

Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 1
Dart OOP Part 3

Lecture 7



Table Of Content

- Introduction
- Polymorphism
- Abstracts classes & implements
- Mixin
- Generics



Introduction

OOP

- ◆ Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.



OOP Concepts

01

Encapsulation

02

Inheritance

03

Polymorphism

04

Abstraction

Polymorphism

Polymorphism

Poly means **many** and morph means **forms**. 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. Similarly, in object-oriented programming, polymorphism is the ability of an object to take on many forms.

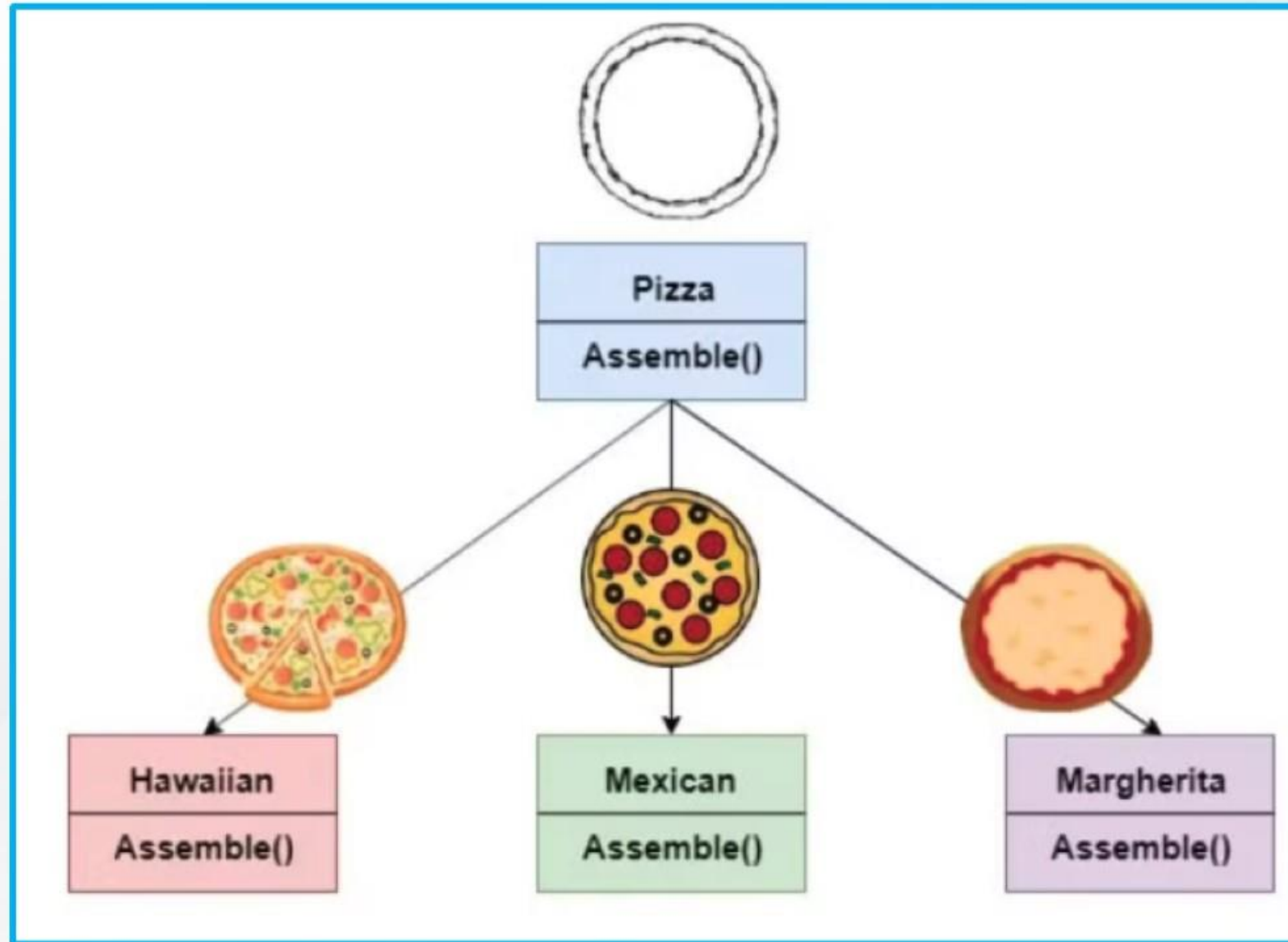
Note: In the real world, polymorphism is updating or modifying the feature, function, or implementation that already exists in the parent class.



Polymorphism

Suppose you're making pizza and wish to make 3 different varieties namely, Margherita, Hawaiian, and Mexican. Each pizza differs concerning the toppings it will have. Instead of writing a program for each variety, we create a class for each recipe and send one message to all, asking them to make the pizza.





```
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();
}
```

```
Name: John
Age: 20
School Name: ABC School
School Address: New York
```

Polymorphism

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. The method in the child class overrides the method in the parent class.

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



Polymorphism

In this example below, there is a class named **Animal** with a method named **eat()**. The **eat()** method is overridden in the child class named **Dog**.

```
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();  
}
```

Animal is eating
Dog is eating



```
class Vehicle {  
    void run() {  
        print("Vehicle is running");  
    }  
}  
  
class Bus extends Vehicle {  
    @Override  
    void run() {  
        print("Bus is running");  
    }  
}  
  
void main() {  
    Vehicle vehicle = Vehicle();  
    vehicle.run();  
  
    Bus bus = Bus();  
    bus.run();  
}
```

Vehicle is running
Bus is running

```
class Car{
    void power(){
        print("It runs on petrol.");
    }
}

class Honda extends Car{
}

class Tesla extends Car{
    @override
    void power(){
        print("It runs on electricity.");
    }
}

void main(){
    Honda honda=Honda();
    Tesla tesla=Tesla();

    honda.power();
    tesla.power();
}
```

It runs on petrol.
It runs on electricity.

```
class Employee{
    void salary(){
        print("Employee salary is \$1000.");
    }
}

class Manager extends Employee{
    @override
    void salary(){
        print("Manager salary is \$2000.");
    }
}

class Developer extends Employee{
    @override
    void salary(){
        print("Developer salary is \$3000.");
    }
}

void main(){
    Manager manager=Manager();
    Developer developer=Developer();

    manager.salary();
    developer.salary();
}
```

Manager salary is \$2000.
Developer salary is \$3000.

Polymorphism

Advantage Of Polymorphism In Dart

- Subclasses can override the behavior of the parent class.
- It allows us to write code that is more flexible and reusable.



Abstract Classes & Implementation

Abstract Classes

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. An abstract class is declared using the keyword **abstract**.

```
abstract class ClassName {  
    //Body of abstract class  
  
    method1();  
    method2();  
}
```



Abstract Method

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 ClassName {  
    //Body of abstract class  
  
    method1();  
    method2();  
}
```



Abstract Class

Why We Need Abstract Class

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



Abstract Class

In this example below, there is an abstract class **Vehicle** with two abstract methods **start()** and **stop()**. The subclasses **Car** and **Bike** implement the abstract methods and override them to print the message.



```
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();
}
```

```
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();
}
```

```
abstract class Shape {
    int dim1, dim2;
    // Constructor
    Shape(this.dim1, this.dim2);
    // Abstract method
    void area();
}

class Rectangle extends Shape {
    // Constructor
    Rectangle(int dim1, int dim2) : super(dim1, dim2);

    // Implementation of area()
    @override
    void area() {
        print('The area of the rectangle is ${dim1 * dim2}');
    }
}

class Triangle extends Shape {
    // Constructor
    Triangle(int dim1, int dim2) : super(dim1, dim2);

    // Implementation of area()
    @override
    void area() {
        print('The area of the triangle is ${0.5 * dim1 * dim2}');
    }
}

void main() {
    Rectangle rectangle = Rectangle(10, 20);
    rectangle.area();

    Triangle triangle = Triangle(10, 20);
    triangle.area();
}
```

Abstract Class

Constructor In Abstract Class

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

Key Points To Remember

- You can't create an object of an abstract class.
- It can have both abstract and non-abstract methods.
- It is used to define the behavior of a class that other classes can inherit.
- Abstract method only has a signature and no implementation.



Mixin

Mixin

Mixins are a way of reusing the code in multiple classes. Mixins are declared using the keyword **mixin** followed by the mixin name. Three keywords are used while working with mixins: **mixin**, **with**, and **on**. It is possible to use multiple mixins in a class.

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



Mixin

Rules For Mixin

- **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.

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



Mixin

In this example below, there are two mixins named **ElectricVariant** and **PetrolVariant**. The **ElectricVariant** mixin has a method **electricVariant()** and the **PetrolVariant** mixin has a method **petrolVariant()**. The **Car** class uses both the **ElectricVariant** and **PetrolVariant** mixins.



```
mixin ElectricVariant {  
    void electricVariant() {  
        print('This is an electric variant');  
    }  
}  
  
mixin PetrolVariant {  
    void petrolVariant() {  
        print('This is a petrol variant');  
    }  
}  
  
// with is used to apply the mixin to the class  
class Car with ElectricVariant, PetrolVariant {  
    // here we have access of electricVariant() and petrolVariant() methods  
}  
  
void main() {  
    var car = Car();  
    car.electricVariant();  
    car.petrolVariant();  
}
```

Mixin

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.



Generics

Generics

Generics

is a way to create a class, or function that can work with different types of data (**objects**). If you look at the internal implementation of **List** class, it is a generic class. It can work with different data types like int, String, double, etc. For example, **List<int>** is a list of integers, **List<String>** is a list of strings, and **List<double>** is a list of double values.

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



Generics

Without Using Generics

Suppose, you need to create a class that can work with both **int** and **double** data types. You can create two classes, one for **int** and another for **double** like this:

```
// Without Generics
// Creating a class for int
class IntData {
    int data;
    IntData(this.data);
}
// Creating a class for double
class DoubleData {
    double data;
    DoubleData(this.data);
}

void main() {
    // Create an object of IntData class
    IntData intData = IntData(10);
    DoubleData doubleData = DoubleData(10.5);
    // Print the data
    print("IntData: ${intData.data}");
    print("DoubleData: ${doubleData.data}");
}
```



Generics

Using Generics

In this example below, there is single class that can work with **int**, **double**, and any other data types using **Generics**.

```
// Using Generics
class Data<T> {
    T data;
    Data(this.data);
}

void main() {
    // create an object of type int and double
    Data<int> intData = Data<int>(10);
    Data<double> doubleData = Data<double>(10.5);

    // print the data
    print("IntData: ${intData.data}");
    print("DoubleData: ${doubleData.data}");
}
```



What is Next ?



Thank You