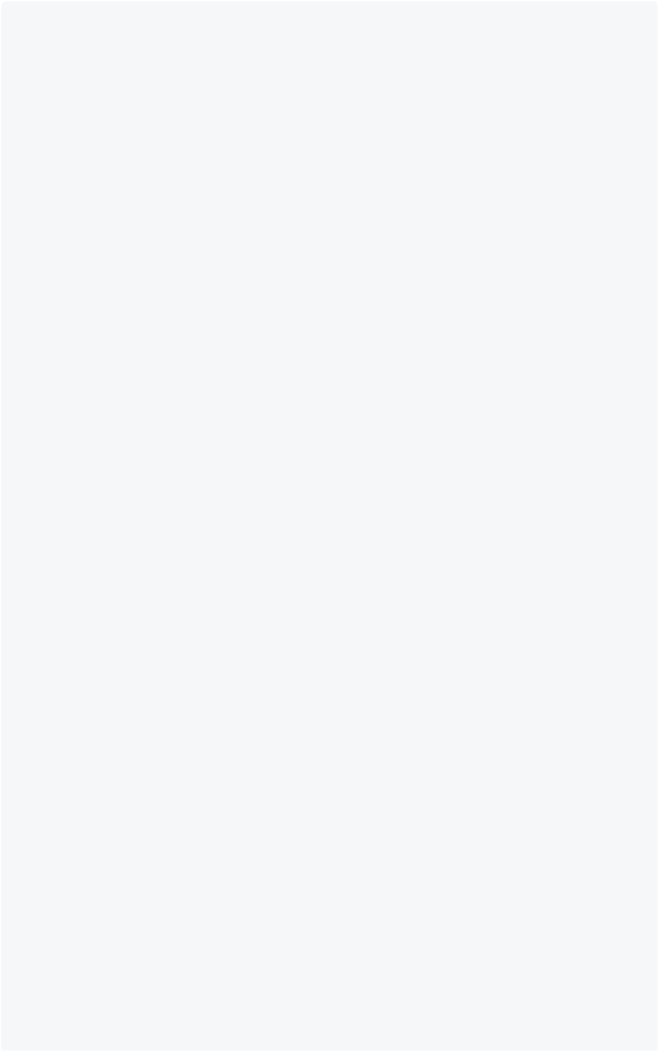


1. Add a variable to the State class to store the Future returned from CameraController.initialize().



1. Create and initialize the controller in the initState() method.
2. Dispose of the controller in the dispose() method.

// A screen that allows users to take a picture using a given camera. class TakePictureScreen extends StatefulWidget {



const TakePictureScreen({

super.key,

required this.camera,

});

final CameraDescription camera;

@override

TakePictureScreenState createState() => TakePictureScreenState(); }

class TakePictureScreenState extends State<TakePictureScreen> { late CameraController \_controller;

late Future<void> \_initializeControllerFuture;

@override

void initState() {

super.initState();

// To display the current output from the Camera,

// create a CameraController.

\_controller = CameraController(

// Get a specific camera from the list of available cameras. widget.camera,

// Define the resolution to use.

ResolutionPreset.medium,

);

// Next, initialize the controller. This returns a Future. \_initializeControllerFuture = \_controller.initialize();

}

@override

void dispose() {

// Dispose of the controller when the widget is disposed. \_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) { // Fill this out in the next steps.

return Container(); }



}

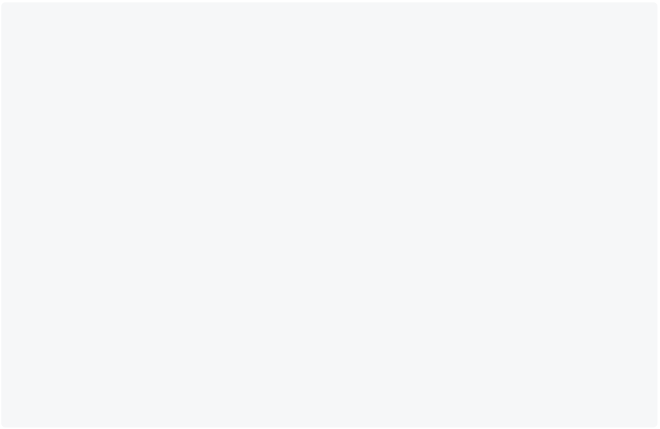
1. **Use a CameraPreview to display the camera’s feed**



Next, use the CameraPreview widget from the camera package to display a preview of the camera’s feed.

Use a [FutureBuilder](https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html) for exactly this purpose.

// You must wait until the controller is initialized before displaying the



// camera preview. Use a FutureBuilder to display a loading spinner until the

// controller has finished initializing.

FutureBuilder<void>(

future: \_initializeControllerFuture,

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.done) {

// If the Future is complete, display the preview.

return CameraPreview(\_controller);

} else {

// Otherwise, display a loading indicator.

return const Center(child: CircularProgressIndicator());

}

},

)

1. **Take a picture with the CameraController**



You can use the CameraController to take pictures using the [takePicture()](https://pub.dev/documentation/camera/latest/camera/CameraController/takePicture.html) method, which returns an [XFile](https://pub.dev/documentation/cross_file/latest/cross_file/XFile-class.html), a cross-platform, simplified File abstraction. On both Android and IOS, the new image is stored in their respective cache directories, and the path to that location is returned in the XFile.



In this example, create a FloatingActionButton that takes a picture using the

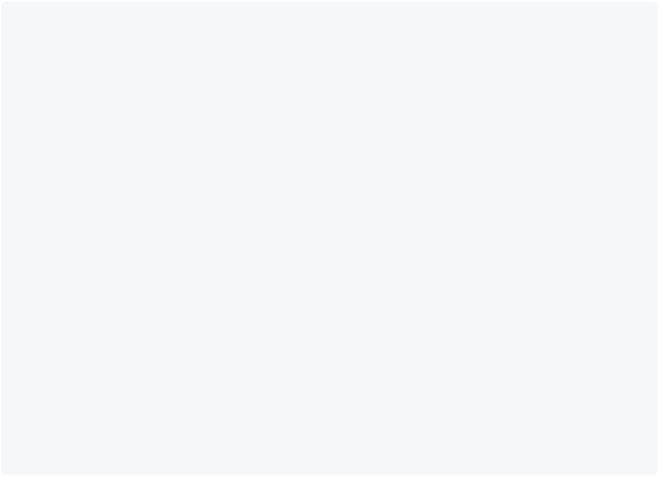
CameraController when a user taps on the button.

Taking a picture requires 2 steps:

1. Ensure that the camera is initialized.
2. Use the controller to take a picture and ensure that it returns a Future<XFile>.

It is good practice to wrap these operations in a try / catch block in order to handle any errors that might occur.

FloatingActionButton(



// Provide an onPressed callback.

onPressed: () async {

// Take the Picture in a try / catch block. If anything goes wrong, // catch the error.

try {

// Ensure that the camera is initialized.

await \_initializeControllerFuture;

// Attempt to take a picture and then get the location // where the image file is saved.

final image = await \_controller.takePicture();

} catch (e) {

// If an error occurs, log the error to the console. print(e);

}

},

child: const Icon(Icons.camera\_alt),

)

If you take the picture successfully, you can then display the saved picture using an Image widget. In this case, the picture is stored as a file on the device.



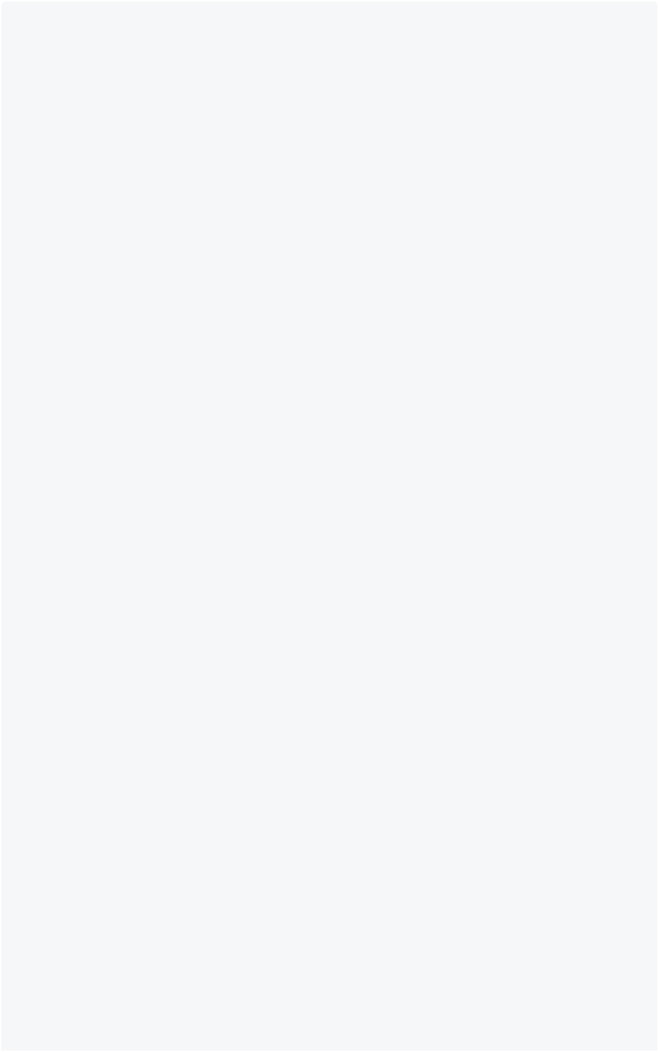
Therefore, you must provide a File to the Image.file constructor. You can create an instance of the File class by passing the path created in the previous step.



Image.file(File('path/to/my/picture.png'));



**Complete example** import 'dart:async'; import 'dart:io';



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

Future<void> main() async {

// Ensure that plugin services are initialized so that `availableCameras()`

// can be called before `runApp()`

WidgetsFlutterBinding.ensureInitialized();

// Obtain a list of the available cameras on the device. final cameras = await availableCameras();

// Get a specific camera from the list of available cameras. final firstCamera = cameras.first;

runApp(

MaterialApp(

theme: ThemeData.dark(),

home: TakePictureScreen(

// Pass the appropriate camera to the TakePictureScreen widget. camera: firstCamera,

),

),

);

}

// A screen that allows users to take a picture using a given camera. class TakePictureScreen extends StatefulWidget {

const TakePictureScreen({

super.key,

required this.camera,

});

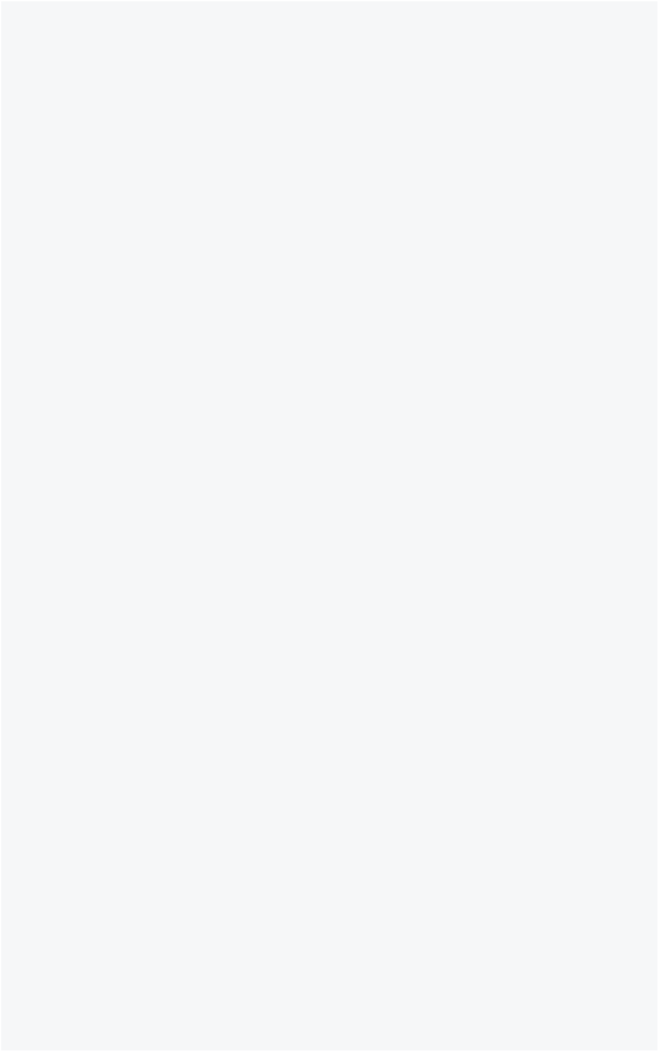
final CameraDescription camera;

@override

TakePictureScreenState createState() => TakePictureScreenState(); }

class TakePictureScreenState extends State<TakePictureScreen> {

late CameraController \_controller;



late Future<void> \_initializeControllerFuture;

@override

void initState() {

super.initState();

// To display the current output from the Camera,

// create a CameraController.

\_controller = CameraController(

// Get a specific camera from the list of available cameras. widget.camera,

// Define the resolution to use.

ResolutionPreset.medium,

);

// Next, initialize the controller. This returns a Future. \_initializeControllerFuture = \_controller.initialize();

}

@override

void dispose() {

// Dispose of the controller when the widget is disposed. \_controller.dispose();

super.dispose();

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Take a picture')),

// You must wait until the controller is initialized before displaying the

// camera preview. Use a FutureBuilder to display a loading spinner until the

// controller has finished initializing.

body: FutureBuilder<void>(

future: \_initializeControllerFuture,

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.done) { // If the Future is complete, display the preview.

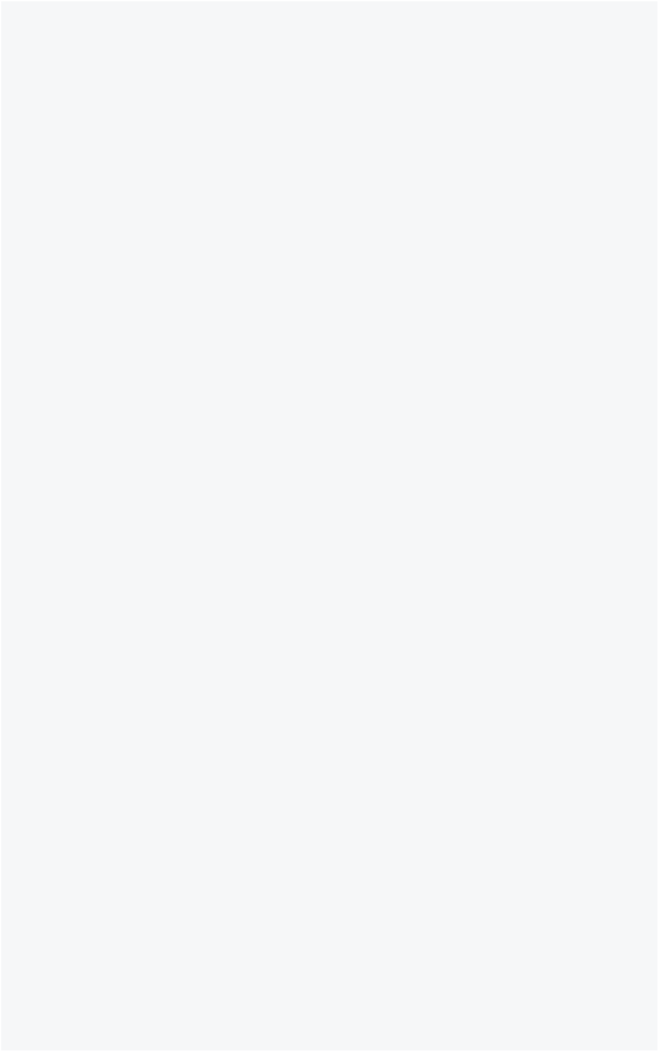
return CameraPreview(\_controller);

} else {

// Otherwise, display a loading indicator.

return const Center(child: CircularProgressIndicator());

}



},

),

floatingActionButton: FloatingActionButton(

// Provide an onPressed callback.

onPressed: () async {

// Take the Picture in a try / catch block. If anything goes wrong,

// catch the error.

try {

// Ensure that the camera is initialized.

await \_initializeControllerFuture;

// Attempt to take a picture and get the file `image` // where it was saved.

final image = await \_controller.takePicture();

if (!context.mounted) return;

// If the picture was taken, display it on a new screen. await Navigator.of(context).push(

MaterialPageRoute(

builder: (context) => DisplayPictureScreen(

// Pass the automatically generated path to

// the DisplayPictureScreen widget.

imagePath: image.path,

),

),

);

} catch (e) {

// If an error occurs, log the error to the console.

print(e);

}

},

child: const Icon(Icons.camera\_alt),

),

);

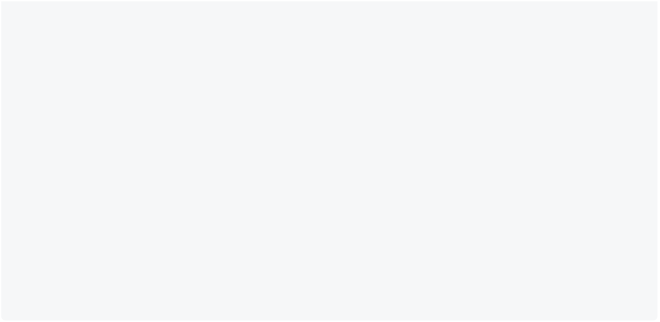
}

}

// A widget that displays the picture taken by the user. class DisplayPictureScreen extends StatelessWidget {

final String imagePath;

const DisplayPictureScreen({super.key, required this.imagePath});



@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text('Display the Picture')), // The image is stored as a file on the device. Use the `Image.file`

// constructor with the given path to display the image. body: Image.file(File(imagePath)),

);

}

}

### Play and pause a video

Playing videos is a common task in app development, and Flutter apps are no exception. To play videos, the Flutter team provides the [video\_player](https://pub.dev/packages/video_player) plugin. You can use the video\_player plugin to play videos stored on the file system, as an asset, or from the internet.

On iOS, the video\_player plugin makes use of [AVPlayer](https://developer.apple.com/documentation/avfoundation/avplayer) to handle playback. On Android, it uses [ExoPlayer](https://google.github.io/ExoPlayer/).

This guide demonstrates how to use the video\_player package to stream a video from the internet with basic play and pause controls using the following steps:

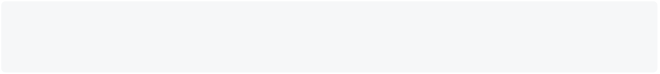
1. Add the video\_player dependency.
2. Add permissions to your app.
3. Create and initialize a VideoPlayerController.
4. Display the video player.
5. Play and pause the video.
6. **Add the video\_player dependency**



This recipe depends on one Flutter plugin: video\_player. First, add this dependency to your project.

To add the video\_player package as a dependency, run flutter pub add:

$ flutter pub add video\_player



1. **Add permissions to your app**



Next, update your android and ios configurations to ensure that your app has the correct permissions to stream videos from the internet.

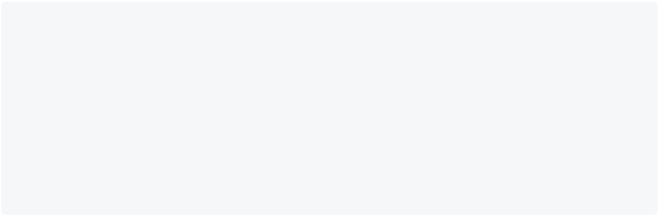


**Android**

Add the following permission to the AndroidManifest.xml file just after the

<application> definition. The AndroidManifest.xml file is found at <project root>/android/app/src/main/AndroidManifest.xml.

<manifest xmlns:android="http://schemas.android.com/apk/res/android"> <application ...>



</application>

<uses-permission android:name="android.permission.INTERNET"/> </manifest>

**iOS**

For iOS, add the following to the Info.plist file found at <project

root>/ios/Runner/Info.plist.

<key>NSAppTransportSecurity</key> <dict>



<key>NSAllowsArbitraryLoads</key> <true/>

</dict>

1. **Create and initialize a VideoPlayerController**



Now that you have the video\_player plugin installed with the correct permissions, create a VideoPlayerController. The VideoPlayerController class allows you to connect to different types of videos and control playback.

Before you can play videos, you must also initialize the controller. This establishes the connection to the video and prepare the controller for playback.

To create and initialize the VideoPlayerController do the following:

1. Create a StatefulWidget with a companion State class



1. Add a variable to the State class to store the VideoPlayerController

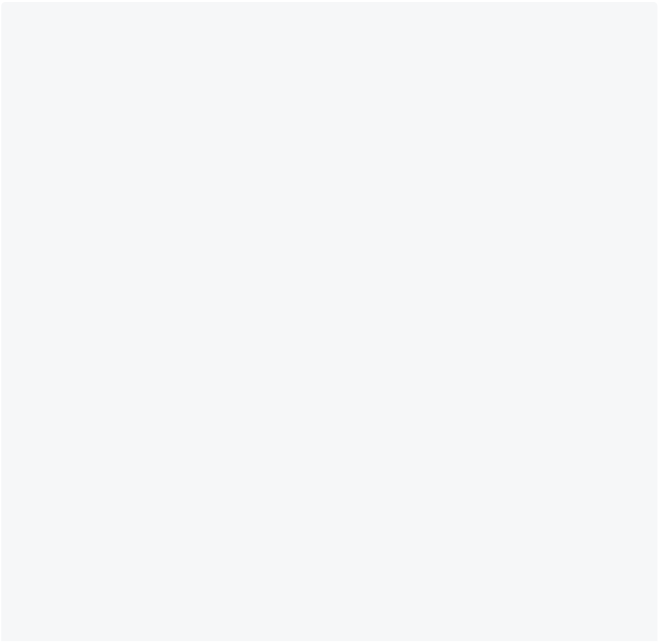


1. Add a variable to the State class to store the Future returned from VideoPlayerController.initialize



1. Create and initialize the controller in the initState method
2. Dispose of the controller in the dispose method

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



@override

State<VideoPlayerScreen> createState() => \_VideoPlayerScreenState(); }

class \_VideoPlayerScreenState extends State<VideoPlayerScreen> { late VideoPlayerController \_controller;

late Future<void> \_initializeVideoPlayerFuture;

@override

void initState() { super.initState();

// Create and store the VideoPlayerController. The VideoPlayerController

// offers several different constructors to play videos from assets, files,

// or the internet.

\_controller = VideoPlayerController.networkUrl(

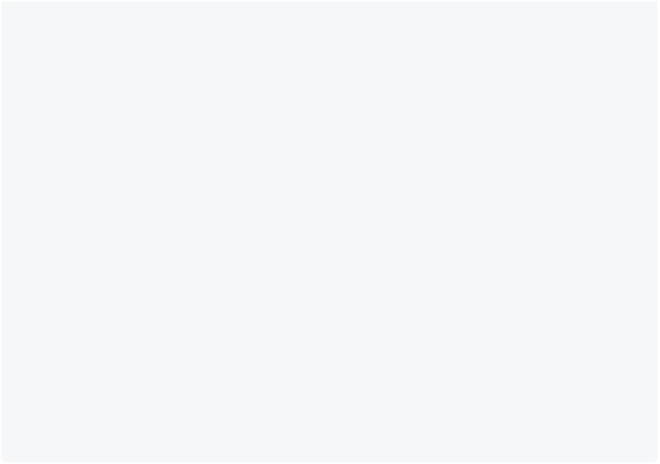
Uri.parse(

'https://flutter.github.io/assets-for-api- docs/assets/videos/butterfly.mp4',

),

);

\_initializeVideoPlayerFuture = \_controller.initialize(); }



@override

void dispose() {

// Ensure disposing of the VideoPlayerController to free up resources.

\_controller.dispose();

super.dispose(); }

@override

Widget build(BuildContext context) {

// Complete the code in the next step. return Container();

}

}

1. **Display the video player**



Now, display the video. The video\_player plugin provides the [VideoPlayer](https://pub.dev/documentation/video_player/latest/video_player/VideoPlayer-class.html) widget to display the video initialized by the VideoPlayerController. By default, the VideoPlayer widget takes up as much space as possible. This often isn’t ideal for videos because they are meant to be displayed in a specific aspect ratio, such as 16x9 or 4x3.

Therefore, wrap the VideoPlayer widget in an [AspectRatio](https://api.flutter.dev/flutter/widgets/AspectRatio-class.html) widget to ensure that the video has the correct proportions.

Furthermore, you must display the VideoPlayer widget after the

\_initializeVideoPlayerFuture() completes. Use FutureBuilder to display a loading spinner until the controller finishes initializing. Note: initializing the controller does not begin playback.

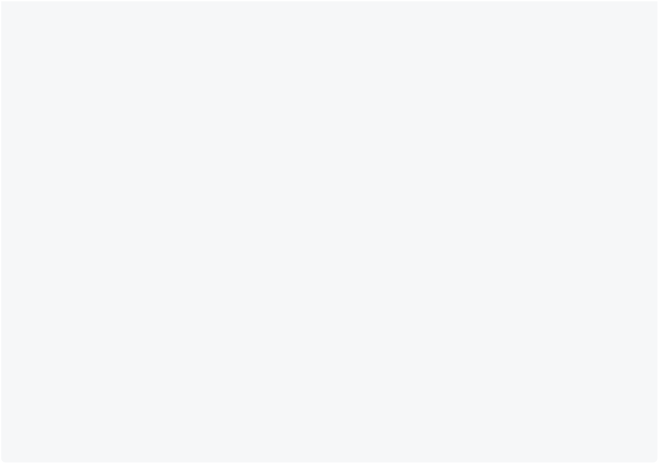
// Use a FutureBuilder to display a loading spinner while waiting for the



// VideoPlayerController to finish initializing.

FutureBuilder(

future: \_initializeVideoPlayerFuture,



builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.done) {

// If the VideoPlayerController has finished initialization, use // the data it provides to limit the aspect ratio of the video. return AspectRatio(

aspectRatio: \_controller.value.aspectRatio,

// Use the VideoPlayer widget to display the video.

child: VideoPlayer(\_controller),

);

} else {

// If the VideoPlayerController is still initializing, show a // loading spinner.

return const Center(

child: CircularProgressIndicator(),

);

}

},

)

1. **Play and pause the video**

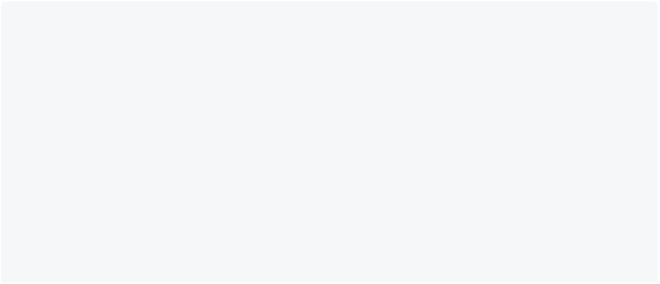


By default, the video starts in a paused state. To begin playback, call the [play()](https://pub.dev/documentation/video_player/latest/video_player/VideoPlayerController/play.html) method provided by the VideoPlayerController. To pause playback, call the

[pause()](https://pub.dev/documentation/video_player/latest/video_player/VideoPlayerController/pause.html) method.

For this example, add a FloatingActionButton to your app that displays a play or pause icon depending on the situation. When the user taps the button, play the video if it’s currently paused, or pause the video if it’s playing.

FloatingActionButton(



onPressed: () {

// Wrap the play or pause in a call to `setState`. This ensures the // correct icon is shown.

setState(() {

// If the video is playing, pause it.

if (\_controller.value.isPlaying) {

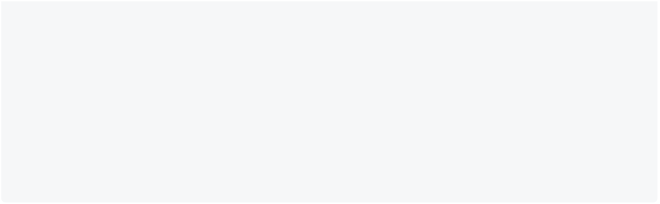
\_controller.pause();

} else {

// If the video is paused, play it.

\_controller.play();

}



});

},

// Display the correct icon depending on the state of the player. child: Icon(

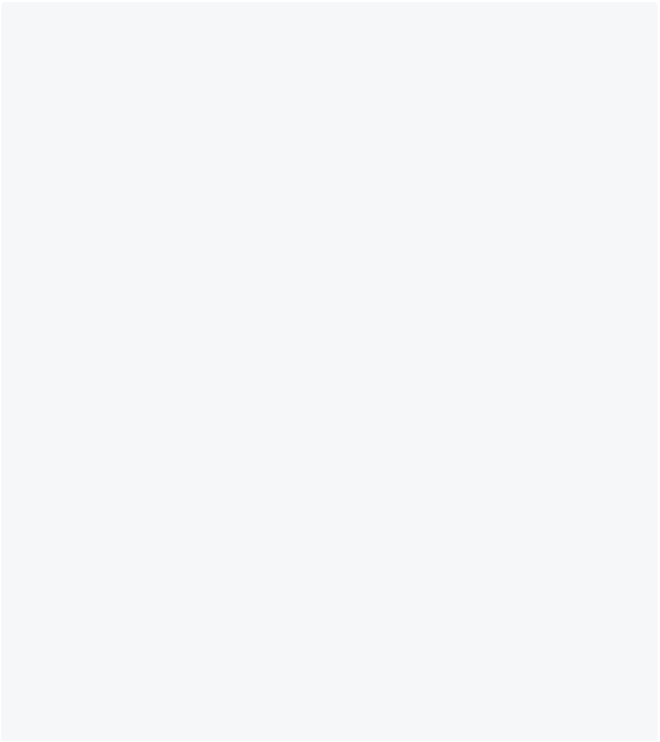
\_controller.value.isPlaying ? Icons.pause : Icons.play\_arrow, ),

)

**Complete example**



import 'dart:async';



import 'package:flutter/material.dart';

import 'package:video\_player/video\_player.dart';

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

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

@override

Widget build(BuildContext context) { return const MaterialApp(

title: 'Video Player Demo',

home: VideoPlayerScreen(),

);

}

}

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

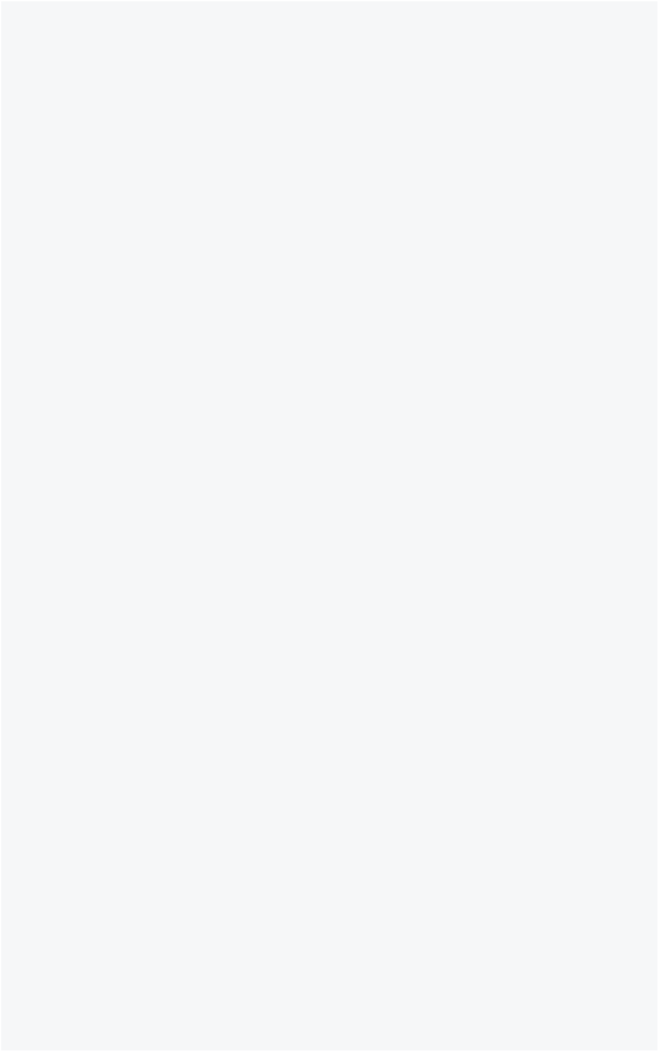
@override

State<VideoPlayerScreen> createState() => \_VideoPlayerScreenState(); }

class \_VideoPlayerScreenState extends State<VideoPlayerScreen> { late VideoPlayerController \_controller;

late Future<void> \_initializeVideoPlayerFuture;

@override



void initState() { super.initState();

// Create and store the VideoPlayerController. The VideoPlayerController

// offers several different constructors to play videos from assets, files,

// or the internet.

\_controller = VideoPlayerController.networkUrl(

Uri.parse(

'https://flutter.github.io/assets-for-api- docs/assets/videos/butterfly.mp4',

),

);

// Initialize the controller and store the Future for later use. \_initializeVideoPlayerFuture = \_controller.initialize();

// Use the controller to loop the video. \_controller.setLooping(true);

}

@override

void dispose() {

// Ensure disposing of the VideoPlayerController to free up resources.

\_controller.dispose();

super.dispose(); }

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Butterfly Video'),

),

// Use a FutureBuilder to display a loading spinner while waiting for the

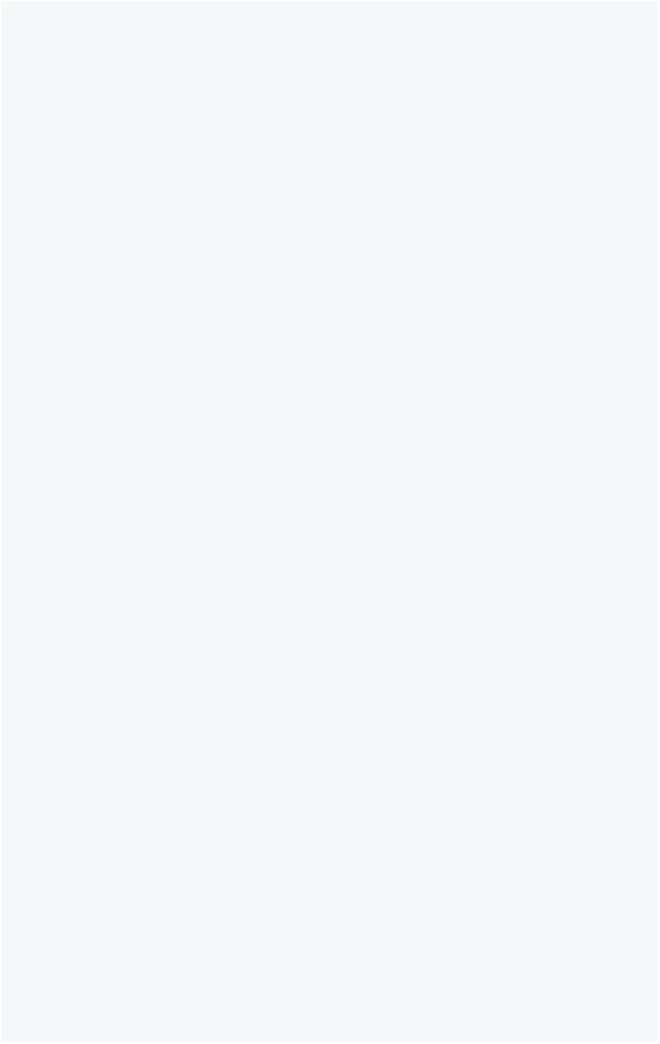
// VideoPlayerController to finish initializing.

body: FutureBuilder(

future: \_initializeVideoPlayerFuture,

builder: (context, snapshot) {

if (snapshot.connectionState == ConnectionState.done) {



// If the VideoPlayerController has finished initialization, use

// the data it provides to limit the aspect ratio of the video.

return AspectRatio(

aspectRatio: \_controller.value.aspectRatio,

// Use the VideoPlayer widget to display the video.

child: VideoPlayer(\_controller),

);

} else {

// If the VideoPlayerController is still initializing, show

a

// loading spinner.

return const Center(

child: CircularProgressIndicator(), );

}

},

),

floatingActionButton: FloatingActionButton(

onPressed: () {

// Wrap the play or pause in a call to `setState`. This ensures the

// correct icon is shown.

setState(() {

// If the video is playing, pause it.

if (\_controller.value.isPlaying) {

\_controller.pause();

} else {

// If the video is paused, play it.

\_controller.play();

}

});

},

// Display the correct icon depending on the state of the player.

child: Icon(

\_controller.value.isPlaying ? Icons.pause : Icons.play\_arrow, ),

),

);

}

}

## Building Flutter APK



To build an APK (Android Package) for a Flutter app, you can use the following steps. Ensure you have Flutter and Dart installed on your machine, and your Flutter environment is set up.

**Step 1: Open a Terminal or Command Prompt**

Open a terminal or command prompt window in the directory where your Flutter project is located.

**Step 2: Run Flutter Build Command**

Run the following command to build the APK:

flutter build apk

This command will create a release APK for your Flutter app. If you want to build a specific flavor or for a specific target device, you can provide additional arguments. For example:

flutter build apk --release

This command generates a release APK. If you want to build a debug version, you can omit the --release flag.

**Step 3: Locate the APK**

After running the build command, you can find the APK file in the following directory:

/build/app/outputs/flutter-apk/app-release.apk

**Step 4: Share the APK**

You can share the generated APK through various methods: **Option 1: Direct File Transfer**

Copy the APK file to your Android device using a USB cable or any other means. You can then install the app manually by opening the APK file on the device.

**Option 2: Cloud Storage**

Upload the APK file to cloud storage services like Google Drive, Dropbox, or any other cloud service of your choice. Share the download link with others.

**Option 3: Email**

Attach the APK file to an email and send it to the recipients. They can then download and install the app.

**Option 4: Create a Release Bundle**

Instead of a standalone APK, you can create an Android App Bundle (AAB), which is a publishing format. The Google Play Store uses this format to generate optimized APKs for different device configurations. To create an AAB, run the following command:

flutter build appbundle

The AAB file will be generated in the build/app/outputs/bundle/release/app-

release.aab directory.

**Step 5: Install the APK**

If you're sharing the APK outside an app store, ensure that the device's security settings allow installations from unknown sources. Users can then tap on the APK file to install the app.

Remember to follow ethical guidelines when distributing apps, and ensure that users are aware of the source from which they are installing the application.

These steps are specific to Android. For iOS, you need to use Xcode and follow different procedures for distribution through the App Store.