

Flutter Foundation Handbook

Prepared By: Choo Xiang Lun

Table of Contents

Chapter 0: Setting up the Development Environment and Installing Necessary Tools.....	3
0.1 Set up - Tutorial Videos.....	3
0.2 Set up - Written Guide.....	3
0.2.1 Installing Flutter SDK.....	3
0.2.2 Installing Visual Studio Code.....	3
0.2.3 Installing Dart and Flutter Visual Studio Code Extensions.....	3
Chapter 1: Introduction to Dart Programming Language.....	5
1.1 What is Dart?.....	5
1.2 Data Types.....	5
1.2.1 int.....	5
1.2.2 double.....	5
1.2.3 bool.....	5
1.2.4 String.....	6
1.2.5 dynamic.....	6
1.3 Variables and Constants.....	6
1.3.1 Variables.....	6
1.3.2 Constants.....	7
1.4 Operators.....	8
1.4.1 Arithmetic Operators.....	8
1.4.2 Comparison Operators.....	8
1.4.3 Logical Operators.....	9
1.5 Comments.....	9
1.6 Control Structures.....	9
1.6.1 if-else statement.....	10
1.6.2 switch statement.....	11
1.6.3 Loops.....	12
1.6.3.1 for loop.....	12
1.6.3.2 while loop.....	12
1.7 Functions.....	13
1.8 List.....	14
1.9 Object-Oriented Programming (OOP).....	14
1.10 Classes.....	14
1.11 Inheritance.....	18
Chapter 2: Introduction to Flutter.....	19
2.1 What is Flutter?.....	19
2.2 Advantages of Flutter.....	19
2.3 Widgets.....	19
2.3.1 Introduction to Widgets.....	19
2.3.2 Types of Widgets.....	19
2.3.3 Creating Widgets.....	20
2.3.4 Layout Widgets.....	20
2.3.5 Styling Widgets.....	20
2.3.6 Interactable Widgets.....	20

Chapter 0: Setting up the Development Environment and Installing Necessary Tools

In this chapter, we will guide you through the process of setting up your development environment for Flutter app development. This section will guide you to install and configure the Flutter SDK, Visual Studio Code, and any other tools you may need.

0.1 Set up - Tutorial Videos

For Windows, follow this Youtube tutorial link: <https://youtu.be/0SRvmcsRu2w>

For MacOS, follow this Youtube tutorial link: <https://youtu.be/NNt3akA9a40>

0.2 Set up - Written Guide

0.2.1 Installing Flutter SDK

Follow this link to download the Flutter SDK: [Install | Flutter](#)

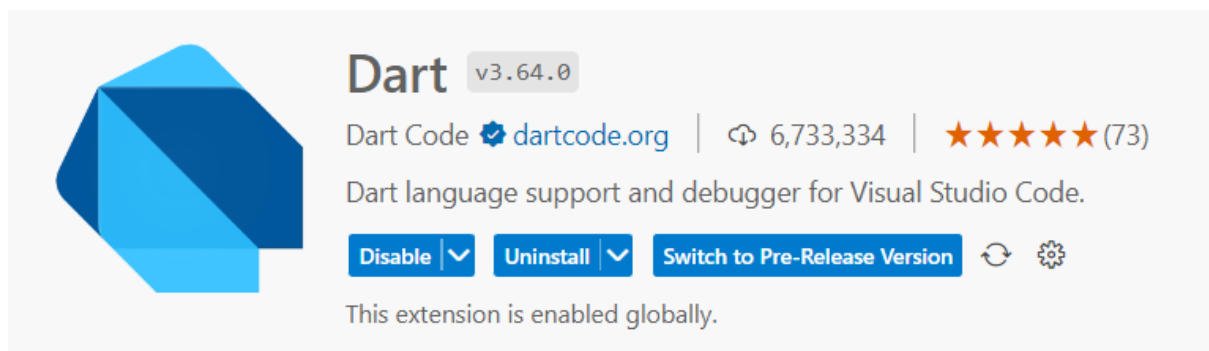
0.2.2 Installing Visual Studio Code

Visual Studio Code (VS Code) is a code editor redefined and optimized for building and debugging modern applications. VS Code is free and available on your favorite platform - Linux, macOS, and Windows. We will use this code editor to write our Dart code to build Flutter applications


Follow this link to download VS Code: <https://code.visualstudio.com/Download>

VS Code provides a lot of keyboard shortcuts to fasten the development process. Follow this link to see all the available keyboard shortcuts: shortcuts: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf>

0.2.3 Installing Dart and Flutter Visual Studio Code Extensions





This Dart extension extends VS Code with support for the Dart programming language, and provides tools for effectively editing, refactoring, running, and reloading Flutter mobile apps.





Flutter

v3.64.0

Dart Code  dartcode.org |  6,152,909 | ★★★★★ (76)

Flutter support and debugger for Visual Studio Code.

[Disable](#) ▼ [Uninstall](#) ▼ [Switch to Pre-Release Version](#)  

This extension is enabled globally.

This Flutter extension adds support for effectively editing, refactoring, running, and reloading Flutter mobile apps. It depends on (and will automatically install) the Dart extension for support for the Dart programming language.

Chapter 1: Introduction to Dart Programming Language

In this chapter, we will introduce you to the **Dart programming language**, which is the **language used to develop Flutter apps**. We will cover basic Dart knowledge that you should acquire before we start to develop a mobile application, which includes syntax, data types, control structures, and etc. In this section, you will learn how to write simple Dart programs and run them using the [DartPad](#).

1.1 What is Dart?

Dart is a general-purpose **programming language** that is used to **build web and mobile applications**. It was **developed by Google in 2011** and was designed to be easy to learn and use. It is an **object-oriented language** that is easy to learn and understand. In this chapter, we will introduce you to the Dart programming language.

1.2 Data Types

Data types in Dart define the **type of data** that can be stored in a variable. In Dart, there are several data types, including `int`, `double`, `bool`, `String`, and `dynamic`.

1.2.1 int

The **int** data type represents **integer values**. For example, you can define an integer variable as follows:

```
int age = 20;
```

1.2.2 double

The **double** data type represents a floating-point **number (decimal number)**. For example:

```
bool isFlutterCool = true;
```

1.2.3 bool

The **bool** data type represents **boolean values (true or false)**, which can be either true or false. For example:

```
bool isFlutterCool = true;
```

1.2.4 String

The **String** data type represents a sequence of characters (some text). The value is wrapped in **double quote or single quote**. For example:

```
String message1 = "Hello, Flutter!";  
// or  
String message2 = 'Hello, Flutter!';
```

1.2.5 dynamic

The **dynamic** data type represents a variable that **can hold any type of data**. For example:

```
dynamic age = 20;  
dynamic price = 10.99;  
dynamic isFlutterCool = true;  
dynamic message = "Hello, Flutter!";
```

Don't rely too much on dynamic type! It might receive runtime errors if used incorrectly.

1.3 Variables and Constants

Variables and constants in Dart are used to store data that can be used later in the program. A variable/constant **has a data type, a name, and a value**.

1.3.1 Variables

To **declare** a variable, use the **var** keyword followed by the **variable name**.

```
var name; // value: null
```

To **initialize** a variable, just **assign a value** to it.

```
name = "John";
```

You can **declare and initialize** a variable in a single line, as such:

```
var name = "John";
```

Here, **name** is the variable name, and **"John"** is the value of the variable. The data type of the variable is **automatically inferred as String**.

You can also **explicitly specify** the data type of a variable **using the data type keyword** that we've learned before. For example:

```
String message = "Hello, Flutter!";
```

Here, **message** is the variable name, and the data type is **explicitly defined as String**.

Variables can be reassigned. Meaning that you can assign a new value to a variable.

```
String message = "Hello, Ali";  
message = "Hello, Abu"; // assigning new value to message variable  
print(message); // output: Hello, Abu
```

1.3.2 Constants

To declare a constant, use the **const** or **final** keyword followed by the variable name and value. You can also **explicitly specify** the data type of a variable **using a data type keyword**. For example:

```
const name1 = "John"; // without type (automatically inferred as String)  
// or  
const String name2 = "John"; // with type (explicitly specified as String)  
// or  
final name1 = "John"; // without type (automatically inferred as String)  
// or  
final String name2 = "John"; // with type (explicitly specified as String)
```

Constants cannot be reassigned. You can't change the value stored in the variable once it is initialized.

```
const message = "Hello, Ali";  
message = "Hello, Abu"; // error!
```

The main difference between final and const is that **const variables are compile-time constants**, this means that the **value** of the const variable **is known at compile time**, while **final variables are run-time constants**.

```
final name;  
name = "ali"; // ok  
  
const name;  
name = "ali"; // error
```

1.4 Operators

In Dart, **operators** are **symbols or keywords** used to **perform certain operations on operands**. An operand can be a variable, value, or expression that an operator is applied to. Dart provides a variety of operators for performing mathematical, logical, and other operations on values.

1.4.1 Arithmetic Operators

Arithmetic Operators perform basic mathematical operations on two data/values and give values as output.

Operators	Symbol	Example	Result
Addition	+	5 + 2	7
Subtraction	-	5 - 2	3
Multiplication	*	5 * 2	10
Division	/	5 / 2	2.5
Integer division	~/	5 ~/ 2	2
Modulo	%	5 % 2	1 (remainder of 5/2)

1.4.2 Comparison Operators

Comparison Operators evaluates two data/values and gives boolean values as output

Operators	Symbol	Example	Result
Equal to	==	10 == 20	True
Not equal to	!=	10 != 20	False
Greater than	>	10 > 20	False
Less than	<	10 < 20	True
Greater than or equal to	>=	10 >= 20	False
Less than or equal to	<=	10 <= 20	True

1.4.3 Logical Operators

Logical Operators evaluates two conditions and gives boolean values as output. And, Or and Not are three most common logical operators in computer programming:

AND			OR			NOT	
P	Q	$P \wedge Q$	P	Q	$P \vee Q$	P	$\sim P$
T	T	T	T	T	T	T	F
T	F	F	T	F	T	F	T
F	T	F	F	T	T		
F	F	F	F	F	F		

Operators	Symbol	Example	Result
And	&&	(5 < 2) && (5 > 3)	False
Or		(5 < 2) (5 > 3)	True
Not	!	!(5 < 2)	True

1.5 Comments

Comments in computer programming are readable explanations or annotations of the code. It is useful to improve the readability of your code. Comments are ignored when the programme is running.

```
// This is a normal, one-line comment.

/// This is a documentation comment, used to document libraries,
/// classes, and their members. Tools like IDEs and dartdoc treat
/// doc comments specially.

/* This is a
   multi-lines comments */
```

1.6 Control Structures

Control structures are used in programming to **control the flow of execution based on certain conditions**. In Dart, there are several types of control structures, including if-else statements, switch statements.

1.6.1 if-else statement

The **if-else statement** is used to **execute a block of code if a certain condition is true** and another block of code if the condition is false. The syntax of a if-else statement is as follows:

```
if (<condition>) {  
    // code to run if condition is true  
} else {  
    // code to run if condition is false  
}
```

For example:

```
var age = 20;  
  
if (age >= 18) {  
    print("You are an adult");  
} else {  
    print("You are not an adult");  
}
```

Here, if the age is greater than or equal to 18, the first block of code will be executed, and if the age is less than 18, the second block of code will be executed. The above if-else statement can be shortened using **conditional expressions** as follows:

```
(age >= 18) ? print("You are an adult") : print("You are not an adult");
```

Here are the syntax of a conditional expression:

```
condition ? expression 1 : expression 2;
```

You can have **more than one if case** by **chaining with multiple if blocks**; in fact, there is no limit.

```
var age = 20;  
  
if (age >= 18) {  
    print("You are an adult");  
} else if (age >= 13) {  
    print("You are a teenager");  
} else {  
    print("You are a child");  
}
```

Here, if the age is greater than or equal to 18, the first block of code will be executed, and if the age is less than 18 but greater or equal to 15, the second block of code will be executed.

1.6.2 switch statement

The switch statement is used to execute different blocks of code based on different values of a variable. Here are the syntax for switch statement:

```
switch (<value to compare>) {  
    case <value>:  
        // your logic here  
        break;  
    case <value>:  
        // your logic here  
        break;  
    default:  
        // code to run if there is no matching case  
}
```

You can have as many cases as you want.

Here is an example of a switch statement:

```
var grade = "A";  
  
switch (grade) {  
    case "A":  
        print("Excellent");  
        break;  
    case "B":  
        print("Good");  
        break;  
    case "C":  
        print("Average");  
        break;  
    default:  
        print("Failed");  
}
```

Here, the code will execute the block of code that matches the value of the grade variable. You can achieve the same thing by chaining multiple if-else blocks. However, a **switch statement is generally more efficient than an if-else statement**.

1.6.3 Loops

Loops are used to **execute a block of code repeatedly**. In Dart, there are two types of loops: for loop and while loop.

1.6.3.1 for loop

A for loop functions by executing a section of code repeatedly until a certain condition has been satisfied. For-loops are typically **used when the number of iterations is known** before entering the loop. A for-loop requires **three parts**: the **initialization part** (runs before the loop starts), the **condition part** (checks after each iteration, continue next loop if true, stop if false), and the **code that will be executed before advancing to the next iteration**. The syntax for for loop is as follows:

```
for (<initialization>; <condition>; <code to execute before next loop>) {  
    // your logic here  
}
```

Here is an example of the for loop:

```
for (int i = 0; i < 5; i++) {  
    print("Hello World!");  
}
```

Here, the code will print *Hello World!* five times.

1.6.3.2 while loop

The while loop is also used to execute a block of code a specific number of times, **based on a given Boolean condition** (in the parenthesis). The syntax of a while loop is as follows:

```
while (<condition>) {  
    // your logic here  
}
```

For example:

```
var i = 0;  
  
while (i < 5) {  
    print(i);  
    i++;  
}
```

Here, the code will print the numbers from 0 to 4.

Checkpoint

Create a program that calculates the sum of all even numbers from 1 to 100. The program should output the sum in the console.

Tips: use modulo, %, to determine whether a value is odd or even.

1.7 Functions

Functions in Dart are used to group a set of statements/code that perform a specific task. A function can have parameters and can return a value.

The syntax to create a method is as follows:

```
<return type> <function name>(<parameter list>) {  
    // your logic here  
}
```

For example:

```
void greet(String name) {  
    print("Hello, $name!");  
}
```

Here, **greet** is the function name, and **name** (type String) is the **parameter**. The parameters of a function are **input to the function**. This function has a **void return type**. This means that the function **does not give any output (returns nothing)**.

You can call the **greet** function as follow:

```
greet("Flutter"); // output: Hello, Flutter!
```

You can also specify the return type of a function by replacing void with the data type of the value that the function returns. For example:

```
int add(int a, int b) {  
    return a + b;  
}
```

Here, **add** is the function name, **a** and **b** (type int) are the parameters, and the function returns the sum of **a** and **b** which is of type int, with the help of the **return** keyword.

You can call the add function as follow:

```
int result = add(1, 2); // use result variable to store the output
print(result); // output: 3
```

Checkpoint

Create a function called `pow` that takes in two `int` parameters, `a` and `b`, that return the value of a^b .

1.8 List

In programming, Lists are a data structure that stores a list of values. The syntax to create a list is as follow:

```
List<list item type> <list name> = [<values>];
```

For example, if we wanted to create a list of our favorite fruits:

```
List<String> favoriteFruits = ["banana", "apple", "mango"];
```

1.9 Object-Oriented Programming (OOP)

Dart is an object-oriented programming language. Object-oriented programming (OOP) is a programming paradigm/approach that is **based on the idea of organizing code into objects that interact with each other**. In simpler terms, it's a way of writing computer programs by breaking them down into **smaller, more manageable pieces**.

At its core, OOP is about designing code in a way that **models the real world**. This means that we can **create objects that represent real-world entities** such as people, cars, or buildings, and we can give these objects **characteristics** such as name, color, or speed. We can also give these objects **behaviors** such as walking, driving, or building.

The **benefits of using OOP** are numerous. By breaking a program down into objects, we can create code that is **modular, reusable, and easy to understand**. This makes it **easier to maintain and update code over time**.

1.10 Classes

Classes in Dart are used to **define objects that have properties and functions**. A class can be thought of as a **blueprint for creating objects**. To define a class, use the **class keyword followed by the class name**. Within the class you can create **properties (class variables)** to define the **characteristics** possessed by the class, you can also create **functions** to define the **behavior** possessed by the class. By convention, class names are written in **PascalCase** (every word begins with an uppercase letter). For example:

```
class Person {
    // properties (characteristics)
    String name = "ali";
    int age = 21;

    // functions (behavior)
    void introduce() {
        print("Hello I am $name, I am $age years old");
    }
}
```

Here, we've created a Person class. The **Person** is the class name, and **String name** and **int age** are the **properties**. The **introduction** is a **function** used to print the name and age of the person.

Now you can use the Person class (blueprint) to create a Person object, as such:

```
Person();
```

You can access all the properties of an object by using the dot operator `.`

```
print(Person().name); // output: ali
print(Person().age); // output: 21
Person().sayHi(); // output: Hello I am ali, I am 21 years old
```

You can store the object inside a variable using the **Person data type**.

```
Person person1 = Person();
Person person2 = Person();
```

person1 and **person2** are **variables** that store the **newly created Person Objects**.

As you can see, we can create multiple Person objects using the Person class. However now all the objects created have the same characteristics (age = 21 and name = ali).

```
print(person1.name); // output: ali
print(person2.name); // output: ali
```

You can create **objects with different characteristics** using a **class constructor**. Modify Person class to include a class constructor.

```
class Person {
    // properties (characteristics)
```

```
late String name;
late int age;

// class constructor
Person(String name, int age) {
    this.name = name;
    this.age = age;
}

// functions (behavior)
void introduce() {
    print("Hello I am $name, I am $age years old");
}
}
```

The above Person class can be modified using a parameterized constructor:

```
class Person {
    // properties (characteristics)
    String name;
    int age;

    // parameterized constructor
    Person(this.name, this.age);

    // functions (behavior)
    void introduce() {
        print("Hello I am $name, I am $age years old");
    }
}
```

Now you can create objects with different characteristics.

```
Person person1 = Person('ali', 20);
Person person2 = Person('abu', 21);
person1.introduce(); // Hello I am ali, I am 20 years old
person2.introduce(); // Hello I am abu, I am 21 years old
```

You can specify named parameters for your constructor. Named parameters are **optional parameters** that can be passed to the constructor **in any order, using the property name**, and they **can be omitted** if they have **default values**.

```
class Person {
    // properties (characteristics)
```



```
String name;
int age;

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

// functions (behavior)
void introduce() {
    print("Hello I am $name, I am $age years old");
}
}
```

Now you can pass values to your constructor in any order:

```
Person person1 = Person(name: "ali", age: 21);
Person person2 = Person(age: 22, name: "abu");
```

Since named parameters are optional you have to use the **required** keyword to indicate that a parameter is required and to make sure that all your properties are initialized. If a required parameter is not passed to the constructor, a compiler-time error will occur.

You can set **default values on named parameters** to avoid using required keywords. If you omit a named parameter with default values while creating the object, default values will be used.

```
class Person {
    // properties (characteristics)
    String name;
    int age;

    // class constructor
    Person({required this.name, this.age = 20});

    // functions (behavior)
    void introduce() {
        print("Hello I am $name, I am $age years old");
    }
}
```

Using default value when creating a new Person object.

```
Person person = Person(name: 'ali');
print(person.age); // output: 20
```

1.11 Inheritance

Inheritance is the process of creating a new class by extending an existing class. The new class **inherits the properties and functions of the existing class** and can also **override or add new properties and functions**. For example:

```
class Student extends Person {
    late String grade;

    Student(String name, int age, String grade) : super(name, age) {
        this.grade = grade;
    }

    void study() {
        print("$name is studying...");
    }

    @override
    void introduce() {
        print("I am a student");
    }
}
```

Student is a new class that extends the Person class. The Student class adds a new grade property and a study function while still possessing the name and age properties from the Person class.

Now you can create a student object like so:

```
Student student = Student("ali", 20, "A");
```

Note that the introduction method is overridden in the Student class.

```
student.introduce(); // output: I am a student
```

Chapter 2: Introduction to Flutter

2.1 What is Flutter?

Flutter is a **mobile app Software Development Toolkit (SDK)** that is used to build high-performance, high-fidelity, apps for iOS and Android, using a **single codebase**. Flutter was **developed by Google in 2017** and is **designed to be easy to learn and use**.

2.2 Advantages of Flutter

Flutter has several advantages over other mobile app SDKs. Some of these advantages include:

1. **Fast development:** Flutter SDK serves a lot of [pre-built material design components](#), allowing you to create apps very quickly. Flutter's hot reload feature allows you to see the changes you make to your code almost instantly.
2. **Single codebase:** Flutter allows you to write code once and use it on both iOS and Android.
3. **Beautiful UI:** Flutter's Material Design and Cupertino widgets allow you to create beautiful and responsive user interfaces.
4. **High performance:** Flutter's high-performance rendering engine allows you to build apps that are fast and responsive.

2.3 Widgets

2.3.1 Introduction to Widgets

Widgets are the building blocks of Flutter apps. They are used to **create all the UI elements in an app**, such as buttons, text fields, images, and more. Widgets are also used to define the layout of an app, and to handle user input. Widgets are just classes that represent UI components in our application.

2.3.2 Types of Widgets

There are two main types of widgets in Flutter: **stateless widgets** and **stateful widgets**.

1. The state (properties) of stateless widgets **cannot change over time**. Examples of stateless widgets include buttons, text fields, and images.
2. The state (properties) of a stateful widget **can change over time**. Examples of stateful widgets include lists, checkboxes, and forms.

2.3.3 Creating Widgets

Widgets are created using the Widget class. The Widget class has a number of properties that can be used to configure the appearance and behavior of a widget.

Some of the most important properties of the Widget class include:

1. **child**: The child of the widget. This can be another widget, or a piece of text, or an image, or anything else that can be displayed on the screen.
2. **width**: The width of the widget.
3. **height**: The height of the widget.
4. **color**: The color of the widget.
5. **style**: The style of the widget. This can be used to change the font, the size, and the alignment of the widget.

2.3.4 Layout Widgets

Layout widgets are used to arrange other widgets on the screen. Some of the most common layout widgets include:

1. **Row**: A layout widget that arranges its children in a horizontal row.
2. **Column**: A layout widget that arranges its children in a vertical column.
3. **Stack**: A layout widget that stacks its children on top of each other.

2.3.5 Styling Widgets

Styling widgets are used to change the appearance of other widgets. Some of the most common styling widgets include:

1. **TextStyle**: A styling widget that changes the font, the size, and the alignment of text.
2. **Color**: A styling widget that changes the color of a widget.
3. **Decoration**: A styling widget that adds a border, a background, or a shadow to a widget.

2.3.6 Interactable Widgets

Interactable widgets are widgets that can be interacted with by the user. Some of the most common interactable widgets include:

1. **Button**: A widget that can be clicked to perform an action.
2. **TextField**: A widget that can be used to enter text.
3. **Checkbox**: A widget that can be used to select an option.
4. **Radio**: A widget that can be used to select one option from a group of options.