What is Object-Oriented Programming (OOP)?

OOP stands for **Object-Oriented Programming**.
It organizes the program into **objects** — which are **real-world entities** like Car, Student, or Employee.

In simple words:

OOP allows you to group **data (variables)** and **functions (methods)** inside a **class** and use them through **objects**.

Class in Dart

A **class** is a blueprint or template for creating objects.
It defines **properties (variables)** and **methods (functions)** that objects will have.

Syntax:

```
class ClassName {
  // properties
  // methods
}
```

Example: Creating a Class

```
class Student {
  // Properties (data members)
  String? name;
  int? age;

  // Method (function inside class)
  void displayInfo() {
    print("Name: $name, Age: $age");
  }
}
```

Object in Dart

An **object** is an **instance of a class**.
It is created using the `new` keyword (optional in Dart).

Syntax:

```
var objectName = ClassName();
```

Example: Creating and Using Object

```
void main() {
  var student1 = Student(); // Object of class Student
  student1.name = "Neeraj";
  student1.age = 23;

  student1.displayInfo(); // Output: Name: Neeraj, Age: 23
```

```
}
```

Note:

- You can create multiple objects from one class.
- Each object has its own copy of data.

Constructor in Dart

A **constructor** is a special function in a class that runs **automatically** when an object is created.
It is mainly used to **initialize variables**.

Syntax:

```
class ClassName {
  ClassName() {
    // initialization code
  }
}
```

Example 1 — Default Constructor

```
class Student {
  String? name;
  int? age;

  // Constructor
  Student() {
    print("Constructor Called!");
  }
}

void main() {
  var student1 = Student(); // Output: Constructor Called!
}
```

Example 2 — Parameterized Constructor

When you want to pass values while creating an object.

```
class Student {
  String name;
  int age;

  // Parameterized constructor
  Student(this.name, this.age);

  void showInfo() {
    print("Name: $name, Age: $age");
  }
}

void main() {
  var student1 = Student("Neeraj", 23);
  student1.showInfo(); // Output: Name: Neeraj, Age: 23
```

```
}
```

Note:

- `this.name` refers to the class variable.
- You can assign values directly while creating the object.

Example 3 — Named Constructor

You can create **multiple constructors** using **named constructors**.

```
class Student {
  String name;
  int age;

  // Default constructor
  Student(this.name, this.age);

  // Named constructor
  Student.guest() {
    name = "Guest";
    age = 0;
  }

  void showInfo() {
    print("Name: $name, Age: $age");
  }
}

void main() {
  var student1 = Student("Neeraj", 23);
  student1.showInfo();

  var student2 = Student.guest();
  student2.showInfo();
}
```

Output:

```
Name: Neeraj, Age: 23
Name: Guest, Age: 0
```

Key Points to Remember

Concept	Description	Example
Class	Blueprint of an object	`class Student {}`
Object	Instance of a class	`var s = Student();`
Constructor	Called automatically when object created	`Student(this.name)`
Named Constructor	Special named version	`Student.guest()`
this Keyword	Refers to current class variable	`this.name = name`

1. `this` keyword in Dart

`this` refers to the **current object** of a class.

It is used when **class variables** and **method parameters** have the same name — to differentiate between them.

Example:

```
class Student {
  String name = "";
  int age = 0;

  // Constructor using this
  Student(String name, int age) {
    this.name = name; // refers to current object
    this.age = age;
  }

  void display() {
    print("Name: $name, Age: $age");
  }
}

void main() {
  var s1 = Student("Neeraj", 23);
  s1.display();
}
```

Output:

```
Name: Neeraj, Age: 23
```

2. Instance Members vs Static Members

Type	Definition	Accessed By	Memory
Instance Members	Belong to each object	Using object name	Each object gets its own copy
Static Members	Belong to the class itself	Using class name	Only one copy shared by all

Example:

```
class Car {
  String color = ""; // instance variable
  static int wheels = 4; // static variable

  void showColor() {
    print("Car color: $color");
  }

  static void showWheels() {
    print("All cars have $wheels wheels");
```

```
  }
}

void main() {
  Car c1 = Car();
  c1.color = "Red";
  c1.showColor();

  // Accessing static member
  Car.showWheels();
}
```

Output:

```
Car color: Red
All cars have 4 wheels
```

3. Inheritance in Dart

Inheritance allows a class (child/subclass) to **use properties and methods of another class** (parent/superclass).

Syntax:

```
class ChildClass extends ParentClass
```

Example:

```
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

class Dog extends Animal {
  void bark() {
    print("Dog is barking");
  }
}

void main() {
  var d = Dog();
  d.eat();  // from parent class
  d.bark(); // from child class
}
```

Output:

```
Animal is eating
Dog is barking
```

Types of Inheritance in Dart

Type	Description	Example
Single Inheritance	One class inherits from another	`class B extends A`
Multilevel Inheritance	A class inherits from a derived class	`A -> B -> C`
Hierarchical Inheritance	Multiple classes inherit from one class	`B extends A, C extends A`
Multiple Inheritance	Not directly supported (can use **mixins** instead)	

Example – Multilevel:

```
class A {
  void methodA() => print("A method");
}

class B extends A {
  void methodB() => print("B method");
}

class C extends B {
  void methodC() => print("C method");
}

void main() {
  var obj = C();
  obj.methodA();
  obj.methodB();
  obj.methodC();
}
```

Output:

```
A method
B method
C method
```

`extends` keyword

Use:
To inherit a class.

```
class Parent {
  void display() {
    print("Parent class method");
  }
}

class Child extends Parent {
  void show() {
    print("Child class method");
  }
}

void main() {
  var c = Child();
```

```
  c.display(); // inherited
  c.show();
}
```

`super` keyword

Use:
To call **parent class constructor** or **methods** from child class.

Example:

```
class Person {
  String name = "";

  Person(this.name);

  void show() {
    print("Person name: $name");
  }
}

class Student extends Person {
  int rollNo;

  Student(String name, this.rollNo) : super(name); // calling parent
constructor

  void display() {
    super.show(); // calling parent method
    print("Roll Number: $rollNo");
  }
}

void main() {
  var s = Student("Neeraj", 101);
  s.display();
}
```

Output:

```
Person name: Neeraj
Roll Number: 101
```

What is Polymorphism?

Polymorphism means **"many forms."**
In OOP, it allows **one method or object** to behave **differently** based on the situation.

There are mainly **two types**:

Type	Description
Compile-time (Static)	Achieved by *method overloading* (Dart doesn't support overloading directly).

Type	Description
Runtime (Dynamic)	Achieved by *method overriding* (Supported in Dart). ✅

2. Method Overriding

When a **child class** provides a **new implementation** of a **method already defined in its parent class**, it's called **method overriding**.

Purpose:
To change or extend the behavior of a method inherited from the parent class.

Example: Method Overriding

```
class Animal {
  void sound() {
    print("Animal makes a sound");
  }
}

class Dog extends Animal {
  @override
  void sound() { // overriding parent method
    print("Dog barks");
  }
}

void main() {
  var obj = Dog();
  obj.sound(); // Calls overridden method
}
```

Output:

```
Dog barks
```

Here, the `Dog` class overrides the `sound()` method from `Animal`.
Even though `sound()` exists in both, the **child class version runs**, not the parent's.

Runtime Polymorphism

When a **parent class reference** is used to **refer to a child class object**, and the **method call is decided at runtime**, that's **runtime polymorphism**.

Example:

```
class Animal {
  void sound() {
    print("Some generic sound");
  }
}

class Dog extends Animal {
  @override
  void sound() {
```

```
    print("Dog barks");
  }
}

class Cat extends Animal {
  @override
  void sound() {
    print("Cat meows");
  }
}

void main() {
  Animal a;  // reference of parent class

  a = Dog();
  a.sound();  // Dog's version runs

  a = Cat();
  a.sound();  // Cat's version runs
}
```

Output:

```
Dog barks
Cat meows
```

Explanation:

- The **object type** (`Dog`, `Cat`) decides which version of `sound()` runs.
- This is **decided at runtime**, not at compile time.
  Hence called **runtime polymorphism**.

Why Method Overriding is Useful

To achieve **runtime polymorphism**
To **reuse code** and **customize** behavior
To **enhance flexibility** — same method name, different results

Example: Combining all concepts

```
class Vehicle {
  void start() {
    print("Vehicle is starting");
  }
}

class Car extends Vehicle {
  @override
  void start() {
    print("Car starts with a key");
  }
}

class Bike extends Vehicle {
  @override
  void start() {
    print("Bike starts with a self start button");
```

```
  }
}

void main() {
  Vehicle v;

  v = Car();
  v.start();

  v = Bike();
  v.start();
}
```

Output:

```
Car starts with a key
Bike starts with a self start button
```

Summary Table

Concept	Description	Example
Polymorphism	One interface, many implementations	Parent class reference → Child object
Method Overriding	Redefining a parent method in child	`@override` keyword used
Runtime Polymorphism	Method called at runtime depending on object type	`Animal a = Dog(); a.sound();`

Encapsulation

Encapsulation means **binding data (variables) and methods (functions)** together and **restricting direct access** to some parts of the object.

In Dart, **encapsulation** is implemented using:

- **Private variables** (using `_underscore`)
- **Getters and Setters** to **access or modify** private data safely.

Private Variables in Dart

In Dart, any identifier that starts with an **underscore (_)** is **private to its library (file)**.

Example:

```
class BankAccount {
  // private variable
  double _balance = 0;

  // getter (to access balance)
  double get balance => _balance;

  // setter (to set balance safely)
  set balance(double amount) {
```

```
    if (amount >= 0) {
      _balance = amount;
    } else {
      print("Invalid amount!");
    }
  }
}

void main() {
  var account = BankAccount();

  account.balance = 1000; // calling setter
  print("Current Balance: ${account.balance}"); // calling getter
}
```

Output:

```
Current Balance: 1000.0
```

Explanation:

1. `_balance` → private variable
2. `get` → used to read private value
3. `set` → used to update private value safely
4. Protects data → this is **encapsulation**

Another Example

```
class Student {
  String _name = '';

  // setter
  set name(String value) {
    if (value.isNotEmpty) {
      _name = value;
    } else {
      print("Name cannot be empty!");
    }
  }

  // getter
  String get name => _name;
}

void main() {
  var s = Student();
  s.name = "Neeraj";
  print("Student name: ${s.name}");
}
```

Encapsulation Benefits

Data hiding
Controlled access
Easy code maintenance
Avoids accidental modification of data

Abstraction

Abstraction means **showing only essential details** and **hiding implementation details**.
It helps simplify complex systems.

In Dart, we achieve abstraction using:

- **Abstract Classes**
- **Interfaces** (`implements` keyword)

Abstract Class

An **abstract class** cannot be instantiated directly.
It is used to **define structure**, not complete implementation.
Subclasses must **override its abstract methods**.

Example:

```
abstract class Shape {
  void area(); // abstract method (no body)
}

class Circle extends Shape {
  double radius;
  Circle(this.radius);

  @override
  void area() {
    print("Area of Circle: ${3.14 * radius * radius}");
  }
}

void main() {
  var c = Circle(5);
  c.area();
}
```

Output:

```
Area of Circle: 78.5
```

Explanation:

- `Shape` is abstract → can't create `Shape()` object
- Child class `Circle` overrides `area()` method

Interface in Dart

In Dart, **there is no separate "interface" keyword**.
Any **class can act as an interface**, and we use **`implements`** keyword.

When a class **implements** another class:

- It must **override all methods** of that class.

Example:

```
class Printer {
  void printDocument() {}
}

class MyPrinter implements Printer {
  @override
  void printDocument() {
    print("Printing document...");
  }
}

void main() {
  var p = MyPrinter();
  p.printDocument();
}
```

Output:

```
Printing document...
```

Abstract Class vs Interface

Feature	Abstract Class	Interface (`implements`)
Can have normal + abstract methods	Yes	No (all must be overridden)
Can have constructors	Yes	No
Can have implemented methods	Yes	No
Keyword used	`abstract`	`implements`
Multiple inheritance	No	Yes (can implement multiple classes)

Example of Multiple Interfaces

```
class A {
  void displayA() {}
}

class B {
  void displayB() {}
}

class C implements A, B {
  @override
  void displayA() => print("Display from A");

  @override
  void displayB() => print("Display from B");
}

void main() {
```

```
  var obj = C();
  obj.displayA();
  obj.displayB();
}
Display from A
Display from B
```

Summary

Concept	Keyword	Purpose
Encapsulation	_, `get`, `set`	Protect and control data access
Abstraction	`abstract`, `implements`	Hide implementation details
Abstract class	`abstract`	Define structure for subclasses
Interface	`implements`	Force a class to implement specific methods

Synchronous vs Asynchronous Programming

Synchronous Programming

In synchronous programming, **code runs line by line** —
each line **waits** for the previous one to finish before executing.

Example:

```
void main() {
  print("Task 1");
  print("Task 2");
  print("Task 3");
}
```

Output:

```
Task 1
Task 2
Task 3
```

Explanation:
Everything executes **in order** — one after another.

Asynchronous Programming

In asynchronous programming, a line of code can **run later** (in the background)
— the program **doesn't wait** for it to finish before moving on.

Example:

```
void main() {
  print("Task 1");
  Future.delayed(Duration(seconds: 2), () {
    print("Task 2");
  });
```

```
  print("Task 3");
}
```

Output:

```
Task 1
Task 3
Task 2
```

Explanation:

- `Future.delayed()` runs asynchronously after 2 seconds.
- The program **does not wait** — it continues running the next lines.

What is a Future in Dart?

A **Future** represents a **value that will be available in the future**.
It's like a **promise** that a value or error will come **later**.

Example:

```
Future<String> fetchData() {
  return Future.delayed(Duration(seconds: 2), () => "Data fetched!");
}

void main() {
  print("Fetching data...");
  fetchData().then((value) => print(value));
  print("Main program continues...");
}
```

Output:

```
Fetching data...
Main program continues...
Data fetched!
```

Explanation:

- `fetchData()` returns a Future.
- `.then()` is used to handle the **result** when the Future completes.

async and await Keywords

These keywords make asynchronous code look **clean and sequential**, like synchronous code.

- `async` → Marks a function as asynchronous.
- `await` → Waits for a Future to complete before moving to the next line.

Example:

```
Future<String> fetchData() async {
```

```
  await Future.delayed(Duration(seconds: 2));
  return "Data fetched successfully!";
}

void main() async {
  print("Fetching data...");
  String result = await fetchData();
  print(result);
  print("Done!");
}
```

Output:

```
Fetching data...
Data fetched successfully!
Done!
```

Explanation:

- `await` pauses execution of the async function until the Future completes.
- Other parts of the program can still run asynchronously.

Future.then()

If you don't use `async/await`, you can handle the result using `.then()`.

Example:

```
Future<String> getUser() {
  return Future.delayed(Duration(seconds: 1), () => "Neeraj");
}

void main() {
  print("Fetching user...");
  getUser().then((user) {
    print("User: $user");
  }).catchError((error) {
    print("Error: $error");
  });
}
```

Output:

```
Fetching user...
User: Neeraj
```

Explanation:

- `.then()` → runs when the Future completes successfully.
- `.catchError()` → handles any exception.

Chaining Futures

You can chain multiple `.then()` calls together to run tasks **in sequence**.

Example:

```
Future<String> step1() async {
  await Future.delayed(Duration(seconds: 1));
  return "Step 1 done";
}

Future<String> step2() async {
  await Future.delayed(Duration(seconds: 1));
  return "Step 2 done";
}

Future<String> step3() async {
  await Future.delayed(Duration(seconds: 1));
  return "Step 3 done";
}

void main() {
  step1()
    .then((value) {
      print(value);
      return step2();
    })
    .then((value) {
      print(value);
      return step3();
    })
    .then((value) {
      print(value);
    })
    .catchError((error) {
      print("Error: $error");
    })
    .whenComplete(() {
      print("All tasks completed!");
    });
}
```

Output:

```
Step 1 done
Step 2 done
Step 3 done
All tasks completed!
```

🧠 **Explanation:**

- Each `.then()` waits for the previous Future to finish.
- `whenComplete()` always runs — similar to `finally` in try-catch.

Async / await vs then()

Feature	async/await	then()
Style	Looks like normal code	Uses chaining
Readability	Easier for long tasks	Good for short async work

Feature	async/await	then()
Error Handling	try-catch block	catchError() method
Example	`await fetchData()`	`fetchData().then(...)`

Enum with Values

Enum (short for *enumeration*) is a special type used to define a group of constant values.
In Dart, from version 2.17+, **enums can also have values, methods, and constructors.**

Example:

```
enum Status {
  success(200),
  notFound(404),
  internalError(500);

  final int code;
  const Status(this.code);
}

void main() {
  print(Status.success);          // Output: Status.success
  print(Status.success.code);     // Output: 200
}
```

Explanation:

- `Status` is an enum having constant values.
- Each enum value (`success`, `notFound`, etc.) has an **integer code**.
- We used a **const constructor** for initialization.

Typedefs

`typedef` is used to **create a custom name for a function type** (alias).
It helps make code more readable and flexible.

Example 1: Function alias

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

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

void calculate(MathOperation operation) {
  print("Result: ${operation(10, 5)}");
}

void main() {
  calculate(add);       // Output: Result: 15
  calculate(subtract);  // Output: Result: 5
}
```

Example 2: Typedef with Class

```
typedef Greet = void Function(String name);

void sayHello(String name) => print("Hello, $name!");

void main() {
  Greet greetUser = sayHello;
  greetUser("Neeraj");
}
```