



Dart

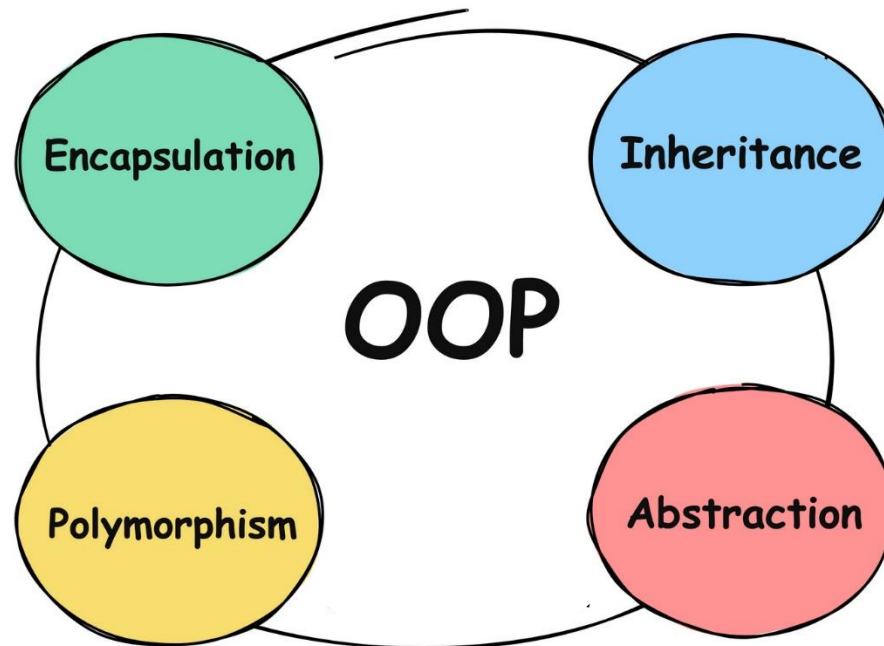


Table of Contents

1. OOP
2. Mixins
3. Enums

Some of the slides are based on Flutter Complete Course [content](#)

OOP

Class Example

 **Concise**

```
void main() {  
    var person = Person(firstName: 'John', lastName: 'Doe', age: 17);  
    print('Full name: ${person.fullName}');  
    print('Is minor: ${person.isMinor()}');  
}
```

```
class Person {  
    // Use final for properties that  
    // are initialized once and won't change  
    final String firstName;  
    final String lastName;  
    int age;  
  
    // Constructor with named parameters  
    // Use required for non-nullable properties to ensure that  
    // a value is provided during object creation  
    Person({required this.firstName, required this.lastName, required this.age});  
  
    // Computed property (getter)  
    String get fullName => '$firstName $lastName';  
  
    // Method to check if the person is a minor  
    bool isMinor() => age < 18;  
}
```

**Instantiate
an object**

Properties

**Primary
constructor**

**Computer
Property**

Method

Class with a computed property

```
class Rectangle {  
  final int width;  
  final int height;  
  Rectangle({required this.width, required  
this.height});  
  bool get isSquare => width == height;  
}
```

Named Constructor

```
class Conference {
  final String name;
  final String city;
  final bool isFree;
  double fee;

  // Primary constructor with optional parameters
  Conference({
    required this.name,
    required this.city,
    this.isFree = true,
    this.fee = 0.0,
  });

  // Named constructor for non-free conferences
  Conference.withFee({
    required this.name,
    required this.city,
    required this.fee,
  }) : isFree = false;
}

void main() {
  // Using the named constructor for a conference with a fee
  var conference = Conference.withFee(name: "Flutter Conference", city: "Doha", fee: 300);
  print('Conference: ${conference.name}, City: ${conference.city}, Fee: ${conference.fee},
        Is Free: ${conference.isFree}');
}
```

cascade operator (..)

- cascade operator (..) allows you to perform a **series of operations** on the same object without having to repeat the object reference for each operation
 - Improved readability: Reduces redundancy and makes the code cleaner

```
class Person {
    String name = '';
    int age = 0;

    void setName(String name) {
        this.name = name;
    }

    void setAge(int age) {
        this.age = age;
    }

    void greet() {
        print("Hello, my name is $name
              and I am $age years old.");
    }
}
```

```
void main() {
    // Without cascade operator:
    var person1 = Person();
    person1.setName("Ali");
    person1.setAge(30);
    person1.greet();

    // With cascade operator:
    var person2 = Person()
        ..setName("Fatima")
        ..setAge(25)
        ..greet();
}
```

Static Properties and Methods

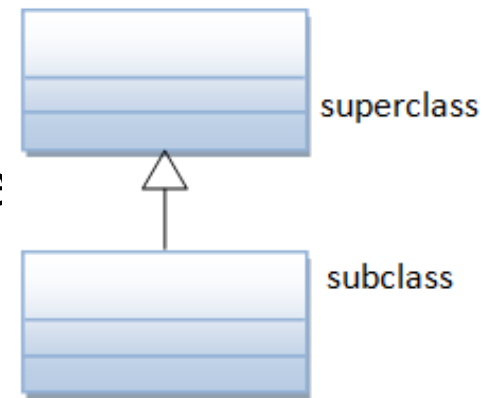
- Static properties/methods belong to the class rather than to any particular object
 - They can be called on the class itself, without creating an instance

```
class Car {  
    // Static property to keep track of the number of cars created  
    static int carCount = 0;  
    // Instance property  
    String model;  
  
    // Constructor  
    Car(this.model) {  
        carCount++; // Increment car count whenever a new car is created  
    }  
  
    // Static method to get the total number of cars  
    static int getCarsCount() {  
        return carCount;  
    }  
}
```


Inheritance

- **Ideas**

- Common properties and methods are placed in a **superclass** (also called *parent class* or *base class*)
- You can create a subclass that **inherits** the properties and methods of the super class
 - Subclass also called *child class*, *subclass* or *derived class*
- Subclass can extend the superclass by **adding new properties/methods** and/or **overriding the superclass methods**



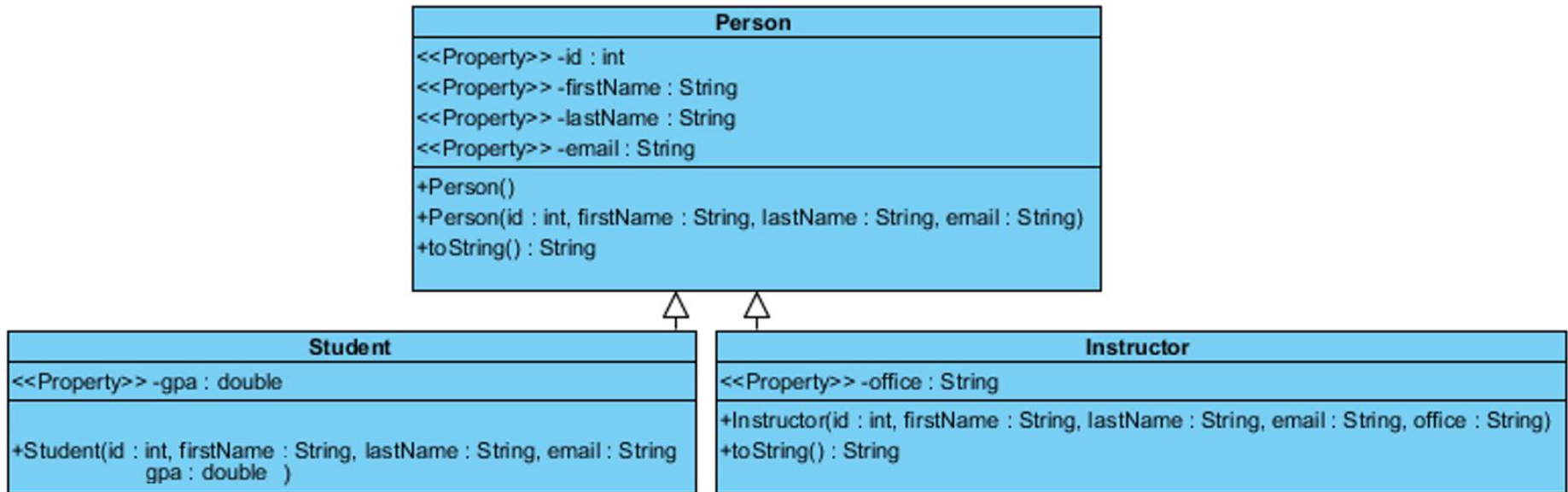
- **Syntax**

```
class SubClass extends SuperClass { ... }
```

- **Motivation**

- Allow **code reuse**. **Common properties and methods are placed in a super class** then inherited by subclasses (i.e., avoids writing the same code twice to ease maintenance)

Inheritance – Person Example



- The Person class has the common properties and methods
- Each subclass can add its own specific properties and methods (e.g., **office** for Instructor and **gpa** for Student)
- Each subclass can **override** (redefine) the parent method (e.g., Instructor class overrode the `toString()` method).

Inheritance – Person Example

```
class Person { ... }
```

```
class Student extends Person {  
    final double gpa;
```

```
    Student(String firstName, String lastName, DateTime dob, this.gpa)  
        : super(firstName, lastName, dob);
```

```
    @override
```

```
    String toString() => '${super.toString()}. GPA: $gpa';  
}
```

Abstract Classes

- Idea
 - Use an abstract class when you want to define a **template** to guarantee that all **subclasses** in a hierarchy will have certain common methods
 - Abstract classes can contain implemented methods and **abstract methods** that are NOT implemented
- Syntax

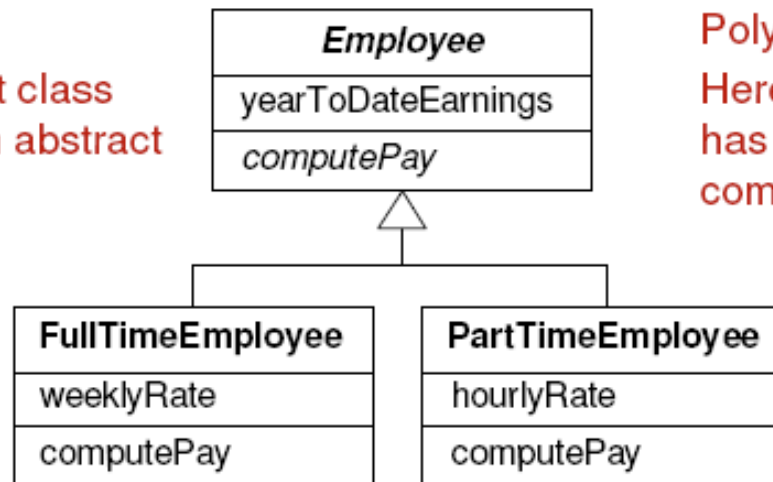
```
abstract class SomeClass {  
    SomeType method1(...) // No body  
    SomeType method2(...) { ... } // Not abstract  
}
```
- Motivation
 - Guarantees that all subclasses will have certain methods => **enforce a common design**
 - Lets you make collections of mixed type objects that can be processed polymorphically

Abstract Classes

- An abstract class has one or more abstract properties/methods that subclasses **MUST** override
 - Abstract properties/methods do not provide implementations because they **cannot be implemented in a general way**
- An abstract class cannot be instantiated

Abstraction:

Employee is an abstract class and *computePay()* is an abstract operation (italicized)



Polymorphism:

Here, each type of Employee has its own version of *computePay()*

Abstract Class Example

Shape.dart

```
abstract class Shape {  
    double area();  
    String get name => 'Shape';  
}
```

Circle.dart

```
class Circle extends Shape {  
    final double radius;  
  
    Circle(this.radius);  
  
    @override  
    double area() => pi * pow(radius, 2);  
  
    @override  
    String get name => 'Circle';  
}
```

Rectangle.dart

```
class Rectangle extends Shape {  
    final double width;  
    final double height;  
  
    Rectangle(this.width, this.height);  
  
    bool get isSquare => width == height;  
  
    @override  
    double area() => width * height;  
  
    @override  
    String get name => isSquare ? 'Square' :  
        'Rectangle';  
}
```

Interfaces

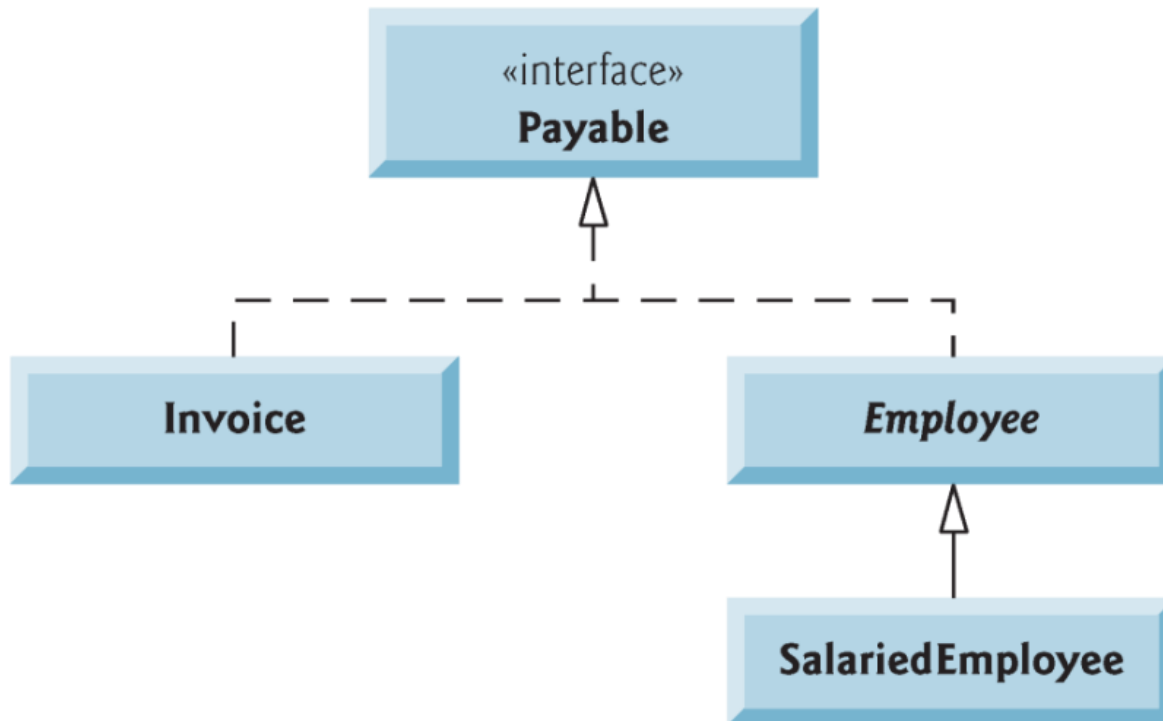
- Idea
 - **Interfaces** are used to define a set of common properties and methods that must be implemented by **classes not related by inheritance**
 - The interface specifies **what** methods a class must perform but does not specify **how** they are performed
- Syntax

```
abstract class SomeInterface {  
    SomeType method1(...)    // No body  
    SomeType fun method2(...) // No body  
}  
class SomeClass() implements SomeInterface {  
    // Real definitions of method1 and method 2  
}
```

- Motivation
 - Interfaces enables requiring that **unrelated classes implement a set of common methods**
 - **Ensure consistency** and guarantee that classes has certain methods:
 - Interface defines a **contract** that implementing classes must adhere to
 - Let us make **collections of mixed type** objects that can processed polymorphically

Interface Example

- A finance system has Employees and Invoices
- Employee and Invoice are not related by inheritance
- But to the company, they are both *Payable*



Interface Example

Payable.dart

```
abstract class Payable {  
    double get amount;  
    String pay();  
}
```

Employee.dart

```
class Employee implements Payable {  
    ...  
    final double salary;  
  
    Employee(this.firstname, this.lastname, this.salary);  
    ...  
    @override  
    String pay() => "Pay $firstname $salary";  
}
```

Invoice.dart

```
class Invoice implements Payable {  
    final String invoiceDate;  
    final double totalAmount;  
  
    Invoice(this.invoiceDate, this.totalAmount);  
    ...  
    @override  
    String pay() => "Invoice $amount is paid";  
}
```

Polymorphism Using interfaces

- A way of coding **generically**
 - way of referencing many related objects as one generic type
 - Cars and Bikes can both `move()` → refer to them as **Transporter** objects
 - Phones and Teslas can both `charge()` → refer to them as *Chargeable* objects, i.e., objects that implement **Chargeable** interface
 - Employees and invoices can both `pay()` → refer to them as *Payable* objects

```
for (var payable in payables) {  
    print ( payable.pay() )  
}
```

Abstract Class vs. Interface

- Abstract classes and interfaces cannot be instantiated
- Abstract classes and interfaces may have abstract methods that must be implemented by the subclasses
- Classes that implement an interface **can be from different inheritance hierarchies**
 - An interface is often used when unrelated classes need to provide **common properties and methods**
 - When a class implements an interface, it establishes a '**IS-A**' relationship with the interface type, enabling interface references to invoke polymorphic methods in a manner similar to how an abstract superclass reference can
- Concrete subclasses that extend an abstract superclass are **all related to one other by inheriting from a shared superclass**
- Classes can extend only ONE abstract class but they may implement more than one interface

Summary

- Inheritance = “factor out” the common properties and methods and place them in a single superclass
 - => Removing code redundancy will result in a smaller, more flexible program that is easier to maintain.
- Interfaces are contracts, can’t be instantiated
 - force classes that implement them to define specified methods
- Polymorphism allows for generic code by using superclass/interface type variables to manipulate objects of subclass type
 - make the client code more **generic** and ease extensibility

Mixins



Mixins

- **Mixins** are a way to reuse code across multiple classes
 - Allowing you to add functionality to a class without extending another class
- **Difference from inheritance:**
 - **Inheritance** allows you to inherit properties and methods from **one** class, establishing an "is-a" relationship (e.g., a Cat is an Animal)
 - Mixins enable a class to "mix in" functionality from multiple sources **without the "is-a" relationship**, giving you more flexibility to add behaviors or functionalities to a class (e.g., a Dog can Swim and Run)

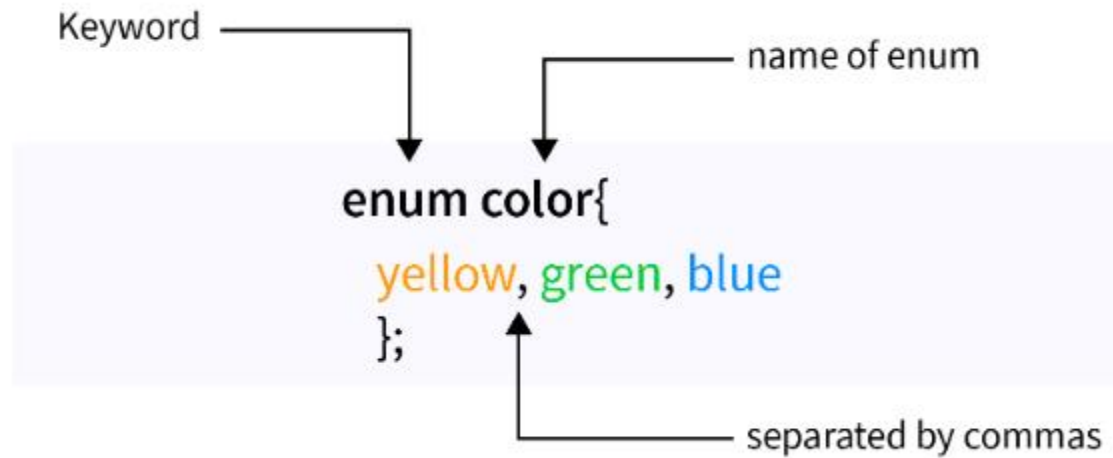
mixins properties

- mixins can define properties, and these properties can be accessed in the classes that use the mixin
- Example:
 - CanFly mixin defines a flyingSpeed property
 - CanSwim mixin defines a swimmingSpeed property
 - The Bird, Fish, and Duck classes access and modify these properties from the mixins
 - The setFlyingSpeed(), setSwimmingSpeed(), and setSpeeds() methods modify the speeds, and those values are used within the methods fly() and swim()

mixins access class properties and methods

- mixins can access properties from the classes that mix them in by adding superclass constraints with the on keyword
 - This allows mixins to require that they be mixed into classes that define certain properties or methods
- Example:
 - The ElectricVehicle and CombustionVehicle mixins use the on keyword with Vehicle, meaning they require that they be mixed into a class that extends Vehicle
 - These mixins access the name property from the Vehicle class to print the vehicle's name in their methods (chargeBattery(), driveOnElectric(), refuel(), and driveOnFuel())
 - The ElectricCar, CombustionCar, and HybridCar classes inherit the name property and behaviors from both their mixins and the Vehicle class, and they can customize the properties such as batteryLevel and fuelLevel

Enums



enum

- enums (short for enumerations) are a way to define a set of named constant values
 - They represent a fixed number of options, such as days of the week, colors, or directions
 - Enums make the code more readable and less error-prone by limiting the possible values
 - Enums have an implicit index starting from 0, which can be accessed using **.index**
 - You can retrieve all enum values using **.values**

```
enum Gender {  
    female, male  
}  
enum Direction {  
    left, right, up, down  
}
```

Enum class

- enum class can have properties, constructors, and methods, similar to regular classes
 - This allows you to attach additional data and functionality to each enum value

```
enum VehicleType {  
    car(120),  
    motorcycle(180),  
    bicycle(25);  
  
    final int maxSpeed;  
  
    const VehicleType(this.maxSpeed);  
  
    void displayInfo() {  
        print('A $name can reach a max speed of $maxSpeed km/h.');    }  
}
```

Dart Resources

- Draft Language
 - Dart language tour <https://dart.dev/language>
- Dart learning resources
 - <https://dart.dev/guides>
 - <https://dart.dev/tutorials>
- Online Dart dev <https://dartpad.dev/>