1. **Introduction of Dart Programming language.**

Dart is a modern, high-performance, object-oriented programming language developed by Google. It is designed for building fast, scalable, and highly performant applications across multiple platforms, including mobile, web, desktop, and backend systems. Dart is the language behind the Flutter framework, which powers cross-platform applications with a single codebase.

1. **Key Features**

* **Object Oriented Language**

1. Dart is based on the concept of objects and classes.
2. This means everything in Dart is an object, and you use classes to organize your code into reusable and logical parts.
3. It’s great for building structured, maintainable, and scalable applications.

**Strong and Sound Type**

1. every variable must have a specific type (e.g., int, String, bool).
2. **Strong typing** ensures you follow rules about how data is used, which prevents many bugs.
3. **Sound type** means the Dart compiler checks your code during development to ensure it's correct, making it safer and more reliable.

* **Cross-Platform**

1. Dart allows you to write code once and run it on multiple platforms, such as Android, iOS, web, desktop, and even embedded devices.
2. This saves time and effort compared to writing separate code for each platform.

* **Null Safety**

1. Dart has a built-in system to prevent null-related errors, which are common causes of crashes.
2. Variables in Dart cannot be null unless you explicitly allow them to be.
3. This feature makes your code more robust and reduces runtime errors. Dart has a built-in system to prevent null-related errors, which are common causes of crashes.
4. Variables in Dart cannot be null unless you explicitly allow them to be.
5. This feature makes your code more robust and reduces runtime errors.

**Fast Performance**

1. Dart is designed to be fast.
2. Its compiler produces optimized code for quick app launch and smooth performance.
3. This is especially important for mobile apps where responsiveness matters.

**AOT( Ahead of time) & JIT( Just in Time ) Compilation**

1. Dart uses AOT compilation to convert your code into machine code before the app is run. This makes apps launch faster and perform better.
2. It also supports JIT compilation, which compiles your code while the app is running. JIT is useful during development as it allows you to test changes instantly (hot reload).

**Asynchronous Programming.**

1. Dart makes it easy to handle tasks that take time, like fetching data from the internet or reading a file.
2. It uses features like async and await to let your app keep running while waiting for these tasks to finish.
3. This ensures the app doesn’t freeze or become unresponsive.
4. **Where Dart is Used?**

1. Mobile App Development (with Flutter)

Dart powers the Flutter framework, enabling developers to build cross-platform mobile applications.

Popular apps like Google Pay, Reflectly, and eBay Motors are built with Dart and Flutter.

2. Web Development

Dart can be compiled to JavaScript, making it suitable for building high-performance web apps.

3. Desktop Applications

Using Flutter, Dart can build native desktop applications for Windows, macOS, and Linux.

4. Backend Development

Dart can be used as a backend language, e.g., with the Shelf package for building web servers or APIs.

1. **Advantages of Dart**
2. Unified Development: Single language for UI, backend, and web development.
3. Hot Reload: Fast iteration during development, especially with Flutter.
4. Null Safety: Minimizes runtime errors caused by null references.
5. Great Tooling: Integrated tools for debugging, testing, and profiling.
6. Easy Learning Curve: Familiar syntax similar to JavaScript, Java, or C#.
7. **Understanding Programming Fundamentals**
8. **Data Types In Dart**

In Dart, data types are used to define the type of data a variable can hold. Dart is a statically typed language, which means every variable has a type that is checked at compile-time. However, with features like var and dynamic, Dart provides flexibility in declaring variables.

**Types of Data Types in Dart.**

1. **Numbers**

Used for numeric values:

- int: For integers ( whole numbers).

- double: For decimal or floating-point numbers.

**Example:**

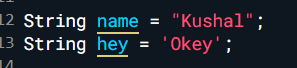
****

1. **Strings**

Used for text or sequences of characters. Strings are enclosed in single (‘) or

double quote(“).

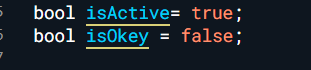
**Example:**



1. **Boolean**

Represents **true** or **false** values, used for logical operations.

**Example:**

****

**Note : There are many more data types in which we will study under the collections in dart.**

1. **Variables**

In Dart, variables are used to store and manage data in memory. A variable is essentially a named container that holds a value, which can be updated or retrieved during the program's execution.

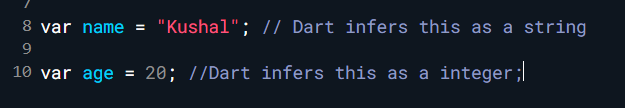
**Declaring Variables in Dart.**

1. **Using var**

 Automatically infers the type of the variable based on the assigned value.

 The variable’s type cannot change after type is inferred.

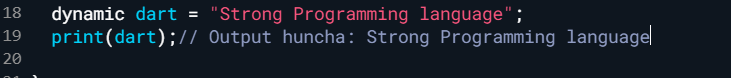
**Example:**



1. **Using dynamic**

The dynamic keyword allows a variable to hold values of any type and its type can change at runtime. Use dynamic only when necessary, as it skips compile-time type checking.

**Example:**

****

1. **Using final and const**

**Final :** A variable whose value can only be set once but is initialized at runtime.

**Const :** A variable that is a compile-time constant.

**Example:**



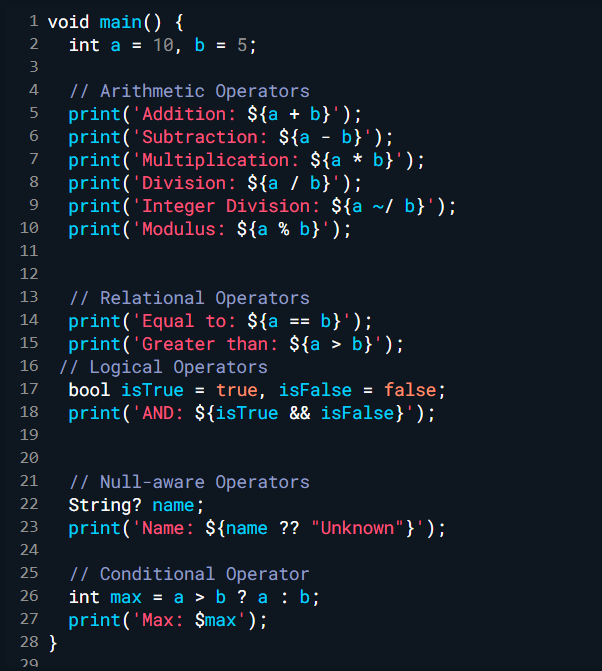
**C. Operators in Dart.**

                Operators in Dart are symbols or keywords used to perform operations on values and variables. They allow you to manipulate data, perform arithmetic, compare values, and control the flow of a program.

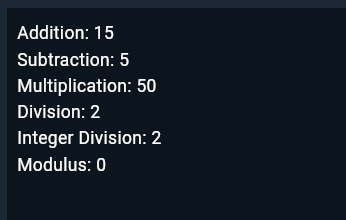
**Types of operators in Dart.**

* + 1. **Arithmetic Operators**
* Addition    “  + ”
* Subtraction    “ - “
* Multiplication “ -\*“
* Division“ /“
* Integer Division“ ~/ “
* Modulus ( Remainder ) “ %“

**Example :**



**Output:**

****

**ii. Relational (Comparison) Operators**

* Equal to   “  == ”
* Not Equal to    “ != “
* Greater Than “ >“
* Less than“ < “
* Greater than or equal to “ >=“
* Less than or equal to “ <=“

**Example :**



**iii. Logical Operators**

- Logical AND “  &&  ”

* Logical OR “  ||  ”
* Logical NOT “  !  ”

**Example :**

**iv. Assignment Operators**

* Assign “  =  ”
* Add and Assign “  +=  ”
* Subtract and assign “ -= ”
* Multiply and assign “  \*=  ”
* Divide and Assign “  /=   ”
* Modulus and Assign “  %=  ”

**Example :**

**v. Bitwise Operators**

* Bitwise AND “  &  ”
* Bitwise OR“  `  ”
* Bitwise XOR “ ^ ”
* Bitwise NOT“  ~  ”
* Left Shift “  <<   ”
* Right Shift “ >>  ”

**Example :**

**vi. Conditial (Ternary) Operators**

* condition ? Exp1 : Exp2

                     If condition is true, evaluate expr1; otherwise, evaluate expr2.

**Example:**

**vii.  Null aware Operators**

 -“  ??  ” (If the left- hand operand is null, return the right -hand operand)

* “  ??=  ” ( Assign value only if the variable is null)
* “ ?.  ” ( Calls a method or accesses a property if the object is not null.)

**Example :**

**viii.  Type Test Operators**

* “  is  “ ( Checks if an object is of a specific type ).
* “  Is! “ ( Checks if an object is not of a specific type ).
* “  as. “ (Typecast operator)

**Example :**

**xi.  Cascade Operator ( . . )**

- Allowing Chaining operators on the same object.

The **cascade operator (..)** in Dart is a powerful feature that allows you to chain multiple operations on the same object without needing to refer to it repeatedly. It makes your code cleaner and easier to read, especially when initializing or modifying objects.

**Example :**

class Person {

String? name;

int? age;

void setName(String name) {

this.name = name;

}

void setAge(int age) {

this.age = age;

}

void showDetails() {

print('Name: $name, Age: $age');

}

}

void main() {

var person = Person()

..setName('Alice')

..setAge(25)

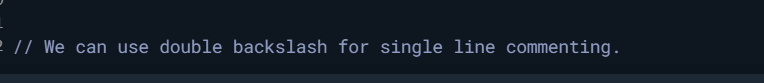
..showDetails(); // Output: Name: Alice, Age: 25

}

**Comments in dart.**

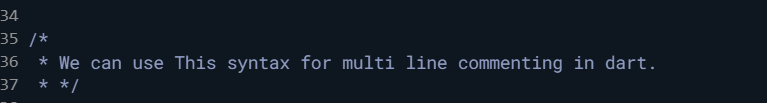
**Single Line Comment.**

Example:



**Multi Line Comment.**

Example:



**Control Flow In Dart Programming.**

**1. Control Flow Statements**

Control flow statements help you decide the flow of your program based on conditions or repeating actions.

**1.1 if, else if, else**

Used to make decisions based on conditions.

Example:

void main() {

int age = 21;

if (age < 18) {

print('Kushal is a teenager.');

} else if (age >= 18 && age < 30) {

print('Kushal is a young adult.');

} else {

print('Kushal is an adult.');

}

}

Explanation:

* If age is less than 18, it prints one thing.
* If it’s between 18 and 30, it prints something else.
* Otherwise, it prints the last option.

**1.2 switch**

Used when you have multiple possible values to check against.

Example:

void main() {

String favoriteColor = 'blue';

switch (favoriteColor) {

case 'red':

print('Kushal loves red!');

break;

case 'blue':

print('Kushal loves blue!');

break;

case 'green':

print('Kushal loves green!');

break;

default:

print('Kushal has a unique taste in colors!');

}

}

Explanation:

* The switch checks the value of favoriteColor.
* Depending on the value, it runs the matching case.
* The default case is like an "else" if no matches are found.

**1.3 Loops**

Loops help you repeat actions multiple times.

**For Loop Example**

void main() {

for (int i = 1; i <= 5; i++) {

print(I am is learning Dart: Day $i');

}

}

Explanation:

* Starts from i = 1 and repeats until i <= 5.
* Prints a message for each day.

**While Loop Example**

void main() {

int counter = 1;

while (counter <= 3) {

print('Hello, Kushal! This is message $counter');

counter++;

}

}

Explanation:

* The loop runs while the condition (counter <= 3) is true.
* After each loop, counter increases by 1.

**2. Functions**

Functions are reusable blocks of code. They help keep your program organized.

**2.1 Defining a Simple Function**

void greetKushal() {

print('Hello, Kushal! Welcome to Dart programming.');

}

void main() {

greetKushal(); // Calling the function

}

Explanation:

* greetKushal is a function that prints a message.
* You can call it whenever you want to reuse the message.

**2.2 Functions with Parameters**

You can pass data into a function using parameters.

Example:

void greetUser(String name, int age) {

print('Hello, $name! You are $age years old.');

}

void main() {

greetUser('Kushal', 25); // Passing your name and age

}

Explanation:

* greetUser takes two inputs: name and age.
* When you call it, you pass values for those inputs.

**2.3 Functions with Return Values**

Functions can return a value for later use.

Example:

int addNumbers(int a, int b) {

return a + b;

}

void main() {

int sum = addNumbers(10, 20);

print('Kushal, the sum is: $sum');

}

Explanation:

* addNumbers adds two numbers and returns the result.
* The result is stored in sum and printed.

**2.4 Arrow Function (Lambda Expression)**

Short way of writing simple functions.

Example:

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

void main() {

int result = multiply(4, 5);

print('Kushal, the product is: $result');

}

Explanation:

* The => is used for single-line functions.
* It returns the result without needing the return keyword.

**Summary**

1. **Control Flow** helps your program decide what to do (e.g., if, switch) or repeat tasks (for, while).
2. **Functions** organize your code and let you reuse logic, with options to accept inputs (parameters) and return outputs.

**Object Oriented Programming In Dart.**

### ****1. Classes and Objects****

**Classes** are blueprints for creating objects, and **objects** are instances of those classes.  
Think of a **class** as a template (like a recipe) and an **object** as the actual item created (like the food made using the recipe).

#### Example:

class Person {

String name = ''; // Property

int age = 0;

void introduce() { // Method

print('Hi, I am $name and I am $age years old.');

}

}

void main() {

Person kushal = Person(); // Creating an object

kushal.name = 'Kushal'; // Setting property

kushal.age = 21;

kushal.introduce(); // Calling the method

}

**Explanation:**

* Person is a class with properties (name, age) and a method (introduce).
* The object kushal is created from the class, and we set its properties and call its method.

### ****2. Constructors****

A **constructor** is a special method that initializes the properties of a class when an object is created.

#### Example:

class Person {

String name;

int age;

// Constructor

Person(this.name, this.age);

void introduce() {

print('Hi, I am $name and I am $age years old.');

}

}

void main() {

Person kushal = Person('Kushal', 21); // Using constructor

kushal.introduce();

}

**Explanation:**

* The constructor Person(this.name, this.age) automatically sets the name and age when creating the object.
* No need to set properties manually!

### ****3. Inheritance****

Inheritance allows one class to inherit properties and methods from another class. This helps reuse code.

#### Example:

class Person {

String name;

Person(this.name);

void introduce() {

print('Hi, I am $name.');

}

}

// Kushal inherits from Person

class Programmer extends Person {

String programming language;

Programmer(String name, this.programmingLanguage) : super(name);// super(name calls the parent class constructor)

void code() {

print('$name loves coding in $programmingLanguage.');

}

}

void main() {

Programmer kushal = Programmer('Kushal', 'Dart');

kushal.introduce(); // From the Person class

kushal.code(); // From the Programmer class

}

**Explanation:**

* Programmer inherits the introduce method from Person.
* It also has its own code method.

### Why is : super(name) needed?

The Programmer class **inherits** from the Person class. This means the Person class is responsible for managing the name property.  
To properly initialize the name property (which belongs to Person), the Programmer class must call the parent class constructor using the super keyword.

### ****4. Polymorphism****

**Polymorphism** means many forms. A subclass can override methods of the parent class to provide specific behavior.

#### Example:

class Person {

String name;

Person(this.name);

void work() {

print('$name is working.');

}

}

class Programmer extends Person {

Programmer(String name) : super(name);

// Overriding the work method

@override

void work() {

print('$name is coding in Dart.');

}

}

void main() {

Person kushal = Programmer('Kushal');

kushal.work(); // Calls the overridden method

}

**Explanation:**

* The Programmer class overrides the work method to give it specific behavior.
* When called, the subclass method is executed.

Note: Key point: The object is of type Person (parent class) but refers to a Programmer (child class).

This is allowed because a Programmer is a Person (inheritance).

This type of behavior is called upcasting.

### ****5. Abstract Classes****

An **abstract class** is like a blueprint. You can’t create objects from it directly, but other classes can inherit from it and provide their own implementation for its methods.

#### Example:

abstract class Animal {

void sound(); // Abstract method

}

class Dog extends Animal {

@override

void sound() {

print('The dog barks.');

}

}

void main() {

Dog myDog = Dog();

myDog.sound();

}

**Explanation:**

* The Animal class has an abstract method sound, which must be implemented by its subclasses.
* You can’t create an Animal object directly.

### ****6. Interfaces****

An **interface** is a way to define behavior that multiple classes can implement.

### Real-World Use Case for Interfaces: ****Device Communication System****

Interfaces are ideal when multiple unrelated classes need to share a common behavior, without enforcing a hierarchy. For example, in a **Device Communication System**, various devices like a Printer, Scanner, and Fax Machine might need to share a common behavior of "connectivity" but implement it in their unique ways.

#### Example:

#### // Interface using an abstract class

#### abstract class Connectable {

#### void connect(); // Abstract method to be implemented

#### }

#### // Printer class implements Connectable

#### class Printer implements Connectable {

#### @override

#### void connect() {

#### print('Printer connected via USB.');

#### }

#### }

#### // Scanner class implements Connectable

#### class Scanner implements Connectable {

#### @override

#### void connect() {

#### print('Scanner connected via Wi-Fi.');

#### }

#### }

#### // Fax Machine class implements Connectable

#### class FaxMachine implements Connectable {

#### @override

#### void connect() {

#### print('Fax Machine connected via Ethernet.');

#### }

#### }

#### void main() {

#### // Create a list of Connectable devices

#### List<Connectable> devices = [

#### Printer(),

#### Scanner(),

#### FaxMachine(),

#### ];

#### // Iterate over devices and call the connect method

#### for (Connectable device in devices) {

#### device.connect(); // Polymorphism in action

#### }

#### }

#### Another Example:

class Reader {

void read() {

print('Kushal is reading.');

}

}

class Writer {

void write() {

print('Kushal is writing.');

}

}

class Author implements Reader, Writer {

@override

void read() {

print('Author Kushal reads a lot.');

}

@override

void write() {

print('Author Kushal writes books.');

}

}

void main() {

Author kushal = Author();

kushal.read();

kushal.write();

}

**Explanation:**

* Author implements the behavior of both Reader and Writer.
* This is useful for sharing functionality between classes.

### ****7. Mixins****

**Mixins** allow you to add functionality to classes without inheritance.

#### Example:

mixin Swimmer {

void swim() {

print('Kushal can swim!');

}

}

mixin Runner {

void run() {

print('Kushal can run!');

}

}

class Athlete with Swimmer, Runner {}

void main() {

Athlete kushal = Athlete();

kushal.swim();

kushal.run();

}

**Explanation:**

* The Athlete class gets functionality from both Swimmer and Runner mixins.

### ****8. Asynchronous Programming (****Futures****,**** async****,**** await****)****

When you need to perform tasks that take time (like downloading data), Dart handles them asynchronously so the app doesn’t freeze.

#### Example:

Future<String> fetchData() async {

print('Fetching data for Kushal...');

await Future.delayed(Duration(seconds: 2)); // Simulates delay

return 'Data fetched for Kushal!';

}

void main() async {

print('Start');

String result = await fetchData();

print(result);

print('End');

}

**Explanation:**

* fetchData simulates a time-consuming task using Future.delayed.
* The await keyword pauses the code until the Future completes.
* The app remains responsive during the wait.

### Summary

* **Classes and Objects:** Blueprints and their instances.
* **Constructors:** Initialize class properties when creating objects.
* **Inheritance:** Share and reuse properties/methods.
* **Polymorphism:** Override parent methods for specific behavior.
* **Abstract Classes:** Define blueprints for subclasses to implement.
* **Interfaces:** Define shared behavior across classes.
* **Mixins:** Add functionality to classes without inheritance.
* **Asynchronous Programming:** Handle time-consuming tasks without freezing the app.

**Error Handling in Dart**

In Dart, **error handling** is the process of managing unexpected situations (exceptions) that may arise during code execution. Dart provides a robust(Strong and Healthy) mechanism for handling these exceptions to ensure your application can gracefully recover or provide helpful feedback.

**1. Exceptions and Try-Catch Blocks**

An **exception** is an event that disrupts the normal flow of your program. Examples include trying to divide by zero, accessing a file that doesn’t exist, or making an invalid API request.

Dart provides the try-catch block to handle exceptions.

**Example: Using Try-Catch Blocks**

void main() {

try {

int result = 10 ~/ 0; // Division by zero throws an exception

print('Result: $result');

} catch (e) {

print('An error occurred: $e');

} finally {

print('Execution completed.');

}

}

**Explanation of the Code**

1. **Try Block:**
   * Code that might throw an exception is wrapped inside the try block.
   * In this case, dividing by zero (10 ~/ 0) causes an exception.
2. **Catch Block:**
   * The catch block catches the exception and lets you handle it.
   * The exception object (e) contains details about the error.
3. **Finally Block (Optional):**
   * The finally block contains code that runs **regardless of whether an exception occurred or not**.
   * Use this for cleanup operations like closing files or network connections.

**Example with More Details: Handling Specific Exceptions**

void main() {

try {

int result = 10 ~/ 0; // Division by zero

print('Result: $result');

} on IntegerDivisionByZeroException {

print('Cannot divide by zero!');

} catch (e) {

print('An unexpected error occurred: $e');

} finally {

print('Program finished.');

}

}

**Key Points:**

* Use on to catch specific exceptions (e.g., IntegerDivisionByZeroException).
* Use catch to handle any exception.

**2. Custom Exception Handling**

Sometimes, you need to create your **own exception types** to handle specific errors in your application.

**Example: Creating and Using Custom Exceptions**

// Define a custom exception

class InvalidInputException implements Exception {

String cause;

InvalidInputException(this.cause);

@override

String toString() => 'InvalidInputException: $cause';

}

void validateName(String name) {

if (name.isEmpty) {

throw InvalidInputException('Name cannot be empty!');

} else {

print('Welcome, $name!');

}

}

void main() {

try {

validateName(''); // Empty name triggers custom exception

} catch (e) {

print('Error: $e');

} finally {

print('Validation process completed.');

}

}

**Explanation of the Code**

1. **Custom Exception Class:**
   * The InvalidInputException class extends Dart's Exception.
   * It has a cause field to store the reason for the error.
   * The toString method is overridden to provide a readable error message.
2. **Throwing the Exception:**
   * In the validateName function, an empty name triggers the custom exception using the throw keyword.
3. **Handling the Exception:**
   * In main, the custom exception is caught in the catch block and displayed to the user.

**Real-World Scenarios for Custom Exceptions**

1. **Validation Errors:**
   * Ensuring user input (e.g., username, email) meets specific criteria.
2. **Business Logic:**
   * Handling application-specific errors like "Insufficient Funds" in a payment system.
3. **Data Integrity:**
   * Throwing an exception when a database operation fails.

**Key Points to Remember**

* **Use try-catch blocks** for handling exceptions in a structured way.
* **Use specific exceptions (on)** to handle known error cases.
* **Create custom exceptions** for domain-specific errors to make debugging easier.
* Always include a finally block (optional) for cleanup tasks like closing files or connections.

By using these techniques, you can ensure your Dart applications are robust, user-friendly, and capable of gracefully handling unexpected issues.