

Mobile Application Development (Flutter Framework)

Dr. Osman Khalid

🌐 Website: <https://bit.ly/3IUccOx>

Last Updated: 26 March 2024

Table of Contents

Dart Programming Language	3
Simple Hello World App	3
Passing arguments through console	3
Variables	4
Variable creation and initialization.	4
Changing value of variable.	4
We use Object data type, we want to change data type of variable.....	4
Explicit data types	4
Nullable type	4
Late variables	4
Final and Const.....	5
Constant.....	5
Variable examples.....	5
Conditional expressions:	6
condition ? expr1 : expr2	6
expr1 ?? expr2.....	6
Comments:.....	6
Builtin-Types:	6
Strings:	7
String concatenation example:	7
Multiline string:.....	7
Records	7
Lists	9
List of Records:.....	9
Sets.....	10

Maps	11
Objects in Dart resembling javascript objects	12
List of map objects	13
Spread operators	13
Control-flow operators	13
Patterns.....	14
Variable assignment.....	15
Switch statements and expressions.....	15
For and for-in loops.....	16
Functions:.....	17
Named parameters	17
Optional positional parameters	18
The main() function.....	19
Functions as first-class objects.....	19
Anonymous functions	20
Arrow notation:.....	21
typedef functions	21
Error handling.	22
Classes.....	22
Simple class example	22
No argument constructor	22
One argument generative constructor	23
Two argument generative constructor	23
Another example of two argument constructor.....	24
Calling a constructor from another constructor within same class.....	24
Named constructors.....	25
Named arguments in a constructor.	25
Immutable objects	26
Optional arguments to a constructor	26
Array of objects.....	27
Printing elements in an array.....	27
Looping through array of objects:.....	28
Use continue to skip to the next loop iteration:	29
Inheritance example.	29
Using parent class constructor in child class.....	29
Calling named argument constructor from Child class of Parent class	30

Flutter widgets.....	31
runApp() function.....	31
MaterialApp Widget:	34
Scaffold Widget:.....	36
MaterialApp and Material Widget.....	38
StatelessWidget	41
Row widget.	45
StatefulWidget	68

Dart Programming Language

Simple Hello World App

Create a Dart console application using Visual Studio Code. The following code will be generated. The calculate() function is defined in lib/dartproject.dart.

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

void main(List<String> arguments) {
  print('Hello world: ${dartproject.calculate()}!');
}
```

Passing arguments through console

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

void main(List<String> arguments) {
  print('Arguments: $arguments');
}
```

To pass the parameters to the above application using console, run the following command in console:

A:\AAA\flutterprojects\dartproject\bin> dart dartproject.dart one two three

Variables

Variable creation and initialization.

```
var name = 'Bob';  
print(name);
```

Changing value of variable.

```
var name = 'Bob';  
name = 10; // not allowed to change data type  
print(name);
```

The following code will work.

We use Object data type, we want to change data type of variable.

```
Object name = 'Bob';  
name = 10;  
print(name);
```

Explicit data types

```
String name = 'Bob';  
print(name);
```

Nullable type

```
String? name1; // Nullable type. Can be `null` or string.  
  
String name2; // Non-nullable type. Cannot be `null` but can be string.  
  
print(name1); // null  
// print(name2); // error
```

Late variables

When you mark a variable as late but initialize it at its declaration, then the initializer runs the first time the variable is used. This lazy initialization is handy in a couple of cases:

The variable might not be needed, and initializing it is costly.

You're initializing an instance variable, and its initializer needs access to this.

```
late String description;

void main() {
    description = 'Feijoadá!';
    print(description);
}
```

Final and Const

A final variable can be set only once; a const variable is a compile-time constant. (Const variables are implicitly final.)

```
void main() {
    final name = 'Bob'; // Without a type annotation
    final String nickname = 'Bobby';

    print(name);
    print(nickname);

    // name = 'Ali'; // This is error as value of final cannot be changed
    // after it is initialized.
}
```

Constant

```
void main() {
    const bar = 1000000; // Unit of pressure (dynes/cm2)
    const double atm = 1.01325 * bar; // Standard atmosphere
}
```

Variable examples

```
void main() {
    var name = 'Voyager I';
    var year = 1977;
    var antennaDiameter = 3.7;
    var flybyObjects = ['Jupiter', 'Saturn', 'Uranus', 'Neptune'];
    var image = {
        'tags': ['saturn'],
        'url': '//path/to/saturn.jpg'
    };

    print(name);
    print(year);
    print(antennaDiameter);
    print(flybyObjects);
}
```

```
print(image);  
}
```

Conditional expressions:

condition ? expr1 : expr2

If *condition* is true, evaluates *expr1* (and returns its value); otherwise, evaluates and returns the value of *expr2*.

```
void main()  
{  
    var isPublic = 'public';  
    var visibility = isPublic=='public' ? 'public' : 'private';  
  
    print(visibility);  
}
```

expr1 ?? expr2

If *expr1* is non-null, returns its value; otherwise, evaluates and returns the value of *expr2*.

```
void main()  
{  
    var var1 = null; // or simply write var var1; this defaults to null  
    var var2 = 10;  
  
    var ans = var1 ?? var2;  
    print(ans);  
}
```

Comments:

// single line , /* multi line */, /// for documentation.

Builtin-Types:

Numbers, Strings, Booleans.

Strings:

String concatenation example:

```
void main() {
var s1 = 'String '
    'concatenation'
    " works even over line breaks.";

    print(s1);
}
```

Multiline string:

```
void main() {

var s1 = '''
You can create
multi-line strings like this one.
''';

    print(s1);
}
```

Records

Records are an anonymous, immutable, aggregate type. Like other [collection types](#), they let you bundle multiple objects into a single object. Unlike other collection types, records are fixed-sized, heterogeneous, and typed.

Records are real values; you can store them in variables, nest them, pass them to and from functions, and store them in data structures such as lists, maps, and sets.

Example:

```
void main() {

// Record type annotation in a variable declaration:
({int a, bool b}) record;

// Initialize it with a record expression:
record = (a: 123, b: true);

print(record);
print(record.a);
print(record.b);
}
```

```
}
```

Records expressions are comma-delimited lists of named or positional fields, enclosed in parentheses:

dart

Example:

Here 'first', "hello", 'last' are positional fields, and others are named.

```
void main() {  
  
var record = ('first', a: 2, "hello", b: true, 'last');  
  
print(record.$1); // Prints 'first'  
print(record.a); // Prints 2  
print(record.b); // Prints true  
print(record.$2); // Prints 'last'  
print(record.$3);  
  
}
```

In a record type annotation, named fields go inside a curly brace-delimited section of type-and-name pairs, after all positional fields. In a record expression, the names go before each field value with a colon after:

```
// Record type annotation in a variable declaration:  
({int a, bool b}) record;  
  
// Initialize it with a record expression:  
record = (a: 123, b: true);
```

The names of named fields in a record type are part of the [record's type definition](#), or its *shape*. Two records with named fields with different names have different types:

```
({int a, int b}) recordAB = (a: 1, b: 2);  
({int x, int y}) recordXY = (x: 3, y: 4);  
  
// Compile error! These records don't have the same type.  
// recordAB = recordXY;
```


In a record type annotation, you can also name the *positional* fields, but these names are purely for documentation and don't affect the record's type:

```
(int a, int b) recordAB = (1, 2);  
(int x, int y) recordXY = (3, 4);  
  
recordAB = recordXY; // OK.
```

Example

```
(int x, int y, int z) point = (1, 2, 3);  
(int r, int g, int b) color = (1, 2, 3);  
  
print(point == color); // Prints 'true'.
```

Example

```
({int x, int y, int z}) point = (x: 1, y: 2, z: 3);  
({int r, int g, int b}) color = (r: 1, g: 2, b: 3);  
  
print(point == color); // Prints 'false'. Lint: Equals on unrelated types.
```

Lists

Perhaps the most common collection in nearly every programming language is the *array*, or ordered group of objects. In Dart, arrays are [List](#) objects, so most people just call them *lists*.

Dart list literals are denoted by a comma separated list of expressions or values, enclosed in square brackets (`[]`). Here's a simple Dart list:

```
void main() {  
  
  var list = [1, 2, 3];  
  
  print(list[0]);  
}
```

List of Records:

```
void main() {  
  
  var list = [(id: 1, name: 'Ali'), (id: 2, name: 'javed')];  
}
```

```
for( var item in list) {  
    print(item.id);  
}  
}
```

Sets

#

A set in Dart is an unordered collection of unique items. Dart support for sets is provided by set literals and the [Set](#) type.

Here is a simple Dart set, created using a set literal:

```
void main() {  
  
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};  
  
print(halogens.elementAt(0));  
print(halogens.elementAt(1));  
  
for (var element in halogens)  
{  
    print(element);  
}  
}
```

To create an empty set, use {} preceded by a type argument, or assign {} to a variable of type Set:

```
void main() {  
  
var names = <String>{};  
// Set<String> names = {}; // This works, too.  
// var names = {}; // Creates a map, not a set.  
  
}
```

Add items to an existing set using the add() or addAll() methods:

```
void main() {  
  
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};  
var elements = <String>{};  
elements.add('fluorine');  
elements.addAll(halogens);  
  
}
```

Maps

In general, a map is an object that associates keys and values. Both keys and values can be any type of object. Each *key* occurs only once, but you can use the same *value* multiple times. Dart support for maps is provided by map literals and the Map type.

```
var gifts = {  
  // Key:    Value  
  'first': 'partridge',  
  'second': 'turtledoves',  
  'fifth': 'golden rings'  
};  
  
var nobleGases = {  
  2: 'helium',  
  10: 'neon',  
  18: 'argon',  
};
```

Add a new key-value pair to an existing map using the subscript assignment operator ([]=):

Dart

```
var gifts = {'first': 'partridge'};  
gifts['fourth'] = 'calling birds'; // Add a key-value pair
```

Retrieve a value from a map using the subscript operator ([]):

Dart

```
void main() {  
  
  var gifts = {'first': 'partridge'};  
  assert(gifts['first'] == 'partridge');  
  
}
```

To print all values of map

```
void main() {  
  
  var nobleGases = {  
    2: 'helium',  
    10: 'neon',  
    18: 'argon',
```

```
};

for(var key in nobleGases.keys) {
  print(nobleGases[key]);
}

}
```

You can create the same objects using a Map constructor:

Dart

```
void main() {

  var gifts = Map<String, String>();
  gifts['first'] = 'partridge';
  gifts['second'] = 'turtledoves';
  gifts['fifth'] = 'golden rings';

  var nobleGases = Map<int, String>();
  nobleGases[2] = 'helium';
  nobleGases[10] = 'neon';
  nobleGases[18] = 'argon';
}
```

Add a new key-value pair to an existing map using the subscript assignment operator ([]=):

Dart

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // Add a key-value pair
```

Objects in Dart resembling javascript objects

We can define javascript type objects using Map:

```
Map<String, String> myObject1 = {
  'name': 'Devin',
  'hairColor': 'brown',
};

print(myObject1);
```

List of map objects

```
var list = [  
  {'name': 'Kamran', 'color' : 'red'},  
  {'name': 'Shahid', 'color' : 'blue'},  
  {'name': 'Zahid', 'color' : 'green'},  
  
];
```

Spread operators

Dart supports the **spread operator** (...) and the **null-aware spread operator** (...?) in list, map, and set literals. Spread operators provide a concise way to insert multiple values into a collection.

For example, you can use the spread operator (...) to insert all the values of a list into another list:

```
void main() {  
  
var list = [1, 2, 3];  
var list2 = [0, ...list];  
assert(list2.length == 4);  
}
```

If the expression to the right of the spread operator might be null, you can avoid exceptions by using a null-aware spread operator (...?):

```
void main() {  
  
var list2 = [0, ...?list];  
assert(list2.length == 1);  
  
}
```

We can modify the map objects using spread operator:

```
Map<String, String> myObject1 = {  
  'name': 'Devin',  
  'hairColor': 'brown',  
};  
  
var myObject2 = {...myObject1, 'name': 'Kamran'};  
print(myObject2);
```

Control-flow operators

Dart offers **collection if** and **collection for** use in list, map, and set literals. You can use these operators to build collections using conditionals (if) and repetition (for).

Here's an example of using **collection if** to create a list with three or four items in it:

```
void main() {  
  
  bool promoActive = false;  
  var nav = ['Home', 'Furniture', 'Plants', if (promoActive) 'Outlet'];  
  print(nav);  
}
```

```
void main() {  
  
  var login = "Manager";  
  
  var nav = ['Home', 'Furniture', 'Plants', if (login case 'Manager')  
    'Inventory'];  
  
  print(nav);  
}
```

Here's an example of using **collection for** to manipulate the items of a list before adding them to another list:

```
void main() {  
  
  var listOfInts = [1, 2, 3];  
  var listOfStrings = ['#0', for (var i in listOfInts) '#$i'];  
  //assert(listOfStrings[1] == '#1');  
  print(listOfStrings);  
}
```

Patterns

Destructuring

<#>

When an object and pattern match, the pattern can then access the object's data and extract it in parts. In other words, the pattern *destructures* the object:

```
void main() {

var numList = [1, 2, 3];
// List pattern [a, b, c] destructures the three elements from numList...
var [a, b, c] = numList;
// ...and assigns them to new variables.
print(a + b + c);

}
```

Variable assignment

A *variable assignment pattern* falls on the left side of an assignment. First, it destructures the matched object. Then it assigns the values to *existing* variables, instead of binding new ones.

Use a variable assignment pattern to swap the values of two variables without declaring a third temporary one:

```
void main() {

var (a, b) = ('left', 'right');
(b, a) = (a, b); // Swap.
print('$a $b'); // Prints "right left".

}
```

Switch statements and expressions

Every case clause contains a pattern. This applies to [switch statements](#) and [expressions](#), as well as [if-case statements](#). You can use [any kind of pattern](#) in a case.

Case patterns are [refutable](#). They allow control flow to either:

- Match and destructure the object being switched on.
- Continue execution if the object doesn't match.

The values that a pattern destructures in a case become local variables. Their scope is only within the body of that case.

```
void main() {

int obj = 3;
const first = 2;
const last = 4;

switch (obj) {
```

```

// Matches if 1 == obj.
case 1:
    print('one');

// Matches if the value of obj is between the
// constant values of 'first' and 'last'.
case >= first && <= last:
    print('in range');

// Matches if obj is a record with two fields,
// then assigns the fields to 'a' and 'b'.
case (var a, var b):
    print('a = $a, b = $b');

default:
}

}

```

[Guard clauses](#) evaluate an arbitrary condition as part of a case, without exiting the switch if the condition is false (like using an if statement in the case body would cause).

```

void main() {
var pair=(20,10);
switch (pair) {
    case (int a, int b):
        if (a > b) print('First element greater');
        // If false, prints nothing and exits the switch.
    case (int a, int b) when a > b:
        // If false, prints nothing but proceeds to next case.
        print('First element greater');
    case (int a, int b):
        print('First element not greater');
}
}

```

For and for-in loops

You can use patterns in [for and for-in loops](#) to iterate-over and destructure values in a collection.

This example uses [object destructuring](#) in a for-in loop to destructure the [MapEntry](#) objects that `a <Map>.entries` call returns:


```
void main() {
  Map<String, int> hist = {
    'a': 23,
    'b': 100,
  };

  for (var MapEntry(key: key, value: count) in hist.entries) {
    print('$key occurred $count times');
  }
}
```

Can also be written as:

```
for (var MapEntry(:key, value: count) in hist.entries) {
  print('$key occurred $count times');
}
```

Functions:

```
bool isNoble(int atomicNumber) {
  return true;
}
```

Although Effective Dart recommends type annotations for public APIs, the function still works if you omit the types:

```
isNoble(int atomicNumber) {
  return true;
}
```

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

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

Named parameters

Named parameters are optional unless they're explicitly marked as required.

When defining a function, use {*param1*, *param2*, ...} to specify named parameters. If you don't provide a default value or mark a named parameter as required, their types must be nullable as their default value will be null:

```
void main() {
  print("Hello world");

  enableFlags(); // will assign default null to bold and hidden
}
```

```
enableFlags(bold:true, hidden: true);
}

void enableFlags({bool? bold, bool? hidden}) {

    print("$bold $hidden");

}
```

To define a default value for a named parameter besides null, use = to specify a default value. The specified value must be a compile-time constant. For example:

```
void main() {
    print("Hello world");
    enableFlags(hidden: false);
}

void enableFlags({bool bold=true, bool hidden=false}) {

    print("$bold $hidden");

}
```

Optional positional parameters

Wrapping a set of function parameters in [] marks them as optional positional parameters. If you don't provide a default value, their types must be nullable as their default value will be null:

```
void main() {
    print("Hello world");

    say("Ali", "hello"); // calling with two arguments
    say("Ali", "hello", "laptop"); // calling with optional argument included
}

void say(String from, String msg, [String? device]) {
    var result = '$from says $msg';
    if (device != null) {
        result = '$result with a $device';
    }
    print(result);
}
```

To define a default value for an optional positional parameter besides null, use = to specify a default value. The specified value must be a compile-time constant. For example:

```

void main() {
  say("Ali", "hello"); // calling with two arguments
}

void say(String from, String msg, [String device = 'carrier pigeon']) {
  print('$from $msg $device');
}

```

The main() function

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

Here's a simple `main()` function:

The file `dartapp.dart` is in the `bin` folder. Run the app like this: `dart run dartapp.dart 1 test`

```

// Run the app like this: dart run args.dart 1 test
void main(List<String> arguments) {
  print(arguments);
}

```

Functions as first-class objects

You can pass a function as a parameter to another function. For example:

```

void main() {

var list = [1, 2, 3];

// Pass printElement as a parameter.
list.forEach(printElement);

}

void printElement(int element) {
  print(element);
}

```

You can also assign a function to a variable, such as:

```

void main() {
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';

print(loudify("hello"));
}

```

Anonymous functions

Most functions are named, such as `main()` or `printElement()`. You can also create a nameless function called an *anonymous function*, or sometimes a *lambda* or *closure*. You might assign an anonymous function to a variable so that, for example, you can add or remove it from a collection.

The following example defines an anonymous function with an untyped parameter, `item`, and passes it to the `map` function. The function, invoked for each item in the list, converts each string to uppercase. Then in the anonymous function passed to `forEach`, each converted string is printed out alongside its length.

```

void main() {

const list = ['apples', 'bananas', 'oranges'];

var result = list.map (

  (item)
  {
    return item.toUpperCase();
  }
);

print(result);

result.forEach((item) {
  print('$item: ${item.length}');
});

/*
// COMBINED FORM

list.map((item) {
  return item.toUpperCase();
}).forEach((item) {
  print('$item: ${item.length}');
});
*/

```

```
}
```

Arrow notation:

Example:

```
void main() {  
  
const list = ['apples', 'bananas', 'oranges'];  
  
    print( list.map((item) => item.toUpperCase()) );  
  
}
```

Example:

```
void main() {  
  
const list = ['apples', 'bananas', 'oranges'];  
  
    list.map((item) => item.toUpperCase())  
        .forEach((item) => print('$item: ${item.length}'));  
  
}
```

typedef functions

```
// 1. Define a typedef for a function that takes two numbers and returns their  
sum  
typedef SumFunction = int Function(int a, int b);  
  
void main() {  
  
    // 2. Implement a function that matches the SumFunction signature  
    int add(int a, int b) => a + b;  
  
    // 3. Use the SumFunction type to declare a variable and assign the add  
function  
    SumFunction sum = add;
```

```
// 4. Call the function using the variable with the SumFunction type
int result = sum(5, 3); // result will be 8

    print("Sum of 5 and 3 is: $result");
}
```

Error handling.

```
void main() {

    misbehave();

}

void misbehave() {
    try {
        dynamic foo = true;
        print(foo++);
    } catch (e) {
        print(e);
    }
}
```

Classes

Simple class example

```
class Point {
    double? x; // Declare instance variable x, initially null.
    double? y; // Declare y, initially null.
}

void main() {
    var point = Point();
    point.x = 4; // Use the setter method for x.
    print(point.x);
}
```

No argument constructor

```
class Car {
```

```

    Car() {
print("no argument constructor called");
    }

/*
//OR
Car();
*/

}

void main() {
Car c = Car();

}

```

One argument generative constructor

```

class Car {
    String model;
    Car(this.model);
}

void main() {
Car c = Car("Toyota");
print(c.model);
}

```

Two argument generative constructor

```

import 'dart:math';

class Point {
    final double x;
    final double y;

    // Sets the x and y instance variables
    // before the constructor body runs.
    Point(this.x, this.y);

    double distanceTo(Point other) {
        var dx = x - other.x;
        var dy = y - other.y;
        return sqrt(dx * dx + dy * dy);
    }
}

```

```

void main() {
    Point p1 = Point(10, 20);
    print("x: ${p1.x}, y: ${p1.y}");
    Point p2 = Point(5, 10);
    print(p2.distanceTo(p1));
}

```

Another example of two argument constructor

If we want to reprocess the variables before assignment to class variables, we can define the constructor as follows.

```

class Car {

    String? name;
    int? age;

    Car(name, age){
        this.name = '$name' ' Ali';
        this.age = age + 30;
    }

}

void main() {
    Car c = Car("Shahid", 45);

    print('${c.name} ${c.age}');

}

```

Calling a constructor from another constructor within same class.

```

class Point {
    int? x;
    int? y;

    Point(int a, int b) {
        x = a;
        y = b;
    }

    Point.fromOrigin() : this(10, 20); // Calls the main constructor
}

```



```

}

void main() {
    var point1 = Point(3, 4);
    var point2 = Point.fromOrigin();
    print('${point1.x} ${point1.y}');
    print('${point2.x} ${point2.y}');
}

```

Named constructors

```

class User {

    User.none();
    User.withId(this.id);
    User.withName(this.name);
    User.withidname(this.id, this.name);

    // we can also place class variables after constructors
    int? id;
    String? name;

}

void main() {

    var user1 = User.none();
    var user2 = User.withId(10);
    var user3 = User.withName("Javed");
    var user4 = User.withidname(10, "Javed");

    print('${user1.id} ${user1.name}');
    print('${user2.id} ${user2.name}');
    print('${user3.id} ${user3.name}');
    print('${user4.id} ${user4.name}');

}

```

Named arguments in a constructor.

```

class Person {
    String? name;
}

```

```

int age;

Person({required this.age, String? name}) { // Name is optional now
    this.name = name ?? "Unknown"; // Default value for name if not provided
}

void main(){
    Person person1 = Person(age: 25); // Specify age first, name is unknown
    print('${person1.age} ${person1.name}');

    Person person2 = Person(name: "Bob", age: 42); // Specify both in any order
    print('${person2.age} ${person2.name}');
}

```

Immutable objects

By adding const, the created object is immutable, its content can't be changed.

```

class Teacher {
    const Teacher(this.name);
    final String name;
}

void main() {
    var teacher = const Teacher("Osman");

    print(teacher.name);

    //teacher.name = "Zahid"; //error
}

```

Optional arguments to a constructor

```

class Person {
    final String name;
    final int? age; // Can be null

    // Constructor with optional age parameter
    const Person(this.name, {this.age});
}

void main() {
    // Create a Person with name and age
    var person1 = Person('Alice', age: 30);
}

```

```
// Create a Person with only name (age will be null)
var person2 = Person('Bob');

print('${person1.name} ${person1.age}');

}
```

Array of objects

```
class Student {

    String name;
    int age;

    // Constructor
    Student(this.name, this.age);

}

void main() {

    // Create an array (List) of MyObject instances
    List<Student> myObjects = [];

    // Add objects to the array
    myObjects.add(Student('Alice', 25));
    myObjects.add(Student('Bob', 30));
    myObjects.add(Student('Charlie', 22));

    // Access and print elements in the array
    for (Student obj in myObjects) {
        print('Name: ${obj.name}, Age: ${obj.age}');
    }
}
```

Printing elements in an array

```
class Student {

    String name;
    int age;

    // Constructor
    Student(this.name, this.age);

}
```

```

}

void main(List<String> arguments) {

// Create an array (List) of MyObject instances
List<Student> myObjects = [];

// Add objects to the array
myObjects.add(Student('Alice', 25));
myObjects.add(Student('Bob', 30));
myObjects.add(Student('Charlie', 22));

// Access and print elements in the array
myObjects.where((student) =>
student.name.contains('Alice')).forEach((student)=>print('Name:
${student.name}, Age: ${student.age}'));

}

```

Looping through array of objects:

```

class Candidate {
    final String name;
    final int yearsExperience;

    Candidate(this.name, this.yearsExperience);

    void interview() {
        print("$name is being interviewed...");
        // Simulate the interview process
        print("$name: Interview went well!");
    }
}

void main() {
    // List of candidates
    List<Candidate> candidates = [
        Candidate("Alice", 6),
        Candidate("Bob", 3),
        Candidate("Charlie", 8),
        Candidate("David", 4),
    ];

    // Filter candidates with 5 or more years of experience and conduct
    interviews

```

```

candidates
    .where((c) => c.yearsExperience >= 5)
    .forEach((c) => c.interview());
}

```

Use **continue** to skip to the next loop iteration:

```

for (int i = 0; i < candidates.length; i++) {
    var candidate = candidates[i];
    if (candidate.yearsExperience < 5) {
        continue;
    }
    candidate.interview();
}

```

Inheritance example.

```

class ParentClass {
    void myfunction() {
        print("This is parent class function");
    }
}

class ChildClass extends ParentClass {
    @override
    void myfunction() {
        super.myfunction();
        print("This is child class function");
    }
}

void main() {
    ChildClass c = ChildClass();
    c.myfunction();
}

```

Using parent class constructor in child class

```

class Person {

String? name;
int? age;

Person(this.name, this.age);
}

```

```

}

class Student extends Person { // Explicitly extend the Person class
    String? regno;

    Student(this.regno, {String? name = "Ali", int? age = 34}) : super(name,
age);
}

void main() {
    Student student = Student("ABC123");
    print(student.name); // Output: Ali
    print(student.age); // Output: 34
    print(student.regno); // Output: ABC123
}

```

Calling named argument constructor from Child class of Parent class

```

class Person {
    String name;
    int age;

    Person({required this.name, required this.age}); // Named arguments
constructor
}

class Student extends Person {
    String id;

    Student({required this.id, required String name, required int age})
        : super(name: name, age: age); // Calling parent constructor with named
arguments
}

void main(){
    Student s = Student(id: "334", name:"Jamal", age:34);
    print('${s.id} ${s.name} ${s.age}');
}

```

Flutter widgets

In flutter, widgets act like tags in html containing information/data. Each widget is represented by class. Widgets call other widgets via class composition principles.

What's the point?

- Widgets are classes used to build UIs.
- Widgets are used for both layout and UI elements.
- Compose simple widgets to build complex widgets.

The core of Flutter's layout mechanism is widgets. In Flutter, almost everything is a widget—even layout models are widgets. The images, icons, and text that you see in a Flutter app are all widgets. But things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.

runApp() function

The runApp() function takes the given Widget and makes it the root of the widget tree.

Example:

Here the Text widget is acting as root widget.

NOTE: When MaterialApp widget is used, we don't need to specify the text direction.

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

void main() {
  runApp(
    const Text(
      "hello world",
      textDirection: TextDirection.ltr,
      style: TextStyle(
        fontSize: 60,
        color: Color.fromARGB(255, 31, 136, 248),
      ),
    ),
  );
}
```

OUTPUT:



Why use const here?

The const keyword used with Text serves two main purposes:

1. **Widget Tree Optimization:** When you use const with a widget, it tells Flutter that the widget and its sub-tree (children) are unlikely to change throughout the lifetime of your app. This allows Flutter to perform some optimizations, such as caching the widget's configuration and avoiding unnecessary rebuilds. This is particularly beneficial for widgets that are expensive to create or render.
2. **Immutability:** Using const with a widget enforces immutability. This means that once the widget is created, its state cannot be changed. This aligns well with the concept of widgets in Flutter, which are supposed to represent a declarative UI state.

Key Points:

- The const keyword is most effective for widgets that are stateless and don't require dynamic updates.
- For widgets that need to update based on user interaction or data changes, using const might not be suitable. In such cases, you'd likely use the StatefulWidget approach.

In summary:

Using `const` with `MaterialApp` in this example helps Flutter optimize the widget tree and enforces immutability, which aligns with the principles of building UIs in Flutter.

Example:

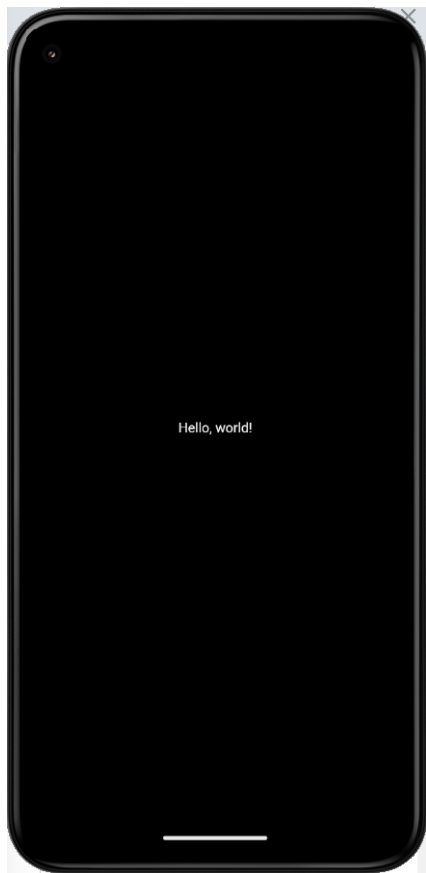
To center align the Text, we use Center widget:

In this example, the widget tree consists of two widgets, the Center widget and its child, the Text widget. The framework forces the root widget to cover the screen, which means the text “Hello, world” ends up centered on screen.

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

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

Output:



MaterialApp Widget:

Many Material Design widgets need to be inside of a `MaterialApp` to display properly, in order to inherit theme data. Therefore, run the application with a `MaterialApp`.

In Flutter, **`MaterialApp`** represents a widget that implements the basic material design visual layout structure. It's typically used as the root widget of a Flutter application. Here's what it provides:

1. **Material Design Components:** `MaterialApp` provides the basic material design components such as `Scaffold`, `AppBar`, `Drawer`, `BottomNavigationBar`, `SnackBar`, and more. These components follow the Material Design guidelines for visual appearance and behavior.
2. **Text Style Warning:** If your app lacks a Material ancestor for text widgets, `MaterialApp` automatically applies an **ugly red/yellow text style**. This serves as a warning to developers that they haven't defined a default text style. Typically, the app's **`Scaffold`** defines the text style for the entire UI.
3. **Routing:** `MaterialApp` provides a navigator that manages a stack of `Route` objects and a route table for mapping route names to builder functions. This allows you to navigate between different screens or "routes" in your app using the **`Navigator`** widget.

4. **Theme:** MaterialApp allows you to define a theme for your entire application using the **theme** property. This includes defining colors, typography, shapes, and other visual properties that are consistent throughout your app.
5. **Localization:** MaterialApp supports internationalization and localization through the **localizationsDelegates** and **supportedLocales** properties. This allows you to provide translations for your app's text and adapt its behavior based on the user's locale.
6. **Accessibility:** MaterialApp includes accessibility features such as support for screen readers and semantic labels, making your app more accessible to users with disabilities.

Overall, MaterialApp serves as the foundation for building Flutter apps with material design principles, providing a consistent and intuitive user experience across different devices and platforms.

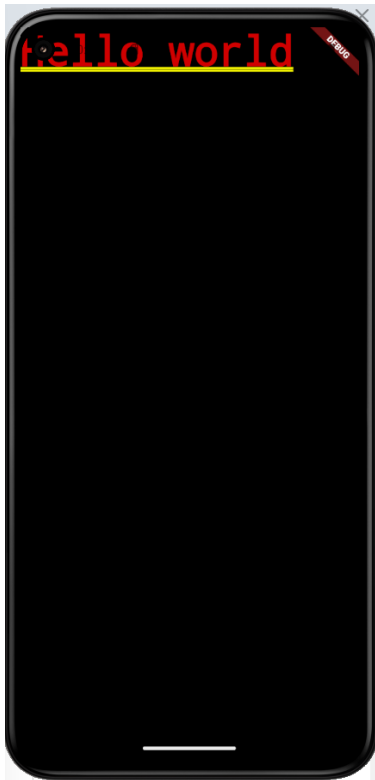
Example:

In this example, MaterialApp widget is made the root widget, and its 'home' property is invoked, in which Text widget is passed.

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

void main() {
  runApp(const MaterialApp(
    home: Text("Hello world"),
  ));
}
```

Output:



Scaffold Widget:

While MaterialApp provides the overall structure and theme for a Flutter app, the Scaffold widget serves as a layout structure for individual screens or "pages" within the app. Here's what the Scaffold widget provides:

1. **App Bar:** Scaffold allows you to easily add an app bar at the top of the screen using the **appBar** property. The app bar typically contains a title, leading and/or trailing actions, and may also include other widgets like tabs or a search bar.
2. **Body Content:** Scaffold's **body** property is where you place the main content of the screen. This can be any widget or combination of widgets, such as text, images, lists, grids, or custom widgets.
3. **Navigation Drawer:** If your app uses a side navigation drawer, Scaffold provides the **drawer** property to easily add one to your screen. The drawer typically contains navigation links or settings options.
4. **Bottom Navigation Bar:** For apps with multiple screens or sections, Scaffold allows you to include a bottom navigation bar using the **bottomNavigationBar** property. This allows users to switch between different sections of the app.

5. **Floating Action Button:** Scaffold provides the **floatingActionButton** property to add a floating action button (FAB) to the screen. FABs are typically used for primary or frequently-used actions, such as adding a new item or starting a new task.
6. **Snackbar:** Scaffold includes methods for displaying snackbars, which are lightweight messages that appear at the bottom of the screen. This can be useful for showing brief feedback or notifications to the user.

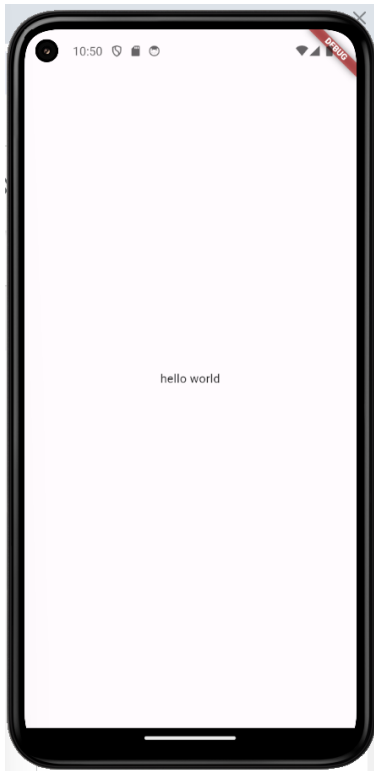
Overall, while `MaterialApp` provides the overall structure and theme for the entire app, `Scaffold` provides a consistent layout structure for individual screens, making it easier to build and organize the UI of your Flutter app.

Example

See how the Point No. 2 of `MaterialApp` is addressed here when we used `Scaffold` widget.

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

void main() {
  runApp(const MaterialApp(
    home: Scaffold(
      body: Center(
        child: Text("hello world")
      )
    )
  ));
}
```



MaterialApp and Material Widget

MaterialApp:

- **Function:** A widget that serves as the foundation for building apps that follow Google's Material Design guidelines.
- **Purpose:** Provides a starting point for your app by configuring essential elements like:
 - **Theme:** Defines the overall look and feel of your app (colors, fonts, etc.)
 - **Routing:** Manages navigation between different screens in your app.
 - **Localization:** Enables support for different languages.
 - **Home:** Sets the initial screen displayed when the app launches.
- **Placement:** Placed at the root of your app's widget tree. There should typically be only one MaterialApp widget in your app.

Material:

- **Function:** A basic widget used for creating UI elements that adhere to Material Design principles.
- **Purpose:** Defines properties related to the visual appearance and behavior of individual UI components, such as:
 - **Elevation:** Creates a shadow effect for a 3D-like feel.
 - **Shape:** Defines the shape of the UI element (rounded corners, etc.).
 - **Clipping:** Controls how the widget's content is displayed within its bounds.
- **Placement:** Used throughout your app's widget tree to build various UI components like buttons, cards, app bars, etc.

Example:

Code snippet

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

void main() {
  runApp(MaterialApp(
    home: Scaffold( // Another Material widget for app structure
      appBar: AppBar(
        title: const Text('My Flutter App'), // Text widget (not directly
from Material)
        backgroundColor: Colors.blue, // Defined in ThemeData (part of
MaterialApp)
      ),
      body: Center(
        child: Material( // Used for specific UI element
          elevation: 4.0, // Material property
          shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(10.0)),
          child: const Text('This is a Material widget'),
        ),
      ),
    ),
  ));
}
```

OUTPUT:



In this example:

- MaterialApp configures the app's theme (including the blue color) and sets the Scaffold as the home screen.
- Scaffold is another Material widget that provides a basic layout structure for most screens in a Material Design app.
- AppBar and Text are not directly from Material, but they can be styled within the MaterialApp's theme.
- The Center widget positions the content in the center.
- Finally, the Material widget is used to create a specific button-like element with elevation and rounded corners.

StatelessWidget

When writing an app, you'll commonly author new widgets that are subclasses of either StatelessWidget or StatefulWidget, depending on whether your widget manages any state. A widget's main job is to implement a `build()` function, which describes the widget in terms of other, lower-level widgets.

Stateless widgets receive arguments from their parent widget, which they store in [final](#) member variables. When a widget is asked to [build\(\)](#), it uses these stored values to derive new arguments for the widgets it creates.

In Flutter, a StatelessWidget is a fundamental building block for user interfaces (UIs). It represents a part of your app's UI that remains static throughout its lifetime. Here's a breakdown of what it means:

Stateless:

- The key characteristic of a StatelessWidget is that it doesn't hold any internal state that can change. This means its appearance is entirely determined by the information passed to it when it's created, and it won't update dynamically on its own.

Widget:

- Everything in Flutter is a widget. Widgets are like building blocks that you assemble to create your app's UI. A StatelessWidget is a specific type of widget that specializes in presenting a fixed UI element.

Building UIs:

- StatelessWidgets are ideal for UI elements whose appearance is based on a set of properties or data provided to them when they are built. They describe how a part of the UI should look at a given point in time.

Benefits:

- Simpler to create and understand compared to StatefulWidgets (which manage state).
- More lightweight and efficient for static UI elements.
- Easier to reason about and test due to their predictable behavior.

Common Use Cases:

- Displaying text, icons, images, or simple layouts.
- Building UI elements that only depend on the data passed to them (e.g., displaying a product name from a list).
- Creating reusable UI components that present a fixed appearance.

In this example, `GreetingText` is a `StatelessWidget` that displays the text "Hello World!" with a specific font size and color. Its appearance remains constant, making it a suitable candidate for a `StatelessWidget`.

When to Use StatelessWidget:

- If a UI element doesn't need to change its appearance based on user interaction or external data updates, use a `StatelessWidget`.
- It's a great choice for building simple and reusable UI components.
- For more complex UIs with dynamic behavior, consider using `StatefulWidget`s.

NOTE:

- Keys are unique identifiers for widgets in Flutter's widget tree.
- They help Flutter track changes to the UI and optimize performance.

Example.

In this example, `GreetingText()` is a custom defined widget.

- **Naming Convention:** In Flutter, custom widgets typically start with a capital letter to differentiate them from built-in widgets provided by the Flutter framework (which are usually in lowercase).
- The **context** object serves as a handle that pinpoints a widget's exact position within the hierarchical structure of widgets that make up your app's UI (User Interface).
- Imagine your app's UI as an inverted tree, with `MaterialApp` at the root and all other widgets branching out from it. Context tells a widget exactly where it sits on that tree.

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

void main() {
  runApp(const MaterialApp(
    home: GreetingText(),
  ));
}

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

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

```

    appBar: AppBar(
      title: const Text('Flutter Greeting'), // Add a title to the app bar
    ),
    body: const Center( // Center the text within the body
      child: Text(
        'Hello World!',
        style: TextStyle(
          fontSize: 32.0,
          color: Colors.blue,
        ),
      ),
    ),
  ),
);
}
}

```

Here's a detailed explanation of the code, focusing on the constructor `const GreetingText ({super.key});`:

1. Constructor Declaration:

- `const GreetingText ({super.key});`: This is the declaration of a constructor for the `GreetingText` class. It has some distinct features:
 - **const Prefix:** The `const` keyword indicates that this constructor creates an immutable instance of the class. This means that once a `GreetingText` object is created, its properties cannot be changed.
 - **Curly Braces:** The curly braces `{}` enclose a single optional parameter, `super.key`.

2. Forwarding Key to Superclass:

- `super.key`: This part of the constructor forwards a special parameter named `key` to the superclass constructor. In this case, the superclass is `StatelessWidget` (since `GreetingText` extends `StatelessWidget`).
- **Key Importance:** Keys are unique identifiers assigned to widgets in Flutter. They play a crucial role in helping the framework efficiently manage the widget tree and determine which widgets need to be rebuilt when the UI state changes.
- **Forwarding Benefits:** By forwarding the `key` to the superclass, `GreetingText` inherits the key management capabilities of `StatelessWidget`, ensuring proper handling of its identity and updates within the widget tree.

3. No Explicit Argument:

- You might be wondering why there's no explicit argument being passed to `super.key`. This is because `key` is an optional parameter that can be specified when creating a `GreetingText` widget. If a `key` is not provided, it will default to `null`.

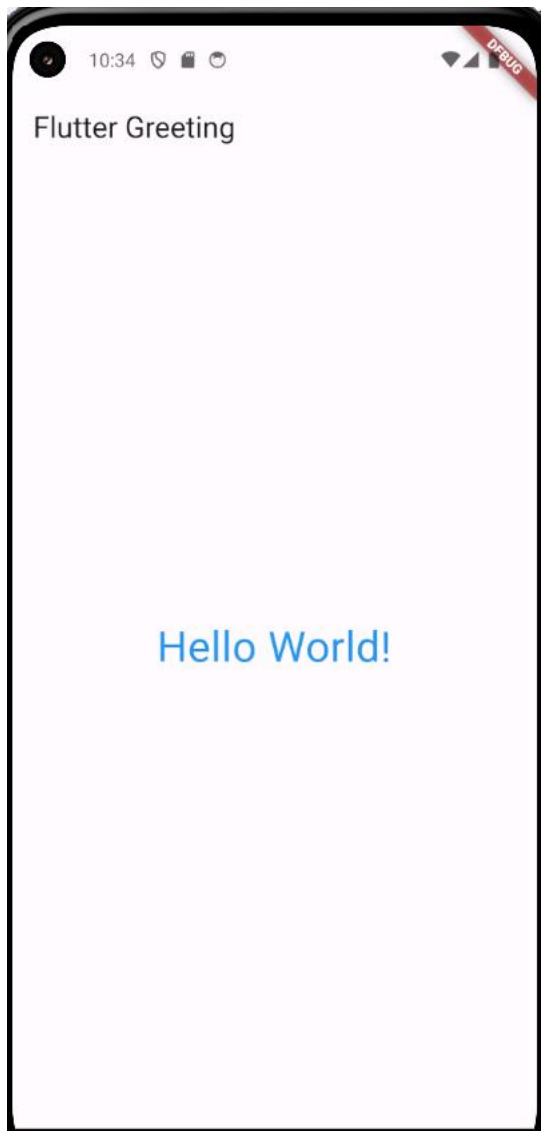
4. Optional Parameter Syntax:

- The curly braces `{}` around `super.key` signify that it's an optional named parameter. However, if you don't provide a `key`, it's perfectly valid to create it without any arguments:

In summary:

- The constructor `const GreetingText ({super.key});` allows for the creation of immutable `GreetingText` widgets with optional keys.
- It forwards the `key` parameter to the superclass for proper widget management.
- Understanding this syntax and the role of keys is essential for effective Flutter development.

OUTPUT:



Row widget.

A widget that displays its children in a horizontal array.

In Flutter, the Row widget is a fundamental layout widget used to arrange its child widgets horizontally in a single row. It's a core building block for creating linear UI elements like:

- Navigation bars with buttons or icons displayed side-by-side.
- Forms with labels and input fields aligned horizontally.
- Layouts with multiple images or text elements displayed in a row.

Key Characteristics:

- **Horizontal Arrangement:** Child widgets are placed from left to right within the available space.

- **Fixed Size:** The Row widget itself doesn't have an intrinsic size. Its size is determined by the combined size of its child widgets and the available space from the parent widget.
- **Multiple Children:** A Row can hold multiple child widgets, allowing you to create complex horizontal layouts.

Customizing Layout:

While child widgets are placed horizontally by default, you can control their alignment and distribution within the Row using two properties:

1. **MainAxisAlignment:** This property determines how the child widgets are positioned along the main axis (horizontal in this case). Options include:
 - MainAxisAlignment.start (default): Aligns child widgets to the left edge of the Row.
 - MainAxisAlignment.center: Centers child widgets horizontally within the Row.
 - MainAxisAlignment.end: Aligns child widgets to the right edge of the Row.
 - MainAxisAlignment.spaceBetween: Distributes child widgets evenly with spaces in between, except for the first and last ones, which are positioned at the edges.
 - MainAxisAlignment.spaceAround: Distributes child widgets evenly with spaces around them.
 - MainAxisAlignment.spaceEvenly: Distributes child widgets with equal space between them, including the first and last ones.
2. **CrossAxisAlignment:** This property controls how the child widgets are positioned along the cross-axis (vertically in this case). Options include:
 - CrossAxisAlignment.start (default): Aligns the top edges of child widgets.
 - CrossAxisAlignment.center: Centers child widgets vertically within the Row.
 - CrossAxisAlignment.end: Aligns the bottom edges of child widgets.
 - CrossAxisAlignment.stretch (stretches child widgets to fill the vertical space of the Row).

Example:

Simple example of Row widget.

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

```

void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Row Example'),
        ),
        body: const Center(
          child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              Text('Item 1', style: TextStyle(color: Colors.black)),
              Text('Item 2', style: TextStyle(color: Colors.black)),
              Text('Item 3', style: TextStyle(color: Colors.black)),
            ],
          ),
        ),
      ),
    );
  }
}

```

Output:



Making Child Widgets Flexible:

If a Row has more child widgets than can fit comfortably in the available space, they might overflow. To handle this, you can wrap child widgets with the Expanded widget. This allows child widgets to flex and share the available horizontal space proportionally.

If the non-flexible contents of the row (those that are not wrapped in [Expanded](#) or [Flexible](#) widgets) are together wider than the row itself, then the row is said to have overflowed. When a row overflows, the row does not have any remaining space to share between its [Expanded](#) and [Flexible](#) children. The row reports this by drawing a yellow and black striped warning box on the edge that is overflowing. If there is room on the outside of the row, the amount of overflow is printed in red lettering.

Example:

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

void main() {
  runApp(const MyApp());
}
```



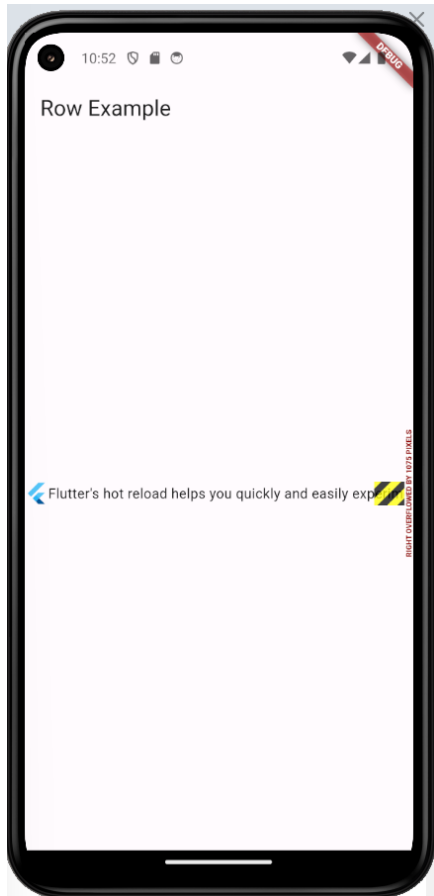
```

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Row Example'),
        ),
        body: const Center(
          child: Row(
            children: <Widget>[
              FlutterLogo(),
              Text(
                "Flutter's hot reload helps you quickly and easily
experiment, build UIs, add features, and fix bug faster. Experience sub-second
reload times, without losing state, on emulators, simulators, and hardware for
iOS and Android."),
              Icon(Icons.sentiment_very_satisfied),
            ],
          ),
        ),
      ),
    );
  }
}

```

Output:



Example:

Controlling text direction in a row:

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

void main() {
  runApp(const MyApp());
}

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

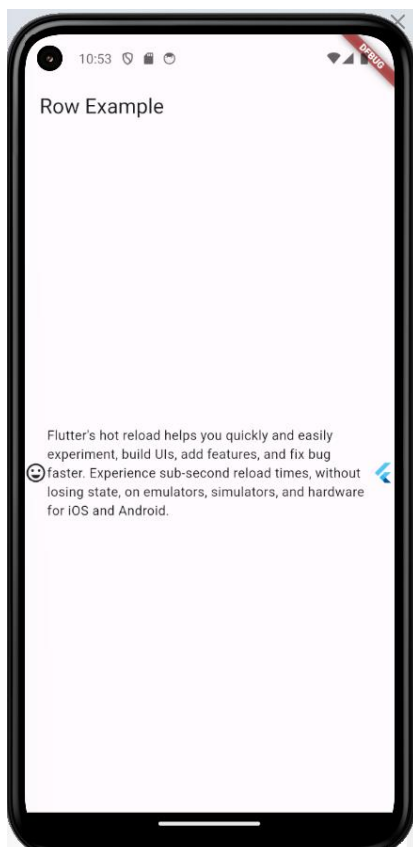
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Row Example'),
        ),
        body: const Center(
          child: Row(
            textDirection: TextDirection.rtl,
```

```

        children: <Widget>[
          FlutterLogo(),
          Expanded(
            child: Text(
              "Flutter's hot reload helps you quickly and easily
experiment, build UIs, add features, and fix bug faster. Experience sub-second
reload times, without losing state, on emulators, simulators, and hardware for
iOS and Android."),
            ),
          Icon(Icons.sentiment_very_satisfied),
        ],
      ),
    ),
  ),
);
}
}

```

Output:



Another example of Expanded widget in Row

Example:

```

import 'package:flutter/material.dart';

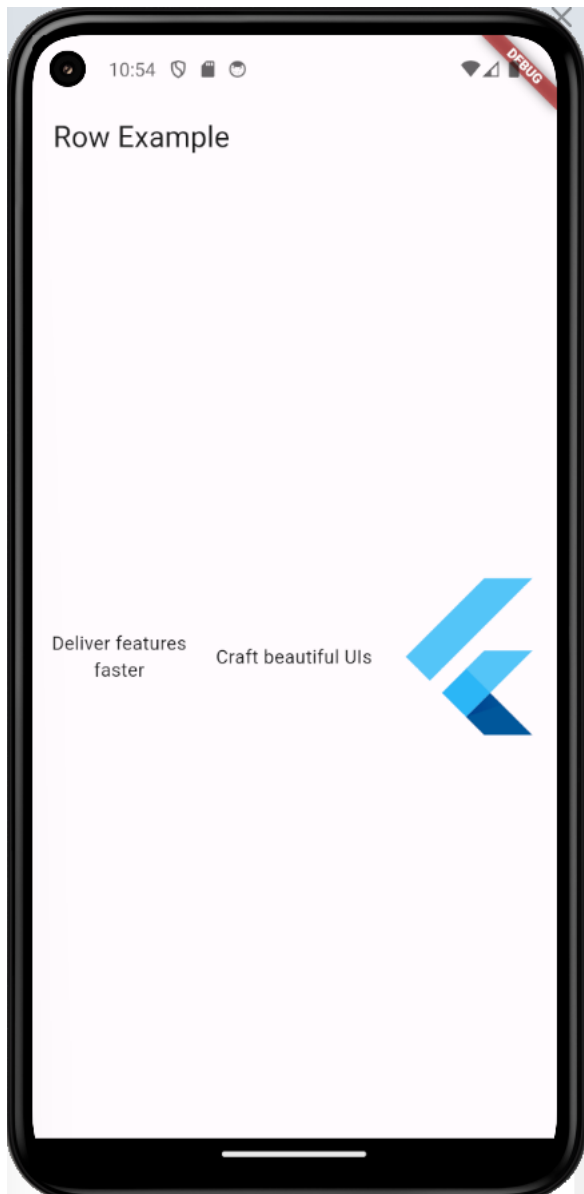
void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Row Example'),
        ),
        body: const Center(
          child: Row(
            children: <Widget>[
              Expanded(
                child: Text('Deliver features faster',
                  textAlign: TextAlign.center),
              ),
              Expanded(
                child: Text('Craft beautiful UIs', textAlign:
TextAlign.center),
              ),
              Expanded(
                child: FittedBox(
                  child: FlutterLogo(),
                ),
              ),
            ],
          ),
        ),
      ),
    );
  }
}

```

Output:



Example of mainAxisAlignment using Text widget

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

void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
```

```
appBar: AppBar(  
  title: const Text('Row Example'),  
)  
,  
body: const Center(  
  child: Row(  
    mainAxisAlignment: MainAxisAlignment.spaceAround,  
    children: <Widget>[  
      Text("ITEM 1"),  
      Text("ITEM 2"),  
      Text("ITEM 3"),  
    ],  
  ),  
)  
,  
)  
);  
}
```

Output:



Example of mainAxisAlignment and crossAxisAlignment using SizedBox and Text widget:

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

void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Row Example'),
        ),
        body: const Center(
          child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceAround,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: <Widget>[
              SizedBox(
                height: 20.0, // Approach 2: Set height of SizedBox
                child: Text('ITEM 1'),
              ),
              SizedBox(
                height: 140.0, // Approach 2: Set height of SizedBox
                child: Text('ITEM 2'),
              ),
              SizedBox(
                height: 20.0, // Approach 2: Set height of SizedBox
                child: Text('ITEM 3'),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Output:



Example of mainAxisAlignment and crossAxisAlignment using Container Widget in Row

NOTE: If the heights of your container widgets are the same as the available space in the Row, you will see no difference in the crossAxisAlignment behavior.

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

void main() {
  runApp(const MyApp());
}

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

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

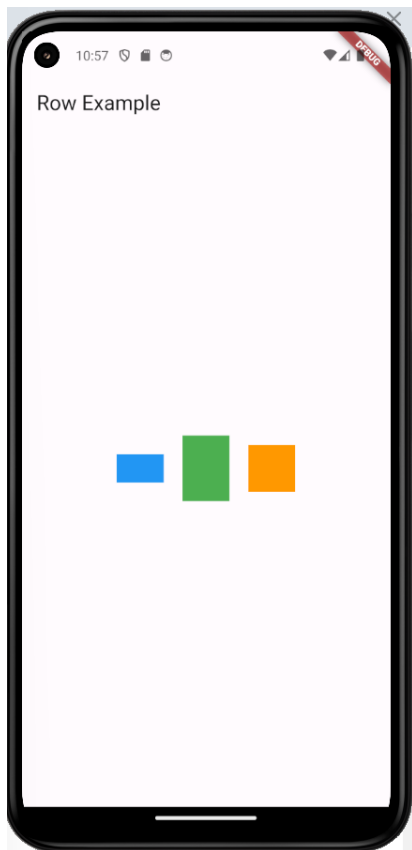


```

        title: const Text('Row Example'),
      ),
      body: Center(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.center,
          crossAxisAlignment: CrossAxisAlignment.center,
          children: [
            Container(
              width: 50,
              height: 30,
              color: Colors.blue,
            ),
            const SizedBox(width: 20),
            Container(
              width: 50,
              height: 70,
              color: Colors.green,
            ),
            const SizedBox(width: 20),
            Container(
              width: 50,
              height: 50,
              color: Colors.orange,
            ),
          ],
        ),
      ),
    ),
  );
}
}

```

Output:



Column widget example

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

void main() {
  runApp(const MyApp());
}

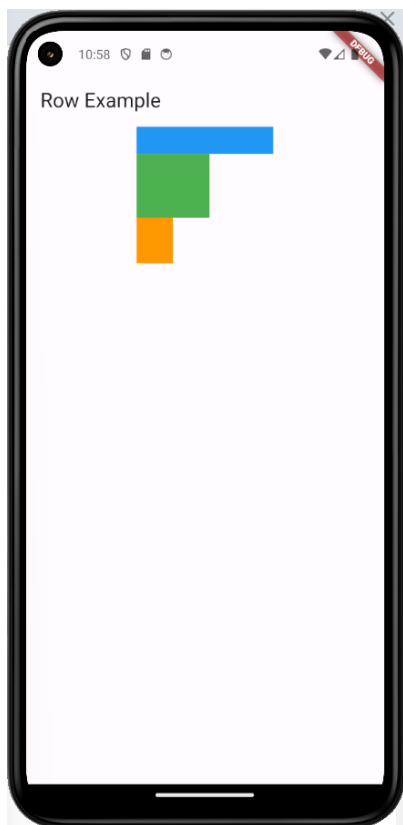
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Row Example'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.start,
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              Container(
```

```

        width: 150,
        height: 30,
        color: Colors.blue,
      ),
      const SizedBox(width: 20),
      Container(
        width: 80,
        height: 70,
        color: Colors.green,
      ),
      const SizedBox(width: 20),
      Container(
        width: 40,
        height: 50,
        color: Colors.orange,
      ),
    ],
  ),
),
);
}
}

```



Example of combining different widgets

- The context object serves as a handle that pinpoints a widget's exact position within the hierarchical structure of widgets that make up your app's UI (User Interface).
- Imagine your app's UI as an inverted tree, with MaterialApp at the root and all other widgets branching out from it. Context tells a widget exactly where it sits on that tree.

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

void main() {
  runApp(
    const MaterialApp(
      title: 'My app', // used by the OS task switcher
      home: SafeArea(
        child: MyScaffold(),
      ),
    ),
  );
}

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

  @override
  Widget build(BuildContext context) {
    // Material is a conceptual piece
    // of paper on which the UI appears.
    return Material(
      // Column is a vertical, linear layout.
      child: Column(
        children: [
          MyAppBar(
            title: Text(
              'Example title',
              style: Theme.of(context) //
                .primaryTextTheme
                .titleLarge,
            ),
          ),
          const Expanded(
            child: Center(
              child: Text('Hello, world!'),
            ),
          ),
        ],
      ),
    );
  }
}
```

```

    ),
  );
}
}

class MyAppBar extends StatelessWidget {
  const MyAppBar({required this.title, super.key});

  // Fields in a Widget subclass are always marked "final".

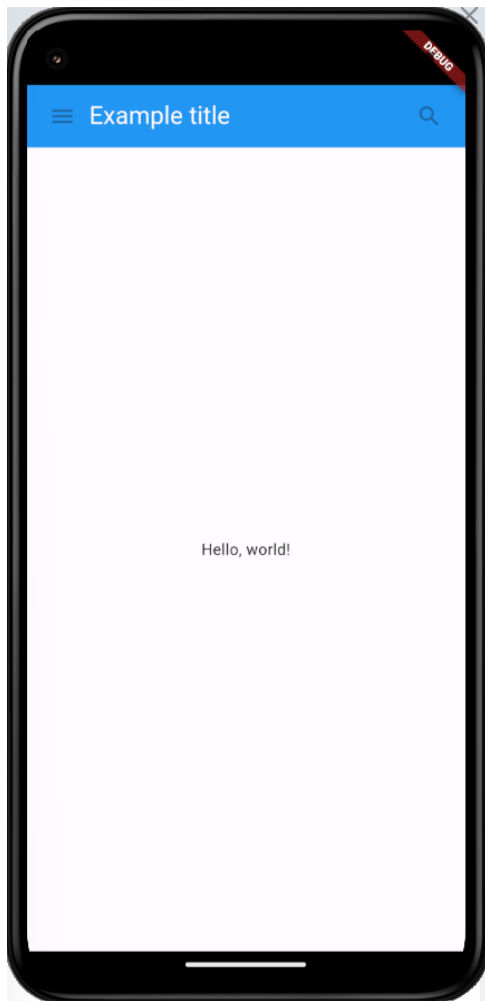
  final Widget title;

  @override
  Widget build(BuildContext context) {
    return Container(
      height: 56, // in logical pixels
      padding: const EdgeInsets.symmetric(horizontal: 8),
      decoration: BoxDecoration(color: Colors.blue[500]),

      // Row is a horizontal, linear layout.
      child: Row(
        children: [
          const IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null, // null disables the button
          ),
          // Expanded expands its child
          // to fill the available space.
          Expanded(
            child: title,
          ),
          const IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
          ),
        ],
      ),
    );
  }
}

```

Output:



Example: Using the MaterialApp and Scaffold Widget with a button

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

void main() {
  runApp(
    const MaterialApp(
      title: 'Flutter Tutorial',
      home: TutorialHome(),
    ),
  );
}

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

  @override
  Widget build(BuildContext context) {
    // Scaffold is a layout for
    // the major Material Components.
```

```

return Scaffold(
  appBar: AppBar(
    leading: const IconButton(
      icon: Icon(Icons.menu),
      tooltip: 'Navigation menu',
      onPressed: null,
    ),
    title: const Text('Example title'),
    actions: const [
      IconButton(
        icon: Icon(Icons.search),
        tooltip: 'Search',
        onPressed: null,
      ),
    ],
  ),
  // body is the majority of the screen.
  body: const Center(
    child: Text('Hello, world!'),
  ),
  floatingActionButton: const FloatingActionButton(
    tooltip: 'Add', // used by assistive technologies
    onPressed: null,
    child: Icon(Icons.add),
  ),
);
}

```

Explanation:

Notice that widgets are passed as arguments to other widgets. The `Scaffold` widget takes a number of different widgets as named arguments, each of which are placed in the `Scaffold` layout in the appropriate place. Similarly, the `AppBar` widget lets you pass in widgets for the `leading` widget, and the `actions` of the `title` widget. This pattern recurs throughout the framework and is something you might consider when designing your own widgets.



Example: Adding a simple button and its function:

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

void main() {
  runApp( MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("This is title"),
        centerTitle: true,
        backgroundColor: Colors.blue,
      ),
      body: Center(
        child: Text("This is body" ) ,
        floatingActionButton: FloatingActionButton(
          onPressed:() => showHelloWorld(),
          child: Text('Click'),
        ),
      ),
    ),
  ));
}
```



```

}

void showHelloWorld() {

    print("Hello, World!");
    // You can replace the print statement with any code to display "Hello,
    World!" on the screen.
}

```

Example: Add a simple button and print message using print function

```

import 'package:flutter/material.dart';

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: MyButton(),
        ),
      ),
    ),
  );
}

class MyButton extends StatelessWidget {

  const MyButton( {super.key});

  @override
  Widget build(BuildContext context) {

    return GestureDetector(
      onTap: () {
        print('MyButton was tapped!');
      },
      child: Container(
        height: 50,
        padding: const EdgeInsets.all(8),
        margin: const EdgeInsets.symmetric(horizontal: 8),
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(5),
          color: Colors.lightGreen[500],
        ),
        child: const Center(
          child: Text('Engage'),

```

```
    ),  
  ),  
);  
}  
}
```

Output:

The output will be shown in blue text of debug console.

Example: Add a simple button and print message using ScaffoldMessenger

```
import 'package:flutter/material.dart';  
  
void main() {  
  runApp(  
    const MaterialApp(  
      home: Scaffold(  
        body: Center(  
          child: MyButton(),  
        ),  
      ),  
    ),  
  );  
}  
  
class MyButton extends StatelessWidget {  
  
  const MyButton( {super.key});  
  
  @override  
  Widget build(BuildContext context) {  
  
    return GestureDetector(  
      onTap: () {  
        ScaffoldMessenger.of(context).showSnackBar(  
          const SnackBar(  
            content: Text('MyButton was tapped!'),  
            backgroundColor: Colors.blue,  
          ),  
        );  
      },  
    ),  
    child: Container(  
      width: 100,  
      height: 50,  
      decoration: BoxDecoration(  
        color: Colors.blue,  
        borderRadius: BorderRadius.circular(10),  
      ),  
    ),  
  ),  
);  
}
```

```

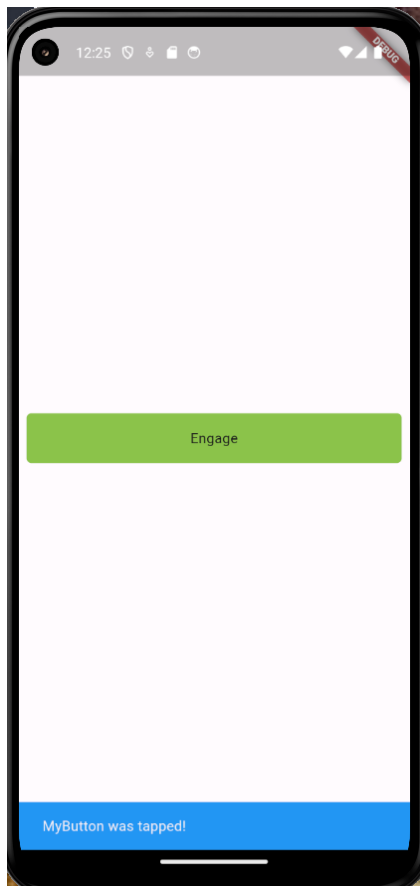
    height: 50,
    padding: const EdgeInsets.all(8),
    margin: const EdgeInsets.symmetric(horizontal: 8),
    decoration: BoxDecoration(
      borderRadius: BorderRadius.circular(5),
      color: Colors.lightGreen[500],
    ),
    child: const Center(
      child: Text('Engage'),
    ),
  ),
);
}
}

```

The [GestureDetector](#) widget doesn't have a visual representation but instead detects gestures made by the user. When the user taps the [Container](#), the GestureDetector calls its [onTap\(\)](#) callback, in this case printing a message to the console. You can use GestureDetector to detect a variety of input gestures, including taps, drags, and scales.

Output:

The blue band at below of screen is output



StatefulWidget

StatefulWidgets are special widgets that know how to generate State objects, which are then used to hold state.

Example: Display message in Text field on button click

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

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Display(),
        ),
      ),
    ),
  );
}
```

```

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

  @override
  State<Display> createState() => _DisplayState();
}

class _DisplayState extends State<Display> {
  String message = ""; // Store the message to display

  void _displayMessage() {
    setState(() {
      message = "Hello World"; // Update the message on button click
    });
  }

  @override
  Widget build(BuildContext context) {
    return Row( // Modified to display the message
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        ElevatedButton(
          onPressed: _displayMessage, // Call the new function on button press
          child: const Text('Display Message'), // Updated button text
        ),
        const SizedBox(width: 16),
        Text(message), // Display the message instead of the Display
      ],
    );
  }
}

```

Example: A counter application

```

import 'package:flutter/material.dart';

class Counter extends StatefulWidget {
  // This class is the configuration for the state.
  // It holds the values (in this case nothing) provided
  // by the parent and used by the build method of the
  // State. Fields in a Widget subclass are always marked
  // "final".

```

```

const Counter({super.key});

@override
State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter = 0;

  void _increment() {
    setState(() {
      // This call to setState tells the Flutter framework
      // that something has changed in this State, which
      // causes it to rerun the build method below so that
      // the display can reflect the updated values. If you
      // change _counter without calling setState(), then
      // the build method won't be called again, and so
      // nothing would appear to happen.
      _counter++;
    });
  }

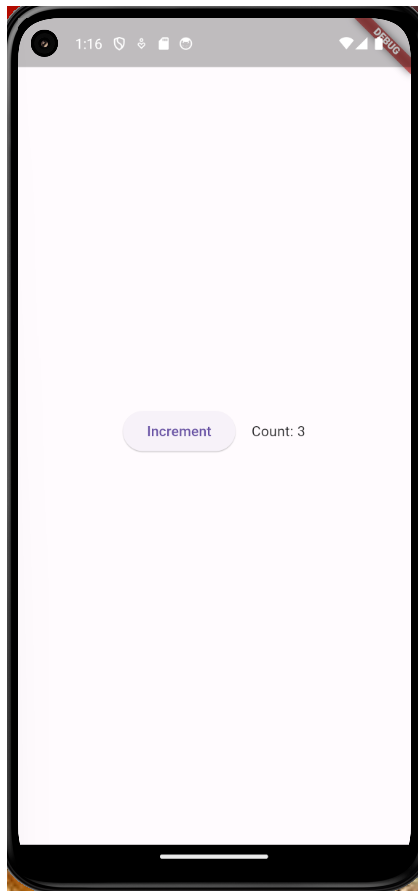
  @override
  Widget build(BuildContext context) {
    // This method is rerun every time setState is called,
    // for instance, as done by the _increment method above.
    // The Flutter framework has been optimized to make
    // rerunning build methods fast, so that you can just
    // rebuild anything that needs updating rather than
    // having to individually changes instances of widgets.
    return Row(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        ElevatedButton(
          onPressed: _increment,
          child: const Text('Increment'),
        ),
        const SizedBox(width: 16),
        Text('Count: $_counter'),
      ],
    );
  }
}

void main() {
  runApp(
    const MaterialApp(

```

```
home: Scaffold(  
  body: Center(  
    child: Counter(),  
  ),  
,  
,  
,  
);  
}
```

Output:



Example: Two counters

```
import 'package:flutter/material.dart';  
  
class Counter extends StatefulWidget {  
  
  const Counter({super.key});  
  
  @override  
  State<Counter> createState() => _CounterState();  
}
```

```

class _CounterState extends State<Counter> {
  int _counter1 = 0;
  int _counter2 = 0;

  void _increment1() {
    setState(() {
      _counter1++;
    });
  }

  void _increment2() {
    setState(() {
      _counter2++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            ElevatedButton(
              onPressed: _increment1,
              child: const Text('Increment1'),
            ),
            const SizedBox(width: 16),
            Text('Count1: $_counter1'),
          ],
        ),
        Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            ElevatedButton(
              onPressed: _increment2,
              child: const Text('Increment'),
            ),
            const SizedBox(width: 16),
            Text('Count2: $_counter2'),
          ],
        ),
      ],
    );
  }
}

```

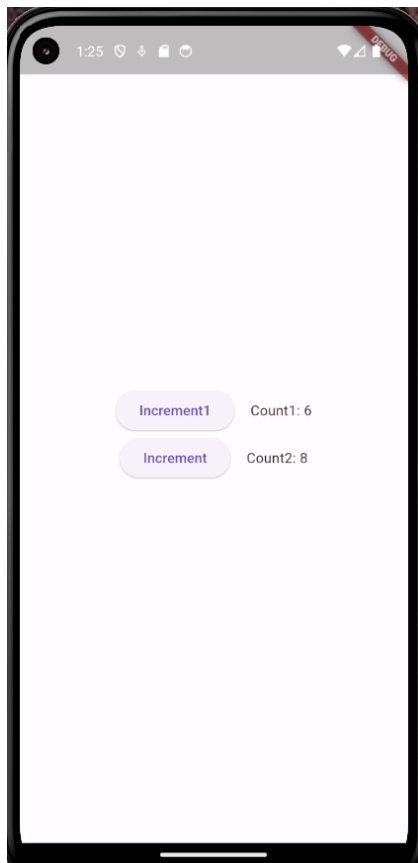


```

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Counter(),
        ),
      ),
    ),
  );
}

```

Output:



Example: Two counters and their sum.

```

import 'package:flutter/material.dart';

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(

```

```

        child: Counter(),
      ),
    ),
  );
}

class Counter extends StatefulWidget {

  const Counter({super.key});

  @override
  State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _counter1 = 0;
  int _counter2 = 0;
  int _sum = 0;

  void _increment1() {
    setState(() {
      _counter1++;
    });
  }

  void _increment2() {
    setState(() {
      _counter2++;
    });
  }

  void _sumcounter() {
    setState(() {
      _sum = _counter1 + _counter2;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column( // Use Column for vertical layout
      mainAxisAlignment: MainAxisAlignment.center, // Center the content
      vertically
      children: <Widget>[
        Row( // First row with a button and Text
          mainAxisAlignment: MainAxisAlignment.center, // Center content in the
          row

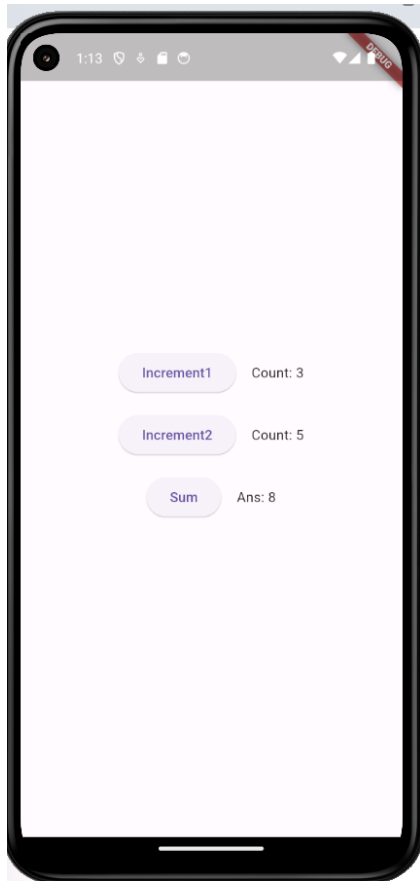
```

```

        children: <Widget>[
          ElevatedButton(
            onPressed: _increment1,
            child: const Text('Increment1'),
          ),
          const SizedBox(width: 16), // Spacing between elements
          Text('Count: $_counter1'),
        ],
      ),
      const SizedBox(height: 16), // Spacing between rows
      Row( // Second row with another button and Text
        mainAxisAlignment: MainAxisAlignment.center, // Center content in the
row
        children: <Widget>[
          ElevatedButton(
            onPressed: _increment2,
            child: const Text('Increment2'),
          ),
          const SizedBox(width: 16), // Spacing between elements
          Text('Count: $_counter2'),
        ],
      ),
      const SizedBox(height: 16), // Spacing between rows
      Row( // Second row with another button and Text
        mainAxisAlignment: MainAxisAlignment.center, // Center content in the
row
        children: <Widget>[
          ElevatedButton(
            onPressed: _sumcounter,
            child: const Text('Sum'),
          ),
          const SizedBox(width: 16), // Spacing between elements
          Text('Ans: $_sum'),
        ],
      ),
    ],
  );
}
}

```

Output:



Example: Introducing separate widgets for counter increment and counter display:

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

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Counter(),
        ),
      ),
    ),
  );
}

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

  @override
```

```

    State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
    int _counter = 0;

    void _increment() {
        setState(() {
            ++_counter;
        });
    }

    @override
    Widget build(BuildContext context) {
        return Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
                CounterIncrementor(onPressed: _increment),
                const SizedBox(width: 16),
                CounterDisplay(count: _counter),
            ],
        );
    }
}

class CounterIncrementor extends StatelessWidget {
    const CounterIncrementor({required this.onPressed, super.key});

    final VoidCallback onPressed;

    @override
    Widget build(BuildContext context) {
        return ElevatedButton(
            onPressed: onPressed,
            child: const Text('Increment'),
        );
    }
}

class CounterDisplay extends StatelessWidget {
    const CounterDisplay({required this.count, super.key});

    final int count;

    @override
    Widget build(BuildContext context) {
        return Text('Count: $count');
    }
}

```

```
}
```

Output:

The output is same to the single counter app.

Example to take user input and display as Text field

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

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: InputDisplay(),
        ),
      ),
    ),
  );
}

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

  @override
  State<InputDisplay> createState() => _InputDisplayState();
}

class _InputDisplayState extends State<InputDisplay> {
  String userInput = ""; // Store user input
  String message = "";

  void _displayInput() {
    setState(() {
      message = userInput; // Update message with user input
    });
  }

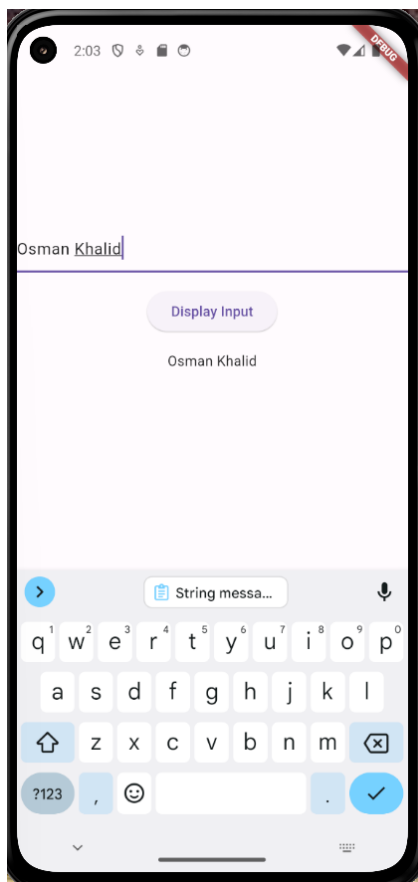
  @override
  Widget build(BuildContext context) {
    return Column( // Use Column for vertical layout
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        TextField( // Text input field
          onChanged: (value) => setState(() => userInput = value), // Update
userInput on change
          decoration: const InputDecoration(
```

```

        hintText: 'Enter your message', // Hint text for the user
      ),
    ),
    const SizedBox(height: 16),
    ElevatedButton(
      onPressed: _displayInput, // Call the function to display input
      child: const Text('Display Input'), // Updated button text
    ),
    const SizedBox(height: 16),
    Text(message), // Display the message
  ],
);
}
}

```

Output:



Example to take inputs from Two text fields and display with button

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

void main() {
  runApp(const MyApp());
}

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

  @override
  State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  // Controllers to store text from each box
  final TextEditingController _textController1 = TextEditingController();
  final TextEditingController _textController2 = TextEditingController();
  String _displayText = ""; // Variable to store combined text

  void _onPressed() {
    // Combine text from controllers and update display text
    setState(() {
      _displayText = "${_textController1.text} - ${_textController2.text}";
    });
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Text Input Example'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              TextField(
                controller: _textController1,
                decoration: const InputDecoration(
                  hintText: 'Enter Text 1',
                ),
              ),
              const SizedBox(height: 10),
              TextField(
```



```

        controller: _textController2,
        decoration: const InputDecoration(
          hintText: 'Enter Text 2',
        ),
      ),
      const SizedBox(height: 20),
      ElevatedButton(
        onPressed: _onPressed,
        child: const Text('Display Text'),
      ),
      const SizedBox(height: 10),
      Text(_displayText),
    ],
  ),
),
),
);
}
}

```

An input form:

```

import 'package:flutter/material.dart';

void main() {
  runApp(const MaterialApp(
    home: ResponsiveForm(),
  ));
}

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

  @override
  State<ResponsiveForm> createState() => _ResponsiveFormState();
}

class _ResponsiveFormState extends State<ResponsiveForm> {
  // Add state variables for form fields
  final _nameController = TextEditingController();
  final _emailController = TextEditingController();
  final _phoneController = TextEditingController();
  final _addressController = TextEditingController();

  @override
  Widget build(BuildContext context) {

```

```

return Scaffold(
  appBar: AppBar(
    title: const Text('User Information Form'),
  ),
  body: SingleChildScrollView(
    padding: const EdgeInsets.all(20.0),
    child: LayoutBuilder(
      builder: (context, constraints) {
        return Row(
          children: [
            Expanded(
              child: _buildForm(context),
            ),
          ],
        );
      },
    ),
  ),
);
}

```

```

Widget _buildForm(BuildContext context) {
  return Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      TextField(
        controller: _nameController,
        decoration: const InputDecoration(
          labelText: 'Name',
        ),
      ),
      const SizedBox(height: 10.0),
      TextField(
        controller: _emailController,
        decoration: const InputDecoration(
          labelText: 'Email',
        ),
        keyboardType: TextInputType.emailAddress,
      ),
      const SizedBox(height: 10.0),
      TextField(
        controller: _phoneController,
        decoration: const InputDecoration(
          labelText: 'Phone Number',
        ),
        keyboardType: TextInputType.phone,
      ),
      const SizedBox(height: 10.0),
    ],
  );
}

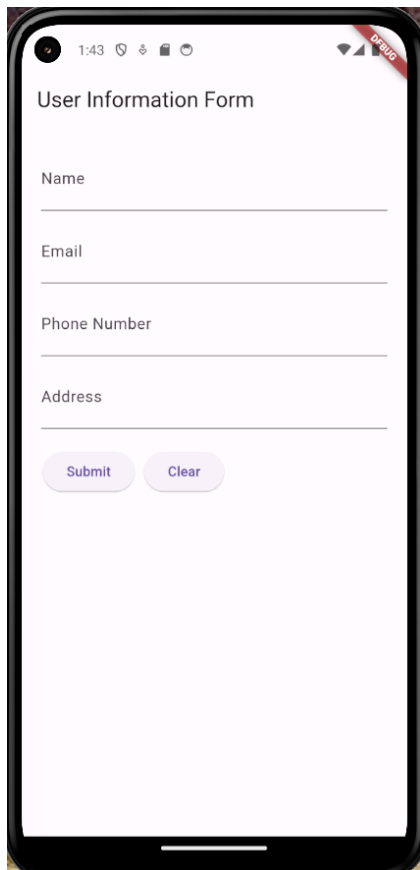
```

```

TextField(
  controller: _addressController,
  decoration: const InputDecoration(
    labelText: 'Address',
  ),
  maxlines: null, // This allows multiline input
),
const SizedBox(height: 20.0),
Row(
  children: [
    const SizedBox(width: 2.0),
    ElevatedButton(
      onPressed: () => { /* Handle Submit Button */},
      child: const Text('Submit'),
    ),
    const SizedBox(width: 10.0),
    ElevatedButton(
      onPressed: () => { /* Handle Clear Button */},
      child: const Text('Clear'),
    ),
  ],
),
],
);
}
}

```

Output:



The image shows a mobile application interface for a 'User Information Form'. The screen is framed by a black border, representing a smartphone. At the top, a status bar displays the time '1:43' and various system icons. The app's title bar is white with a red 'demo' label in the top right corner. The form itself has a light pink background and contains four text input fields labeled 'Name', 'Email', 'Phone Number', and 'Address'. Below these fields are two rounded buttons: 'Submit' in blue and 'Clear' in light purple. The bottom of the screen shows a white home indicator bar.

1:43

User Information Form

Name

Email

Phone Number

Address

Submit Clear