DART

COMPLETE GUIDELINE

FIRST EDITION

AUTHOR

MOHAMMAD SANJID DAD

Guide you to be a complete developer

About the Author

Hello there, this is Mohammad Sanjid Dad an Undergraduate Student B.Sc. in CSE at RSTU Known as Rajshahi Science & Technology University. I'm passionate about software development, problem-solving, & contributing to collaborative projects that create real-world impact.

Over the past few years, I've dedicated more than 10,000 Hours to learning, building software, and strengthening my understanding of core programming concepts.

I prefer meaningful and efficient work, and I dislike repeating the same tasks unnecessarily, which drives me to automate, optimize, and improve workflows wherever possible.

Free time, I enjoy playing Football and Badminton, which help me stay focused and energized.

I live in Natore, Bangladesh, a culturally rich place known for its Famous *Kacha Golla sweet* and *Patul,* Often called the *"Mini Cox's Bazar."* I'm proud to be the youngest son in the Dad Family. My Mother is a dedicated Homemaker, and my Father is a Businessman who has always supported my academic and personal growth.

Contents

Dart Fundamentals

1.1. Variable

There is two way to create dart variable.

1. Explicit Type Declaration.
2. Inference Type Declaration.

Example 1:

Explicit Type Declaration means use the data-type when creating variable.

String variable_name = **"**Sanjid Dad**";** → **String Type Variable.**

Again,

Bool variable_name = true**;** → **Boolean Type Variable.**

Example 2:

Inference Type Declaration means use the Keyword that can store all type of variable. In Dart we use **- var, dynamic, final, const.**

var variable_name **=** "New York"**;** → **String Type Variable.**

dynamic value **=** 42**;** → **Integer Type Variable.**

const PI **=** 3.14159**;** → **Integer Type Variable.**

final name **= "**Sanjid Dad**";** → **String Type Variable.**

1.2. Functions

We can create Function in Dart by righting **Function return type** then **Function name** and **so on . . .** There are 7 type of Function we can create. Example: **void** and **all the data-type kind.**

Syntax:

return_type Function_name **(Parameters) {**

→ **Function Body**

return value**;**

}

Example 1:

Void Function_name **(Parameters) {**

→ **Function Body**

}

This Function return nothing.

Example 2:

String Function_name **(Parameters) {**

→ **Function Body**

return value**;**

}

This Function return String Value.

1.2.1. Arrow Functions

This Function is also called single-expression functions means this Function contains only one syntax.

Example 3:

return_type Function_name **(Parameters) =>** One-line expression**;**

1.2.2. Lambdas Functions

In Dart programming language **Anonymous Functions** are also called **Lambdas Function**.

This Function creates without name. They're commonly used when you need a quick, throwaway function. Especially as arguments to Higher-Order Functions. **Example: ForEach, map, where, Etc.**

Syntax:

(Parameters) {

→ **Function Body**

};

Or,

(Parameters) => Expression**;** → **Arrow Function / Single Line Expression**

Example 1:

In Dart, ForEach is a method used to iterate over elements in a collection such as a **List, Set,** or **Map**. It takes a function as an argument and applies that function to each element.

List<String> names = [**'SADIA SULTANA'**, **'SANJID DAD'**, **'KANIZ MUN'**];

names.forEach ((**element**) => **print**(**element**));

It takes parameter for accessing the value in the collections.

Example 2:

In Dart, where is a method used to filter elements in a collection based on a condition. It takes parameter as a collection data and return the filter value. The return value data-type is **Iterable<String>**

List<String> names = [**'SADIA SULTANA'**, **'SANJID DAD'**, **'KANIZ MUN'**];

Iterable<String> = **names.where** ((**element**) => **element.startsWith ('S'))**;

1.3. Control Flow

There are 2 type control flow statement in dart programming.

1. Conditional Statement. → **[IF, Else, Else..IF, Switch]**
2. Looping Statement. → **[For, While, do-While]**

1.4. Dart Collections

There are 5 type collection in dart programming.

1. **List** (**Ordered**, **Indexed collection** — like **Arrays**) → **[Variable1, Variable2, Variable3]**
2. **Set** (**Unordered**, **No Duplicates**) → **{Variable1, Variable2, Variable3}**
3. **Map** (**Key-Value Pairs**, like **Dictionaries in Python**) → **{Key: Value, Key: Value}**
4. **Queue** (From **dart:collection**, supports **FIFO**) → **Came From dart package.**
5. **Iterable** (Base Type **List**, **Set**, **etc.**) → **.map (), .where (), .reduce ()**

1.4.1. List Declare and Create.

Every programming list are same. We can create list 2 way.

1. Normal Way. → **By Just Saying List Name.**
2. With Data-Type. → **By Saying What Data-Type List Contain.**

Example 1:

List student_name = [**'SADIA SULTANA'**, **'SANJID DAD'**]; **→ By Saying List Name**

Example 2:

List<String> student_name = [**'SADIA SULTANA'**, **'SANJID DAD'**]

→ By Saying What Data-Type List Contain.

Here, we say that this list contains **String** data-type variable. **Next, Set** is same as List collection with some difference.

Example 3:

List<dynamic> student_name = [**'SADIA SULTANA'**, **100, True**]

→ By Saying What Data-Type List Contain

Here, the **List** contains 3 type of data-type. That's why we say the **List** type is **dynamic**.

1.4.2. Map Declare and Create.

Like List we can create **Map** in 2 way. One is Just saying by the name other is specify the data-type that what data that **Map** contains.

Example 1:

Map student_details = **{**

'NAME': **'Sanjid Dad'**,

'**AGE'**: **22**,

'GENDER': **'Male'**,

'ID': **'0732310005101054'**

}

Example 2:

Map<String, dynamic> student_details = **{**

'NAME': **'Sanjid Dad'**,

'**AGE'**: **22**,

'GENDER': **'Male'**,

'ID': **'0732310005101054'**

}

1.4.3. Dart Query

Dart Query is same as **Data-Structure Query**. It supports **FIFO**, means **First in First out**. Here, the Query came from dart package name **dart:collection.**

Example 1:

List<String> names = **['KANIZ MUN', 'SADIA SULTANA', 'SANJID DAD'];**

Queue<String> student_name = **Queue ();** → **Create an Empty Query.**

student_name.addAll (names);

Summary Table

Type	Ordered	Duplicates	Indexed	Example Syntax
List	Yes	Yes	Yes	**[1, 2, 3]**
Set	No	No	No	**{1, 2, 3}**
Map	N/A	Key: No	Keyed	**{'A': 1, 'B': 2}**
Queue	Yes	Yes	No	**Queue.From ([1, 2])**

1.5. Null Safety

Null Safety in Dart is a feature that helps you avoid null reference errors. By clearly distinguishing between **nullable and non-nullable** variables.

Non-nullable types: Variables can't be null unless you explicitly say so.

Nullable types: You must add? to indicate a variable can hold null.

Example 1:

Non-nullable → **String** name = **'Alice';**

name = **Null;** → **Give us Error For Assigning Null Value**

We, can't use **Null** in any variable because variable can't be **Null** in **Dart Programming.**

Let be an example, we want to declare a String data-type variable then, we want to assign the value on the run time state. But in dart we can't create a Nullable variable normally we can do in other programming language. Ex:

String student_name; → **This** student_name **Variable Hold Nothing Means Null.**

We can't create any variable like that in **Dart Programming Language**. So, what should we do. Here, we use **Null Safety** that allow us to create a nullable variable in dart programming. To do that we use some **Operators**. Ex:

Operators	Descriptions	Example
?	Use to create a **Null Variable**	**String?** name**;**
!	Say that the **Variable Not Null**	**To take Input From User**

Example 2:

String name**;**

print(**name**); → **Give us Error Because in Dart Variable Can't be Null**

String? name**;**

print(**name**); → **Give us no Error Because we use Null Safety Operators**

Example 3:

Import 'dart:io';

String name = **stdin.readLineSync ();**

→ Here, it gives us Error Because the user Input can't be Null

So,

Import 'dart:io';

String name = **stdin.readLineSync()!;**

→ Here, it gives us no Error Because we use ! Operators.

Otherside, we can use **late** Keyword to say that we will assign the value later.

Example 4:

Import 'dart:io';

late String name**;**

void user_name() **{**

name = **stdin.readLineSync()!;**

→ We take Input From user and assign to the variable name

}

Here, without late, Dart would require immediate assignment**.** We can also create **Nullable** Functions **OR** a **Nullable** Functions Parameters.

1.6. String Interpolation

String Interpolation in Dart allows you to embed variables or expressions directly inside a string. Using **$** or **${}** it's a clean and readable way to build dynamic strings.

Example 1:

```
String name = ' MOHAMMAD SANJID DAD ';
print ('||| HELLO || $name |||');          → Using the $ to Interpolation the String
```

Example 2:

```
final apples = 3;
final bananas = 2;
print ('Total Fruits: ${apples + bananas}');
```

→ Using the ${} to Interpolation the String

To interpolate, use double quotes "..." or triple quotes """...""".

1.7. Exception Handling

Exception Handling in Dart allows you to gracefully handle errors during runtime without crashing your program.

Here, some common Exception name:

- **Invalid Input**
- **Division By Zero**
- **File Not Found**
- **Network Issues**

To **Handel Exception** we use **try & catch** section. Ex:

```
try {
        // Code | Statement
} catch (e) {
        // Code | Statement
}
```

Here, we place our code in **try** block. If we get some **Errors** then it raises to **catch** section and the error will be store in the parameters **e**.

Example 1:

```
try {
        int result = 10 ~/ 0;          → Integer division by zero Exception
        Print (result);
```

} **catch** (e) {

Print ('An Error Occurred: $e'**);** → **Raises IntegerDivisionByZeroException**

}

Now, we learn the basic about the **Exception Handling**. In the **catch** section we pass the parameter **e**. This **e** means **exception name**. We can pass another parameter to the **stack trace** of the error.

Example 2:

try {

int result = **10 ~/ 0**; → **Integer division by zero Exception**

print(**result**);

} **catch** (e, s) {

Print ('An Error Occurred: **$e**'**);** → **Raises IntegerDivisionByZeroException**

Print ('Stack Trace: **$s**'**);** → **Print the all Stack Trace the Error**

}

Now, we learn **Catch Error and Stack Trace**. Now, we can use **on** keyword to catch some **Specific Exceptions**.

Example 3:

try {

int num = **int.parse (**'ABC'**);** → **Raises Format Exception**

} **on FormatException catch** (e) {

Print (e**);** → **Print FormatException Error**

}

Now, we can use the **finally keyword** to show the user that all error is handled carefully. Ex:

finally {

Print ('All Error are Handled Carefully.'**);**

}

1.7.1. Throwing My Own Exception & Custom Exception

To create my own **exception**, we use **throw keyword** and use **Exception Function** that take parameter to prompt the user to see the results.

Example 1:

```
void check_age (final age) {
        if (age < 18) {
                throw Exception ('Age must be 18 or older');
                → Create My Own Exception
        }
}
void main () {
        try {
                check_age (16);          → Throw Exception 'Age must be 18 or older'
        } catch (e) {
                print(e);                → Print the Exception in the Output.
        }
}
```

Dart Object-Oriented Programming

2.1. Dart Class & Objects

Dart Class & Object is same as every programming language. We can create class outside the **main Function**. Ex:

class Class_name **{**

- ➔ **Constructors**
- ➔ **Methods and Attributes**

}

Creating Object:

Class_name object_name = **Class_constructor (**Parameter**);**

Class means, a **Blueprints** and **Objects** are the **Instances** of the **Class**. Just like in other **Object-Oriented Languages**.

2.2. Dart Class Constructors

A Constructor in Dart is a special function that's automatically called when an **Object** is created. Mainly, it is used to initialize fields **(Properties)** of a class.

In Dart, we use 7 type of **Constructors**:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Named Constructor**
4. **Const Constructor**
5. **Factory Constructor**
6. **Redirecting Constructor**

2.2.1. Default Constructor

Default Constructor Has **no parameters.** Is automatically provided by Dart if no constructor is defined in your class.

There are 2 way to call default constructor:

1. Implicit Default Constructor

2. Explicit Default Constructor

2.2.1.1. Implicit Default Constructor

IF you don't write any constructor, Dart gives you one that is called Implicit Default Constructor. Ex:

```
class Car {
    String brand = 'Toyota';
    void display () {
        print ('Brand: $brand');
    }
}

Car BMW = Car ();
```

Here, **Car ()** is the default constructor automatically provided by Dart.

2.2.1.2. Explicit Default Constructor

Here, we right **Constructor** to create a default **Constructor**. We can right **Constructor Empty** | **Constructor with Initialization.**

Example 1:

```
class Student {
    String name;
    int age;
    Student () {                    → Default Constructors Just by Creating Name.
        name = 'Unknown';
        age = 0;
    }
    void show () {
        print ('Name: $name, Age: $age');
    }
}
```

In Dart, once you define any constructor, Dart no longer provides the default one automatically.

IF you need both a default and a parameterized constructor, define both explicitly.

2.2.2 Parameterized Constructor

A Parameterized Constructors lets you pass values to the class when creating an object, so you can initialize fields directly at the time of **Object Creation**.

Example 1:

```
class Student {
        String name;
        int age;
        Student (String n, int a) {
        → Parameterized Constructors with 2 Parameters
                name = n;
                age = a;
        }
}
Student Sanjid_dad = Student ('Sanjid Dad', 22);
→ Object Creation with Passing Parameters
```

Example 2:

```
class Student {
        String name;
        int age;
        Student (this.name, this.age);
        → Parameterized Constructors with this Keyword.
}
```

By using **this** Keyword, we can assign value direct to the **attribute or variable**. Here, we don't Have to assign value like we do in our last example.

Example 3:

```
class Student {

        String name;

        int age;

        Student ({this.name = 'Unknown', this.age = 0});

        → Constructors with assign value.

}

Student Sanjid_dad = Student (name = 'Sanjid Dad', age = 22);

→ Object with Passing Parameters with Name.
```

Here, when we use **{}** in the **Class Constructors** that time we must initialize the value in the **Constructor**.

Otherwise it causes **Error**. Here, we can use one other way by using **required Operators. Required Operator** means we mast assign the value when we creating the **Objects.**

Example 4:

```
class Student {

        String name; int age;

        Student ({required this.name, required this.age});

        → Parameters with required Keyword.

}

Student Sanjid_dad = Student (name = 'Sanjid Dad', age = 22);

→ Object with Passing Parameters with Name.
```

Here, we use **required Operator** to tell the **compiler** that when we create an **Object** we will assign the **Perimeters**. We will see a lot of work with it when we create **Fultter** or **Develop Applications**.

Example 5:

```
class Student {

        String name; int age;
```

Student ([this.name = 'Unknown', this.age = 0]);

→ Constructor with Positional Parameters.

}

Student Sanjid_dad = **Student** (**'Sanjid Dad', 22**);

→ Object with Passing Parameters with Name.

Here, we create a **Class Constructor** that take **2 Parameters** those **Parameters** are **Positional Parameters**. Means we must assign the value when we create it.

Otherside, when we create an **Object** that time we must follow by the position like we do in **Error Handling**. That time **First Parameter** is **Error type or name** & **Second Parameter is the Stack Tress**. Likewise, Here **First is the name** & **Second is the age.**

IF we don't give any parameter when we create the object that's fine the compiler didn't give us any **Error**.

2.2.3 Named Constructor

A Named Constructor lets you create multiple constructors in a class with different names. This is useful when you want to initialize objects in different ways.

Example 1:

class Student **{**

late int age**;**

late String name**;**

Student (**this.name, this.age**)**;**

→ Default Parameterize Constructor with this Keyword.

Student.guest (**this.name, this.age**)**;**

→ Name Parameterize Constructor with this Keyword.

}

Student S1 = Student **('Ali', 21);** **→ Create Object with Default Constructors.**

Student S2 = Student**.guest ('Ali', 21);** **→ Create Object with Named Constructors.**

Here, we create one **Named Constructors**. But we can create as many as we want **Named Constructors**. By doing that we can create very big project or we can solve multiple problem with one **Class**.

2.2.4 Const Constructor

A Const Constructor creates an object that is **compile-time constant** — meaning its value is fixed and cannot change after it's created. These constructors improve performance and memory efficiency, especially in **Flutter** apps.

Example 1:

```
class Student {
    final String name;
    final String gender;
    const Student (this.name, this. gender);
    → Const Constructor with this Keyword.
}
```

In Dart Programming, when we use **const Constructor** we must use **final** to all **Attribute in the Class**. For the **const Constructor** we can create a **const Object** or **final Object**. When we use **const constructor** we must take all **attribute as a parameter** to work the program. **Because, all attribute is final.**

Example 2:

```
class Student {
    final String name;
    final String gender;
    const Student (this.name, this.gender);
    → Const Constructor with this Keyword.
}
const student_one = Student ('Sanjid Dad', 'Male');
→ Const Object For Const Constructor
final student_one = const Student ('Sanjid Dad', 'Male');
→ Final Object For Const Constructor
```

In Flutter development, we use most of the time this **const constructor**. Because, **const constructor** is more efficient by the other **constructors**.

Summary Table

Feature	Const Constructor	Normal Constructor
Immutable	Yes	No
Compile-time	Yes	No
Performance	**Optimized**	**Less Efficient**
Requires final	Yes	No

2.2.5 Factory Constructor

A Factory Constructor is a special type of constructor in **Dart** that **doesn't always create a new instance** of a class. **Instead, it can:**

- Return a **cached instance**.
- Return a **subclass**.
- Perform **logic before returning an object**.
- Use **Factory** when you don't want to **return a new object every time**.

2.2.5.1. Cached In Factory Constructors

In Dart, a factory constructor can **cache (store)** and **reuse previously created objects** instead of making a **new one each time**.

It means **the object is saved in memory (in a variable or map)** so the next time you need the same object, **Dart just returns the existing one — not a new copy.**

That means, in another constructor we can create as many as **Object** we want. Also, we can create **Object** that use same parameters **(Means Same Object with Same Parameters But Different Name)**.

Example 1:

```
class Student {
    late String student_name;
    Student (this.student_name) {
        Print ('Hello, this is default constructor. Hi, $student_name.');
    }
}
```

Student user_one **= Student (**'Sanjid Dad'**);**

Student user_two **= Student (**'Sanjid Dad'**);**

Here, we create a **class name Student** that take one **parameter student name**. We can create many Object using the same parameter. That repeat our object same again and again allow as **create same instance again and again**.

In Dart, we can control it by creating **Factory Constructor**. We cannot create an **Instance of Factory Constructor**.

Factory Constructor return the **named constructor or const constructor** with **some condition**. That **condition is do we create the same instance previously**. **IF – YES –** then return the same instance that we create previously. **Otherwise, create a new one**.

Now, the question where we check. We check in the **Cache (Memory – Variable or Map).**

Example 2:

```dart
class Student {
  late String student_name;

  static final Map<String, Student> _cache = {};

  factory Student(String student_name) {
    if (_cache.containsKey(student_name)) {
      return _cache[student_name]!;
    } else {
      Student user = Student._internal(student_name);
      _cache[student_name] = user;
      return user;
    }
  }

  Student._internal(this.student_name);
}
```

In the above example we create a **Class name Student** then create an **Attribute name student_name** the create a **Map name _cache** that contains the **Instance/Object** we create by the **Student Class**.

Then we create a Factory **Constructors** that check the condition that if the Object we create now is created in the previous then just return the **same Object**. **Otherwise create a new Object** then return it.

Remember that **Factory Constructors** doesn't create an Object it checks some condition then return the **named constructors**.

Key Differences Between Factory and Const/Normal Constructors

Feature	Factory Constructor	Const / Normal Constructor
Returns new object	Not always	Always
Logic allowed	Yes	No logic
Subclass return	Yes	No

2.2.6. Redirecting Constructors

A Redirecting Constructor is a constructor that calls another constructor in the same class to reuse initialization logic. Instead of repeating code, you can redirect one constructor to another using **(:)** syntax.

Example 1:

```
class Student {
    String student_name;
    int student_age;
    Student (this.student_name, this.student_age) {
    → Para. Constructor with direct initializer
        Print ('Student name: $name. Student age: $age.');
    }
    Student.guest (String name, int age) : this (name, age);
    → Redirect Constructor with Parameters
}
```

Notes Table

Redirecting constructors cannot have a **function body.**
You can redirect to any constructor in the same class **(even a named one).**
You **can't redirect to a factory constructor.**

2.3. Class Inheritance

Inheritance is a key concept in **Object-Oriented Programming (OOP)**. It allows one class (called the child or subclass) to inherit the properties and methods of another class (called the parent or superclass). **Dart supports only single inheritance** – a class can inherit from one superclass.

Example 1:

```
class Parent {                          → Parent Class Contain Method sayHello ()
    void sayHello () {
        print ("Hello from Parent");
```

```
        }
}
class Child extends Parent {          → Child Class Inherited Parent Class All Data
        void sayHi () {
                print ("Hi from Child");
        }
}

Child c = Child ();                                  → Create Child Class Object Name c
c.sayHello ();                        → Call Parent Class Method by Child Class Object
c.sayHi ();                           → Call Child Class Method by the Same Object
```

In Class Inheritance, we always create **Child Class Object.** Then by the **Object** we call the **Parent Class Method** or **Attribute**.

2.3.1. Parent Class Constructor Inheritance By Super Keyword

In Dart, we can inherit **Parent Class Constructor** by **super Keyword**. The Syntax look like:

Child_Class_Constructor () **: super** ();

We can do this by using **(:) Operator** and **super ().** It will call the parent class constructor and all the attribute or method that constructor contains.

Like wise we can Inherit **Parent Class Method** by **Super Keyword**. The Syntax look like:

super**.Parent_Class_Method_Name** ();

Example 1:

Here, we use **Class Inheritance, Constructor Inheritance** also **Method and Attribute Inheritance** using the **super Keyword** and so. . . on,

2.4. This, Static and Method Overriding In Dart

We will see **this, static, Method Overriding** one by one. To learn those, you must complete the section we complete.

2.4.1. Method Overriding

Method Overriding is same in every language. But, in dart we have some difference. Let's see one by one:

In Dart, you override a method when you want to provide a **specific implementation** for a method that already exists in the **parent class.**

Example 1:

```
class Parent {
    void greet () {
        print ("Hello from Parent");
    }
}
class Child extends Parent {
    @override
    void greet () {
        print ("Hello from Child");
    }
}
```

In Dart, you can override method by using the **@override Syntax. In overriding,** you must right the same name and the parameter in the Child Class.

Rules of Overriding in Dart

Rule	Description
Must use same method name	Including same parameters
Can change method body	But not return type (unless it's a subtype)
Use **@override**	For clarity and safety
Can call **super.method()**	To include parent logic

2.4.2. This Keyword In Dart

The **this** keyword in Dart is a reference to the current instance of a class. It's used inside a class. For:

- Access instance variables and methods
- Differentiate between class fields and constructor parameters
- Pass the current instance as a parameter

2.4.3. Static Keyword In Dart

In Dart, the **static keyword** is used to declare **class-level variables or methods** — meaning they belong to the **class**, not to any **specific object**.

That means when we create a static variable or method we say **it belong to the Class** not the **Object**. Here, the **Object means the Class Object.**

What Does static Do:

- You can access static members without creating an instance of the class.
- Useful for utility methods, shared data, or constants.

We can access the static variable or method by right - **ClassName.static_variable_name**

Example 1:

```dart
class Shape {
  late String name;

  static const PI = 3.14165;

  Shape(this.name) {...}

  void _print_shape_name() {...}
}

class Rectangle extends Shape {
  late final length;
  late final weight;

  Rectangle(String shape_name, this.weight, this.length) : super(shape_name);

  int calculate_area() {
    try {
      print('THE AREA OF THE ${super.name}: ${length * weight}');
      return length * weight;
    } catch (exception, stack) {
      print(exception);
      print('Stack: ${stack}');
      return 0;
    }
  }
}
```

The Output:

```
THIS IS A Circle
THE AREA OF THE Circle: 78.54124999999999
THIS IS A Rectangle
THE AREA OF THE Rectangle: 50
```

Here, in this example we use the **static variable** name **PI**. We access the variable by right the **Class** name **(.) Variable** name. Ex:

Shape.PI

2.5 Abstract Classes & Interfaces

Dart supports object-oriented programming, and like other OOP languages, it provides **abstract classes and interfaces** for creating flexible and reusable code structures. **While Dart** doesn't have a **separate interface keyword**. Every class in Dart can act as an interface.

```dart
class Circle extends Shape {
  late final radius;

  Circle(String shape_name, this.radius) : super(shape_name);

  double calculate_area() {
    try {
      print('THE AREA OF THE ${super.name}: ${Shape.PI * (radius * radius)}');
      return Shape.PI * (radius * radius);
    } catch (exception, stack) {
      print(exception);
      print('Stack: ${stack}');
      return 0;
    }
  }
}

void main() {
  Circle C = Circle('Circle', 5);
  C.calculate_area();
  Rectangle R = Rectangle('Rectangle', 10, 5);
  R.calculate_area();
}
```

Abstract Classes

- Is a class that **cannot be instantiated directly.**
- Can have abstract methods **(methods without implementation).**
- Can have **implemented methods, fields, and constructors**
- Is designed to be **inherited by subclasses.**

Example 1:

```
abstract class Animal {
    void make_sound ();
    → Abstract Method That Don't Contains Method Body.
    void breathe () {                    → Normal Method That Contains Method Body.
        print ("Breathing...");
    }
}
```

In Dart Programming, we can create both **Abstract Method or Normal Method**. **But**, some other **Programming Language** we can't create a **Normal Method in Abstract Class. Now,** as we see we can't create an **Object From Abstract Class. So,** we must **extend** it by other **Sub Class** then we can create an **Object** and access the method and attribute.

When we extend the **Abstract Class** we must assign value to the **Abstract Attribute or Method.**

Example 2:

```
abstract class Animal {
    void make_sound ();
    → Abstract Method That Don't Contains Method Body.
    void breathe () {
        → Normal Method That Contains Method Body.
        print ("Breathing...");
    }
}
class Dog extends Animal {                    → Extends Abstract Class Name Animal.
    @override
    void makeSound () {
        → Override The Abstract Method & Assign Method Body.
        print("Bark");
    }
}
```

Now, we can use **Factory Constructor** in **Abstract Class** too. But there is some **Problem That We Can Not Create an Instantiated in Factory Constructor.**

Here, An Instantiated means. . .

ClassName._initials ();

We cannot right it in the **Factory Constructor**. Because, **Abstract Class Doesn't Support Instantiated.**

So, we must create another class then **Extends the Abstract Class** then **Return the Extends Class Name.**

```dart
abstract class Skeleton {
  late final String type;

  static final Map<String, Skeleton> _cache = {};

  factory Skeleton(String type) {
    if (_cache.containsKey(type)) {
      return _cache[type]!;
    } else {
      _cache[type] = ConcreteSkeleton(type);
      return ConcreteSkeleton(type);
    }
  }

  Skeleton._initials(this.type);

  String section_name();

  void print_details() {...}
}
```

```dart
class ConcreteSkeleton extends Skeleton {
  late final String name;

  ConcreteSkeleton(String type) : super._initials(type);

  @override
  String section_name() {
    return "THE SKELETON IS ${super.type} TYPE. NAME ${name}";
  }
}
```

Interface In Dart

In Dart, **every class is an interface**. You can use any class as an interface by using the **implements keyword**.

When you implement a class, **you must override all its methods and fields**, **even if they have implementations**.

Example 1:

```
class Vehicle {
    void start () {
        print ("Starting engine...");
    }
}

class Car implements Vehicle {
    @override
    void start () {
        print ("Car started");
    }
}
```

Unlike **extends**, **implements** don't inherit **implementation**, only the **structure (method signatures).** When you create a **Class with Argument Constructor**. IF you **implement** the **Class** then the **Constructor** will be no use **it will auto disable.**

Scenario	**Use**
You want to **define base functionality**	Abstract Class **(extends)**
You want to **enforce a structure only**	Interface **(implements)**
You need to **share implementation and logic**	Abstract class **(extends)**

2.6 Mixins in Dart

A Mixin in Dart, is a way to reuse code across multiple classes without using traditional inheritance. They allow you to add functionality to classes in a **flexible, composable way**.

- To share **methods and properties across different classes**
- Avoid **limitations of single inheritance**
- Provide reusable behaviors without needing a **class hierarchy**

Rules For Mixins

- Use the **mixin Keyword**
- Mixins cannot be **instantiated (Like the Abstract Class)**
- A class can use **multiple mixins**
- Mixins **can extend or apply constraints** using **on Keyword**

Syntax:

```
mixin MixinName {
    → Methods and Attributes.
}
class ClassName with MixinName {
    → Can Access Methods and Attributes.
}
```

Example 1:

```
mixin Logger {
    void log (String message) {
        print ("LOG: $message");
    }
}
class FileManager with Logger {
    void saveFile () {
        log ("File saved");
    }
}
```

You can use multiple mixins by **(,)** sign,

Example 2:

```
mixin A {
    void a () => print('A');
}
mixin B {
    void b () => print('B');
}
class C with A, B {}
```

By that we can use **A & B** mixin **methods and attribute**.

Now Mixins with Constraints, means we can use **extends & mixin** at a same time. To do that we **first use mixin** then **extends**.

Example 3:

```
class Animal {
    void eat () => print ("Animal eats");
}
mixin CanRun on Animal {
    void run () => print ("Running...");
}
class Dog extends Animal with CanRun {}
```

2.7 Getters and Setters

In Dart, getters and setters are special methods that let you **read and write private or protected fields** of a class in a safe and controlled way. They allow **encapsulation**, which is a key principle in **object-oriented programming.**

A Getter is used to **retrieve a value** from a private field & **A Setter** is used to **update or change a value** of a private field.

Benefit	Description
Encapsulation	Control access to internal fields
Validation logic	Add checks in setters before assigning values

Example 1:

```
class Person {
    String _name = '';                          → Private Attribute or Variable.
    String get name => _name;                   → Get the Private Variable Value.
    set name (String value) {                   → Set the Private Variable with Condition.
        if (value.isNotEmpty) {
            _name = value;
        } else {
            Print ("Name cannot be empty");
```

```
            }
        }
    }
    void main () {
        Person SanjidDad = Person ();
        SanjidDad.name = 'Sanjid Dad';
        print (SanjidDad.name);
    }
```

In Dart Programming Language, we can use **Condition** in the **Setters** that make our **Setter stronger & more Controlled. Here,** the **setter and getter** must be the **same name**. Here, in the example we use **name**. **Setter** must take **one or more parameters** as an input and assign to the **Private Variable**. Then, **Getter** return the **Private Variable** that we can **Print**.

In the main Function, we use **[SanjidDad.**name **= '**Sanjid Dad**';]** this to set the name in the private variable.

Behind the scene, Dart called the **[set** name **(**String value**)] Method** & set in the Private Variable. **Then,** we **Print the Getters** by **Print** the same thing **[print (**SanjidDad**.name);]**

Advanced Dart Programming

3.1. Asynchronous Programming

Asynchronous programming in Dart allows your app to perform time-consuming tasks **(File I/O, API Calls, or Timers)** without freezing the program. It is needed because: **To avoid blocking operations,**

1. Network requests **2. File Read/Write** **3. Database Access**

4. Delays

In Asynchronous Programming we use the concepts:

Concept	Keyword/Type	Purpose
Future	Future<T>	Represents a value that will be available in the future.
async	async	Marks a function that has await calls means Function is waiting for result.
await	await	Waits for the future to complete
then	.then()	Handles future results (chaining)

3.1.1. Future In Dart

A Future in Dart, represents a **potential value or error** that will be available at some point in the future usually the result of an asynchronous operation like **reading a file, making an HTTP request, or querying a database**.

Think of a Future like a promise to return a value later, **not immediately.**

Syntax:

Future<String> method_name **() {**

→ Code Operations return or print data.

}

States Of a Future:

- **Uncompleted** – Waiting for the result.
- **Completed with data** – Successfully returned a value.
- **Completed with error** – Failed with an exception.

Example 1:

```
Future<String> print_data () {
    return Future.delayed (
        Duration (seconds: 2),
        () => 'Data Loaded',
    );
}
```

Here, in this example this **Future will complete after 2 seconds.** Until then, it's **Incomplete**. IF came with value then **Complete with data** otherwise **Complete with error.**

Here, the program returns the String **'Data Loaded'**.

3.1.2. Async In Dart

In Dart Programming, the **async keyword** used to mark a function as asynchronous.

Meaning, **it returns a Future (implicitly)**. You can use **await** inside it to pause execution until the awaited future completes.

We use async, to write non-blocking code that waits for **time-consuming operations. Like:**

- Network requests.
- File I/O.
- Timers.
- Database calls.

Without Freezing the App or Function Flow.

Syntax:

```
Future<String> method_name () async {
    → Code Operations return or print data.
}
```

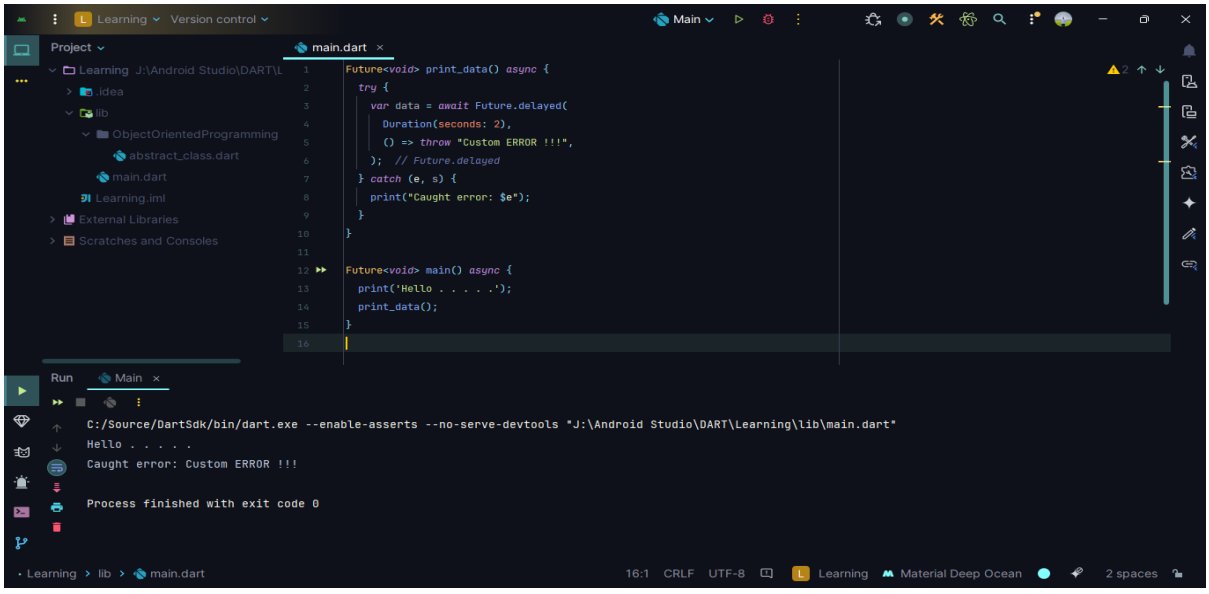
This is same as Future. You just put **async** in the middle between **the () and the {}.**

Example 1:

```
Future<String> print_data () async {
        return Future.delayed (
                Duration (seconds: 2),
                () => 'Data Loaded After 2 Second',
        );
}
void main () async {
        print('Start');
        String data = print_data ();
        print (data);
        print ('End');
}
```

Here, in this example print **Start** then after 2 second print **Data Loaded after 2 second** then print **End. IF** we call the Function that use **Future and async** then we must use **async in the main Function to** because it only prints when it **Find async.**

Example 2:



Like this we can use **main Function** as a **Future<void>** to print the **Future Function** name **print_data**

3.1.3. Await In Dart

The await keyword in Dart is used **inside an async function** to pause execution until a Future completes. It waits for an **asynchronous operation to finish** before **continuing the next line of code**.

Example 1:

```
Future<String> print_data () async {
        await Future.delayed (Duration (seconds: 2));
        return 'Data Loaded After 2 Second';
}
Future<void> main () async {
        print('Start');
        String data = await print_data ();
        print (data);
}
```

3.2 Generics (List<T>, Map<K, V>) In Dart

Generics in Dart allow you to write flexible, reusable, and type-safe code. **Instead of specifying a data type directly**, you use **type parameters like T, K, and V.**

3.2.1. What Are Generics

Generics let you write code that works with any data type, while still maintaining type safety at compile time. Some Common Generic Types in Dart:

1. **List<T> (A List Of Elements Of Type T).**
2. **Map<K, V> (A Map With Keys Of Type K & Values Of Type V).**

Example 1:

```
List<int> numbers = [1, 2, 3];
List<String> names = ['Alice', 'Bob'];
```

Here, the we use **List<T>**. **T** means the **type that the list contains**.

Map<String, int> scores **= {**

'Math': 95**,**

'Science': 90**,**

};

Here, the we use **Map<K, V>**. **K** means the **map key** & the **V** means the **type that the map contains**. **Now,** we can create our own generics type. **Here,** an example,

```
class CustomType<T> {
  late T value;

  CustomType(this.value);

  T print_data() {
    print('Value: ${value}');
    return this.value;
  }
}

void main() {
  CustomType<String> number_type = CustomType<String>('Sanjid Dad');
  number_type.print_data();
}
```

Again, using the int data-type,

```
class CustomType<T> {
  late T value;

  CustomType(this.value);

  T print_data() {
    print('Value: ${value}');
    return this.value;
  }
}

void main() {
  CustomType<int> number_type = CustomType<int>(100);
  number_type.print_data();
}
```

Why Use Generics,

Benefit	**Description**
Type safety	Catch errors at compile time
Reusability	Write code once, use with any type
Better performance	Avoids runtime type checks
Cleaner code	Avoids casting

3.3. Extensions In Dart

Extensions in Dart allow you to add **new functionality to existing classes** - even if you **didn't write or own those classes** - without modifying the original source code.

An extension lets you define **additional methods, getters, or setters for a class** outside of the class itself. This is especially useful for some data-type operations. **Like: String, int, double, Etc.**

Syntax:

```
extension user_define_name on data-type/class {
        → Your Own Code Like Logic or Syntax.
}
```

Example 1:

```
extension number_extension on int {
        bool isEven () => this % 2 == 0;
        bool squared () => this * this;
}
void main () {
        int number = 100;
        print (number.isEven);          → Print True or False by the Number.
        print (number.squared());       → Print the Squared Number.
}
```

Here, In this example we create an Extension For int name **squared ()** it squared the numbers then return the numbers. It is useful when you to check some conditions to the variable Before you use it in the **Operations**.

Example 2:

```
extension email_validator on String {
        bool isValidEmail () => contains ('@') && contains ('.');
}
```

void main **() {**

String email **= "**test@example.com**";**

print (email.isValidEmail()**);** **→ Print True or False by the Number.**

}

Here, Is a good example that, Spouse you take input the user. Then you **check the input is it valid or not**. **IF the input is valid** then use it on the **Operation**. **Otherwise** show user **Error**.

3.4 Typedefs In Dart

In Dart, typedef is used to create a **custom name** For a **Function Type**. **A typedef (**Short For Type Definition**)** allows you to name a Function type and then use that name to **declare variables, parameters, or Fields that accept Functions of that type.**

There are two Syntax in Typedefs. First one is only use by **creating** the **Function name** with or without **Parameter** & the second one is creating a variable by use the **return type then Function name** with or without **Parameters**.

Syntax:

Typedef Function_name **(Parameters);** **→ First Syntax Options.**

Typedef variable_name = **return_type** Function_name **(Parameters);**

→ Second Syntax Options.

In Dart, we can create a **Function model** then create as many as **Function** we want use by the **Function model**. Then, what we do just **create a main variable the Typedefs** & call the **Function by the variable**.

This is a core concept in **Functional Programming: Passing Functions as values.**

Example 1:

```dart
typedef Greet = Function({required String name, required int age});

print_greet({required String name, required int age}) {
  print("Hello $name, you are $age years old.");
}

void main() {
  Greet greet = print_greet;
  greet(name: "Sanjid", age: 22);
}
```

Here, we create a **Typedefs Function** then create a **Function name print_greet** that **contains same Parameters** in the **Typedefs Functions.**

Then, we create a **variable name greet** in the **main Function** by the **type Greet (What We Create in the Typedefs)** then **assign the value as a Function** we create earlier name **print_greet.**

Then, pass the **Parameters** to call the **Function**.

Example 2:

```
typedef Operation = int Function(int a, int b);

int add(int a, int b) => a + b;

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

void execute(Operation op, int x, int y) => print("Result: ${op(x, y)}");

void main() {
  execute(add, 4, 5);
  execute(multiply, 4, 5);
}
```

Here, you created a **template (typedef)** to use **math operations**, and then reused it to pass different **Functions (add, multiply)** as **arguments** into another **Function (execute)**.

Why Use Typedefs,

Concept	Descriptions
Typedefs	Creates a name for a function signature
Cleaner code	Helps reduce repetition in complex signatures
Use cases	Callbacks, Flutter widgets, Functional programming

3.5 Enums (Including Enhanced Enums) In Dart

In Programming Languages, we use **constant.** Every **Programming Language** use different type **Syntax Constant.** In **Dart Programming,** we use **Enums (short for enumeration)** it is a special type in Dart used to define a **fixed set of constant values.**

Syntax:

Enum variable_name **{**

- ➔ **Constant Fields.**

}

To access the element and work with it we use **Enums as a Class & Object**. First create an **Enums** then **create a variable** to the **main Function** to access the Enums.

Example 1:

```
enum Colorname {                          → Create an Enum name Colorname
        red,
        green,
        white,
}
Void main () {
        Colorname select_color = Colorname.red;
        → Create a variable For Enum & Assign value.
        Print (select_color);             → Print the variable select_color.
        Print (select_color.name);
        → Print the Name Of the Enum Constant By Variable.
}
```

Now, How we work with Enums,

```
enum Colorname {                          → Create an Enum name Colorname
        red,
        green,
        white,
}
Void describe_color (Colorname color) {
        switch (color) {
                case Colorname.red:
                        print ("Red color");
                        break;
```

```
            case Colorname.green:
                print ("Green color");
                break;
            case Colorname.white:
                print ("White color");
                break;
            case Colorname.orange:
                print ("Orange color");
                break;
        }
    }
```

Here, we use **switch case**, For **each Enum Constant** we create **each Block Code** that print some data. **Remember that** when we **create an Enum we must declare and assign value to all. IF** we **miss one** then the code will give us **Error.**

Why Use Enums:

1. Improves code **readability.**
2. Prevents use of **magic strings or numbers.**
3. **Enforces** type safety.
4. Used in **state management, switch cases, options,** etc.

3.5.1. Enhanced Enums

Enhanced Enums take traditional **Enums** to the next level. They allow you to define **Fields, Constructors, Methods, and Even Implement Interfaces** or **Mixins** — making **Enums** powerful and expressive.

An enhanced enum is just **like a class:**

1. You can define Fields. **2. Add a Constructor** **3. Define Methods**

4. Implement Interfaces **5. Use switch, name, index, and more.**

In Dart Programming, Enhanced Enums is more like **Class**. Like the **Class** we can create **Constructor, Method, Fields** also we can **Implement Interfaces** and we can use **switch, name, index** or more in **Operations**.

In Class we must create an **Object** to work with it. **But,** in **Enums** we do not create any **Object.** We call directly by the **name or index.**

Example 1:

```
Enum Vehicle {
    Car (max_speed: 120),
    → Classic Enums What we Create In the Past.
    Bike (max_speed: 80),
    → Classic Enums What we Create In the Past.
    Truck (max_speed: 60);
    → Classic Enums What we Create In the Past.

    final int max_speed;
    → Create New Filed Name max_speed.

    const Vehicle ({required this.max_speed});
    → Create Constructor Like Classes.

    void describe () {                    → Create Method Name describe.
        print ('The $name can go up to $max_speed km/h');
    }
}
```

In this example, we see we can create all the thing we can create in **Dart Classes.**

```
Void main () {
    Vehicle.Car.describe ();              → Call the describe Function.
    Print (Vehicle.Bick.max_speed);       → Print the max_speed Variable.
}
```

Here, In Dart Programming we must create an **Object** to use the **Class. Enums** are same as **Class. But,** we do not need any **Object** to use the **Enums Value. Because,** In this Enums we create **3 Constructors not only Constructors it is named Constructors.**

So, we can say that we can use **Enums** by the **Constructors.**

By Saying --- Enums_name.Constructors_name.Method_name

We use **(.) Operators** to access the **Variable** or **Method** in the **Enums** same as **Classes & Objects.**

Rules For **Enhanced Enums:**

1. Must declare **const Constructors.**
2. **Fields** must be **Final.**

Example 2:

```dart
abstract class Text {
  String greet();
}

enum Greeting implements Text {
  morning("Good morning 😊😊"),
  night("Good night 😴😴");

  final String message;

  const Greeting(this.message);

  @override
  String greet() => message;
}

void main() {
  print(Greeting.night.greet());
}
```

Here, we create **Abstract Class** name **Text.** Then **Implement** the **Text Class** in the **Enums. So,** we can **Implement** in **Enums** we **cannot Extend Enums.**

Enums & Enhanced Enums Summary:

Feature	Basic Enum	Enhanced Enum
Fixed values	Yes	Yes
Fields & methods	No	Yes
Constructors	No	Yes
Implements	No	Yes

3.6 Cascading Notation In Dart

The Cascade Notation (..) in Dart allows you to perform multiple operations **on the same object without repeating its name.** It improves code readability and conciseness, especially when configuring objects with many properties or methods.

Syntax:

Object_name.**Property =** Value**;** → **Access & Assign with . Notations.**

..Property = Value**;** → **Assign Value With .. Cascading Notation.**

Why Use Cascade Notation

Without Cascade	**With Cascade**
person.**name** = "Alice"**;**	person**..name** = "Alice"
person.**age** = 30**;**	**..age** = 30
person.**greet ();**	**..**greet **();**

Example 1:

```dart
class Person {
  late String name;
  late int age;

  void greet() {
    print("Hello, my name is $name and I am $age years old.");
  }
}

void main() {
  var person = Person()
    ..name = "Sanjid Dad"
    ..age = 22
    ..greet();
}
```

Here, we create a **Class** name **Person** then create **two Variables**. **One name & one age**. Then create a **method** name **greet**.

Now in the main **Function** we create an **Object** name **Person** then we use **Cascading notation (..)** to assign the value in **name & age variable**. Then same as we **call the method by using the Cascading notation**.

Here, one thing to remembers we do not give any **semicolon (;)** to any **syntax**.

Example 2:

List<String> names **= []**

..add("Alice**")**

..add("Bob**")**

..add("Charlie**");**

We can also use the **Cascading Notation** in **Method Chaining.**

Dot	Meaning
.	Call or access directly
..	Call and return the object again

3.7 Tear - Off Syntax In dart

The Tear-Off Syntax in Dart allows you to **Pass Functions** or refer to **Methods or Constructors** without calling them. It's like taking a **Function or Method** and **"Tearing it Off"** from its **Object or Class** to use it as a First-Class Function.

Syntax:

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

void main() {
  var sayHello = greet;
  sayHello("Alice");
}
```

Here, in this example we **First Create a Method** then in the main **Function** we **First Create a Variable name sayHello.** In that **Variable** we assign the **Function name** not the **Function Fully.**

In Class & Constructors Example:

```dart
class Skeleton {
  late String type;

  Skeleton(this.type);
}

void main() {
  final create_new = Skeleton.new;
  final user = create_new('HUMAN');
  print('Skeleton type: ${user.type}');
}
```

Here, in this example we create a class name **Skeleton** the create a variable name type & a constructor. So, this class take a parameter name type. Now, In the main **Function** we First create a variable For **Class reference** by saying **Class_name.new (Tear Off)** without calling it.

Then we use the variable & call the constructor. It behaves exactly like calling **Skeleton("HUMAN").** A new **Skeleton** object is created with type **HUMAN**.

We can also **Tear-Off - Method, Static Variable, and more.**

3.8 Null-Awareness Operators & Spread Operators In Dart

We already reach the end our **Advance Dart learning Section**. There are one more listen to go. Now, In this section we learn **Null-Awareness Operations.**

There are 4 type Null-Awareness Operators:

Null-Aware Operators	Name
??	IF - Null
?.	Null-Aware Access
??=	IF - Null Assignment
!	Null Assertion

3.8.1.1 IF – Null Operators (??)

Here, it returns the **Right-Hand Side** IF the **Left-Hand Side** is **Null.**

Example 1:

```
void main () {
    String? name;          → Create a Variable that is Null with Null – Operators.
    name = name ?? 'Sanjid Dad';
    → It Return the Right-Side Value IF the Variable is Null.
    print (name);
}
```

Here, name ?? 'Sanjid Dad' return **Sanjid Dad**. Because the name is **Empty.**

3.8.1.2. Null-Aware Access Operators (?.)

Here, It **Calls** a **Method or Accesses** a property only **IF** the **Object is Not Null.**

Example 1:

```
void main () {
    String? name;          → Create a Variable that is Null with Null – Operators.
    print (name?.length);          → Here It Print Null (Instead Of Throwing an Error)
}
```

Here name?.length print **Null**. Because the name is **Empty & it not Throwing an Error.**

3.8.1.3. IF - Null Assignment Operators (??=)

Here, It **Assigns** a value only **IF the Variable is Null.**

Example 1:

```
void main () {
    String? name;          → Create a Variable that is Null with Null – Operators.
    name ??= 'Sanjid Dad';
    → It Assign the Right-Side Value IF the Variable is Null.
    print (name);
}
```

Here name ??= 'Sanjid Dad**'** assign **Sanjid Dad** in the **name Variable.** Because the name is **Empty.**

3.8.1.4. Null Assertion Operators (!)

This Operators is Special. Because, it **Forced Dart Programming Language** to **Commit** that the **Variable is Null.** It doesn't matter **IF** the variable is not **Null.**

Example 1:

```
void main () {
    String? name = 'Sanjid Dad';
    → We Create a Null Variable. But, we assign the value.
    Print (name!);          → This Operators it Check Is the Variable Null.
}
```

Here, (!) Operators check IF the **Variable is Non-Null or Not. IF** the **Variable** is not null then work as it is. **But, IF** the **Variable is Null** then it **Print Errors.**

3.8.2. Spread Operator in Dart (..., ...?)

In Dart Programming, Spread is use to **insert variable** to **another variable.** Most case's it is use in **List's. Also,** we can do it manually by using the **List Methods.**

Here, we use 2 type Spread:

1. **. . . (**Spread**).**
2. **. . . ? (**Null Aware Spread**).**

Example 1:

List<int> numbers_list **= [1, 2, 3];**

List<int> numbers_list_two **= [0, . . . numbers_list, 4];**

→ Use Spread to add the numbers_list List.

Here, in the **number_list_two** the **List** will be **[0, 1, 2, 3, 4].** This is most use in the **Application Development.**

Example 2:

List<int>? numbers_list**;**

List<int> numbers_list_two **= [0, . . .? numbers_list, 4];**

→ Use Spread to add the numbers_list List.

Same, as **Example 1**. **Here,** in the **number_list_two** the **List** will be **[0, 4].** It doesn't give us **Error** because we add **Null Check Operators.** This is most use in the **Application Development.**

Dart Functional Programming

4.1 Anonymous Functions / Lambdas In Dart

An anonymous Function (also called a **Lambda or inline Function**) is a **Function without a name.**

You usually use them when you need a Short, Temporary Function, often as a callback or inside **Higher-Order Functions** like forEach, map, or where.

It is same what we learn in our **1st Function chapters.** Also, the **Higher-Order Function.**

4.2 Higher-Order Functions In Dart

A Higher-Order Function is a Function that does at least one of the following:

1. **Takes another Function as a parameter.**
2. **Returns a Function as its result.**

In Dart (and many modern languages), **Functions** are **First-Class Objects**, meaning you can:

1. **Store them in variables.**
2. **Pass them as arguments.**
3. **Return them From other Functions.**

4.2.1. Takes Another Function as a Parameter in Dart

Dart allow you to pass **Functions as a Function Parameters.** It allows you to access more control or strength in the Function.

Example 1:

```dart
void greet(String name) {
  print('Hello, ${name}');
}

void greet_morning(Function name, String user_name) {
  name(user_name);
}

void main() {
  greet_morning(greet, 'Sanjid Dad');
}
```

Here, in the example we First create a **Function name greet** that take a parameter called **name.** Then, we create a **Function greet_morning** that take **two parameters, one name Function type** then **user_name String type.**

In the main Function we call the **greet_morning Function** then pass **two parameters** one is **greet Function** & Second is **String**.

In the output in rint **Hello, Sanjid Dad.**

4.2.2. Returns a Function as its Result in Dart

Dart allow you to **return a Function in the Function.** Means we can create a **Function** that can **return** another **Function.**

Many times, in **Real Life** example we use that to return a **Function** instead returning a **Single or Multiple Values.**

Example 1:

```
Function multiplyBy(int numbers) {
  return (int value) => value * numbers;
}

void main() {
  var triple = multiplyBy(3);
  print(triple(4));
}
```

Here, is a Real-Life Example:

In Dart Programming **map, forEach, where** are the **Higher-Order Function** we use in **Real-Life.**

4.3 Closures In Dart

A Closure in Dart is a **Function** that **"remembers" the variables** From the **Scope** in which it was **created**, even after that **Scope Has Finished Executing.**

Closures capture variables From their **surrounding environment** and **Hold on to them.**

Example 1:

```
Function counter() {
  int count = 0;

  return () {
    count++;
    print("Count is $count");
  };
}

void main() {
  var increment = counter();

  increment();
  increment();
  increment();
}
```

Here, in this example we create **counter Function** that **returns an Anonymous Function** (**a Closure**). That **Function** remembers the value **Count** even after **Counter Has Finished.** Every time we call **increment ()**, it still **Has access to count.**

Property	**Description**
Captures variables	Remembers variables from the outer scope.
Keeps state	Can **Keep and Change** data even after outer **Function** ends.
Used **For Callbacks**	Very useful **For event Handling,** data processing, etc.

Example 2:

```
Function multiplyBy(int multiplier) {
  return (int value) => value * multiplier;
}

void main() {
  var doubleIt = multiplyBy(2);
  var tripleIt = multiplyBy(3);

  print(doubleIt(5));
  print(tripleIt(5));
}
```

Here, Each Closure remembers its own **multiplier** value**.**

4.4 Collection Methods In Dart Programming

To Finish this chapter or the **Advance Dart Programming** we must know all the **Collection Methods. Because,** in our **Real-Life** we use it a lot.

Here, we already learn all the **Collections (List, Set, Map).**

4.4.1. List<T> Methods In Dart

Here, all the Methods in **List<T>:**

1.	add (element)	**Adds a single element to the list**
2.	addAll (iterable)	**Adds multiple elements**
3.	insert (index, value)	**Inserts element at a given index**
4.	insertAll (index, iterable)	**Inserts multiple elements at index**
5.	remove (value)	**Removes the first occurrence of value**
6.	removeAt (index)	**Removes element at index**
7.	removeLast ()	**Removes the last element**
8.	removeWhere (test)	**Removes elements that satisfy a condition**
9.	retainWhere (test)	**Keeps only elements that satisfy a condition**
10.	clear ()	**Removes all elements**
11.	contains (element)	**Returns true if list contains the element**
12.	indexOf (element)	**Returns the index of the first match**
13.	lastIndexOf (element)	**Returns last index of the element**
14.	elementAt (index)	**Returns the element at a specific index**
15.	forEach (action)	**Loops through each item**
16.	map (f)	**Transforms each item and returns a new Iterable**
17.	where (test)	**Filters the list based on a condition**
18.	reduce (callback)	**Reduces list to single value (requires non-empty)**
19.	fold (initial, callback)	**Accumulates values starting from initial**
20.	any (test)	**Returns true if any item satisfies condition**
21.	every (test)	**Returns true if all items satisfy condition**
22.	expand (f)	**Expands each element to multiple elements**
23.	join ([separator])	**Joins list into a single string**
24.	sort ([compare])	**Sorts the list (in-place)**
25.	reversed	**Returns a reversed iterable**
26.	sublist (start, [end])	**Returns part of the list**
27.	asMap ()	**Converts list to map with index as key**
28.	shuffle ()	**Randomly shuffles elements**
29.	fillRange (start, end, [fill])	**Fills range with value**
30.	replaceRange (start, end, iterable)	**Replaces range with new elements**

4.4.2. Set<T> Methods In Dart

Here, all the Methods in **Set<T>:**

1.	add (value)	**Adds value if not present**
2.	addAll (values)	**Adds all values**
3.	remove (value)	**Removes value**
4.	removeWhere (test)	**Removes all elements satisfying condition**
5.	retainWhere (test)	**Keeps only elements satisfying condition**
6.	contains (value)	**Checks if value is in the set**
7.	clear ()	**Removes all elements**
8.	union (otherSet)	**Returns union of two sets**
9.	intersection (otherSet)	**Returns common elements**
10.	difference (otherSet)	**Returns items not in otherSet**
11.	forEach (action)	**Iterates over each element**
12.	lookup (value)	**Returns matching element if present**

4.4.3. Map<K, V> Methods In Dart

Here, all the Methods in **Map<K, V>:**

1.	addAll (other)	**Adds all key-value pairs from another map**
2.	remove (key)	**Removes a key and its value**
3.	removeWhere (test)	**Removes entries that match the test**
4.	clear ()	**Removes all entries**
5.	containsKey (key)	**Checks if a key exists**
6.	containsValue (value)	**Checks if a value exists**
7.	forEach (action)	**Loops through each key-value pair**
8.	putIfAbsent (key, ifAbsent)	**Adds key if absent**
9.	update (key, updateFn)	**Updates value if key exists**
10.	updateAll (updateFn)	**Updates all values**
11.	keys	**Returns iterable of all keys**
12.	values	**Returns iterable of all values**
13.	entries	**Returns iterable of MapEntry objects**
14.	map ((key, value) => ...)	**Transforms the map**

Dart Package & Library Management

5.1 What is Package & Library Management in Dart

Dart Has a Robust System For managing **Packages and Libraries,** allowing you to use **External Code** or **Organize** your own code **Efficiently.**

This is especially essential when working on larger Dart or **Flutter Projects.**

In the **Package & Library Management** we mainly use some sections. Like:

1. Using **pubspec. yaml (The Package Configuration File)**
2. Importing Libraries. **(Use import Keyword)**
3. **Creating and Publishing** your own **Packages.**

5.2 Package Configuration In Dart

Package Configuration in Dart refers to **How Dart projects manage** and **locate external packages**, **dependencies**, and **libraries —** controlled through a **Central File** and related tools.

In Dart we use a **pubspec. yaml** Central Configuration **File For** any **Dart & Flutter Projects.** It defines the project **metadata, dependencies, SDK constraints, assets,** and more.

5.2.1. Creating pubspec.yaml File

To create the **pubspec. yaml** Central Configuration File. We First create a **Project with a name.** Then create a **bin Folder** For the **main File** & the **pubspec. yaml** File. **Ex:**

My-Project/

|------- pubspec.yaml

|------- bin/

|------- main.dart

In the **pubspec.yaml** we right some Fields **(name, description, version, environment, dependencies, dev_dependencies)** that contain Projects **metadata.**

```yaml
name: Learning
description: A Command Line Application
version: 1.0.0

environment:
  sdk: '>=3.0.0 <4.0.0'

dependencies:
  http: ^1.1.0

dev_dependencies:
  test: ^1.24.0
```

Here, every Filed contains some specific data.

Field	Description
name	Name of your project/package
description	A Brief description
version	Version number (semver: major.minor.patch)
environment	Specifies Compatible Dart SDK Versions
dependencies	Packages needed for runtime **(Editable)**
dev_dependencies	Packages needed only development **(Editable)**

Here, In the **dependencies** and the **dev_ dependencies section** we can add as many we want in our projects. **For example,** we use **Http Package.**

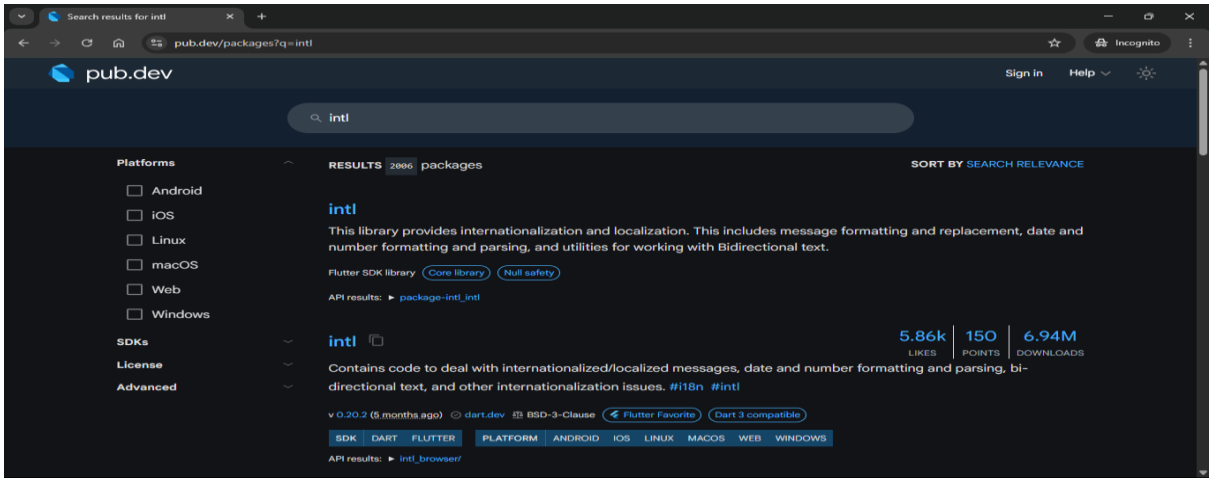
We can Find all **dependencies** in the **pub.dev (The Official Website For Dart Packages).**

5.2.2. Adding Dependencies in pubspec.yaml File

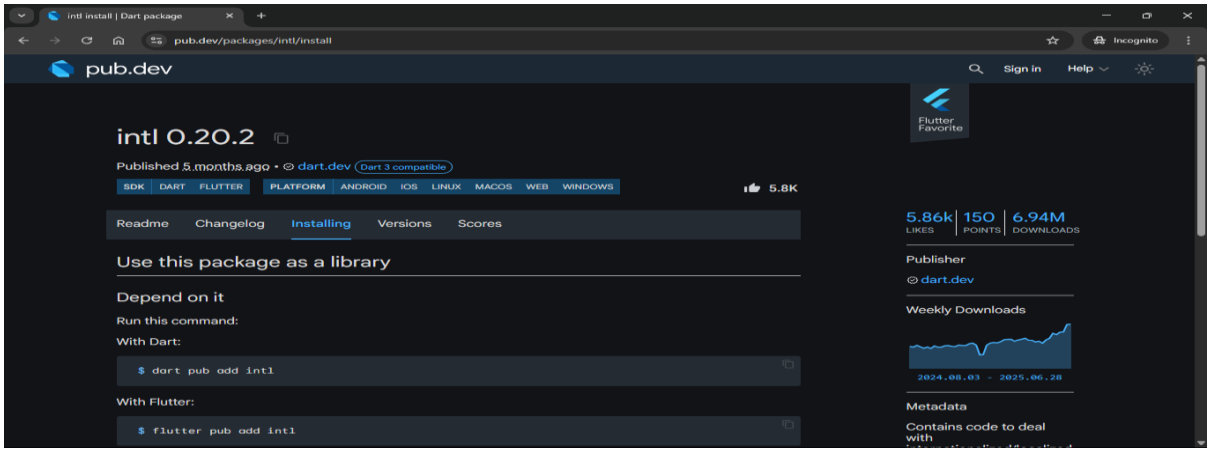
For adding the **dependencies** in the center configuration File we use **pub.dev** website. Here, in the same project what we create in previous example we adding a package name **intl.** This package is used For **internationalization.**

It Helps you Handle **date/time formatting, number formatting, currencies, pluralization, and messages** based on **locale**. **At First,**

Go to the pub.dev Website then search For intl package,



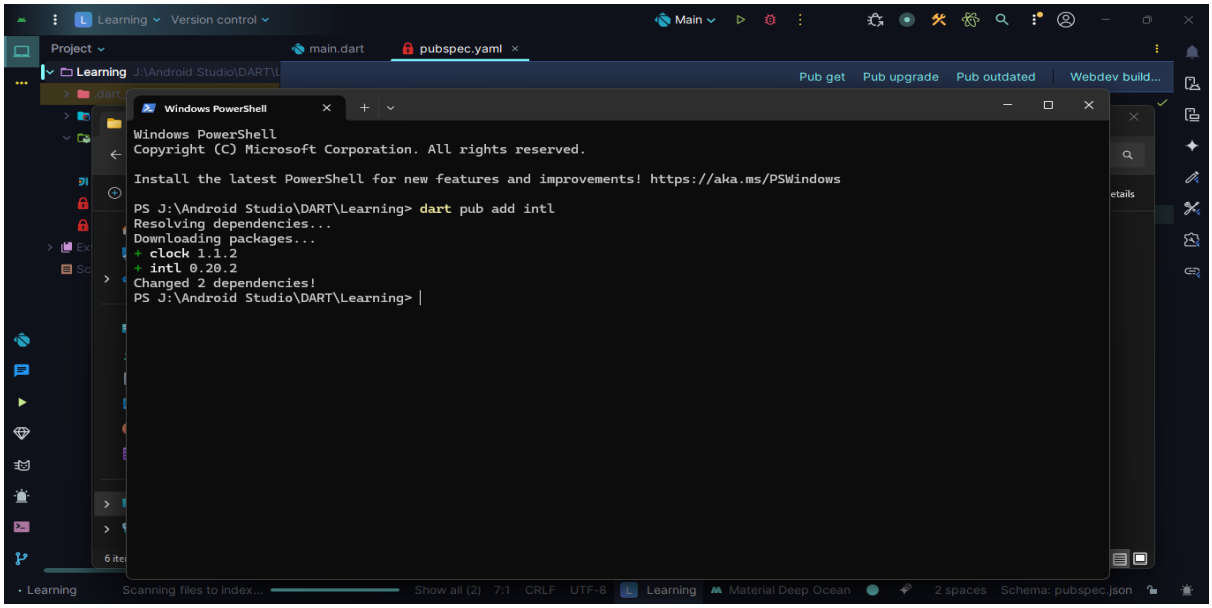
Now, we can see there is a **package name intl** with a **Likes | Points | Downloads.** Now then, go to the package and go to the Installing tab.



We can see, there is a **Command with Dart,**

$ dart pub add intl → **The Command For Download the intl Package.**

Copy the command and go to your project root Folder and open a terminal then paste the command and Enter.



By that we successfully added **intl Package** in out **Project.** IF we check our **pubspec File** we will see now there is **two** dependencies. **One is Http and Other one is intl.**

That's How we can add as many as package we want in our project.

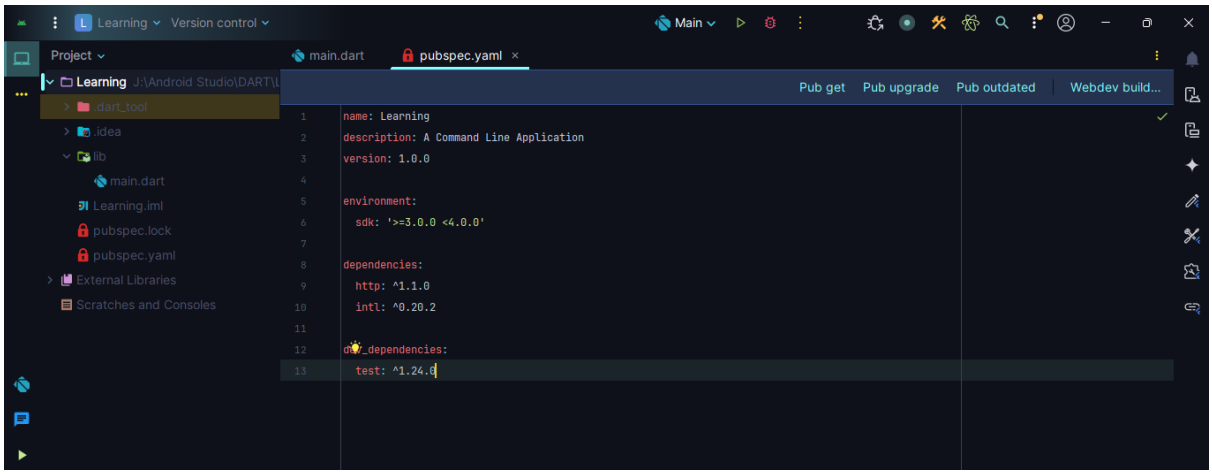
In the next section we will see How we **use that Package** to work in our **main.dart** File**.**

5.3 Importing Libraries or Packages In Dart

In Dart Programming Language, we can use another person created **Library or Packages** From the **pub.dev** Website. To do that we must **import** their code to our code.

In Dart Programming Language, we use three places **Library or Packages:**

1. **Built-in Dart Libraries.**
2. **External Packages (From pub.dev).**
3. **Your own Project Files.**



5.3.1. Importing Built-in Libraries

In Dart Programming, we use many **Built-in Libraries** like **math Libraries** For Mathematical Functions.

Now, math is a **Built-in Library** means this is created when the Language is created. So, IF we want to use that in our project we must **import** it to our code or **main File. To do that we say:**

import 'dart:math'**;** → **It Importing the math Libraries All File.**

Now we can use the math Libraries Function in our code.

Example 1:

```dart
import 'dart:math';

void main() {
  print(sqrt(64));
}
```

Here, in this example we **First import the math Library** then in the **main Function** we use a **math Library Function** name **sqrt(number).**

Here, the **sqrt(number)** take one parameters type integer and in the output, it returns the **Square Root** Of the **Numbers** as **double data type.**

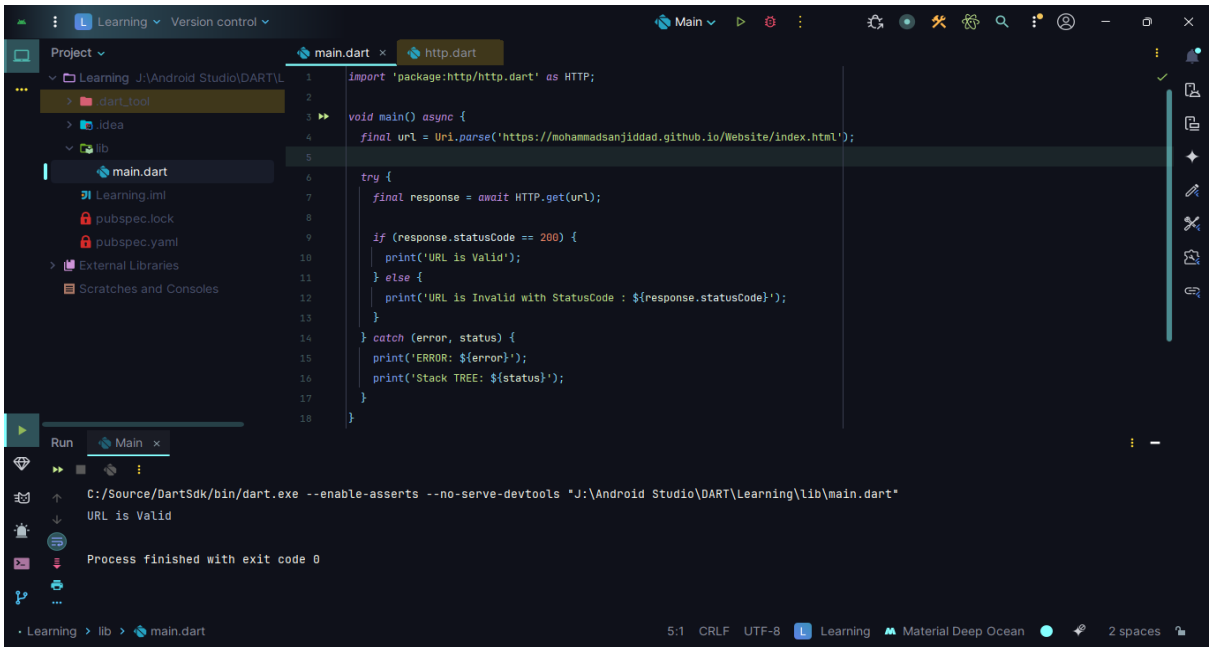
So, this example prints **8.0.**

5.3.2. Importing External Packages (From pub.dev)

We already see How to import a Library or Package. In our previous chapters we see we use **pubspec.yaml** File to download the External Packages For pub.dev websites.

We work with the same project we create in our previous chapters.

Example 1:



Here, in this example we **First import** the **Package** and say it should work as the name **HTTP** by typing **as HTTP.** Now, in the **main Function** we create a **Final Variable name url**. It checks that the website link is valid or not and **returns** some **response.**

In the next section we create another variable name **response** that await some time and give us the response credential then we check a condition IF **(statusCode == 200)** then print **"URL is Valid"**. Otherwise print **"URL is Invalid".** Also, we do this in the **try and catch** Block For Healing Errors.

5.3.3. Create Our Own Custom Package or Library

In Dart Programming, we Have many ways to create a **Custom Library or Package**. In the earlier chapter we see How to create a project**.**

Package or Project create it by same way. There are not very much different.

We can create project by manually in the IDE or use this command,

dart create **Application_name** → **Create Dart Project**

As For the **Library or Package** we can we can do the same as create by manually in the IDE or we can use this command,

dart create -t package **Application_name** → **Create Dart Package**

First go to the location where we want to create the project then we open the terminal in that directory & pest the command.

Now, when we create the package by the command. Dart by default generates some **Folders**

Or some Code. The Project looks like,

My-Project/

|------- pubspec.yaml → **Package metadata.**

|------- lib/

|------- My-Project.dart → **Main Entry Point For Package.**

|------- src/

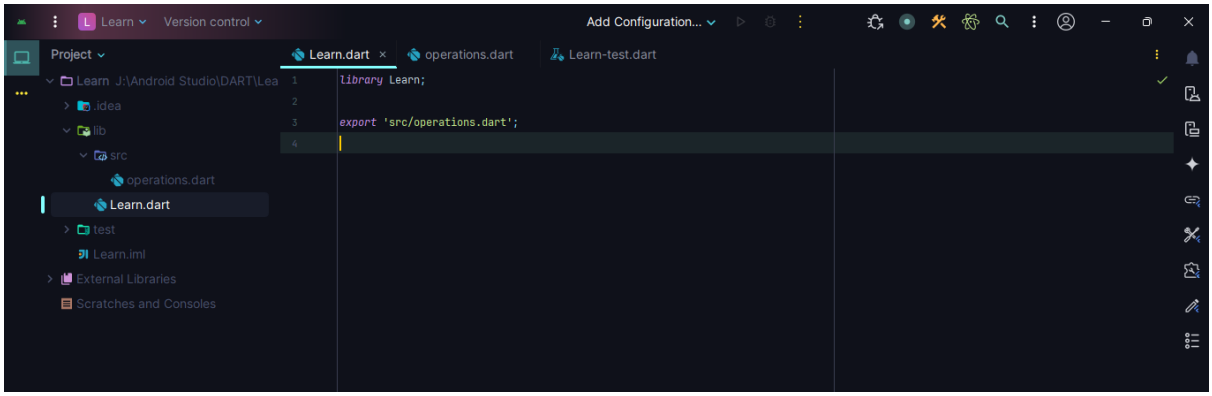
|------- operations.dart → **All Logic Or Code / Internal Logic**

|------- test/

|------- My-Project-test.dart → **Dart Testing File to test All Code**

Now, when we create it manually we Have to create all the Folder or the File one by one.

Now, in the entry point **My-Project.dart File** we right some code it is same in every Projects,



Here, in this example we create a project manually. We name this project **Learn**. Now, in the **Lean** project we create **lib, src, test Folder** & inside the **lib Folder** we create **src** and the **Learn.dart File.** Now, in the **Learn.dart** File we say that is a **Library** and we **export** the **operations.dart File**.

> ***In Dart,*** *the export keyword is used to* ***Re-Export a Library File*** *so that other* ***Files Importing*** *your* ***Library*** *gain access to everything inside the* ***exported File.***
> *Without needing to import it directly.*
> *We export package's main library File (Like* ***lib/Learn.dart****) to control what parts your package is* ***exposed publicly.***

Now, we done editing our **Entry Point File.** It's time to edit our **operations.dart File** where we right our all Code or Logic.

Now, in the **operations.dart File** we create **two Functions.** One is **greet (name)** that return a **String.** Other one, **multiply (number_one, number_two)** that return a **double.**

Now, We come close to Finish. There is one File left that is **test File.** We must test all our **Functions or Logics** after we use it in another projects.

Although we don't learn the **Dart Test** yet. We learn this in our next upcoming chapters. As For now we can leave it as it is.

So, we complete our **Custom Library Creation Chapter**.

Dart Testing & Debugging

6.1 What Is Dart Testing

Dart testing is the process of writing and running automated tests to check that your Dart code works as expected. It helps catch **bugs early, verify logic, and ensure your app remains stable** when you make **Changes.**

Why Testing Is Important

- **Catch Bugs Early.**
- **Ensure Code Behaves Correctly Over Time.**
- **Improve Code Quality and Confidence.**
- **Support Refactoring and Future Updates.**

Dart uses the **test package** — **Official testing Framework** — For Writing and Running dart testing.

In Dart Programming we use 3 type testing,

1. **Unit Test. - Test Individual Functions/Methods.**
2. **Widget Test** *(Flutter).* **- Test UI Components.**
3. **Integration Test** *(Flutter).* **- Test Full user Flows.**

In pure Dart Programming (non-Flutter**),** we mostly write **unit tests.**

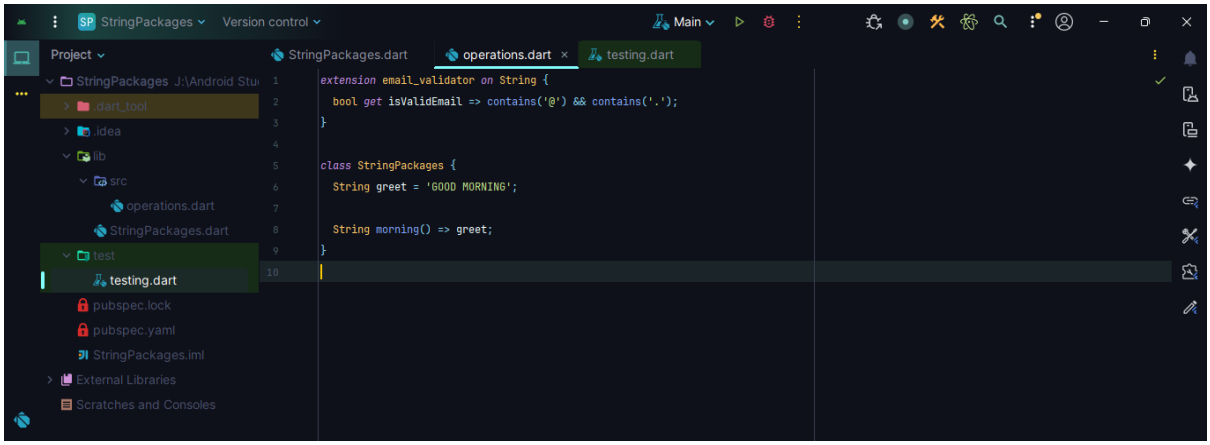
In Dart Programming, we use testing in the Packages or Libraries to insurer there is no error. So, Here we create a Package then test the **Functionality** by using **unit test.**

6.2 Unit Testing In Dart

Unit tests are tests that check small pieces of code — usually individual **Functions, Methods, or Classes** — to make sure they work correctly.

First, we create our Library or packages. Here, we created named **StringPackages.** In the String package in the **lib Folder** we create a **src Folder** in the **src Folder** we create a **Dart File** name **operations.**

then right some code,



Here, we create an **extension on String** to **Check** that the given **String** is **Email** or not.

6.2.1. What Is test (), expect (a, m) Function In Dart Testing

To test our Functions one by one we use the **test () Function.** Inside the **test Function use expect () Function.**

Example 1:

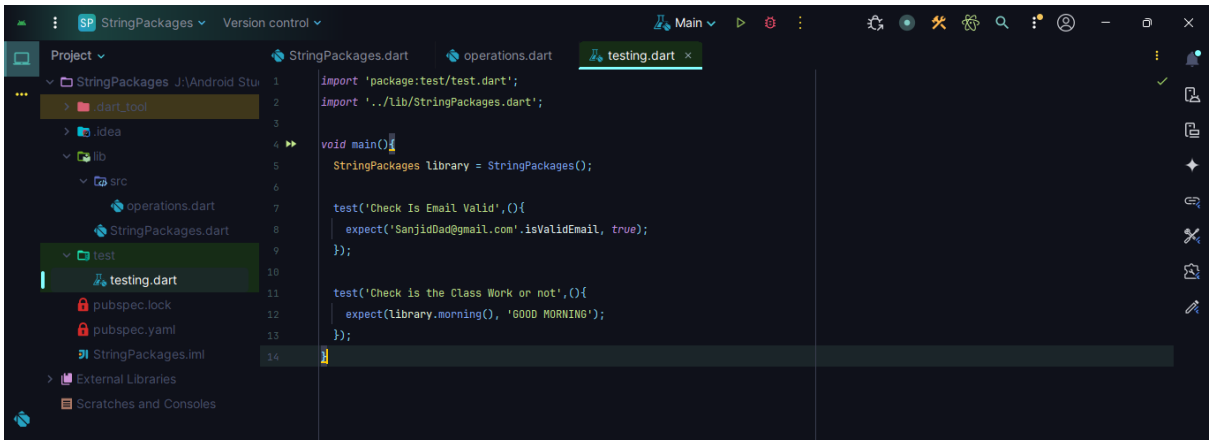
```
import 'package:test/test.dart';                    → Import the test Packages.
import '../lib/StringPackages.dart';                → Import our main Project File.
void main () {
    test ('Check Is Email Valid', () {
        expect ('SanjidDad@gmail.com'. isValidEmail, true);
    });
}
```

Here, in this example we create a **main Function** then we use **test () Function.** Here, the **test Function** take **2 parameters,** one is **discerptions** and the second one is a **Function** then it tests the **Function** is valid or not.

Inside the **test Function** we use **expect Function** that also take **2 parameters,** one is the **Functions or Logic** and the second one is the **result that Functions or Logic.**

In this way we can test every Functions or Logic or Classes one by one.

Example 2:



Here, we test the Class that contain **Method** name **morning ().**

6.2.2. What Is group (), setUp (), tearDown () Function In Dart Testing

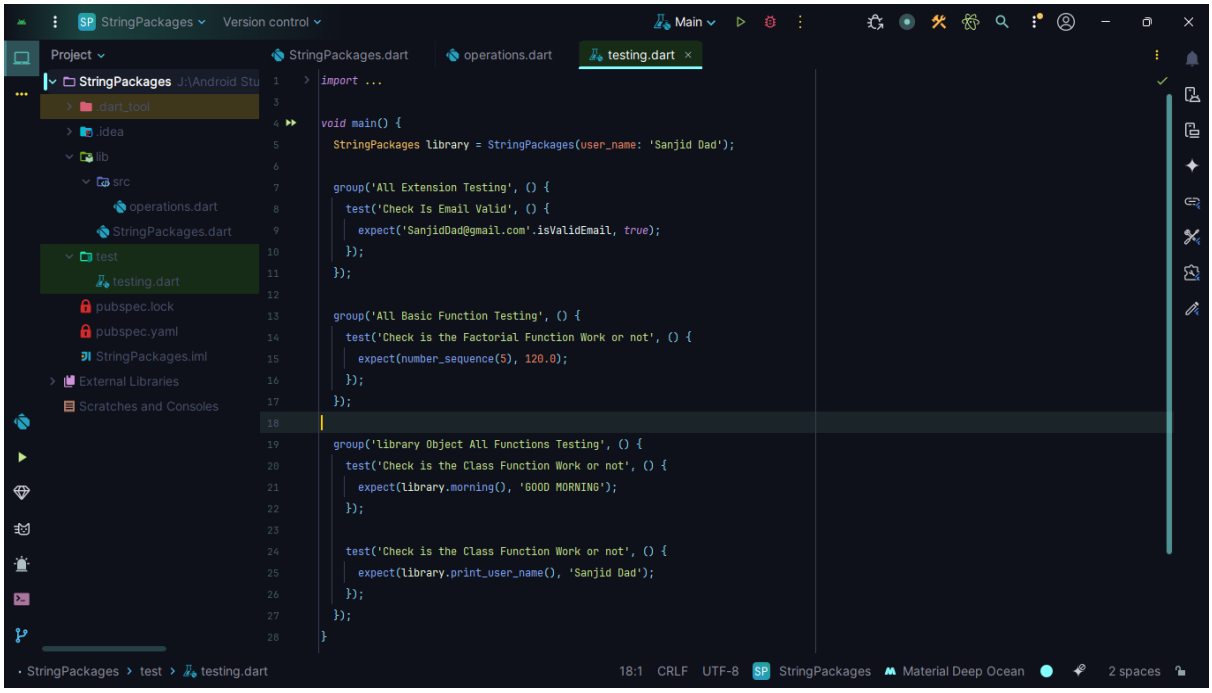
Now, in this point we already use **dart unit test**. In the earlier **Chapter** we see we can **test Functions or Classes Methods also.** So, those are two different think one is **under the Class** one **not**.

Here, we can group those one by one. **Example:**

1. **One For Normal Functions.**
2. **One For extension.**
3. **One For Classes Methods.**

So, to do that we use **group Function** so we can easily track them.

Example 1:



In this example we see almost all the testing Functions. We use two more Functions but those Function are not use like that. Those Functions are optional. Example:

1. **setUp ()** Function.
2. **tearDown ()** Function.

In Dart's testing Framework (using the **test** package), the **setUp () Function** is used to run setup code before each test case in a test suite.

It Runs before every individual **test ()** in a group or **File,** Helps initialize shared Objects, Variables, or State Keeps your test Code Clean, Organized, and DRY (Don't Repeat Yourself).

The tearDown () Function is same as setUp () Function, one difference it run at the last.

6.3 Debugging In Dart

Debugging is the process of **Finding and Fixing** errors or bugs in your Dart code. Dart provides multiple tools and **techniques For debugging** both Dart and Flutter apps.

Some Common way to debug In Dart

Method	Description
print ()	Simple, fast way to see values
assert ()	Validates assumptions during dev
IDE Debugging Tools	Visual debugging with Breakpoints
DevTools	Advanced profiling (mostly for Flutter)

6.3.1. Print () Debugging (Quick & Easy)

Print is very good when we want to check something quickly. It is used to output small answers also it prints output only in the console or terminals.

Example 1:

```
void main () {
    int a = 5; int b = 3;
    int sum = a + b;
    print ("Sum is: $sum");          → Print the sum in the terminal.
}
```

6.3.2. Assert () (Only in Debug Mode)

Assert Function only used in the **debugging mode.** Its main purpose is to **Check Condition.** Ensure **Conditions Hold true** during development. **IF the assertion Fails,** Dart throws an **error** and the error will be the else **Block Code.**

Example 1:

```
void login (String? username) {
    assert (username! = null, 'Username cannot be null');
    print ("Welcome, $username");
}
```

6.3.3. Using dart devtools (Advanced)

In dart programming we usually don't use **dart devtools.** We use it mostly in **Flutter** and **web applications.**

So, we don't use it Here,

Features in devtools,

1. **Timeline performance.**
2. **Memory usage.**
3. **Network tracking.**
4. **Widget inspector (Flutter).**

Summary Table

Tool	Best For
Print ()	Quick checks in console apps
Assert ()	Validating logic in development
Breakpoints	Inspecting app state line-by-line
DevTools	Flutter debugging & profiling

THANK YOU 💕 💕