Classes and Objects in Dart

What is a Class?

A **class** is like a **blueprint** for creating objects.

Blueprint is also like a model.

It defines:

- What **properties** (variables/data) an object will have
- What **methods** (functions/behavior) the object can do

In simple terms:
If a **house** is an object, the **class** is the house plan.

Syntax of a Class in Dart:

```
class Person {
  String name = '';
  int age = 0;
  void sayHello() {
    print("Hello, my name is $name and I am $age years old.");
  }
}
```

Here:

- name and age are **fields** (variables)
- sayHello() is a **method**

What is an Object?

An **object** is a real-world **instance of a class**.

It’s like building an actual house from the blueprint.

Creating an Object

```
void main() {
  Person p1 = Person();  // creating an object
  p1.name = "Sakif";
  p1.age = 23;
  p1.sayHello();  // Output: Hello, my name is Sakif and I am 23 years old.
}
```

Here:

- p1 is an object of the Person class
- You can access the object's properties and methods using the dot (.) operator

Constructor in Dart

A **constructor** is a special function used to **create objects** and optionally **initialize values**.

Default Constructor:

Automatically provided by Dart if you don't write any.

Custom Constructor:

```
class Person {
  String name;
  int age;
  // Constructor
  Person(this.name, this.age);
  void sayHello() {
    print("Hello, my name is $name and I am $age years old.");
  }
}
void main() {
  Person p2 = Person("Sakif", 23);
  p2.sayHello();  // Output: Hello, my name is Sakif and I am 23 years old.
}
```

this.name refers to the **class's field**, and name is the constructor parameter.

Summary Table

Term	**Meaning**
Class	A blueprint(model) for creating objects
Object	An actual instance created from a class
Field	A variable inside a class
Method	A function inside a class
Constructor	A special method for object creation and value setting

Real-Life Example:

```
class Car {
  String brand;
  int modelYear;

  Car(this.brand, this.modelYear);

  void displayInfo() {
    print("Brand: $brand, Year: $modelYear");
  }
}

void main() {
  Car car1 = Car("Toyota", 2022);
  Car car2 = Car("Tesla", 2024);

  car1.displayInfo();  // Brand: Toyota, Year: 2022
  car2.displayInfo();  // Brand: Tesla, Year: 2024
}
```

Each object (car1, car2) holds different data but comes from the same class (Car).

Error Handling & Null Check Operators in Dart

Part 1: Error Handling in Dart

What is Error Handling?

Error handling lets you **control what happens when your code runs into a problem** — instead of crashing, you can catch the error and respond gracefully.

Why Use Error Handling?

- Prevent app crashes
- Show user-friendly messages
- Handle unexpected behavior safely

Basic Error Handling Syntax

```
try {
  // Code that might throw an error
} catch (e) {
  // Code to handle the error
}
```

Example:

```
void main() {
  try {
    int result = 10 ~/ 0; // ~/ is integer division
    print(result);
  } catch (e) {
    print("Error occurred: $e");
  }
}
```

Output:

Error occurred: IntegerDivisionByZeroException

Optional: Use on and finally

```
void main() {
  try {
    int result = 10 ~/ 0;
    print(result);
  } on IntegerDivisionByZeroException {
    print("You can't divide by zero.");
  } catch (e) {
    print("Some other error: $e");
  } finally {
    print("This always runs.");
  }
}
```

- on: For specific exception types
- catch: For all types of errors
- finally: Runs no matter what (success or error)

Throwing Custom Errors

```
void checkAge(int age) {
  if (age < 18) {
    throw Exception("You must be 18 or older.");
  } else {
    print("Welcome!");
  }
}
```

Part 2: Null Check Operators in Dart

What is null in Dart?

In Dart, null means **no value** or **nothing**.

By default, variables in Dart cannot hold null unless you specifically allow them to do so. Dart uses **null safety** to help prevent errors that happen when a variable is unexpectedly null.

Nullable and Non-nullable Types

- int age; → This variable **must always have a value**. It is **non-nullable**.
- int? age; → This variable **can be null**. It is **nullable**.
  The ? after the type tells Dart that this variable is allowed to hold a null value.

Example:

```
void main() {
  int? age;     // Nullable integer. Default value is null.
  print(age);   // Output: null
  age = 20;     // Now assigning value to it.
  print(age);   // Output: 20
}
```

This example shows how int? lets a variable stay empty (null) until you assign a real value.

What is Null Safety?

Null safety helps you **avoid null-related errors** by checking whether a variable can or cannot be null.

Common Null Check Operators

1. ? — Nullable Type

Allows a variable to hold null.

String? name;

2.

Says: “I'm sure this is not null.”
Will **crash** if it actually is null.

String? name = "Sakif";

print(name!); // Safe

name = null;

// print(name!); // Error: Null check operator used on a null value

3. ?? — Null Coalescing Operator

Provides a default value if the variable is null.

String? city;

print(city ?? "Unknown"); // Output: Unknown

4. ??= — Null-Aware Assignment

Assigns a value **only if** the variable is null.

```
String? country;
country ??= "Bangladesh";
print(country); // Bangladesh
```

5. ?. — Null-Aware Access

Calls a method or accesses a property only if the object is not null.

```
String? user = null;
print(user?.length); // Output: null (doesn't crash)
```

Summary Table

Operator	Name	Purpose
?	Nullable type	Allows variable to be null
!	Null assertion	Trusts variable is not null
??	Null coalescing	Provides fallback value
??=	Null-aware assignment	Assigns only if variable is null
?.	Null-aware access	Safely access method/property if not null

Combining Error Handling and Null Checks

```
void main() {
  String? name;

  try {
    print(name!.toUpperCase()); // Will throw error if name is null
  } catch (e) {
    print("Caught error: $e");
  }
}
```

Final Notes

- Always use null checks before using variables that might be null.
- Use try-catch to safely handle operations like file reading, API calls, or risky calculations.
- Dart's null safety makes code more stable and less error-prone.

Constructors in Dart

What is a Constructor?

A **constructor** is a special method in a class that is automatically called when an object is created.
It is used to **initialize fields** and set default or custom values when a new instance of the class is created.

A constructor:

- Has the **same name as the class**
- **Does not have a return type** (not even void)

Syntax

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

Types of Constructors

1. Default Constructor

If no constructor is written, Dart provides a default constructor automatically.

```
class Person {
  String name = "Sakif";
  void greet() {
    print("Hello, my name is $name");
  }
}

void main() {
  Person p = Person();  // Default constructor is called
  p.greet();
}
```

2. Parameterized Constructor

Used to assign values during object creation.

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

  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

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

void main() {
  Person p = Person("Sakif", 23);
  p.display();
}
```

3. Shorthand Constructor (Using this)

A shortened version of the parameterized constructor using this.

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

  Person(this.name, this.age);

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

4. Named Constructor

Allows multiple constructors with different names for flexibility.

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

  Student(this.name, this.age);

  Student.named() {
    name = "Default Name";
    age = 18;
  }

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

void main() {
  Student s1 = Student("Mishad", 22);
  s1.display();

  Student s2 = Student.named();
  s2.display();
}
```

5. Constant Constructor

Used to create compile-time constant objects. All fields must be marked as final.

```
class Point {
  final int x;
  final int y;

  const Point(this.x, this.y);
}

void main() {
  const p1 = Point(1, 2);
  const p2 = Point(1, 2);

  print(identical(p1, p2));  // true
}
```

Summary Table

Constructor Type	Description	Example Syntax
Default Constructor	Auto-created if no constructor is defined	ClassName()
Parameterized	Takes values via parameters	ClassName(type param)
Shorthand	Uses this.param syntax for assignment	ClassName(this.param)
Named Constructor	Alternative named constructor	ClassName.named()
Const Constructor	For immutable compile-time objects	const ClassName(param)

Advanced Constructor Parameters in Dart

1. Positional Parameters (Default Style)

This is the basic constructor style, where parameters are passed in order.

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

  Person(this.name, this.age);
}
```

Usage:

```
var p = Person("Sakif", 23);
```

Both parameters must be passed in the correct order.

2. Optional Positional Parameters [ ]

These parameters are optional and passed **in position**, using square brackets.

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

  Person(this.name, [this.age]);

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

Usage:

```
Person p1 = Person("Sakif", 23);
Person p2 = Person("Mishad"); // age will be null
```

3. Named Parameters { }

Named parameters allow arguments to be passed by **name**, not position.
They use **curly braces**.

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

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

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

Usage:

```
Person p = Person(name: "Sakif", age: 23);
```

4. Optional Named Parameters (Without required)

Named parameters become optional if required is not used. Fields should be nullable or have default values.

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

  Person({this.name, this.age});
}
```

Usage:

```
Person p1 = Person();  // name and age are null
Person p2 = Person(age: 25);  // only age is provided
```

5. Combining Default Values with Named Parameters

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

  Person({this.name = "Guest", this.age = 18});

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

Usage:

```
Person p = Person();  // Uses default values
```

Summary Table

Syntax	Meaning	Example
(this.a, this.b)	Positional, required	Person("Sakif", 22)
([this.a, this.b])	Positional, optional	Person("Sakif")
({this.a, this.b})	Named, optional	Person(name: "Sakif")
({required this.a})	Named, required	Person(name: "Sakif", age: 23)
({this.a = val})	Named with default values	Person()

Conclusion

- Use **required** for named parameters that must be given.
- Use **{}** for named parameters (good for clarity and order doesn't matter).
- Use **[]** for optional positional parameters (less readable for many params).
- Combine with default values or nullable types to make constructors flexible and safe.

Encapsulation in Dart (OOP)

What is Encapsulation?

Encapsulation is one of the four core principles of Object-Oriented Programming (OOP).
It refers to the concept of **hiding internal data** of a class and only exposing it through **controlled access methods** such as **getters** and **setters**.

Purpose of Encapsulation

- To protect data from direct access or modification
- To enforce data validation rules
- To keep the class self-contained
- To improve code reusability, security, and maintainability

Encapsulation in Dart

Dart achieves encapsulation using:

1. **Private variables** — by prefixing variable names with an underscore _
2. **Getter methods** — to read private values
3. **Setter methods** — to update private values with validation if needed

Without Encapsulation (Not Recommended)

```
class BankAccount {
  double balance = 0;
}

void main() {
  BankAccount account = BankAccount();
  account.balance = -1000; // Direct access; no validation
}
```

This allows invalid values to be set, which may break business rules.

With Encapsulation (Recommended)

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

  // Getter
  double get balance => _balance;

  // Setter with validation
  set balance(double amount) {
    if (amount >= 0) {
      _balance = amount;
    } else {
      print("Invalid balance. Cannot be negative.");
    }
  }
}

void main() {
  BankAccount account = BankAccount();
  account.balance = 5000;      // Uses setter
  print(account.balance);      // Uses getter, prints: 5000
  account.balance = -1000;     // Rejected due to validation
  print(account.balance);      // Still 5000
}
```

Custom Getter and Setter Example

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

  // Getter
  String get name => _name;
```

```
  // Setter with validation
  set name(String newName) {
    if (newName.length > 1) {
      _name = newName;
    } else {
      print("Name must be at least 2 characters long.");
    }
  }
}

void main() {
  Student s = Student();
  s.name = "A";      // Invalid
  s.name = "Sakif";  // Valid
  print(s.name);     // Output: Sakif
}
```

Summary Table

Concept	Description	Example
Private Variable	Prevents direct access outside the class	double _balance = 0;
Getter	Provides read-only access	double get balance => _balance;
Setter	Provides controlled write access	set balance(double val) { ... }

Key Benefits of Encapsulation

- Prevents invalid or unauthorized access to data
- Allows validation before data is changed
- Keeps object behavior predictable and secure
- Promotes modular and organized code

Inheritance in Dart (OOP)

What is Inheritance?

Inheritance is an object-oriented programming (OOP) concept where **one class (child/subclass)** can **inherit** properties and methods from **another class (parent/superclass)**.

It allows code reuse, avoids duplication, and supports hierarchical relationships between classes.

Key Terminology

Term	Meaning
Superclass	The parent class being inherited from
Subclass	The child class that inherits from the superclass
extends	The keyword used to perform inheritance in Dart
super	Used inside subclass to refer to the superclass constructor or members

Basic Syntax

```
class Parent {
  void greet() {
    print("Hello from Parent");
  }
}

class Child extends Parent {
  void sayName() {
    print("This is Child");
  }
}
```

```
void main() {
  Child obj = Child();
  obj.greet();     // Inherited from Parent
  obj.sayName();   // Defined in Child
}
```

Output:

```
Hello from Parent
This is Child
```

Types of Inheritance in Dart

Dart supports **Single Inheritance** but allows **mixins** and **interfaces** to simulate multiple inheritance behavior.

1. Single Inheritance (One class inherits from one parent)

```
class Animal {
  void eat() => print("Eating...");
}

class Dog extends Animal {
  void bark() => print("Barking...");
}

void main() {
  Dog d = Dog();
  d.eat();   // Inherited
  d.bark();  // Own method
}
```

2. Multilevel Inheritance (Child inherits from a child class of another class)

```
class Grandparent {
  void heritage() => print("Old Traditions");
}

class Parent extends Grandparent {
  void teach() => print("Parent Teaching");
}

class Child extends Parent {
  void learn() => print("Child Learning");
}

void main() {
  Child c = Child();
  c.heritage(); // From Grandparent
  c.teach();    // From Parent
  c.learn();    // From Child
}
```

3. Hierarchical Inheritance (Multiple classes inherit from the same parent)

```
class Vehicle {
  void start() => print("Vehicle started");
}

class Car extends Vehicle {
  void carFeature() => print("Has AC");
}

class Bike extends Vehicle {
  void bikeFeature() => print("Has two wheels");
}
```

```
void main() {
  Car car = Car();
  car.start();
  car.carFeature();

  Bike bike = Bike();
  bike.start();
  bike.bikeFeature();
```

Constructor in Inheritance

```
class Parent {
  Parent() {
    print("Parent Constructor");
  }
}

class Child extends Parent {
  Child() {
    print("Child Constructor");
  }
}

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

Output:

Parent Constructor

Child Constructor

Dart automatically calls the superclass constructor first.

Using super Keyword

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

  void show() {
    print("This is $name");
  }
}

class Child extends Parent {
  String name = "Child";

  void showNames() {
    print(name);      // Child
    print(super.name); // Parent
  }
}

void main() {
  Child c = Child();
  c.showNames();
}
```

Method Overriding in Dart

Method overriding allows a subclass to redefine a method inherited from its superclass, providing specialized behavior

// The subclass **replaces** or **modifies** the behavior of a method that was originally defined in its **superclass**.

🔧 Code Example

```
// Parent class
class Vehicle {
  String brand;
  String model;
  int year;

  Vehicle(this.brand, this.model, this.year);

  void displayInfo() {
    print("Brand : $brand");
    print("Model : $model");
    print("Year : $year");
  }

  double calculateRentalPrice(int days) {
    return days * 50; // Base price per day
  }
}

// Subclass
class Car extends Vehicle {
  int door;
  Car(String brand, String model, int year, this.door) : super(brand, model, year);
```

```
  @override
  double calculateRentalPrice(int days) {
    return super.calculateRentalPrice(days) + 20 * door;
  }
}

void main() {
  Car car = Car("Toyota", "Camry", 2024, 5);
  print("Car Information:");
  car.displayInfo();
  print("Rental price for 5 days: \$${car.calculateRentalPrice(5)}");
}
```

Explanation

- The class Vehicle defines a method calculateRentalPrice to return a base rental cost.
- The class Car inherits from Vehicle and overrides calculateRentalPrice using the @override annotation.
- Within the overridden method, super.calculateRentalPrice(days) calls the original logic from the parent class, and 20 * door adds additional cost based on the number of doors.

🖨️ **Output**

Car Information:

Brand : Toyota

Model : Camry

Year : 2024

Rental price for 5 days: $350

(Calculation: 5 × 50 = 250 base + 5 × 20 = 100 door charge → total = 350)

Summary Table

Concept	Description
@override	Declares that a method from the parent class is being redefined
super.method()	Calls the original method from the superclass
Method Signature	Must be identical to the parent method (name, parameters, return type)
Purpose	Enables subclasses to provide specialized behavior while reusing logic

Use Cases for Method Overriding

- Customizing inherited behavior
- Extending base functionality with subclass-specific logic
- Implementing polymorphism in class hierarchies

Access Modifiers in Dart (Important in Inheritance)

Modifier	Description	Can be Inherited?
Public	Normal members (no _)	Yes
Private	Members starting with _ (file-private)	No (not inherited outside the file)

Inheritance vs Composition

Concept	Meaning
Inheritance	"is-a" relationship
Composition	"has-a" relationship (using classes inside classes)

Example:

- A Car **is a** Vehicle → Inheritance
- A Car **has a** Engine → Composition

Key Points

- Dart supports **single inheritance** only.
- A subclass can override methods from the superclass.
- Use super to access superclass members.
- Private members (_var) are not inherited.
- Constructors in parent classes are automatically called first.

Summary Table

Feature	Example Syntax
Inheritance	class Child extends Parent
Call superclass method	super.methodName()
Override method	@override
Constructor order	Parent constructor → Child

Polymorphism in Dart (OOP)

What is Polymorphism?

Polymorphism is an Object-Oriented Programming (OOP) concept that allows **one method or object to behave in different ways** based on the context.

The word *polymorphism* comes from Greek:

- *Poly* = many
- *Morph* = form
  → **"Many Forms"**

Why Use Polymorphism?

- To **reuse** code across different classes
- To write **flexible and extendable** programs
- To allow the **same method name** to work differently for different object types
- To support **runtime method resolution** (via overridden methods)

Types of Polymorphism in Dart

Dart supports **two main types**:

Type	Description
Compile-time Polymorphism	Achieved via method overloading (not directly supported in Dart)
Runtime Polymorphism	Achieved via method overriding (supported in Dart)

1. Runtime Polymorphism (Method Overriding)

Dart supports **runtime polymorphism** through **method overriding**, where a subclass provides a new implementation for a method defined in the parent class.

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

class Dog extends Animal {
  @override
  void makeSound() {
    print("Dog barks");
  }
}

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

void main() {
// Polymorphism: A parent class reference (Animal) can hold child class objects (Dog, Cat).
  // The method that gets executed depends on the actual object type, not the reference type.
  Animal animal;
  animal = Dog();
  animal.makeSound();  // Output: Dog barks
  animal = Cat();
  animal.makeSound();  // Output: Cat meows
}
```

Explanation:

- The same variable animal refers to different object types (Dog, Cat).
- The method makeSound() behaves **differently** depending on the actual object — this is **polymorphism in action**.

2. Compile-Time Polymorphism (Not directly supported)

Dart does **not support method overloading** by default that means we **cannot create multiple methods with the same name and different parameters** in the same class like in Java or C++.

However, it can be simulated using optional parameters:

```
class Calculator {
  void add(int a, [int? b]) {
    if (b != null) {
      print("Sum = ${a + b}");
    } else {
      print("Value = $a");
    }
  }
}

void main() {
  Calculator calc = Calculator();
  calc.add(10);       // Output: Value = 10
  calc.add(10, 20);   // Output: Sum = 30
}
```

Real-World Analogy

Imagine a **"draw()"** method:

- A Circle class can override it to draw a circle
- A Square class can override it to draw a square

We will call the same method draw(), but the result is different depending on the object. That is **polymorphism**.

Summary Table

Concept	Supported in Dart?	Description
Method Overriding	Yes	Redefining a parent method in a subclass
Method Overloading	No (not directly)	Multiple methods with same name (simulate using optional params)
Polymorphism (runtime)	Yes	Subclass behavior via overridden methods

Key Benefits of Polymorphism

- Improves **code readability and organization**
- Supports **interface-based programming**
- Reduces code duplication
- Encourages **open-closed principle** (open for extension, closed for modification)

Polymorphism vs Inheritance vs Abstraction

Concept	Description
Inheritance	Allows a class to inherit from another class
Abstraction	Hides complexity using abstract classes or interfaces
Polymorphism	Allows objects to behave differently via overridden methods

Abstraction in Dart (OOP)

What is Abstraction?

Abstraction is an OOP principle that allows defining the **structure of a class without implementing all its functionality**.
It helps to **hide internal details** and only expose the essential features.

In Dart, **abstraction is achieved using the abstract keyword**.

Key Characteristics of Abstraction in Dart

- An **abstract class** cannot be instantiated (i.e., no objects can be created from it directly).
- It can contain **abstract methods** (methods without a body) and **non-abstract methods**.
- Classes that **extend** an abstract class must **override and implement** its abstract methods.
- Abstract classes act as **blueprints** or **contracts** for subclasses.

Example: Abstract Class with Implementation

```
abstract class BaseApiServices {
  void postApi(var data);   // Abstract method
  void getApi();            // Abstract method
}
```

- BaseApiServices is an abstract class.
- It declares two methods: postApi() and getApi() — without implementations.

Implementing the Abstract Class

```
class NetworkServicesApi extends BaseApiServices {
  String name;

  NetworkServicesApi(this.name);  // Constructor

  @override
  void postApi(var data) async {
    print("post api hit");
    await Future.delayed(Duration(seconds: 2));
    print("User Logged in");
    print(data['email']);
  }

  @override
  void getApi() {
    print("get api hit");
  }
}
```

- The class NetworkServicesApi **extends** BaseApiServices.
- It provides concrete implementations of both postApi() and getApi().

Usage in Main Function

```
void main() {
  NetworkServicesApi networkServicesApi = NetworkServicesApi("test");

  Map<String, String> data = {
    'email': 'test@gmail.com',
    'password': '123123'
  };

  networkServicesApi.postApi(data);
}
```

- An object of NetworkServicesApi is created and its postApi() method is called.
- Output:
- post api hit
- User Logged in
- test@gmail.com

Key Points

Feature	**Description**
abstract class	Defines a class that cannot be instantiated directly
Abstract methods	Methods without implementation (no body)
Implementation	Must be provided in subclasses
@override	Used to indicate that a method is being redefined
Purpose	Enforces a contract-like structure across different implementations

Summary Table

Term	Explanation
Abstract Class	A class with at least one method without implementation
Concrete Class	A class that implements all abstract methods from its superclass
Instantiation Allowed?	No, objects cannot be created directly from an abstract class
Inheritance Syntax	class Child extends AbstractClass

Use Case of Abstraction

- Creating APIs and service layers where multiple implementations might exist.
- Defining consistent behavior across different modules.
- Enforcing architectural contracts (e.g., repositories, data sources, service layers).

Polymorphism with Abstract Classes

Polymorphism also works with abstract classes and interfaces.

```
abstract class Shape {
  void draw();
}

class Circle extends Shape {
  @override
  void draw() {
    print("Drawing Circle");
  }
}

class Square extends Shape {
  @override
  void draw() {
    print("Drawing Square");
  }
}

void main() {
  Shape shape;

  shape = Circle();
  shape.draw();  // Drawing Circle

  shape = Square();
  shape.draw();  // Drawing Square
}
```

Interface in Dart (OOP)

What is an Interface?

An **interface** in object-oriented programming defines a set of **methods that a class must implement**.

It acts as a **contract**: any class implementing an interface **must override** and provide a concrete implementation of its methods.

In Dart:

- Every **class can act as an interface**.
- Dart does **not have a separate interface keyword**.
- Interfaces are implemented using the implements keyword.
- Unlike extends, which allows inheritance of methods, implements **forces full method reimplementation**.

Example 1: Using a Class as an Interface

```
class Laptop {
  void turnOn() {
    print("Laptop turn on");
  }

  void turnOff() {
    print("Laptop turn off");
  }
}

class MacBook implements Laptop {
  @override
  void turnOn() {
    print("Macbook turn on");
  }
```

```
  @override
  void turnOff() {
    print("Macbook turn off");
  }
}

void main() {
  MacBook macBook = MacBook();
  macBook.turnOn();   // Output: Macbook turn on
  macBook.turnOff();  // Output: Macbook turn off
}
```

Key Observations:

- MacBook is **not extending**, but **implementing** Laptop.
- MacBook must override **all methods** from Laptop, even if they have default implementations.
- This provides **strong abstraction** and **complete control** over the new behavior.

Example 2: Interface via Abstract Class

```
abstract class Animal {
  void sound();  // abstract method

  void eat() {
    print("The animal is eating"); // concrete method
  }
}

class Dog extends Animal {
  @override
  void sound() {
    super.eat();       // calling superclass method
    print("Dog Barks");
  }
```

```
}

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

void main() {
  Dog dog = Dog();
  dog.sound();   // Output: The animal is eating\nDog Barks
}
```

Key Observations:

- Animal is an **abstract class** with both **abstract** and **concrete** methods.
- Dog and Cat extend Animal and provide their own implementations of sound().
- In this approach, **concrete methods can be reused**, unlike with implements.

Syntax Comparison

Purpose	**Keyword**	**Behavior**
Inheritance	extends	Inherits methods and properties, can override selectively
Interface	implements	Must override **all** methods from the interface class

Interface vs Abstract Class in Dart

Feature	**Interface (implements)**	**Abstract Class (abstract)**
Syntax	class A implements B	abstract class A {}
Can contain body?	No, must reimplement everything	Yes, can include both abstract and concrete methods
Allows partial override?	No (must override all)	Yes (can override only necessary ones)
Multiple implementation?	Yes (comma-separated interfaces)	Dart does not support multiple inheritance

Implementing Multiple Interfaces

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

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

class C implements A, B {
  @override
  void methodA() => print("C overrides A");

  @override
  void methodB() => print("C overrides B");
}
```

- A class can implement **multiple interfaces** separated by commas.
- All methods from all interfaces must be overridden.

Summary Table

Concept	**Description**
Interface in Dart	Any class used with implements to enforce method signatures
Implementation	All methods must be overridden
Abstract class use	Can act as interface, but also contain reusable logic
implements keyword	Forces a class to fulfill all method requirements

---

Key Use Cases of Interfaces

- Creating service contracts (e.g., AuthService, DatabaseService)
- Enforcing consistent method structure across unrelated classes
- Supporting **polymorphism** by treating multiple objects through a shared interface

Polymorphism vs Inheritance vs Abstraction

Concept	Description
Inheritance	Allows a class to inherit from another class
Abstraction	Hides complexity using abstract classes or interfaces
Polymorphism	Allows objects to behave differently via overridden methods

Mixins in Dart (OOP)

What is a Mixin?

A **mixin** is a class-like structure used to **reuse code across multiple classes** without using inheritance.
It allows a class to "borrow" functionality from one or more mixins without forming a parent-child relationship.

Dart uses the mixin keyword to define mixins and the with keyword to apply them.

Key Characteristics of Mixins

- Mixins **cannot be instantiated**.
- Mixins do **not have constructors**.
- Multiple mixins can be applied to a single class.
- The class that uses mixins retains its own inheritance chain.

Syntax

```
mixin MixinName {
```

```
  // methods or properties to reuse
}

class SomeClass with MixinName {
  // can use methods from MixinName
}
```

Example Explanation

Mixins Defined

```
mixin Logger {
  void log(String message) {
    print(message);
  }
}

mixin validation {
  String? validatePassword(String value) {
    if (value.isEmpty) {
      return "Password cannot be empty";
    }
    if (value.length < 6) {
      return "Password cannot be less than 6";
    }
    return null;
  }
}
```

- Logger provides a method for logging messages.
- validation provides a method to validate password strings.

Class Using Mixins

```
class Person with Logger, validation {
  String email;
  String password;

  Person(this.email, this.password);

  void displayInfo() {
    if (validatePassword(password) != null) {
      log(validatePassword(password).toString());
    }
    log("User email ${email} and password is ${password}");
  }
}
```

- The Person class **uses** both Logger and validation mixins.
- This allows Person to access log() and validatePassword() methods **as if they were defined in the class**.

Main Function and Output

```
void main() {

  Person person = Person('test@gmail.com', '1234');

  person.displayInfo();

}
```

Output:

Password cannot be less than 6

User email test@gmail.com and password is 1234

Why Use Mixins?

- To avoid **code duplication**
- To **separate reusable functionality** (e.g., logging, validation, caching)
- To compose classes with **multiple behaviors** without deep inheritance trees

Comparison: Inheritance vs Mixin

Feature	Inheritance (extends)	Mixin (with)
Relationship	Parent-child (is-a)	Behavior sharing (has functionality)
Constructors	Inherited	Not allowed in mixins
Method override	Yes	Yes
Multiple usage	Single parent only	Multiple mixins allowed

Mixin Rules in Dart

- Must be declared using mixin keyword (since Dart 2.1)
- Cannot have constructors
- Can implement interfaces or extend Object
- Must not extend or be extended like a normal class

Multiple Mixins Order

class A with M1, M2 {}

- If both mixins define the same method, the **last one (M2)** takes precedence.

Summary Table

Concept	**Keyword**	**Purpose**
Mixin	mixin	Define reusable behavior
Apply mixins	with	Apply one or more mixins to a class
Limitation	No constructor, no instantiation	

Use Cases

- Logging (Logger)
- Input validation (validation)
- Caching
- Error handling
- Analytics or tracking

Enums in Dart

What is an Enum?

Enum (short for *enumeration*) is a **special type** that represents a **fixed set of constant values**.

Enums are commonly used to define a group of related options or states (e.g., days of the week, user roles, network statuses, gender, etc.).

Syntax

enum EnumName { value1, value2, value3 }

- **Example**

enum Gender { Male, Female, Others }
enum Status { loading, error, success }

- Gender has 3 possible values: Male, Female, and Others.
- Status has 3 possible values: loading, error, and success.

Using Enums in a Class

```
class Person {
  String name;
  Gender gender;

  Person(this.name, this.gender);
}
```

- Here, gender is a variable of type Gender.
- It must be assigned one of the enum values: Gender.Male, Gender.Female, or Gender.Others.

Using Enums in a switch Statement

```
void main() {
  Person person = Person("Mishad", Gender.Female);

  switch (person.gender) {
    case Gender.Male:
      print("Male");
      break;
    case Gender.Female:
      print("Female");
      break;
    case Gender.Others:
      print("Others");
      break;
  }
}
```

Output:

Female

- switch-case is often used with enums for decision-making.
- Each case must include a break (unless returning immediately or falling through intentionally).

Iterating Over Enum Values

```
for (var value in Gender.values) {
  print(value);         // Output: Gender.Male, Gender.Female, etc.
  print(value.name);    // Output: Male, Female, etc.
}
```

Enum in Real-World Use Cases

- Status for API states (loading, success, error)
- Gender in user profiles
- UserRole for access control
- OrderStatus in e-commerce apps

Dart Variable and Value Declarations

This document provides a comprehensive overview of all declaration types in Dart, including var, final, const, late, dynamic, Object, nullable types, and static.

1. var

- Automatically infers the variable's type from the initial value.
- Once assigned, the type is fixed and cannot be changed.
- Mutable (can be reassigned with the same type).

```
var name = "Mishad";   // Inferred as String
name = "Sakif";        // Valid
// name = 10;          // Invalid: Cannot assign int to String
```

2. Explicit Type Declaration

- Used to declare variables with a specific type.
- Ensures strict type safety.

```
String name = "Sakif";
int age = 25;
bool isStudent = true;
```

3. dynamic

- Disables static type checking.
- Allows assigning values of any type at runtime.
- Type can change at any point.

```
dynamic data = "Text";
data = 42;         // Valid
data = true;       // Valid
```

4. Object

- Can hold values of any type, but retains some level of type safety.
- Safer alternative to dynamic.

```
Object value = "Text";
value = 30;        // Valid
value = false;     // Valid
```

5. Nullable Types (?)

- Allows variables to hold null.
- Used when a value is optional or uninitialized.

```
String? username = null;
int? number;
```

6. final

- Value must be assigned only once.
- Initialization occurs at runtime.
- Cannot be reassigned afterward.

```
final country = "Bangladesh";
// country = "India"; // Error: final variable cannot be reassigned
```

7. const

- Value must be assigned at compile time.
- Cannot be reassigned or changed.
- Used for compile-time constants only.

```
const pi = 3.1416;
const List<String> cities = ["Dhaka", "Chittagong"];
```

8. late

- Used to declare a non-nullable variable that will be initialized later.
- Avoids immediate assignment during declaration.
- Must be assigned before being accessed.

```
late String token;

void initialize() {
  token = "abc123";
}

void main() {
  initialize();
  print(token);
}
```

9. late final

- Combines the features of late and final.
- Allows assignment later, but only once.

```
late final String userId;

void setup() {
  userId = "U001";
}
```

10. static

- Declares class-level members.
- Shared among all instances of the class.
- Commonly used for constants, utility functions, and counters.

```
class Config {
  static const String apiUrl = "https://example.com";
}

void main() {
  print(Config.apiUrl);
}
```

11. Declaration Comparison Table

Keyword	Type Inferred	Reassignable	Initialized At	Nullable	Notes
var	Yes	Yes	Declaration	No	Fixed type after first assignment
dynamic	Yes	Yes	Any time	Yes	Disables static type safety
Explicit type	No	Yes	Declaration	Yes/No	Strongly typed
final	Optional	No	Runtime	No	Assigned once only
const	Optional	No	Compile time	No	Strictest: compile-time only
late	Optional	Yes/No	Before first use	No	Delayed initialization for non-nullables
late final	Optional	No	Before first use	No	Assigned once, after declaration
Nullable types	Optional	Yes	Any time	Yes	Allows null assignment
static	Optional	Varies	Class-level	Varies	Used inside classes

How Flutter Uses OOP Concepts Basic Idea

1. Class and Object

- **OOP Meaning**: Class is a blueprint; Object is an instance of a class.
- **Flutter Usage**: Every widget is a class, and every widget you place on screen is an object.

Example:

```
class MyTextWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text("Hello");
  }
}


void main() {
  runApp(MyTextWidget()); // Object created from MyTextWidget class
}
```

Used in: StatelessWidget, StatefulWidget, Text, Container, etc.

2. Inheritance (extends)

- **OOP Meaning**: A child class inherits properties/methods from a parent class.
- **Flutter Usage**: StatelessWidget and StatefulWidget are subclasses of Widget.

Example:

```
class MyBox extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(); // Container is also a Widget (inherited)
  }
}
```

✅ **Used in**: StatelessWidget extends Widget, State extends StatefulWidget, and most custom widgets.

3. Method Overriding (@override)

- **OOP Meaning**: A subclass redefines a method of its parent class.
- **Flutter Usage**: Most commonly, widgets override build() to return UI.

Example with StatelessWidget:

```
class MyText extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text("Overridden build");
  }
}
```

Example with StatefulWidget:

```
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState(); // Overriding createState
}

class _CounterState extends State<Counter> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return Text('Count: $count'); // Overriding build
  }
}
```

Used in:

- build() method in both widgets
- createState() in StatefulWidget

4. Encapsulation

- **OOP Meaning**: Hiding internal details using private variables and methods.
- **Flutter Usage**: The _ prefix is used to **hide widget state** and logic from the outside.

Example:

```
class _MyPrivateWidget extends StatelessWidget {
  final String _message = "Hidden inside widget";

  @override
  Widget build(BuildContext context) {
    return Text(_message);
  }
}
```

Used in:

- State classes like _MyWidgetState (starts with _)
- Private variables like _counter

5. Polymorphism

- **OOP Meaning**: One interface, many implementations — same method behaves differently based on object type.
- **Flutter Usage**:
    - A Widget list can hold any type of widgets: Text, Icon, Row, etc.
    - All are treated as Widget, but behave differently.

Example:

```
List<Widget> items = [
  Text("One"),
  Icon(Icons.star),
  ElevatedButton(onPressed: () {}, child: Text("Tap"))
];


Column(children: items); // All widgets are treated polymorphically
```

Used in:

- List<Widget> holding different subclasses
- Layouts like Row, Column, Stack, etc.

6. Abstraction

- **OOP Meaning**: Hiding complex logic behind simple interfaces.
- **Flutter Usage**: Flutter widgets like Text, ElevatedButton, TextField provide a simple interface to build complex UIs.

Example:

TextField() // Internally has rendering, controller logic, focus, etc.

- Developer just calls TextField(), no need to understand inner logic.

Used in:

- All high-level Flutter widgets
- Built-in classes like StatefulWidget, Theme, Navigator

Summary Table (with Widget Examples)

OOP Concept	**Flutter Widget Example**	**Description**
Class/Object	Text(), MyWidget()	Every widget is a class and its use on screen is an object
Inheritance	class MyWidget extends StatelessWidget	Inherits from base Widget class
Overriding	@override Widget build(BuildContext)	Redefining parent method to return widget tree
Encapsulation	_MyState, _counter	Hiding state variables and logic
Polymorphism	List<Widget> with Text, Icon, Button	One interface (Widget), multiple forms
Abstraction	TextField(), AppBar()	Easy-to-use widgets hide complex internal logic

Real Widget Lifecycle using OOP

```
class MyCounter extends StatefulWidget {
  @override
  _MyCounterState createState() => _MyCounterState(); // Inheritance + Override
}

class _MyCounterState extends State<MyCounter> {
  int _count = 0; // Encapsulated

  void _increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text("Count: $_count"),
        ElevatedButton(onPressed: _increment, child: Text("Add")),
      ],
    );
  }
}
```