502071

Cross-Platform Mobile Application Development

Dart Programming Language

1

OUTLINE

1. Introduction to Dart
2. Conditions and loops
3. Function in Dart
4. Collections in Dart
5. Null Safety
6. Asynchronous Programming

Introduction to Dart

- Dart is an open-source programming language developed by Google.
- Dart is an object-oriented language with C-style syntax which can optionally trans compile into JavaScript.
- It is meant for both server side as well as the user side.
- It supports programming concepts like interfaces, classes. Unlike other programming languages, Dart doesn’t support arrays (but list).

Introduction to Dart

- Dart is a client-optimized language for fast apps on any platform.
- Dart is Object-oriented language and is quite similar to that of Java Programming. Dart is extensively use to create single-page websites and web-applications.
- The Dart SDK comes with its compiler – the Dart VM and a utility dart2js which is meant for generating Javascript equivalent of a Dart Script so that it can be run on those sites also which don’t support Dart.

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.

Dart vs. 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.

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 gained popularity in recent days because of flutter.
- Dart 3.2 was released in November 2023
- Refer to https://dart.dev/guides/language/evolution for more information.

Install Flutter and Dart

- The Flutter SDK includes the full Dart SDK, and has Dart's dart command-line interface in its bin folder.
- To install and run Flutter, your development environment must meet these minimum requirements:
  - Operating Systems: macOS, Windows 10 or later (64-bit), x86-64 based.
  - Disk Space: 3 GB (does not include disk space for IDE/tools).
  - Tools: Flutter depends on these tools being available in your environment.
    - Windows PowerShell 5.0 or newer (this is pre-installed with Windows 10)
    - Xcode (macOS only)
    - Git for Windows 2.x, with the Use Git from the Windows Command Prompt option.

Install Flutter and Dart

1. Download the following installation bundle to get the latest stable release of the Flutter SDK:

   flutter_windows_3.3.10-stable.zip

   For other release channels, and older builds, see the SDK releases page.

2. Extract the zip file and place the contained `flutter` in the desired installation location for the Flutter SDK (for example, `C:\src\flutter`).

Update your path

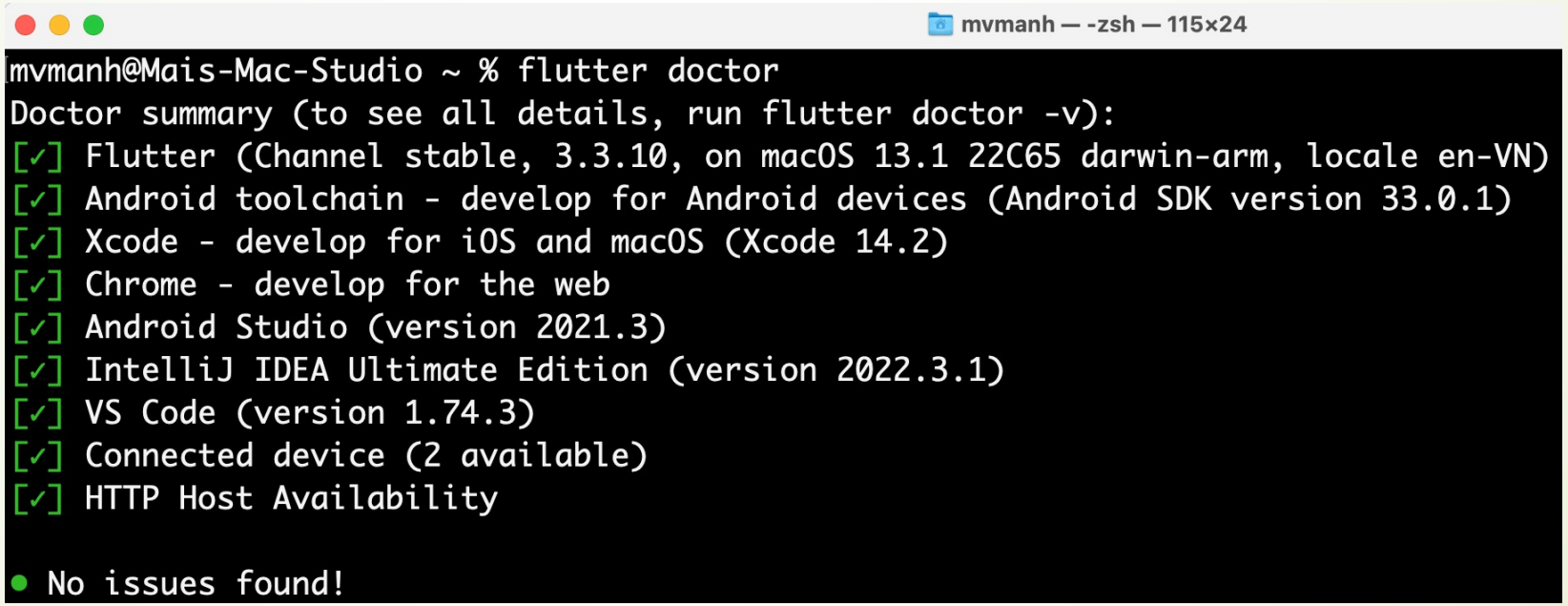
If you wish to run Flutter commands in the regular Windows console, take these steps to add Flutter to the `PATH` environment variable:

- From the Start search bar, enter 'env' and select **Edit environment variables for your account**.
- Under **User variables** check if there is an entry called **Path**:
  - If the entry exists, append the full path to `flutter\bin` using `;` as a separator from existing values.
  - If the entry doesn't exist, create a new user variable named `Path` with the full path to `flutter\bin` as its value.

You have to close and reopen any existing console windows for these changes to take effect.

Install Flutter and Dart

- Run flutter doctor
  - This command checks your environment and displays a report of the status of your Flutter installation. Check the output carefully for other software you might need to install or further tasks to perform (shown in **bold** text).

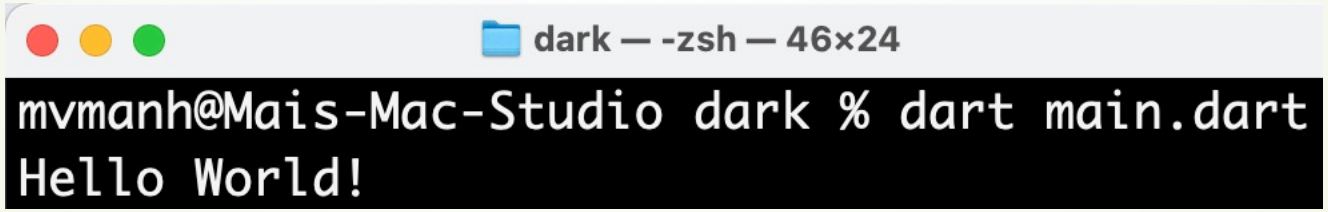


Basic Dart Program

- 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!");
}
```

- To run the program, use the following command: **dart main.js** where main.js is the file containing the above source code.



Basic Dart Program

- Basic Dart Program To Join One Or More Variables
  - This joining process in dart is called string interpolation.

```
void main(){
    var firstName = "John";
    var lastName = "Doe";
    print("Full name is $firstName $lastName");
}
```

Variables in Dart

- 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";
```

- They are called data types.
  - **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 value. E.g. 10.0, -10.2, 85.698 [Decimal is included]
  - **num**: For storing any types of number. E.g. 10, 20.2, -20 [both int and double]
  - **bool**: For storing true or false value. E.g. true, false [Only stores true or false values]
  - **var**: For storing any value. E.g. ‘Bimal’, 12, ‘z’, true

String In Dart

- String data types help you to store text data. You can use single or double, or triple quotes to represent String. Single line String is written in single or double quotes, whereas multi-line strings are written in triple quotes.

```
void main() {
    String text1 = 'This is an example of a single-line string.';
    String text2 = "This is an example of a single line string";
    String text3 = """This is a multiline line
    string using the triple-quotes.
    This is tutorial on dart strings.
    """;
    print(text1);
    print(text2);
    print(text3);
}
```

String In Dart

- 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.

```
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
}
```

String In Dart

- 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.

Null Safety

Null Safety

- 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.
- Null safety avoids null errors, runtime bugs, vulnerabilities, and system crashes which are difficult to find and fix.

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.
- In null safety, variables cannot be null unless you explicitly specify that they can.

```dart
String processName(String name) {
    var firstName = name.split(' ')[0];
    return firstName.toLowerCase();
}

void main() {
    String name = 'Nguyen Minh Tuan';
    print(processName(name)); // okay
    print(processName(null)); // compile-error
}
```

```dart
void main() {
    int a1 = 20; // okay
    int a2 = null; // compile-error
}
```

Nullable Data Type

- To specify that a variable can be null, you add a question mark (?) to the type in variable declaration.

```dart
void main() {
    String? name = 'Nguyen Minh Tuan';
    print(name);

    name = null; // okay
    print(name);
}
```

Nullable Data Type

- A nullable type contains null in addition to its own values of the type.
- To mark an existing type nullable, you place a question mark after the type.
  - int? – a nullable integer such as 1, 2, and null.
  - double? – a nullable double such as 3.14, 2.5, and null.
  - bool? – nullable boolean such as true, false, and null.
  - String? – a nullable string such as ‘Hello’, ‘Bye’, and null.
  - Point? a nullable user-defined class Point. For example, point(10,20) and null
  - Student? a nullable user-defined class Student. For example, Student(‘Duy’, 18) and null

Working with nullable types

- With null safety, Dart makes it impossible to forget to add the code that handles null. Because you really cannot do much with null unless you deal with the null possibility.
- For example, Dart will not allow you to run the following code:

```
void main() {
    String? message;
    print(message.length); // compile error
}
```

- You have to put the statement inside if check or use the **?.** operator before accessing properties of nullable object.

```
String? message;
if (message == null)
    print('Can not get length of a null string');
else
    print(message.length);
```

```
void main() {
    String? message;
    print(message?.length);
}
```

Type Promotion

- Type promotion allows you to assign a value to a nullable variable without requiring any extra work.

```
void main() {
    String? message;
    message = 'Hello';
    print(message.length); // NO error
}
```

- In this example, the variable message is a nullable string type. However, Dart can see that the message is not null because we assign a value to it before accessing the length property.
- Therefore, Dart implicitly promotes the type of the message variable from **String?** to **String** automatically.

Condition and loop

Conditions In Dart

- When you write a computer program, you need to be able to tell the computer what to do in different situations. With conditions, you can control the flow of the dart program.
- Types Of Condition
  - If Condition
  - If-Else Condition
  - If-Else-If Condition
  - Switch case

Conditions In Dart

- The easy and most common way of controlling the flow of a program is through the use of an if statement.

```
void main()
{
    var age = 20;
    if(age >= 18){ // câu lệnh if
        print("You are voter.");
    }

    if(age >= 18){ // câu lệnh if-else
        print("You are voter.");
    }else{
        print("You are not voter.");
    }
}
```

Ternary Operator

- 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.

```
condition ? exprIfTrue : exprIfFalse
```

```
void main() {
    int num1 = 10;
    int num2 = 15;
    int max = (num1 > num2) ? num1 : num2;

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

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

Switch Case In Dart

- You can use a Switch case as an alternative to the if-else-if condition.

```
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;
    }
}
```

For Loop

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

```
void main() {

    for (int i = 1; i <= 10; i++) {
        print(i);
    }
    for(int i = 50; i <= 100; i++){
        if(i%2 == 0){
            print(i);
        }
    }
}
```

Foreach Loop

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

```
void main(){
    List<String> players=['Ronaldo','Messi','Neymar','Hazard'];
    for(String player in players){
        print(player);
    }
}
```

Foreach Loop

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

```
void main(){
    List<String> players=['Ronaldo','Messi','Neymar','Hazard'];
    for(String player in players){ // cách 1
        print(player);
    }
    players.forEach((names) => print(names)); // cách 2
}
```

Foreach Loop

- 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> players = ['Ronaldo','Messi','Neymar','Hazard'];
    players.asMap().forEach((index, value) => print("$value index is $index"));
}
```

```
mvmanh@Mais-Mac-Studio dark % dart main.dart
Ronaldo index is 0
Messi index is 1
Neymar index is 2
Hazard index is 3
```

While Loop

- In **while loop**, the loop's body will run until and unless the condition is true. You must write conditions first before statements.

```
void main() {
    int i = 1;
    while (i <= 10) {
        print(i);
        i++;
    }
}
```

Exception Handling

- 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.

```
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');
    }
}
```

Function in Dart

Function In Dart

- **Functions** are the block of code that performs a specific task. The function helps reusability of the code in the program.
- Advantage Of Function
  - Avoid Code Repetition
  - Easy to divide the complex program into smaller parts
  - Helps to write a clean code

```
void add(int num1, int num2){
    int sum = num1 + num2;
    print("The sum is $sum");
}
void main(){
    add(10, 20);
}
```

Function parameter In Dart

- 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.

```dart
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");
}
```

Function parameter In Dart

- 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/myam.

```
void printInfo(String name, String gender, [String title = "sir/myam"]) {
    print("Hello $title $name your gender is $gender.");
}

void main() {
    printInfo("John", "Male");
    printInfo("John", "Male", "Mr.");
    printInfo("Kavya", "Female", "Ms.");
}
```

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.
- Named parameters must either have **default** values or are marked as **required**.

```
void printInfo({String name = '', String gender = 'Male'}) {
    print("Hello $name your gender is $gender.");
}

void main() {
    printInfo(gender: "Male", name: "John");
    printInfo(name: "Sita", gender: "Female");
}
```

Named parameter In Dart

- Named parameters must either have **default** values or are marked as **required**.

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

void main() {
    printInfo(gender: "Male", name: "John");
    printInfo(name: "Sita", gender: "Female");
}
```

Anonymous Function

- You already saw function like main(), add(), etc. These are the named functions, which means they have a certain name.
- Not every function needs a name. If you remove the return type and the function name, the function is called anonymous function.

```
void main() {
    const fruits = ["Apple", "Mango", "Banana", "Orange"];
    fruits.forEach((fruit) {
        print(fruit);
    });
}
```

```
    (fruit) {
        print(fruit);
    }
```

Collection 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.
- Types Of Lists
  - Fixed Length List
  - Growable List [**Mostly Used**]

```dart
// Integer List
List<int> ages = [10, 30, 23];

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

// Mixed List (dynamic)
var mixed = [10, "John", 18.8];
```

List In Dart

- 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.

List In Dart

- 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]);
}
```

List In Dart

- 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));
}
```

- Find The Length Of The List
  - You can find the length of List by using **.length** property.

```
void main() {
    var list = [210, 21, 22, 33, 44, 55];

    print(list.length);
}
```

List In Dart

- 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);
}
```

- Add Item To List

```
void main() {
    var evenList = [2,4,6,8,10];
    print(evenList);
    evenList.add(12);
    print(evenList);
}
```

List In Dart

- Insert Item To List

```dart
void main() {
    List myList = [3, 4, 2, 5];
    print(myList);
    myList.insert(2, 15);
    print(myList); // [3, 4, 15, 2, 5]
}
```

- Remove Item From List

```dart
void main() {
    List myList = ["A", "B", "C", "D"];
    myList.remove("C");
    myList.removeAt(2);
    print(myList); // [A, B]
}
```

List In Dart

- Loops In List

```dart
List<int> list = [10, 20, 30, 40, 50];
list.forEach((n) => print(n)); // cách 1

for (int x in list) { // cách 2
    print(x);
}
```

- Multiply All Value By 2 Of All List

```dart
List<int> list = [10, 20, 30, 40, 50];
var douledList = list.map((n) => n * 2);

print((douledList)); // 20, 40, 60, 80, 100
```

List In Dart

- Combine Two Or More List In Dart

```dart
List<int> l1 = [1, 2, 3, 4];
List<int> l2 = [5, 6, 7, 8];
List<int> l3 = [0, ...l1, ...l2, 9]; // spread operator
print(l3); // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

- Conditions In List

```dart
bool sad = false;
var cart = ['milk', 'ghee', if (sad) 'Beer'];
print(cart); // milk, ghee
```

- Where In List

```dart
List<int> numbers = [2, 4, 6, 8, 10, 11, 12, 13, 14];
List<int> even = numbers.where((n) => n.isEven).toList();
print(even); // 2, 4, 6, 8, 10, 12, 14
```

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{}.**
- The list allows you to add **duplicate items**, but the Set doesn't allow it.
- The time complexity of checking whether a Set contains a specific element is typically **O(1)** on average.

Set In Dart

- Set Properties In Dart
  - **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.

```
// 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}");
```

Set In Dart

- Check The Available Value

```dart
Set<String> fruits = {"Apple", "Orange", "Mango"};
print(fruits.contains("Mango"));
print(fruits.contains("Lemon"));
```

- Add & Remove Items In Set

```dart
Set<String> fruits = {"Apple", "Orange", "Mango"};
fruits.add("Lemon");
fruits.add("Grape");
print(fruits); // {Apple, Orange, Mango, Lemon, Grape}

fruits.remove("Apple");
print(fruits); // {Orange, Mango, Lemon, Grape}
```

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.
- 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.

```
Map<String, String> countryCapital = {
    'USA': 'Washington, D.C.',
    'India': 'New Delhi',
    'China': 'Beijing'
};
print(countryCapital["USA"]);
```

Map In Dart

- Map Properties In Dart
  - **keys**: To get all keys.
  - **values**: To get all values.
  - **isEmpty**: Return true or false.
  - **isNotEmpty**: Return true or false.
  - **length**: returns the length of the Map.

```
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 empty: ${expenses.isEmpty}");
print("Length of map is: ${expenses.length}");
```

Map In Dart

- Adding and Updating Element in Map

```
Map<String, String> countryCapital = {
    'USA': 'New York',
    'India': 'New Delhi'
};
// Adding New Item
countryCapital['Japan'] = 'Tokio';
// Updating Item
countryCapital['USA'] = 'Washington, D.C.';
// Removing Item
countryCapital.remove('India');
```

Map In Dart

- Map Methods In Dart
  - **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.

OOP in Dart

Class In Dart

- 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**.

```
class Animal {
    String? name;
    int? numberOfLegs;
    int? lifeSpan;

    void display() {
        print("Animal name: $name.");
        print("Number of Legs: $numberOfLegs.");
        print("Life Span: $lifeSpan.");
    }
}
```

Access Modifier

- Dart does not have traditional access modifiers like private, protected, or public.
- Dart uses an underscore (_) prefix to indicate that a variable or method is intended for internal use within the library or package.

```dart
class MyClass {

    String _privateVariable;

    void _privateMethod() {
        // ...
    }
}
```

Object In Dart

- An object is a self-contained unit of code and data. Objects are created from templates called class.
- In OOP, 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 classes. An object is an instance of a class.

```
Animal a1 = new Animal();
a1.name = 'Tom';
a1.numberOfLegs = 4;
a1.lifeSpan = 15;

print(a1); // Instance of 'Animal'
a1.display(); // Animal name: Tom. Number of Legs: 4. Life Span: 15.
```

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.

```
Person person = Person("John", 30);
```

- If you don't define a constructor for class, then you need to set the values of the properties manually

```
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.

Constructor In Dart

```dart
class Student {
    String? name;
    int? age;
    int? rollNumber;

    // Constructor
    Student(String name, int age, int rollNumber) {
        print("Constructor called"); // check if the constructor is called
        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}");
}
```

Write Constructor Single Line

- You can also write the constructor in short form. You can directly assign the values to the properties.

```dart
class Person {
    String? name;
    int? age;
    String? subject;
    double? salary;

    // Constructor in short form
    Person(this.name, this.age, this.subject, this.salary);
}

void main() {
    Person person = Person("John", 30, "Maths", 50000.0);
    print(person);
}
```

Parameterized Constructor

- Parameterized constructor is the constructor that takes parameters. It is used to pass the values to the constructor at the time of object creation.

```
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() {
    Student student = Student(name: "John", age: 20, rollNumber: 1);
    print(student);
}
```

Parameterized Constructor

- Parameterized Constructor With Default Values In Dart

```
class Student {
    String? name;
    int? age;

    // Constructor
    Student({String? name = "John", int? age = 0}) {
        this.name = name;
        this.age = age;
    }
}

void main(){
    Student student = Student();
    print("Name: ${student.name}");
    print("Age: ${student.age}");
}
```

Named Constructor

- 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.
- However, We can create multiple constructors with the same name using **named constructors**.
- Named constructors improves code readability. It is useful when you want to create multiple constructors with the same name.

Named Constructor

```
class Student {
     String? name;
     int? age;
     int? rollNumber;

     Student() {
          print("This is a default constructor");
     }
     Student.create(String name, int age, int rollNumber) {
          this.name = name;
          this.age = age;
          this.rollNumber = rollNumber;
     }
}

void main() {
     Student student = Student.create("John", 20, 1);
     print(student);
}
```

Dart class vs Non-nullable

- By default, A class containing non-null properties must provide a required constructor method otherwise there will be a compile error.

```
class Student {
    String name; // error
    int age; // error
}
```

- To solve this error, we have three solutions as follow:

```
class Student {
    String? name;
    int? age;
}
```

```
class Student {
    late String name;
    late int age;
}
```

```
class Student {
    String name;
    int age;
    Student({required this.name, required this.age});
}
```

Asynchronous Programming

Asynchronous Programming

- What Is **Sync**hronous 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.

```
void main() {
    print('Start program');
    downloadFile('movie.mp4'); // 30 seconds
    downloadFile('music.mp3'); // 5 seconds
    print('End program'); // called after 35 seconds
}
```

Asynchronous Programming

- What is **Async**hronous Programming?
  - The 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.

```
void main() {
    print('Start program');
    downloadFile('movie.mp4', () => print('Download movie complete'));
    downloadFile('music.mp3', () => print('Download music complete'));
    print('End program'); // called almost immediately
}
```

Asynchronous Programming

- Why We Need Asynchronous
  - To Fetch Data From Internet
  - To Write Something to Database
  - To Read Data From File, and
  - To Download File etc
- To Perform asynchronous operations in dart you can use the Future class and the async and await keywords.

Future In Dart

- The Future represents a value 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.
- Future represents the result of an asynchronous operation and can have 2 states:
  - **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>.
- If the asynchronous operation performed by the function fails due to any reason, the future completes with an error.

Future In Dart

```dart
String normalFunc() {
    return "Hello";
}

Future<String> specialFunc() {
    return Future.delayed(Duration(seconds: 3), () => "Hello");
}

void main() {
    print(normalFunc()); // Hello
    print(specialFunc()); // Instance of 'Future<String>'
    // program wait for 3s and then exit
}
```

Async and Await

- 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.

```dart
String normalFunc() {
    return "Hello";
}

Future<String> specialFunc() {
    return Future.delayed(Duration(seconds: 3), () => "Hello");
}

void main() async {
    print(normalFunc()); // Hello
    // program wait for 3s
    print(await specialFunc()); // Hello
}
```