Abstraction and Polymorphism

Topcis	- Abstract classes and abstract methods, - Implementing Interfaces, - Polymorphism basics, - Real-world modeling with classes (Car, Dog, BankAccount)

Class Content: Abstraction & Polymorphism

1. Lesson Objectives

By the end of this class, you will be able to:

- **Define** Abstraction: Hiding complex implementation details and showing only functionality.
- **Create** `abstract` classes and methods to define templates for subclasses.
- **Understand** the difference between `extends` (Inheritance) and `implements` (Interfaces).
- **Implement** an Interface to force a class to adhere to a specific "contract."
- **Apply** Polymorphism to treat different objects (e.g., `Car`, `Boat`) as a single type (e.g., `Vehicle`).
- **Model** a real-world scenario using a `BankAccount` system.

---

2. Core Concepts (Lecture & Demo)

Part 1: Abstraction (Separating the “what” from the “how”)

Abstraction allows you to hide complex implementation details and expose only the necessary features of an object. We achieve this in two ways:

A. Abstract Classes (extends)

An abstract class serves as a base hierarchy. It represents a conceptual "is-a" relationship.

- **Cannot be instantiated:** You cannot write `new Employee()`.
- **Partial Implementation:** Can contain both abstract methods (no body) and concrete methods (with body).
- **State:** Can maintain internal state (fields/variables) that children inherit.
- **Constructor:** Can have constructors to initialize that state.
- **Use Case:** When creating a family of related classes that share common logic but need specific implementations for certain behaviors.

```
// Abstract Base Class
abstract class Employee {
  String name;
  String id;

  Employee(this.name, this.id);

  // Concrete Method: Shared logic for all employees
  void clockIn() {
    print('$name ($id) clocked in at ${DateTime.now()}');
  }

  // Abstract Method: Specific logic per role (must be overridden)
  double calculateSalary();
}

class Developer extends Employee {
  double hourlyRate;
  int hoursWorked;

  Developer(String name, String id, this.hourlyRate, this.hoursWorked)
      : super(name, id);

  @override
  double calculateSalary() {
    return hourlyRate * hoursWorked;
  }
}
```

B. Interfaces (implements)

Unlike Java or C#, Dart does not have an interface keyword. Every class is implicitly an interface.

- **Contract Enforcement:** When you use `implements`, you must override *every* public field and method.
- **No Inheritance:** You do not inherit code or logic from the parent; you only inherit the "shape" (signatures).
- **Multiple Implementation:** A class can implement multiple interfaces.
- **Use Case:** When unrelated classes need to share a capability (a **"can-do"** relationship).

> ⚠️ From Dart 3.0, `interface` keyword was introduced. To achieve a pure interface, you need to combine it with `abstract` and use `abstract interface` before `class`

```
// Acts as an Interface
class Logger {
  void log(String message) {
    print('Default logging: $message');
  }
}

class Database {
  void connect() {}
}

// Implementation
// 'implements' forces us to redefine log(), ignoring the code in Logger
class FileLogger implements Logger {
  @override
  void log(String message) {
    print('Writing to file: $message');
  }
}

// Multiple Interfaces
class SecureService implements Logger, Database {
  @override
  void log(String message) { /* ... */ }

  @override
  void connect() { /* ... */ }
}
```

Part 2: Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass.

A. Compile-time Polymorphism (Static Binding)

In many languages, this is achieved via Method Overloading (same method name, different parameters).

- **Crucial Note:** Dart **does not** support traditional Method Overloading.
- **The Dart Way:** Dart achieves compile-time flexibility through **Optional and Named parameters**.

```
class Printer {
  // Dart style "Overloading"
  void printData(String data, {bool isBold = false, String? prefix}) {
    String output = data;
    if (prefix != null) output = "$prefix $output";
    if (isBold) output = "**$output**";

    print(output);
  }
}

void main() {
  var p = Printer();
  p.printData("Hello");               // Form 1
  p.printData("Hello", isBold: true);   // Form 2
  p.printData("Hello", prefix: ">>>");  // Form 3
}
```

B. Runtime Polymorphism (Dynamic Binding)

This is achieved via Method Overriding. The runtime environment determines which method to call based on the actual object type, not the variable type.

- **Requires:** Inheritance (`extends`) or Implementation (`implements`).
- **Mechanism:** The parent reference holds a child object.
- **Use Case:** Handling a collection of different objects uniformly.

```
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 renderShapes(List<Shape> shapes) {
  for (var shape in shapes) {
    // Runtime Polymorphism:
```

```
    // The runtime checks if 'shape' is actually a Circle or Square
    // and calls the correct draw() method.
    shape.draw();
  }
}
```

3. In-Class Exercise (Combined)

Goal: Model a Banking System using Abstraction and Polymorphism.

The Scenario:

We need a system for a bank. All bank accounts have a balance and can deposit(). However, withdraw() works differently depending on the account type:

1. **Savings Account:** Cannot withdraw if the balance goes below 0.
2. **Checking Account:** Can withdraw below 0, but charges a fee (Overdraft).

Instructions:

1. **Create an Abstract Class `BankAccount`:**
    - Property: `double balance`.
    - Constructor: Initialize `balance`.
    - Concrete Method: `void deposit(double amount)` (increases balance).
    - Abstract Method: `void withdraw(double amount)` (no body).
2. **Create `SavingsAccount` (extends BankAccount):**
    - Override `withdraw`: Check if `balance >= amount`. If yes, subtract. If no, print "Insufficient funds".
3. **Create `CheckingAccount` (extends BankAccount):**
    - Override `withdraw`: Subtract the amount. If the resulting balance is negative, print "Overdraft fee applied" and subtract an extra $10.
4. **Polymorphism Test:**
    - In `main()`, create a `List<BankAccount>`.

- Add one `SavingsAccount` (start with $100) and one `CheckingAccount` (start with $100).
- Loop through the list and try to `withdraw(150)` from each.
- Print the final balance of each.

4. Summary Comparison

Feature	Abstract Class	Interface (Dart Implementation)
Keyword	`abstract class A` / `extends A`	`class A` / `interface class A` / `abstract interface class A` / `implements A`
Logic Sharing	**Yes**. Child inherits actual code.	**No**. Child must rewrite all logic.
State (Fields)	Can have variables/state.	Fields are treated as getters/setters you must override.
Multiplicity	Single Inheritance only.	Multiple Interfaces allowed.
Best For	"Is-A" relationship (Dog is an Animal).	"Can-Do" relationship (Dog implements Swimmable).

5. Key Takeaways

- **Abstract Class:** A partial blueprint. Cannot be created, only extended.
- **Abstract Method:** A rule. Subclasses *must* implement it.
- **Interface (`implements`):** A contract. You must build everything yourself, inheriting nothing.
- **Polymorphism:** Treating a `Dog` as an `Animal`. It allows for flexible lists and functions that accept generic types but run specific code.