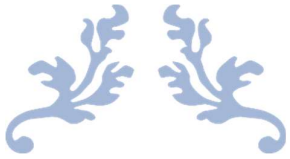


# FLUTTER INTERNSHIP

Prepared for Interns at XEPOS

JULY 15, 2025

MUHAMMAD SAMAR HASHIM

# Contents

# TASK NUMBER 1:

## What is int data type?

*The data type int stands for integer it's a basic data type used in many programming languages to represent whole numbers (numbers without decimal points).*

- *It Has **no decimal** or fractional part*
- *It Can be **positive**, **negative**, or **zero**.*

## What is meant by double data type?

*The **double** data type is a numeric data type used in many programming languages to represent **real numbers** (i.e., numbers with decimal points) with **double precision**.*

- ***Double** stands for **"double-precision floating-point number"**.*
- It uses **64 bits** (in most languages) to store the number, allowing:
    - *Greater **range** (very large/small numbers).*
    - *More **precision** (more digits after the decimal).*

## WHAT IS STRING?

*A string is a data type used in programming to represent text a sequence of characters.*

*A **string** can contain:*

- ***Letters**: "Hello"*
- ***Numbers** (as characters): "1234"*

## What is meant by bool?

*In programming, **bool** (short for **boolean**) is a **data type** that can have one of two possible values:*

- *True*
- *False*

## What is dynamic data type?

*A **dynamic data type** refers to a data type that is **determined at runtime**, rather than at compile time. This concept is typically found in **dynamically typed programming languages**.*

## What is var data type?

*The **var** data type is a special keyword used in some programming languages to declare a variable **without explicitly specifying its type**. The **type is inferred automatically by the compiler** based on the assigned value.*

- *var stands for **"variable".***
- *It tells the language to **infer the data type from the value** you assign.*
- *It's part of **type inference** systems.*

# OPERATORS

### 1. Arithmetic Operators

*These are used to perform basic math operations on numbers***.**

| Operator | Name | Example | Result |
|---|---|---|---|
| + | Addition | 5+3 | 8 |
| - | Subtraction | 5-3 | 2 |
| * | Multiplication | 5*3 | 15 |
| / | Division | 10/2 | 5 |
| % | Modulus | 10%3 | 1 |

### 2. Comparison Operators (Relational Operators)

These compare two values and return a Boolean result (True or False).

| Operators | Name | Example | Result |
|---|---|---|---|
| == | Equals too | 5==5 | True |
| != | Not equals too | 3 != 0 | True |
| > | Greater then | 7>3 | True |
| < | Less then | 4<2 | False |
| >= | Greater than or equals too | 5>=5 | True |
| <= | Less then or equals too | 3<=4 | True |

### 3. Logical Operators

Used to combine or manipulate **Boolean values** (True/False).

| Operator | Meaning | Example | Result |
|---|---|---|---|
| And | Logical and | True and false | False |
| Or | Logical or | True or false | True |
| not | Logical not | Not true | False |

**Write a Dart program that declares variables for name (String), age (int), salary (double), is Employed (bool), and years of Experience (dynamic). Perform arithmetic operations (add 10 to age, multiply salary by 1.5, increment years of Experience) and print a formatted string with all variables.**

```
void main() {
  String name = 'Ahmed';
  int age = 30;
  double salary = 50000.0;
  bool isEmployed = true;
  dynamic yearsOfExperience = 5;

  age += 10;
  salary *= 1.5;
  yearsOfExperience++;
  print('''
Employee Information
Name: $name
Age (after 10 years): $age
Salary (after 1.5x increase): \$${salary.toStringAsFixed(2)}
Currently Employed: ${is Employed ? "Yes" : "No"}
incremented Years of Experience: $yearsOfExperience
''');}
```

## OUTPUT:

```
Employee Information
Name: Ahmed
Age (after 10 years): 40
Salary (after 1.5x increase): $75000.00
Currently Employed: Yes
incremented Years of Experience: 6
```

**Create a program that takes two hardcoded numbers and performs all arithmetic operations (addition, subtraction, multiplication, division, modulus). Handle division-by-zero errors with conditional checks.**

```
void main() {
  int num1 = 20;
  int num2 = 0;

  int addition = num1 + num2;
um2 = $modulus');
  } else {  print('Addition: $num1 + $num2 = $addition');

  int subtraction = num1 - num2;
  print('Subtraction: $num1 - $num2 = $subtraction');

  int multiplication = num1 * num2;
  print('Multiplication: $num1 * $num2 = $multiplication');

  if (num2 != 0) {
    double division = num1 / num2;
    print('Division: $num1 / $num2 = $division');
  } else {
    print('Division: Error - Division by zero is not allowed.');
  }

  if (num2 != 0) {
    int modulus = num1 % num2;
    print('Modulus: $num1 % $n
    print('Modulus: Error - Division by zero is not allowed.');
  }
```

## OUTPUT:

```
Numbers: num1 = 20, num2 = 0

Addition: 20 + 0 = 20
Subtraction: 20 - 0 = 20
Multiplication: 20 * 0 = 0
Division: Cannot divide by zero.
Modulus: Cannot perform modulus by zero.
```

**Experiment with dynamic and var types. Write a program that changes the type of a dynamic variable (e.g., from int to String) and prints the results. Compare with var to understand type inference limitations?**

```
void main() {
  dynamic dynVar = 42;
  print('dynamic before: $dynVar (${dynVar.runtimeType})');

  dynVar = 'Now I am a String';
  print('dynamic after: $dynVar (${dynVar.runtimeType})');

  var inferredVar = 100;
  print('var before: $inferredVar (${inferredVar.runtimeType})');
}
```

**OUTPUT:**

```
dynamic before: 42 (int)
dynamic after: Now I am a String (String)
var before: 100 (int)
```

## Research Task

**Write a 100- word summary in your report explaining the difference between final, const, and var.**

*In Dart, var declares a variable with an inferred type that can be reassigned but not retyped. The final keyword defines a single-assignment variable: its value is set once and cannot change, though it can be assigned at runtime. In contrast, const declares compile time constants, meaning the value must be known at compile time and is deeply immutable. Use final for runtime constants and const for compile time constants. Unlike var, both final and const ensure immutability after assignment, helping improve code safety and predictability. Only var allows reassignment and flexibility.*

# TASK (DAY 2)

## TASK 1

**What is if-else?**

Used to execute a block of code if a condition is true; otherwise, execute another block.

int number = 10;

```
if (number > 0) {
  print('Positive');
} else {
  print('Non-positive');
}
```

## TASK 2

**What is switch?**

Used to compare a variable against multiple constant values (cases).

```
int day = 2;
  case 1:
    print('Monday');
switch (day) {

    break;
  case 2:
    print('Tuesday');
    break;
  default:
    print('Other day');
}
```

## TASK 3

**What is nested control structure?**

*Control structures placed inside one another (e.g., an if inside another if or a switch inside an if.*

*int score = 85;*

```
if (score >= 60) {
  if (score >= 90) {
    print('Excellent');
  } else {
    print('Passed');
```

```
  }
} else {
  print('Failed');
}
```

## TASK 4

**Implement for, while, and do-while loops with break and continue.**

### For loop:

```
void main() {
  print('--- FOR LOOP ---');
  for (int i = 1; i <= 5; i++) {
    if (i == 3) {
      print('Skipping 3 using continue');
      continue;
    }
    if (i == 5) {
      print('Breaking at 5');
      break;
    }
    print('i = $i');
  }
```

### WHILE LOOP:

```
int j = 1;
  while (j <= 5) {
    if (j == 2) {
      j++;
      print('Skipping 2 using continue');
      continue;}
    if (j == 4) {
      print('Breaking at 4');
      break;   }
    print('j = $j');
    j++;
  }
```

### DO WHILE LOOP:

```
int k = 1;
  do {
```

```
    if (k == 2) {
      k++;
      print('Skipping 2 using continue');
      continue;
    }
    if (k == 4) {
      print('Breaking at 4');
      break;
    }
    print('k = $k');
    k++;
  } while (k <= 5);
}
```

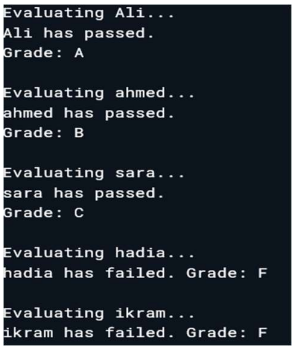
## TASK 5

**Solve complex problems using control flow.**

```
void main() {
  List<Map<String, dynamic>> students = [
    {'name': 'ali', 'score': 95},
    {'name': 'ahmed', 'score': 82},
    {'name': 'sara', 'score': 67},
    {'name': 'hadia', 'score': 49},
    {'name': 'ikram', 'score': 30}
  ];

  for (var student in students) {
    String name = student['name'];
    int score = student['score'];

    print('\nEvaluating $name...');

    if (score >= 50) {
      print('$name has passed.');

      if (score >= 90) {
        print('Grade: A');
      } else if (score >= 75) {
        print('Grade: B');
      } else if (score >= 60) {
        print('Grade: C');
      } else {
        print('Grade: D');
      }
    } else {
      print('$name has failed. Grade: F');
    }
```

```
  }
}
```

**OUTPUT:**

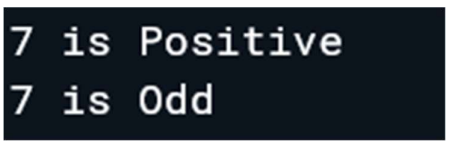



## TASK 6

**Write a program that takes a hardcoded integer and uses if-else to categorize it as "Positive," "Negative," or "Zero." If positive, check if its even or odd and print the result**

```
void main() {
  int number = 7;
  if (number > 0) {
    print('$number is Positive');
    if (number % 2 == 0) {
      print('$number is Even');
    } else {
      print('$number is Odd');
    }
  } else if (number < 0) {
    print('$number is Negative');
  } else {
    print('The number is Zero');
  }
}
```

**OUTPUT:**




## TASK 7

**Implement a FizzBuzz program that prints numbers from 1 to 100. For multiples of 3, print "Fizz"; for multiples of 5, print "Buzz"; for multiples of both, print "FizzBuzz."**

```
void main() {
  for (int i = 1; i <= 100; i++) {
    if (i % 3 == 0 && i % 5 == 0) {
      print('FizzBuzz');
    } else if (i % 3 == 0) {
      print('Fizz');
    } else if (i % 5 == 0) {
      print('Buzz');
    } else {
      print(i);
    }
  }
}
```

**OUTPUT:**

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
```

## TASK 8

**Create a program that prints a 5x5 star pattern (e.g., a square of asterisks) using nested for loops. Extend it to print a triangular pattern.**

```
void main(){
  int row = 5;
  for(int i = 0;i<row;i++){
    print(" "*(row-i)+"* "*i);
  }
}
```

**OUTPUT**

TASK 9

**Read the Dart documentation on control flow. Write a 100-word summary in your report comparing if-else and switch statements, including when to use each**

*In Dart, if-else statements are used for evaluating complex, ranged, or Boolean conditions. They offer flexibility when comparing variables against multiple unrelated values or when using logical operators (&&, ||). On the other hand, switch statements are more efficient and readable when checking a single variable against multiple constant values. Use switch when you have a fixed set of possible outcomes (like menu options or status codes), and if-else when conditions require more complex logic or range checks. Dart's switch also supports break, continue, and default for managing unmatched cases effectively.*

# TASK (DAY 3)

## Write reusable named functions with required and optional parameters.

**EXAMPLE NO 1:**

```
void greetUser({required String name, String greeting = "Hello"}) {
  print('$greeting, $name!');
}
```

## Use arrow syntax for concise functions.

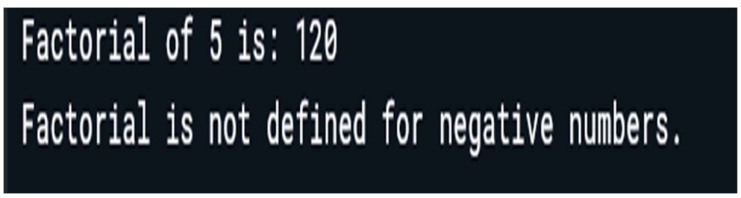
```
int addNumbers({required int a, required int b, bool showResult = true}) {
  int result = a + b;
  if (showResult) print('The sum is: $result');
  return result;
}
```

## Implement recursive and higher-order functions.

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

**Create a higher-order function that takes a list of numbers and a function to apply (e.g., square or double). Apply the function to each element and return the new list.**

```
int factorial(int n) {
  if (n < 0) {
    return -1;
  } else if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
void main() {
  int number = 5;
  int result = factorial(number);

  if (result != -1) {
    print('Factorial of $number is: $result');
  } else {
    print('Factorial is not defined for negative numbers.');
  }

  number = -2;
  result = factorial(number);

  if (result != -1) {
    print('Factorial of $number is: $result');
  } else {
    print('Factorial is not defined for negative numbers.');
  }
}
```




**Write a dart program with three functions: one to calculate the square of a number, one with optional parameters to print user details (name, age, role), and an arrow function to check if a number is even?**

```
void main() {

  print('Square of 6: ${square(6)}');

  print('\nUser Details :');
  printUserDetails(name: 'hadi', age: 30, role: 'salesman');

  print('\nUser Details :');
  printUserDetails();

  print('\nCheck even numbers:');
  print('Is 4 even? ${isEven(4)}');
  print('Is 7 even? ${isEven(7)}');
}

int square(int number) {
  return number * number;
}

void printUserDetails({String name = 'Unknown', int? age, String role = 'Guest'}) {
  print('Name: $name');
  if (age != null) {
    print('Age: $age');
  }
  print('Role: $role');
}

bool isEven(int number) => number % 2 == 0;
```

```
Square of 6: 36

User Details :
Name: hadi
Age: 30
Role: salesman

User Details :
Name: Unknown
Role: Guest

Check even numbers:
Is 4 even? true
Is 7 even? false
```

## Create a higher-order function that takes a list of numbers and a function to apply (e.g., square or double). Apply the function to each element and return the new list?

```
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];
```

```
  int square(int x) => x * x;
  int doubleValue(int x) => x * 2;

  List<int> squaredList = applyFunction(numbers, square);
  List<int> doubledList = applyFunction(numbers, doubleValue);

  print('Original List: $numbers');
  print('Squared List: $squaredList');
  print('Doubled List: $doubledList');
}

List<int> applyFunction(List<int> list, int Function(int) func) {
  return list.map(func).toList();
}
```

```
Original List: [1, 2, 3, 4, 5]
Squared List: [1, 4, 9, 16, 25]
Doubled List: [2, 4, 6, 8, 10]
```

**Read the Dart documentation on functions. Write a 100-word summary in your report explaining the benefits of named vs. optional parameters?**

*Dart supports both named and optional parameters, offering flexibility in function design.* ***Named parameters*** *improve code readability and clarity, especially when a function has many parameters or when their order might be confusing. They allow callers to specify only the arguments they need, by name.* ***Optional parameters****, which can be either named or positional, allow developers to omit some arguments, using default values instead. This simplifies function calls and reduces boilerplate. Named optional parameters are preferred in most cases because they make intent clearer and reduce errors due to misordered arguments. They enhance code maintainability and self-documentation.*

## TASK (DAY 4)

### What is list?

*A* ***List*** *is an ordered collection of items. It allows duplicates and elements are accessed by index.*

### What is set?

*A* ***Set*** *is an unordered collection of* ***unique*** *items (no duplicates).*

## What is map?

*A **Map** is a collection of key-value pairs.*

## Use iteration methods (for Each, map, where) and loops.

```
void main() {
  // List example
  List<int> numbers = [1, 2, 3, 4, 5];

  // forEach
  print('Using forEach on List:');
  numbers.forEach((n) => print('Number: $n'));

  // map
  List<int> squares = numbers.map((n) => n * n).toList();
  print('Squares using map: $squares');

  // where
  List<int> evenNumbers = numbers.where((n) => n.isEven).toList();
  print('Even numbers using where: $evenNumbers');

  // for loop
  print('Using for loop:');
  for (int i = 0; i < numbers.length; i++) {
    print('Index $i: ${numbers[i]}');
  }

  // while loop
  print('Using while loop:');
  int i = 0;
  while (i < numbers.length) {
    print('Number at $i: ${numbers[i]}');
    i++;
  }

  // do-while loop
  print('Using do-while loop:');
  int j = 0;
  do {
    print('Number at $j: ${numbers[j]}');
    j++;
  } while (j < numbers.length);

  // Set example
  Set<String> fruits = {'apple', 'banana', 'orange'};

  print('\nUsing forEach on Set:');
  fruits.forEach((fruit) => print('Fruit: $fruit'));

  // Map example
```

```
Map<String, int> scores = {
  'ahmed': 90,
  'ali': 85,
  'muskan': 95,
};

print('\nUsing forEach on Map:');
scores.forEach((key, value) {
  print('$key scored $value');
});

print('\nUsing for loop with Map keys:');
for (String name in scores.keys) {
  print('$name => ${scores[name]}');
}}
```

**OUTPUT**

```
Using forEach on List:
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Squares using map: [1, 4, 9, 16, 25]
Even numbers using where: [2, 4]
Using for loop:
Index 0: 1
Index 1: 2
Index 2: 3
Index 3: 4
Index 4: 5
Using while loop:
Number at 0: 1
Number at 1: 2
Number at 2: 3
Number at 3: 4
Number at 4: 5
Using do-while loop:
Number at 0: 1
Number at 1: 2
Number at 2: 3
Number at 3: 4
Number at 4: 5

Using forEach on Set:
Fruit: apple
Fruit: banana
Fruit: orange
```

```
Using forEach on Map:
ahmed scored 90
ali scored 85
muskan scored 95

Using for loop with Map keys:
ahmed => 90
ali => 85
muskan => 95
```

**Create a program that initializes a List of 10 numbers, sorts it, filters out even numbers using where, and transforms it using map to square each number.**

```
void main() {
  List<int> numbers = [7, 2, 10, 5, 8, 1, 3, 6, 4, 9];
  print('Original List: $numbers');
  numbers.sort();
  print('Sorted List: $numbers');
  List<int> oddNumbers = numbers.where((n) => n.isOdd).toList();
  print('Odd Numbers (filtered): $oddNumbers');
  List<int> squaredOdds = oddNumbers.map((n) => n * n).toList();
  print('Squared Odd Numbers: $squaredOdds');
}
```

```
Original List: [7, 2, 10, 5, 8, 1, 3, 6, 4, 9]
Sorted List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Odd Numbers (filtered): [1, 3, 5, 7, 9]
Squared Odd Numbers: [1, 9, 25, 49, 81]
```

**Write a program that creates two Sets of numbers, performs union, intersection, and difference operations, and prints the results.**

```
void main() {
Set<int> setA = {1, 2, 3, 4, 5};
  Set<int> setB = {4, 5, 6, 7, 8};
 print('Set A: $setA');
  print('Set B: $setB');
 Set<int> unionSet = setA.union(setB);
  print('\nUnion of A and B: $unionSet');
  Set<int> intersectionSet = setA.intersection(setB);
```

```
  print('Intersection of A and B: $intersectionSet');
Set<int> differenceSet = setA.difference(setB);
  print('Difference of A - B: $differenceSet');
Set<int> differenceBA = setB.difference(setA);
  print('Difference of B - A: $differenceBA');
}
```

```
Set A: {1, 2, 3, 4, 5}
Set B: {4, 5, 6, 7, 8}

Union of A and B: {1, 2, 3, 4, 5, 6, 7, 8}
Intersection of A and B: {4, 5}
Difference of A - B: {1, 2, 3}
Difference of B - A: {6, 7, 8}
```

**Build a program that uses a Map to represent a store inventory (item names as keys, quantities as values). Implement functions to add, remove, and update items, and print the inventory.**

```
void main() {
  Map<String, int> inventory = {};

  addItem(inventory, 'Apples', 10);
  addItem(inventory, 'Bananas', 20);
  addItem(inventory, 'Oranges', 15);

  print('\nInitial Inventory:');
  printInventory(inventory);

  updateItem(inventory, 'Bananas', 25);

  removeItem(inventory, 'Oranges');

  print('\nFinal Inventory:');
  printInventory(inventory);
}

void addItem(Map<String, int> inventory, String item, int quantity) {
  if (inventory.containsKey(item)) {
    print('Item "$item" already exists. Use updateItem to change quantity.');
  } else {
    inventory[item] = quantity;
    print('Added "$item" with quantity $quantity.');
  }
}

void removeItem(Map<String, int> inventory, String item) {
  if (inventory.containsKey(item)) {
    inventory.remove(item);
    print('Removed "$item" from inventory.');
  } else {
```

```
    print('Item "$item" not found.');
  }
}

void updateItem(Map<String, int> inventory, String item, int newQuantity) {
  if (inventory.containsKey(item)) {
    inventory[item] = newQuantity;
    print('Updated "$item" to quantity $newQuantity.');
  } else {
    print('Item "$item" not found. Use addItem to add it.');
  }
}

void printInventory(Map<String, int> inventory) {
  if (inventory.isEmpty) {
    print('Inventory is empty.');
  } else {
    inventory.forEach((item, quantity) {
      print('$item: $quantity');
    });
  }
}
```

```
Added "Apples" with quantity 10.
Added "Bananas" with quantity 20.
Added "Oranges" with quantity 15.

Initial Inventory:
Apples: 10
Bananas: 20
Oranges: 15
Updated "Bananas" to quantity 25.
Removed "Oranges" from inventory.

Final Inventory:
Apples: 10
Bananas: 25
```

## Read the Dart documentation on collections. Write a 100-word summary in your report comparing List, Set, and Map use cases.

*Dart provides three primary collection type **List**, **Set**, and **Map** each suited for different use cases. A **List** is an ordered collection that allows duplicates, ideal for maintaining sequences like to-do lists or user inputs. A **Set** is an unordered collection of unique items, useful for removing duplicates or checking membership efficiently. A **Map** stores key-value pairs, making it perfect for structured data like configurations, inventory, or lookup tables. Lists are best when order matters, Ses when uniqueness is needed, and Maps when associating*

*keys to values. Choosing the right collection improves performance, readability, and logic clarity in Dart applications.*

## Task (day 5)

### Definition of OOP:

*OOP (Object-Oriented Programming) is a programming paradigm based on the concept of "objects", which can contain:*

*Data in the form of fields.*

*Code in the form of procedure.*

*OOP helps structure software in a more natural and modular way, especially for larger systems. The four main principles of OOP are:*

***Encapsulation***

*Hiding the internal state and requiring all interaction to be performed through methods.*

***Abstraction***

*Exposing only relevant details and hiding complexity.*

***Inheritance***

*Creating new classes from existing ones.*

***Polymorphism***

*Using a single interface to represent different types of behavior.*

### Implement inheritance and polymorphism.

***Inheritance**:*

*Electric Car and Diesel Car inherit from Car.*

***Polymorphism**:*

*start_engine() and display_info() are overridden in each subclass to behave differently.*

***Code Reuse**: Shared logic (like stopping the engine) remains in the base class.*

## Use getters and setters for encapsulation

**Example of usage:**

```
class Rectangle {
  double _width;
  double _height;
  Rectangle(this._width, this._height);
  double get width => _width;
  set width(double value) {
    if (value > 0) {
      _width = value;
    } else {
      throw ArgumentError('Width must be positive');
    }
  }
  double get height => _height;
  set height(double value) {
    if (value > 0) {
      _height = value;
    } else {
      throw ArgumentError('Height must be positive');
    }
  }
  double get area => _width * _height;
  void displayInfo() {
    print('Rectangle: width=$_width, height=$_height, area=$area');
  }
}
```

## Write a program that defines a Car class with properties (model, year, mileage). Include getters/setters and a method to display details.

```
class Car {
  String _model;
  int _year;
  double _mileage;
  Car(String model, int year, double mileage)
      : _model = model,
        _year = year,
```

```
      _mileage = mileage {
    if (year < 1886) {
      throw ArgumentError('Year must be 1886 or later.');
    }
    if (mileage < 0) {
      throw ArgumentError('Mileage cannot be negative.');
    }
  }
  String get model => _model;
  set model(String value) => _model = value;

  // Getter and setter for year, with validation
  int get year => _year;
  set year(int value) {
    if (value < 1886) {
      throw ArgumentError('Year must be 1886 or later.');
    }
    _year = value;
  }
  double get mileage => _mileage;
  set mileage(double value) {
    if (value < 0) {
      throw ArgumentError('Mileage cannot be negative.');
    }
    _mileage = value;
  }
  void displayDetails() {
    print('Car Details:');
    print('  Model: $_model');
    print('  Year: $_year');
    print('  Mileage: ${_mileage.toStringAsFixed(1)} km');
  }
}

void main() {
  var car = Car('Toyota Camry', 2020, 30000);
  car.displayDetails();

  print('\n-- After updating --');
  car.model = 'Honda Accord';
  car.year = 2022;
  car.mileage = 35250.7;
  car.displayDetails();}
```

```
Car Details:
  Model: Toyota Camry
  Year: 2020
  Mileage: 30000.0 km

-- After updating --
Car Details:
  Model: Honda Accord
  Year: 2022
  Mileage: 35250.7 km
```

**Create a Vehicle class and two subclasses, ElectricCar and Gas. Carwithuniqueproperties (e.g., batteryCapacity, fuelType). Implement a method over ridden in each subclass**

```
class Vehicle {
  String model;
  int year;
  double mileage;

  Vehicle(this.model, this.year, this.mileage);
  void describe() {
    print('Vehicle: $model, Year: $year, Mileage: ${mileage.toStringAsFixed(1)} km');
  }
}

class ElectricCar extends Vehicle {
  double batteryCapacity;
  ElectricCar(String model, int year, double mileage, this.batteryCapacity)
      : super(model, year, mileage);

  @override
  void describe() {
    print('Electric Car: $model, Year: $year, Mileage: ${mileage.toStringAsFixed(1)} km, '
        'Battery: ${batteryCapacity.toStringAsFixed(1)} kWh');
  }
}

class GasCar extends Vehicle {
  String fuelType;

  GasCar(String model, int year, double mileage, this.fuelType)
      : super(model, year, mileage);

  @override
  void describe() {
    print('Gas Car: $model, Year: $year, Mileage: ${mileage.toStringAsFixed(1)} km, '
        'Fuel Type: $fuelType');
  }
```

```
}
void main() {
  Vehicle generic = Vehicle('Generic Model', 2010, 120000.0);
  ElectricCar tesla = ElectricCar('Tesla Model 3', 2023, 15000.0, 75.0);
  GasCar toyota = GasCar('Toyota Corolla', 2021, 40000.0, 'Petrol');

  generic.describe();
  tesla.describe();
  toyota.describe();
}
```

```
Vehicle: Generic Model, Year: 2010, Mileage: 120000.0 km
Electric Car: Tesla Model 3, Year: 2023, Mileage: 15000.0 km, Battery: 75.0 kWh
Gas Car: Toyota Corolla, Year: 2021, Mileage: 40000.0 km, Fuel Type: Petrol
```

## Design a system with a Person class and a Student subclass. The student class should track grades in a List and include a method to calculate the average grade

```
class Person {
  String name;
  int age;
Person(this.name, this.age);
 void printInfo() {
    print('Person: $name, Age: $age');
  }
}
class Student extends Person {
  List<double> _grades = [];
  Student(String name, int age) : super(name, age);
 List<double> get grades => List.unmodifiable(_grades);
  void addGrade(double grade) {
    if (grade < 0 || grade > 100) {
  throw ArgumentError('Grade must be between 0 and 100');
    }
    _grades.add(grade);
  }
 double calculateAverage() {
    if (_grades.isEmpty) {
      return 0.0;
 }
    double sum = _grades.fold(0.0, (prev, g) => prev + g);
    return sum / _grades.length;  }
 @override
  void printInfo() {
    super.printInfo();
```

```
    print('  Grades: ${_grades.map((g) => g.toStringAsFixed(1)).join(', ')}');
    print('  Average Grade: ${calculateAverage().toStringAsFixed(2)}');
  }
}
void main() {
  var person = Person('Ahmed', 30);
  person.printInfo();
  print('\n---');
var student = Student('ali', 20);
  student.addGrade(85.0);
student.addGrade(92.5);
  student.addGrade(76.0);
  student.printInfo(); }
```

```
Person: Ahmed, Age: 30

---
Person: ali, Age: 20
  Grades: 85.0, 92.5, 76.0
  Average Grade: 84.50
```

**Read the Dart documentation on classes. Write a 100-word summary in your report on encapsulation and why getters/setters are used?**

*Encapsulation in Dart refers to bundling a class's data (private fields) and behavior (methods) while hiding internal state from external access. By marking fields with an underscore (_), Dart restricts direct access outside the library, ensuring controlled interaction. Getters and setters form the public interface for these fields: getters allow safe read access, while setters enable validation and enforce invariants before modifying internal data. This ensures data integrity, supports future refactoring (since implementation can change without altering how clients access properties), and allows computed or side-effect behaviors when reading or writing.*

# TASK (DAY 6)

## Question 1:

**Understand asynchronous programming with Futures**

*A Future<T> represents a computation that completes with a value (of type T) or an error at some later time. Dart functions that perform time-consuming work (like network I/O) typically return Futures. You can consume a Future by registering callbacks with .then() and .catchError(), or more seamlessly using async/await. async functions automatically return a Future, and await pauses execution until the future completes, without blocking the event loop Reddit+13Dart+13Dart+13.*

## Question 2:

**Use async/await for complex asynchronous operations**

*Marking a function as async converts its return type to Future, even if it returns directly. Within an async function, using await pauses execution until each awaited future completes.*

## Question 3:

**Handle errors in asynchronous code**

**You can handle errors in asynchronous code using:**

- *try/catch inside async functions, surrounding any await expressions.*
- *catchError() on futures when using .then() chains.*
- *catchError() captures errors from the original future or in chained callbacks.*
- *For functions that may throw synchronously before returning a Future, wrap logic in Future.sync() to funnel errors into the asynchronous flow properly.*

**Write a program with a Future that simulates fetching user data after 3 seconds. Use async/await to handle the result.**

```
Future<String> fetchUserData() async {
  return await Future.delayed(
    Duration(seconds: 3),
    () => 'salar, age 28',
```

```
  );
}

Future<void> main() async {
  print('Fetching user data...');
  try {
    final userData = await fetchUserData();
    print('User data received: $userData');
  } catch (e) {
    print('Failed to fetch user data: $e');
  }
}
```

```
Fetching user data...
User data received: salar, age 28
```

## Create a program that runs three Futures with different delays (2, 3, 5 seconds). Use Future.wait to wait for all to complete and print the results in order.

```
import 'dart:async';
Future<String> task(String name, int sec) async {
  await Future.delayed(Duration(seconds: sec));
  return '$name done after $sec seconds';
}
Future<void> main() async {
  print('Starting tasks...');
  final results = await Future.wait<String>([
    task('First', 2),
    task('Second', 3),
    task('Third', 5),
  ]);
  print('All tasks completed in order:');
  for (var r in results) {
    print(r);
  }
}
```

```
Starting tasks...
All tasks completed in order:
First done after 2 seconds
Second done after 3 seconds
Third done after 5 seconds
```

**Write a program with a Future that may throw an error (e.g., invalid data). Use try-catch to handle the error and print a user-friendly message.**

```
import 'dart:async';

Future<int> validateAge(int age) async {
  return Future.delayed(Duration(seconds: 1), () {
    if (age < 0 || age > 150) {
      throw Exception('Invalid age: $age');
    }
    return age;
  });
}

Future<void> main() async {
  print('Validating age...');
  try {
    final age = await validateAge(-5);
    print('Age is valid: $age');
  } catch (e) {
    print('Error: ${e.toString()}');
  }
}
```

```
Validating age...
Error: Exception: Invalid age: -5
```

**Read the Dart documentation on Futures. Write a 100-word summary in your report on the difference between Future. then and async/await.**

Using Future.then() registers callbacks that execute when the future completes, enabling chained asynchronous operations (e.g., fetch().then(process).catchError(...)) [rm-dart.web.app+15dart.dev+15medium.com+15](). In contrast, async/await provides a cleaner, sequential style: marking a function async returns a Future, and using await pauses execution until the future completes, allowing readable, try/catch-wrapped logic . Compared to .then(), async/await improves clarity, flattens deeply nested callbacks, and simplifies error handling. Dart's linter even recommends preferring async/await for maintainable asynchronous code.

# TASK (DAY 7)

**Write a program combining concepts from (variables, control flow, functions, collections, classes, async). Example: A student man agement system with a Student class, a List of students, and an async function to fetch student data.**

```
import 'dart:async';
class Person {
  String name;
  int age;
  Person(this.name, this.age);
  void printInfo() => print('Person: $name, Age: $age');
}
class Student extends Person {
  List<double> grades = [];
  Student(String name, int age) : super(name, age);
  void addGrade(double g) {
    if (g < 0 || g > 100) throw ArgumentError('Grade must be 0–100');
    grades.add(g);  }
 double get average =>
      grades.isEmpty ? 0 : grades.reduce((a, b) => a + b) / grades.length;
  @override
  void printInfo() {
    super.printInfo();
    print(
        'Grades: ${grades.map((g) => g.toStringAsFixed(1)).join(', ')}; Average:
${average.toStringAsFixed(2)}');
  }
}
Future<List<Student>> fetchStudents() async {
  await Future.delayed(Duration(seconds: 2));
  var s1 = Student('Aliya', 20)..addGrade(88)..addGrade(92);
  var s2 = Student('ali', 22)..addGrade(75)..addGrade(80)..addGrade(85);
  return [s1, s2];
}
Future<void> main() async {
  print('Loading students...');
  try {
    var students = await fetchStudents();
    print('Students loaded: ${students.length}');
    for (var s in students) {
      s.printInfo();
    }
  } catch (e) {
    print('Failed to load students: $e');
  }
}
```

```
Loading students...
Students loaded: 2
Person: Aliya, Age: 20
Grades: 88.0, 92.0; Average: 90.00
Person: ali, Age: 22
Grades: 75.0, 80.0, 85.0; Average: 80.00
```

## Write a 300-word reflection report in day7/reflection.md on challenges, learnings, and how you plan to apply Week 1 skills in Flutter development.

In Dart, **variables** like var, final, or const form the foundation for representing data states new values wrapped in immutable or mutable containers. Strong typing promotes clarity and early error detection. *Control flow* structures if/else, for/while, switch dictate program logic flow, enabling dynamic responses to variable states and guiding execution paths.

**Functions**, including first-class and higher-order ones, encapsulate behavior. You can pass them as arguments, return them, or attach them to collections for expressive data transformations using map, where, and closures Dart+9FasterCapital+9Kinda Technical+9. Such patterns promote modularity and reuse, ensuring that logic remains DRY and testable.

**Collections** like Lists, Maps, and Sets serve as essential containers. Lists maintain ordered data (e.g., student grades), Maps link keys to values (e.g., user profiles), and Sets ensure uniqueness. Iteration over these collections, especially using functional operators, makes data operations concise and powerful.

**Classes** structure related data and behavior together. Encapsulation with private fields (_) and public getters/setters offers control and validation. Inheritance and polymorphism allow shared behavior among derived types while enabling custom overrides. This aligns with Dart's object-oriented philosophy, supporting maintainable application design.

Asynchronous operations using **async**/await and Future bring non-blocking concurrency into Dart. By marking functions async and using await, you can write intuitive, sequential-looking code that waits for asynchronous results such as network calls or I/O without freezing the UI thread Dart+7Dart+7Dart+7. Error handling in async flows becomes straightforward with try/catch.

Together, these elements variables, flow control, modular functions, robust data collections, structured classes, and clean async allow Dart developers to build programs that are **readable**, **maintainable**, **type-safe**, and **user-friendly**, especially in interactive apps like Flutter. The harmonious blend of these concepts empowers developers to tackle real-world complexity with clarity and reliability.