Object Oriented Programming in Dart

💡 **Object-oriented programming (OOP)** is a programming method that uses objects and their interactions to design and program applications.

Advantages

- It increases reusability and decreases complexity. (It reduces the repetition of code)
- It makes the code easier to maintain, modify and debug.
- OOP is based on objects, which are data structures containing data and methods.
- OOP is a way of thinking about programming that differs from traditional procedural programming.
- OOP can make code more modular, flexible, and extensible.

📌 **SUMMARY:** The main purpose of OOP is to break complex problems into smaller objects.

Class

→ In object-oriented programming, a class is a blueprint for creating objects. A class defines the properties and methods that an object will have.

→ You can declare a class in dart using the **class** keyword followed by class name and braces {}. It's a good habit to write class name in **PascalCase**.

```
class ClassName {
// properties or fields are used to store the data.
```

```
// methods or functions are used to perform the operations.
}

class Person {
  String? name;
  String? phone;
  bool? isMarried;
  int? age;

  void displayInfo() {
    print("Person name: $name.");
    print("Phone number: $phone.");
    print("Married: $isMarried.");
    print("Age: $age.");
  }
}
```

Object

→ **In object-oriented programming**, an object is a self-contained unit of code and data. Objects are created from templates called classes. An object is an instance of a class.

→ You can create many objects of a class. Each object will have its own copy of the properties.

→ In object-oriented programming, instantiation is the process of creating an instance of a class. In other words, you can say that instantiation is the process of creating an object of a class.

```
ClassName objectName = ClassName();

// Here bicycle is object of class Bicycle.
Bicycle bicycle = Bicycle();
```

Constructor

→ **A constructor** is a special method used to initialize an object. It is called automatically when an object is created, and it can be used to set the initial values for the object's properties.

→ If you don't define a constructor for class, then you need to set the values of the properties manually.

→ The constructor's name should be the same as the class name.

→ Constructor doesn't have any return type.

```
class ClassName {
  // Constructor declaration: Same as class name
  ClassName() {
    // body of the constructor
  }
}

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

  // Constructor
  Student(String name, int age, int rollNumber) {
    // Checking the constructor is called or not.
    print(
        "Constructor called");
  // This keyword refer to the property of the class
        this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
}
```

```
void main() {
  // Here student is object of class Student.
  Student student = Student("John", 20, 1);
}
```

→ Other constructor types

```
// Constructor in short form
Person(this.name, this.age, this.subject, this.salary);
//Call
Person person = Person("John", 30, "Maths", 50000.0);

// Constructor with optional parameters
Employee(this.name, this.age, [this.subject = "N/A",
this.salary=0]);
//Call
Employee employee = Employee("John", 30);

// Constructor with named parameters
Chair({this.name, this.color});
//Call
Chair chair = Chair(name: "Chair1", color: "Red");

// Constructor with named parameters
Student({String? name, int? age, int? rollNumber})
//Call
Student student = Student(name: "Joe", age: 21,
rollNumber : 15);

// Constructor with named parameters
Student.namedConstructor(String name, int age,
int rollNumber) {
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
// Call
```

```
Student student = Student.namedConstructor("John", 20, 1);

// Named constructor
Mobile.namedConstructor(this.name, this.color,
[this.price = 0]);
// Call
var mobile2 = Mobile.namedConstructor("Apple", "White");

// Constructor with default parameters
Student({String? name = "John", int? age = 0})
// Call
Student student = Student();
```

Default Constructor

→ The constructor which is automatically created by the dart compiler if you don’t create a constructor is called a default constructor. A default constructor has no parameters

```
// Constructor
  Laptop() {
    print("This is a default constructor");
  }
```

Encapsulation

→ **Encapsulation** means **hiding data** within a library, preventing it from outside factors. It helps you control your program and prevent it from becoming too complicated.

→ Why encapsulation is important?

- **Data Hiding**: Encapsulation hides the data from the outside world. It prevents the data from being accessed by the code outside the class. This is known as data hiding.
- **Testability**: Encapsulation allows you to test the class in isolation. It will enable you to test the class without testing the code outside the class.

- **Flexibility**: Encapsulation allows you to change the implementation of the class without affecting the code outside the class.
- **Security**: Encapsulation allows you to restrict access to the class members. It will enable you to limit access to the class members from the code outside the library.

→ Dart doesn't support keywords like **public**, **private**, and **protected**. Dart uses **_** (underscore) to make a property or method private.

- Declaring the class properties as **private** by using **underscore(_)**.
- Providing public **getter** and **setter** methods to access and update the value of private property.

→ **Getter** and **setter** methods are used to access and update the value of private property. **Getter** methods are used to access the value of private property. **Setter** methods are used to update the value of private property.

Use of Getter and Setter

→ Advantages of the getter and setter

- Validate the data before reading or writing.
- Restrict the read and write access to the properties.
- Making the properties read-only or write-only.
- Perform some action before reading or writing the properties.

```
class Employee {
  // Private properties
  int? _id;
  String? _name;

// Getter method to access private property _id
  int getId() {
    return _id!;
  }
```

```
// Getter method to access private property _name
  String getName() {
    return _name!;
  }
// Setter method to update private property _id
  void setId(int id) {
    this._id = id;
  }
// Setter method to update private property _name
  void setName(String name) {
    this._name = name;
  }

}

void main() {
  // Create an object of Employee class
  Employee emp = new Employee();
  // setting values to the object using setter
  emp.setId(1);
  emp.setName("John");

  // Retrieve the values of the object using getter
  print("Id: ${emp.getId()}");
  print("Name: ${emp.getName()}");
}
```

📌 **U**sing **underscore (_)** before a variable or method name makes it **library private** not **class private. Therefore, you need to put your class that has private properties apart from the file has main class.**

→ You can also define **getter** and **setter** using **get** and **set** keywords

```
class Vehicle {
  String _model;
  int _year;

  // Getter method
  String get model => _model;

  // Setter method
  set model(String model) => _model = model;

  // Getter method
  int get year => _year;

  // Setter method
  set year(int year) => _year = year;
}
```

Read-Only Properties

→ You can control the properties's access and implement the encapsulation in the dart by using the read-only properties. You can do that by adding the **final** keyword before the properties declaration.

```
class Student {
 // Late keyword also may be used
  final _schoolname = "ABC School";

  String getSchoolName() {
    return _schoolname;
  }
}
```

Inheritance (is-a relation)

→ Inheritance is a sharing of behaviour between two classes. It allows you to define a class that extends the functionality of another class.
The **extend** keyword is used for inheriting from parent class.

```
class ParentClass {
  // Parent class code
}

class ChildClass extends ParentClass {
  // Child class code
}


class Person {
  // Properties
  String? name;
  int? age;

  // Method
  void display() {
    print("Name: $name");
    print("Age: $age");
  }
}
// Here In student class, we are extending the
// properties and methods of the Person class
class Student extends Person {
  // Fields
  String? schoolName;
  String? schoolAddress;

  // Method
  void displaySchoolInfo() {
    print("School Name: $schoolName");
    print("School Address: $schoolAddress");
  }
}

void main() {
  // Creating an object of the Student class
  var student = Student();
  student.name = "John";
```

```
  student.age = 20;
  student.schoolName = "ABC School";
  student.schoolAddress = "New York";
  student.display();
  student.displaySchoolInfo();
}
```

Types of Inheritance

1. **Single Inheritance** - In this type of inheritance, a class can inherit from only one class. In Dart, we can only extend one class at a time.
2. **Multilevel Inheritance** - In this type of inheritance, a class can inherit from another class and that class can also inherit from another class. In Dart, we can extend a class from another class which is already extended from another class.
3. **Hierarchical Inheritance** - In this type of inheritance, parent class is inherited by multiple subclasses. For example, the **Car** class can be inherited by the **Toyota** class and **Honda** class.
4. **Multiple Inheritance** - In this type of inheritance, a class can inherit from multiple classes. **Dart does not support multiple inheritance.** For e.g. **Class Toyota extends Car, Vehicle {}** is not allowed in Dart.

📌 **Dart does not support multiple inheritance.**

→ Super keyword is used to call the method of the parent class. (Methods and constructors)

```
//Super in methods of a class
class Laptop {
  // Method
    void show() {
        print("Laptop show method");
    }
```

```
}

class MacBook extends Laptop {
    void show() {
    // Calling the show method of the parent class
        super.show();
        print("MacBook show method");
    }
}

void main() {
  // Creating an object of the MacBook class
  MacBook macbook = MacBook();
  macbook.show();
}

//Super in constructor
class Employee {
  // Constructor
  Employee(String name, double salary) {
    print("Employee constructor");
    print("Name: $name");
    print("Salary: $salary");
  }
}

class Manager extends Employee {
  // Constructor
  Manager(String name, double salary) : super(name, salary) {
    print("Manager constructor");
  }
}

void main() {
  Manager manager = Manager("John", 25000.0);
}
```

Polymorphism

→ Polymorphism is the ability of an object to take on many forms. As humans, we have the ability to take on many forms. We can be a student, a teacher, a parent, a friend, and so on.

Polymorphism by method overloading

→ Method overriding is a technique in which you can create a method in the child class that has the same name as the method in the parent class

```
class ParentClass{
void functionName(){
  }
}
class ChildClass extends ParentClass{
@override
void functionName(){
  }
}

// Example of method overriding
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

class Dog extends Animal {
  @override
  void eat() {
    print("Dog is eating");
  }
}

void main() {
  Animal animal = Animal();
  animal.eat();
```

```
  Dog dog = Dog();
  dog.eat();
}
```

Static Keyword

→ If you want to define a variable or method that is shared by all instances of a class, you can use the **static** keyword. Static members are accessed using the class name. It is used for **memory management**.

→ A static variable is a variable that is shared by all instances of a class. It is declared using the static keyword. It is initialized only once when the class is loaded. It is used to store the **class-level data**.

→ A static method is shared by all instances of a class. It is declared using the static keyword. You can access a static method without creating an object of the class.

→You don’t need to create an instance of a class to access a static variable or call a static method.

```
// Static variable declaration
class ClassName {
  static dataType variableName = value;
}

// Static variable access
ClassName.variableName

// Static method declaration
class ClassName{
static returnType methodName(){
  //statements
}}
```

```
//Static method access
ClassName.methodName();
```

Enum

→ An enum is a special type that represents a fixed number of constant values. An enum is declared using the keyword **enum** followed by the enum's name.

Advantages Of Enum In Dart

- It is used to define a set of named constants.
- Makes your code more readable and maintainable.
- It makes the code more reusable and makes it easier for developers.

Characteristics Of Enum

- It must contain at least one constant value.
- Enums are declared outside the class.
- Used to store a large number of constant values.

```
enum enumName {
  constantName1,
  constantName2,
  ...
  constantNameN
}

enum Gender { Male, Female, Other }

enum days {
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thrusday,
  Friday,
```

```
  Saturday
}
```

Enhanced Enum

→ Enhanced Enum is a generator for enum extensions and generators from strings. You can declare enums with members.

```
enum CompanyType {
  soleProprietorship("Sole Proprietorship"),
  partnership("Partnership"),
  corporation("Corporation"),
  limitedLiabilityCompany("Limited Liability Company");

  // Members
  final String text;
  const CompanyType(this.text);
}

void main() {
  CompanyType soleProprietorship =
                        CompanyType.soleProprietorship;
  print(soleProprietorship.text);
}
```

Abstract Class

→ Abstract classes are classes that cannot be initialized. It is used to define the behavior of a class that can be inherited by other classes.

→ You cannot create an object of abstract classes.

Subclasses of an abstract class must implement all the abstract methods of the abstract class. It is used to achieve abstraction

```
abstract class ClassName {
  //Body of abstract class
  method1();
```

```
  method2();
}
```

→ An abstract method is a method that is declared without an implementation. It is declared with a semicolon (;) instead of a method body.

```
abstract class Vehicle {
  // Abstract method
  void start();
  // Abstract method
  void stop();
}

class Car extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Car started');
  }

  // Implementation of stop()
  @override
  void stop() {
    print('Car stopped');
  }
}

class Bike extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Bike started');
  }

  // Implementation of stop()
  @override
  void stop() {
```

```
    print('Bike stopped');
  }
}

void main() {
  Car car = Car();
  car.start();
  car.stop();

  Bike bike = Bike();
  bike.start();
  bike.stop();
}
```

→ You can’t create an object of an abstract class. However, you can define a constructor in an abstract class. The constructor of an abstract class is called when an object of a subclass is created.

```
abstract class Bank {
  String name;
  double rate;

  // Constructor
  Bank(this.name, this.rate);

  // Abstract method
  void interest();

  //Non-Abstract method: It have an implementation
  void display() {
    print('Bank Name: $name');
  }
}

class SBI extends Bank {
  // Constructor
  SBI(String name, double rate) : super(name, rate);
```

```
  // Implementation of interest()
  @override
  void interest() {
    print('The rate of interest of SBI is $rate');
  }
}

class ICICI extends Bank {
  // Constructor
  ICICI(String name, double rate) : super(name, rate);

  // Implementation of interest()
  @override
  void interest() {
    print('The rate of interest of ICICI is $rate');
  }
}

void main() {
  SBI sbi = SBI('SBI', 8.4);
  ICICI icici = ICICI('ICICI', 7.3);

  sbi.interest();
  icici.interest();
  icici.display();
}
```

Interface

→ It is a contract that defines the capabilities of a class. It is used to achieve abstraction.

```
class InterfaceName {
  // code
}

class ClassName implements InterfaceName {
```

```
  // code
}
```

→ In dart there is no keyword **interface** but you can use **class** or **abstract class** to declare an interface. All classes implicitly define an interface. Mostly **abstract class** is used to declare an interface.

```
// creating an interface using abstract class
abstract class Person {
  canWalk();
  canRun();
}

class Student implements Person {
 // implementation of canWalk()
  @override
  canWalk() {
    print('Student can walk');
  }

// implementation of canRun()
  @override
  canRun() {
    print('Student can run');
  }
}
```

Multiple Inheritance

→ **Multiple inheritance** means a class can inherit from more than one class. In dart, you can't inherit from more than one class. But you can implement multiple interfaces in a class.

```
class ClassName implements Interface1, Interface2,
                Interface3 {
  // code
}
```

```
// abstract class as interface
abstract class Area {
  void area();
}
// abstract class as interface
abstract class Perimeter {
  void perimeter();
}
// implements multiple interfaces
class Rectangle implements Area, Perimeter {
    // properties
  int length, breadth;

 // constructor
  Rectangle(this.length, this.breadth);

// implementation of area()
  @override
  void area() {
      print('The area is ${length * breadth}');
  }
// implementation of perimeter()
  @override
  void perimeter() {
    print('The perimeter is ${2 * (length + breadth)}');
  }
}
```

extends	implements
Used to inherit a class in another class.	Used to inherit a class as an interface in another class.
Gives complete method definition to sub-class.	Gives abstract method definition to sub-class.
Only one class can be extended.	Multiple classes can be implemented.
It is optional to override the methods.	Concrete class must override the methods of an interface.
Constructors of the superclass is called before the sub-class constructor.	Constructors of the superclass is not called before the sub-class constructor.
The super keyword is used to access the members of the superclass.	Interface members can't be accessed using the super keyword.
Sub-class need not to override the fields of the superclass.	Subclass must override the fields of the interface.

Mixin

→ Mixins are a way of reusing the code in multiple classes. It is possible to use multiple mixins in a class.

- **Mixin** can't be instantiated. You can't create object of mixin.
- Use the **mixin** to share the code between multiple classes.
- **Mixin** has no constructor and cannot be extended.
- It is possible to use multiple **mixins** in a class.

→ The **with** keyword is used to apply the mixin to the class. It promotes DRY(Don't Repeat Yourself) principle.

```
mixin Mixin1{
  // code
}

mixin Mixin2{
  // code
}

class ClassName with Mixin1, Mixin2{
  // code
}
```

→ Sometimes, you want to use a mixin only with a specific class. In this case, you can use the **on** keyword.

```
abstract class Animal {
  // properties
  String name;
  double speed;

  // constructor
  Animal(this.name, this.speed);

  // abstract method
  void run();
}

// mixin CanRun is only used by class that extends Animal
mixin CanRun on Animal {
  // implementation of abstract method
  @override
  void run() => print('$name is Running at speed $speed');
}

class Dog extends Animal with CanRun {
  // constructor
  Dog(String name, double speed) : super(name, speed);
}

void main() {
  var dog = Dog('My Dog', 25);
  dog.run();
}

// Not Possible
// class Bird with Animal { }
```

→ What Is Allowed For Mixin

- You can add properties and static variables.
- You can add regular, abstract, and static methods.
- You can use one or more mixins in a class.

→ What Is Not Allowed For Mixin

- You can’t define a constructor.
- You can’t extend a mixin.
- You can’t create an object of mixin.

Factory Constructor

→ All of the constructors that you have learned until now are **generative constructors**. Dart also provides a special type of constructor called a **factory constructor**.

- Factory constructor must return an instance of the **class** or **sub-class**.
- You can’t use **this** keyword inside factory constructor.
- It can be **named** or **unnamed** and called like normal constructor.
- It can’t access **instance members** of the class.

→ A **factory constructor** gives more flexibility to create an object. Generative constructors only create an instance of the class. But, the factory constructor can return an instance of the **class or even subclass**. It is also used to return the **cached instance** of the class.

```
class ClassName {
  factory ClassName() {
    // TODO: return ClassName instance
  }

  factory ClassName.namedConstructor() {
    // TODO: return ClassName instance
  }
}
```

```
class Area {
  final int length;
  final int breadth;
  final int area;

  // private constructor
  const Area._internal(this.length, this.breadth) :
                                                    area =

  // Factory constructor
  factory Area(int length, int breadth) {
    if (length < 0 || breadth < 0) {
      throw Exception("Length and breadth must be positive");
    }
    // redirect to private constructor
    return Area._internal(length, breadth);
  }
}
```

Generics

→ Generics is a way to create a class, or function that can work with different types of data (objects).

```
// General Structure
class ClassName<T> {
  // code
}


class Data<T> {
  T data;
  Data(this.data);
}
// Call
Data<int> intData = Data<int>(10);
Data<double> doubleData = Data<double>(10.5);
```

```
// Dart implementation of Map class
abstract class Map<K, V> {
  // code
  external factory Map();
}

// Define generic method
T genericMethod<T>(T value) {
  return value;
}
```

→ Generics type variables are used to define the type of data that can be used with the class. In the above example, **T** is a type variable. You can use any name for the type variable. A few typical names are **T**, **E**, **K**, and **V**.

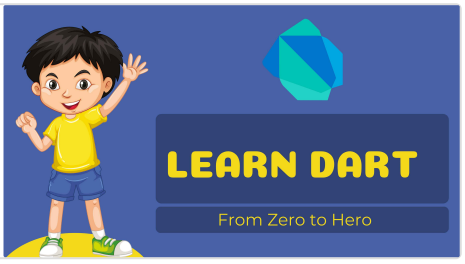
Name	Work
T	Type
E	Element
K	Key
V	Value

📚 References

OOP In Dart

Learn Dart Programming

https://dart-tutorial.com/object-oriented-programming/



✍🏽 Author → Serhat Kumas

https://www.linkedin.com/in/serhatkumas/

SerhatKumas - Overview

Computer engineering student who loves coding in different fields instead of focusing on a one spesific area. - SerhatKumas

https://github.com/SerhatKumas