Technical College of Zakho

Computer Information Systems

Mobile Application Development I

1. install and run Dart and Flutter

Lecturer:

Sipan M. Hameed

www.sipan.dev

2025-2026

Contents

Run Dart and Flutter

To install and run **Dart and Flutter** on Windows in VS Code, follow this guide.

Crucial Note: Do **not** install the Dart SDK separately. The Flutter SDK already includes the full Dart SDK. Installing both can cause path conflicts.

1.1. Phase 1: Install Dependencies

1.1.1. Install Git:

Flutter relies on Git. If you don't have it, download and install it from git-scm.com.

- *During installation, just click "Next" through all the default options.*

1.2. Phase 2: Install the Flutter SDK

1.2.1. Download Flutter:

Go to the Flutter Windows Install page and download the **stable zip file**.

Extract:

- Extract the zip file to a clean folder path (e.g., C:\src\flutter).
- **Do not** put it in C:\Program Files (this causes permission issues).

Update your Path (Critical):

- Press the **Windows Key**, type env, and select **"Edit the system environment variables"**.
- Click **Environment Variables...**.
- Under **User variables** (top box), find Path and double-click it.
- Click **New** and paste the path to the bin folder:

```
C:\src\flutter\bin
```

- Click **OK** on all windows.

1.3.Phase 3: Run Flutter Doctor

1. Open a **new** Command Prompt or PowerShell window.
2. Type the following command and press Enter:

PowerShell:

```
flutter doctor
```

- *Note: This command checks your system. It is normal to see "X" marks next to Android Studio or Visual Studio if you haven't installed them yet.*
- For now, as long as you see a green checkmark next to **Flutter**, you are ready to proceed with VS Code.

1.4.Phase 4: Set up VS Code

1.4.1. Open Visual Studio Code.

1.4.1.1. Install the Extension:

- Click the **Extensions** icon on the left sidebar (or press Ctrl+Shift+X).
- Search for **"Flutter"**.
- Install the official extension by **Flutter** (this will automatically install the Dart extension too).

1.5.Phase 5: Create and Run Your First App

Now let's create a real Flutter app and run it. The easiest way to test this without installing heavy Android tools is to run it as a **Windows Desktop App**.

Create the Project:

- In VS Code, press Ctrl+Shift+P to open the Command Palette.
- Type Flutter: New Project and select it.
- Select **Application**.

- Choose a folder where you want to save your project.
- Name your project hello_flutter (lowercase with underscores).

Select Your Device:

- Look at the bottom right corner of the VS Code blue status bar. It might say "No Device" or "Windows (windows-x64)".
- Click it and select **Windows (desktop)** (or **Chrome** if you prefer running it in a browser).

Run the App:

- Press **F5** on your keyboard (or go to **Run > Start Debugging**).
- *Wait a moment.* The first time you build, it takes a minute or two.

Success! You should see a window pop up with the standard Flutter counter app.

Android Emulator

let's set up the **Android Emulator**. This allows you to test your app on a virtual phone running on your screen.

This process involves installing **Android Studio** (which contains the emulator tools) and then connecting it to VS Code.

1.5.1. Phase 1: Install Android Studio

Even though you will write code in VS Code, you need Android Studio for its tools.

1. **Download:** Go to the Android Studio download page and download the installer.
2. **Install:** Run the installer.
   - Make sure **"Android Virtual Device"** is checked during setup.
   - Click "Next" through all defaults until finished.

1.5.2. Phase 2: Install the "Hidden" Command-Line Tools

This is the most common step beginners miss, causing flutter doctor errors.

1. Open **Android Studio**.
2. On the Welcome screen, look for a button that says **More Actions** (usually three dots or a dropdown icon) and select **SDK Manager**.
   - *If you don't see the Welcome screen, go to **Tools > SDK Manager** in the top menu.*
3. In the new window, click the **SDK Tools** tab (in the middle of the window).
4. Check the box next to **Android SDK Command-line Tools (latest)**.
5. Click **Apply**, then **OK** to install them.

1.5.3. Phase 3: Accept Android Licenses

Google requires you to legally accept their licenses via the command line.

1. Close Android Studio.
2. Open your **Command Prompt** or **PowerShell**.
3. Run this command:

PowerShell

```
flutter doctor --android-licenses
```

4. It will ask you to review licenses. Keep typing y and hitting **Enter** until it says "All SDK package licenses accepted."

1.5.4. Phase 4: Create Your Virtual Phone

1. Open **Android Studio** again.
2. Click **More Actions > Virtual Device Manager** (or **Device Manager**).
3. Click **Create Device** (or the big + button).
4. **Select Hardware:** Choose a device like **Pixel 5** or **Pixel 6**. Click **Next**.
5. **System Image:** Click the **Download** arrow next to a recent Android version (like **R** or **S** or **Tiramisu**).
   - *Wait for the download to finish.*
6. Select that downloaded system version and click **Next**.
7. Click **Finish**.

You can now close Android Studio. You won't need to open it again.

1.5.5. Phase 5: Run Your App in VS Code

1. Open **VS Code** and your Flutter project.
2. Look at the bottom right status bar. Click where it says **Windows (desktop)** or **No Device**.
3. You should now see your new **Android Emulator** in the list. Select it.
   - *The emulator phone will launch on your screen. Give it a minute to boot up.*
4. Press **F5** to run your app.

Success! Your app should now be running on the virtual Android phone.

Install Flutter manually

Learn how to install and set up the Flutter SDK manually.

Learn how to install and manually set up your Flutter development environment.

Tip

If you've never set up or developed an app with Flutter before, follow Get started with Flutter instead.

If you're just looking to quickly install Flutter, consider installing Flutter with VS Code for a streamlined setup experience.

1.5.6. Choose your development platform

The instructions on this page are configured to cover installing Flutter on a **Windows** device.

If you'd like to follow the instructions for a different OS, please select one of the following.

1.5.7. Download prerequisite software

1.5.7.1. Before installing the Flutter SDK, first complete the following setup.

Install Git for Windows

Download and install the latest version of Git for Windows.

For help installing or troubleshooting Git, reference the Git documentation.

1.5.8. Set up an editor or IDE

For the best experience developing Flutter apps, consider installing and setting up an editor or IDE with Flutter support.

1.5.9. Install and set up Flutter

To install the Flutter SDK, download the latest bundle from the SDK archive, then extract the SDK to where you want it stored.

1.5.10. Download the Flutter SDK bundle

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

Create a folder to store the SDK

Create or find a folder to store the extracted SDK in. Consider creating and using a directory at (C:\src\flutter).

1.5.11. Note

Select a location that doesn't have special characters or spaces in its path and doesn't require elevated privileges.

1.5.12. Extract the SDK

Extract the SDK bundle you downloaded into the directory you want to store the Flutter SDK in.

```
C:\src\flutter
```

1.5.13. Add Flutter to your PATH

Now that you've downloaded the SDK, add the Flutter SDK's bin directory to your PATH environment variable. Adding Flutter to your PATH allows you to use the flutter and dart command-line tools in terminals and IDEs.

Determine your Flutter SDK installation location

```
1. C:\src\flutter\bin
```

Copy the absolute path to the directory that you downloaded and extracted the Flutter SDK into.

Navigate to the environment variables settings

1. Press Windows + Pause.

If your keyboard lacks a Pause key, try Windows + Fn + B.

The **System > About** dialog opens.

Click Advanced System Settings > Advanced > Environment Variables....

The **Environment Variables** dialog opens.

2. **Add the Flutter SDK bin to your path**
    1. In the **User variables for (username)** section of the **Environment Variables** dialog, look for the **Path** entry.
    2. If the **Path** entry exists, double-click it.

The **Edit Environment Variable** dialog should open.

a. Double-click inside an empty row.
b. Type the path to the bin directory of your Flutter installation.

For example, if you downloaded Flutter into a develop\flutter folder inside your user directory, you'd type the following:

```
C:\src\flutter
```

c. Click the Flutter entry you added to select it.
d. Click **Move Up** until the Flutter entry sits at the top of the list.
e. To confirm your changes, click **OK** three times.

3. If the entry doesn't exist, click **New...**.

The **Edit Environment Variable** dialog should open.

a. In the **Variable Name** box, type Path.
b. In the **Variable Value** box, type the path to the bin directory of your Flutter installation.

For example, if you downloaded Flutter into a develop\flutter folder inside your user directory, you'd type the following:

```
C:\src\flutter
```

c. To confirm your changes, click **OK** three times.

1.5.14. Apply your changes

To apply this change and get access to the flutter tool, close and reopen all open command prompts, sessions in your terminal apps, and IDEs.

1.5.15. Validate your setup

To ensure you successfully added the SDK to your PATH, open command prompt or your preferred terminal app, then try running the flutter and dart tools.

```
flutter –version
```

```
dart --version
```

If either command isn't found, check out Flutter installation troubleshooting.

1.5.16. Continue your Flutter journey

Now that you've successfully installed Flutter, set up development for at least one target platform to continue your journey with Flutter.

Introduction to Dart

1.6.Hello World

Every app requires the top-level `main()` function, where execution starts. Functions that don't explicitly return a value have the `void` return type. To display text on the console, you can use the top-level `print()` function:

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

Read more about the `main()` function in Dart, including optional parameters for command-line arguments.

1.7.Variables

Even in type-safe Dart code, you can declare most variables without explicitly specifying their type using `var`. Thanks to type inference, these variables' types are determined by their initial values:

```
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',
};
```

Built-in types

Information on the types Dart supports.

The Dart language has special support for the following:

- Numbers (int, double)
- Strings (String)
- Booleans (bool)
- Records ((value1, value2))
- Functions (Function)
- Lists (List, also known as *arrays*)

- Sets (Set)
- Maps (Map)
- Runes (Runes; often replaced by the characters API)
- Symbols (Symbol)
- The value null (Null)

1.8.Numbers

Dart numbers come in two flavors:

`int`

Integer values no larger than 64 bits, depending on the platform. On native platforms, values can be from -2^{63} to $2^{63} - 1$. On the web, integer values are represented as JavaScript numbers (64-bit floating-point values with no fractional part) and can be from -2^{53} to $2^{53} - 1$.

`double`

64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard.

Both `int` and `double` are subtypes of `num`. The num type includes basic operators such as +, -, /, and *, and is also where you'll find `abs()`, `ceil()`, and `floor()`, among other methods. (Bitwise operators, such as >>, are defined in the `int` class.) If num and its subtypes don't have what you're looking for, the `dart:math` library might.

Integers are numbers without a decimal point. Here are some examples of defining integer literals:

```
var x = 1;
var hex = 0xDEADBEEF;
```

If a number includes a decimal, it is a double. Here are some examples of defining double literals:

```
var y = 1.1;
var exponents = 1.42e5;
```

You can also declare a variable as a num. If you do this, the variable can have both integer and double values.

```
num x = 1; // x can have both int and double values
```

```
x += 2.5;
```

Integer literals are automatically converted to doubles when necessary:

```
double z = 1; // Equivalent to double z = 1.0.
```

1.1.Strings

A Dart string (`String` object) holds a sequence of UTF-16 code units. You can use either single or double quotes to create a string:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

You can put the value of an expression inside a string by using `${`*`expression`*`}`. If the expression is an identifier, you can skip the `{}`. To get the string corresponding to an object, Dart calls the object's `toString()` method.

Constant

```
// These work in a const string.
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// These do NOT work in a const string.
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = [1, 2, 3];
```

1.2.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.

1.2.1. Record syntax

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

```
var record = ('first', a: 2, b: true, 'last');
```

Record type annotations are comma-delimited lists of types enclosed in parentheses. You can use record type annotations to define return types and parameter types. For example, the following `(int, int)` statements are record type annotations:

```
(int, int) swap((int, int) record) {
  var (a, b) = record;
  return (b, a);
}
```

Fields in record expressions and type annotations mirror how parameters and arguments work in functions. Positional fields go directly inside the parentheses:

```
// Record type annotation in a variable declaration:
(String, int) record;

// Initialize it with a record expression:
record = ('A string', 123);
```

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.
```

This is similar to how positional parameters in a function declaration or function typedef can have names but those names don't affect the signature of the function.

For more information and examples, check out Record types and Record equality.

1.2.2. Record fields

Record fields are accessible through built-in getters. Records are immutable, so fields do not have setters.

Named fields expose getters of the same name. Positional fields expose getters of the name `$<position>`, skipping named fields:

```
var record = ('first', a: 2, b: true, 'last');

print(record.$1); // Prints 'first'
print(record.a); // Prints 2
print(record.b); // Prints true
print(record.$2); // Prints 'last'
```

To streamline record field access even more, check out the page on Patterns.

1.2.3. Record types

There is no type declaration for individual record types. Records are structurally typed based on the types of their fields. A record's *shape* (the set of its fields, the fields' types, and their names, if any) uniquely determines the type of a record.

Each field in a record has its own type. Field types can differ within the same record. The type system is aware of each field's type wherever it is accessed from the record:

```
(num, Object) pair = (42, 'a');

var first = pair.$1; // Static type `num`, runtime type `int`.
var second = pair.$2; // Static type `Object`, runtime type `String`.
```

Consider two unrelated libraries that create records with the same set of fields. The type system understands that those records are the same type even though the libraries are not coupled to each other.

Tip

While you can't declare a unique type for a record shape, you can create type aliases for readability and reuse. To learn how and when to do so, check out Records and typedefs.

1.2.4. Record equality

Two records are equal if they have the same *shape* (set of fields), and their corresponding fields have the same values. Since named field *order* is not part of a record's shape, the order of named fields does not affect equality.

For 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'.
({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.
```

Records automatically define `hashCode` and `==` methods based on the structure of their fields.

1.2.5. Multiple returns

Records allow functions to return multiple values bundled together. To retrieve record values from a return, destructure the values into local variables using pattern matching.

```
// Returns multiple values in a record:
(String name, int age) userInfo(Map<String, dynamic> json) {
  return (json['name'] as String, json['age'] as int);
}

final json = <String, dynamic>{'name': 'Dash', 'age': 10, 'color': 'blue'};

// Destructures using a record pattern with positional fields:
var (name, age) = userInfo(json);

/* Equivalent to:
  var info = userInfo(json);
  var name = info.$1;
  var age  = info.$2;
*/
You can also destructure a record using its named fields, using the colon :
syntax, which you can read more about on the Pattern types page:
({String name, int age}) userInfo(Map<String, dynamic> json)
// ···
// Destructures using a record pattern with named fields:
final (:name, :age) = userInfo(json);
```

You can return multiple values from a function without records, but other methods come with downsides. For example, creating a class is much more verbose, and using other collection types like `List` or `Map` loses type safety.

Note

Records' multiple-return and heterogeneous-type characteristics enable parallelization of futures of different types, which you can read about in the `dart:async` documentation.

1.2.6. Records as simple data structures

Records only hold data. When that's all you need, they're immediately available and easy to use without needing to declare any new classes. For a simple list of data tuples that all have the same shape, a *list of records* is the most direct representation.

Take this list of "button definitions", for example:

```
final buttons = [
  (
    label: "Button I",
    icon: const Icon(Icons.upload_file),
    onPressed: () => print("Action -> Button I"),
  ),
  (
    label: "Button II",
    icon: const Icon(Icons.info),
    onPressed: () => print("Action -> Button II"),
  )
];
```

This code can be written directly without needing any additional declarations.

1.3. 1️⃣ Arithmetic (Math) Operators in Dart

Operator	Meaning	Example	Result
`+`	Addition	`5 + 2`	`7`
`-`	Subtraction	`5 - 2`	`3`
`*`	Multiplication	`5 * 2`	`10`
`/`	Division (double)	`5 / 2`	`2.5`
`~/`	Integer division	`5 ~/ 2`	`2`
`%`	Modulus (remainder)	`5 % 2`	`1`
`++`	Increment	`a++`	adds 1
`--`	Decrement	`a--`	subtracts 1

1.3.1. Example

```
void main() {
  int a = 10;
  int b = 3;

  print(a + b);   // 13
  print(a / b);   // 3.3333
  print(a ~/ b);  // 3
  print(a % b);   // 1
}
```

1.4. 2️⃣ Relational (Comparison) Operators

These **always return `bool`** (`true` or `false`).

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	`true`
`!=`	Not equal	`5 != 3`	`true`
`>`	Greater than	`5 > 3`	`true`
`<`	Less than	`5 < 3`	`false`
`>=`	Greater or equal	`5 >= 5`	`true`
`<=`	Less or equal	`3 <= 5`	`true`

1.4.1. Example

```
void main() {
  int x = 10;
  int y = 20;

  print(x > y);    // false
  print(x <= y);   // true
  print(x == y);   // false
}
```

1.5. 3️⃣ Logical Operators

Used to **combine conditions**.

Operator	Meaning	Example
`&&`	AND	`a > 5 && b < 10`
||	OR	`a > 5 || b < 10`
`!`	NOT	`!isLoggedIn`

1.5.1. Truth Table (Important for exams!)

A	B	A && B	A || B
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

1.5.2. Example

```
void main() {
  int age = 20;
  bool hasID = true;

  print(age >= 18 && hasID); // true
  print(age < 18 || hasID);  // true
  print(!hasID);             // false
}
```

1.6. 4️⃣ Combined Example (Real-world logic)

```
void main() {
  int score = 75;

  if (score >= 50 && score <= 100) {
    print("Passed");
  } else {
    print("Failed");
  }
}
```

1.7. 5️⃣ Common Tricky Points ⚠️

🔹 == vs =

```
a == b    // comparison
a = b     // assignment ❌ (very common mistake)
```

- / vs ~/

```
print(5 / 2);    // 2.5  (double)
print(5 ~/ 2);   // 2    (int)
```

- Short-circuit logic

```
false && expensiveFunction(); // function NOT called
true || expensiveFunction();  // function NOT called
```

1.8. 6 Quick Practice (Try mentally)

1 What is the output?

```
print(10 > 5 && 3 < 1);
```

2 What is printed?

```
int x = 7;
print(x % 2 == 0);
```

3 What type is the result?

```
var r = 5 / 2;
```

1.9. if / else conditions in Dart

1️⃣ Basic if Statement

Executes code **only if the condition is true**.

```
void main() {
  int age = 20;

  if (age >= 18) {
    print("Adult");
  }
}
```

📌 Condition **must be bool** (no 0/1 like C).

2️⃣ if – else

Two paths: **true** or **false**

```
void main() {
  int marks = 45;

  if (marks >= 50) {
    print("Pass");
  } else {
    print("Fail");
  }
}
```

3️⃣ if – else if – else (Multiple Conditions)

Checked **top** → **bottom** (first true block runs).

```
void main() {
  int score = 82;

  if (score >= 90) {
    print("A");
  } else if (score >= 75) {
    print("B");
  } else if (score >= 50) {
    print("C");
  } else {
    print("F");
  }
}
```

⚠️ Order matters!

4️⃣ Nested if Statements

if inside another if.

```
void main() {
  int age = 22;
  bool hasID = true;

  if (age >= 18) {
    if (hasID) {
      print("Allowed");
    } else {
      print("ID required");
    }
  } else {
    print("Underage");
  }
}
```

5️⃣ Logical Conditions in if

- **AND (&&)**

```
if (age >= 18 && hasID) {
  print("Access granted");
}
```

- **OR (||)**

```
if (age < 12 || age > 60) {
  print("Discount");
}
```

- **NOT (!)**

```
if (!isLoggedIn) {
  print("Please log in");
}
```

6️⃣ if with Comparison Operators

```
int x = 10;

if (x == 10) {
  print("Equal");
}

if (x != 5) {
  print("Not five");
}
```

7️⃣ Ternary Operator (condition ? expr1 : expr2)

Short **if–else** expression.

```
int age = 16;

String result = age >= 18 ? "Adult" : "Minor";
print(result);
```

📌 Must return a value.

8️⃣ if as an Expression (Dart-style)

```
String grade;
int marks = 70;

grade = marks >= 50 ? "Pass" : "Fail";
```

9️⃣ Common Mistakes ❌ (Exam Traps)

- ❌ **Using = instead of ==**

```
if (a = 5) {}    // ERROR
```

- ❌ **Non-boolean condition**

```
if (1) {}        // ERROR
```

- ❌ **Missing braces (logic bug)**

```
if (x > 0)
  print("Positive");
  print("Always runs"); // ❌
```

✔ Correct:

```
if (x > 0) {
  print("Positive");
}
```

🔟 Real-World Example (Complete)

```
void main() {
  int balance = 500;
  int withdraw = 300;

  if (withdraw <= balance) {
    balance -= withdraw;
    print("Withdraw successful");
  } else {
    print("Insufficient balance");
  }
}
```

🧪 **Practice (Try before scrolling)**

- **Q1**

```
int x = 15;

if (x % 3 == 0 && x % 5 == 0) {
  print("FizzBuzz");
} else if (x % 3 == 0) {
  print("Fizz");
} else if (x % 5 == 0) {
  print("Buzz");
}
```

- **Q2**

What is printed?

```
int a = 5;

if (a > 10) {
  print("A");
} else if (a > 3) {
  print("B");
} else {
  print("C");
}
```

Dart functions

1️⃣ What is a Function in Dart?

A **function** is a reusable block of code that:

- may take **parameters**
- may **return a value**

```
returnType functionName(parameters) {
  // body
}
```

2️⃣ Basic Function (No parameters, no return)

```
void greet() {
  print("Hello Dart");
}

void main() {
  greet();
}
```

📌 void → returns nothing.

3️⃣ Function with Parameters

```
void printSum(int a, int b) {
  print(a + b);
}

void main() {
  printSum(3, 4);
}
```

4️⃣ Function with Return Value

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

void main() {
  int result = add(5, 6);
  print(result);
}
```

5️⃣ Arrow (Short) Functions =>

For **single-expression functions**.

```
int square(int x) => x * x;

void main() {
  print(square(4)); // 16
}
```

6️⃣ Optional Positional Parameters [ ]

Parameters that may be omitted.

```
void showInfo(String name, [int? age]) {
  print("Name: $name");
  print("Age: ${age ?? 'Not provided'}");
}

void main() {
  showInfo("Ali");
  showInfo("Ali", 20);
}
```

📌 Optional params must be **nullable** or have defaults.

7️⃣ Optional Named Parameters { }

Very common in Dart & Flutter.

```
void registerUser({String? name, int? age}) {
  print("Name: $name, Age: $age");
}

void main() {
  registerUser(name: "Sara", age: 22);
}
```

8️⃣ Required Named Parameters

```
void login({required String username, required String password}) {
  print("User: $username");
}

void main() {
  login(username: "admin", password: "1234");
}
```

9️⃣ Default Parameter Values

```
void greet(String name, {String country = "Iraq"}) {
  print("Hello $name from $country");
}

void main() {
  greet("Sipan");
  greet("Sipan", country: "Turkey");
}
```

🔟 Return Multiple Values (Using Records – Dart 3)

```
(int, int) minMax(int a, int b) {
  return (a < b ? a : b, a > b ? a : b);
}

void main() {
  var (min, max) = minMax(3, 7);
  print("Min: $min, Max: $max");
}
```

1️⃣1️⃣ Anonymous (Lambda) Functions

Functions **without names**.

```
void main() {
  var add = (int a, int b) {
    return a + b;
  };

  print(add(3, 4));
}
```

Arrow lambda:

```
var multiply = (int a, int b) => a * b;
```

1️⃣2️⃣ Functions as Parameters (Higher-Order)

```
void calculate(int a, int b, int Function(int, int) operation) {
  print(operation(a, b));
}

void main() {
  calculate(5, 3, (x, y) => x + y);
  calculate(5, 3, (x, y) => x * y);
}
```

1️⃣3️⃣ Recursive Functions

Function calling itself.

```
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial(n - 1);
}

void main() {
  print(factorial(5)); // 120
}
```

1️⃣ 4️⃣ Common Exam Mistakes ❌

- ❌ **Missing return**

```
int sum(int a, int b) {
  a + b; // ERROR
}
```

✔️ Correct:

```
return a + b;
```

- ❌ **Wrong parameter order**

```
login("admin", password: "123"); // ERROR
```

🧪 Practice Exercises (Try!)

- **Exercise 1**

Write a function that returns the **largest of 3 numbers**.

- **Exercise 2**

Write a function that checks whether a number is **prime**.

- **Exercise 3**

Convert this function to an arrow function:

```
int cube(int x) {
  return x * x * x;
}
```