Lecture 3 - Dart OOP

🔵 Object and Class

Dart's OOP Features:

- Dart is an object-oriented programming language that supports key OOP concepts like `classes`, `objects`, `inheritance`, and `abstract classes`.

Default Constructor:

- A constructor `created` by the compiler when `no constructor` is `declared` in the class.

```dart
// Importing the math library for mathematical operations.
import 'dart:math' as math;

// Defining a class named Circle.
class Circle {
  // Declaring a property radius and initializing it to 1.
  double radius = 1;

  // Method to calculate the area of the circle.
  double findArea() => math.pi * radius;
}

void main() {
  // Creating an instance of the Circle class using the default constructor.
  var c1 = Circle();
  // Printing the area of the circle.
  print('The area is ${c1.findArea()}');
}
```

No-Arg Constructor:

- A constructor with `no parameters`.

```dart
import 'dart:math' as math;

class Circle {
  double radius = 1;

  // No-argument constructor with radius set to 5.
  Circle() {
    this.radius = 5;
  }
```

```
  double findArea() => math.pi * radius;
}

void main() {
  // Creating an instance of the Circle class using the no-argument constructor.
  var c1 = Circle();
  print('The area is ${c1.findArea()}');
}
```

Parameterized Constructor:

- A constructor that `accepts parameters`.

```
import 'dart:math' as math;

class Circle {
  double radius = 1;

  // Parameterized constructor accepting radius.
  Circle(double radius) {
    this.radius = radius;
  }

  double findArea() => math.pi * radius;
}

void main() {
  // Creating an instance of the Circle class using a parameterized constructor.
  var c1 = Circle(5);
  print('The area is ${c1.findArea()}');
}
```

Optional Parameter Constructor:

- A constructor with optional `named parameters`.

```
import 'dart:math' as math;

class Circle {
  double radius = 1;

  // Optional parameter constructor with named parameter radius.
  Circle({double radius = 1}) {
    this.radius = radius;
  }
```

```
  double findArea() => math.pi * radius;
}

void main() {
  // Creating an instance of the Circle class using an optional parameter
constructor.
  var c1 = Circle(radius: 5);
  print('The area is ${c1.findArea()}');
}
```

Named Constructor:

- Used to declare multiple constructors in a single class.

```
import 'dart:math' as math;

class Circle {
  double radius = 1;

  // Named constructor with a named parameter radius.
  Circle.radius(this.radius);

  // Method to calculate the area of the circle.
  double findArea() => math.pi * radius;
}

void main() {
  // Creating an instance of the Circle class using a named constructor.
  var c1 = Circle.radius(6);
  print('The area is ${c1.findArea()}');
}
```

🔒 Access Modifiers

Private Modifier:

- In Dart, there are only two access modifiers: `private` and `public`.
- Dart doesn't have specific keywords like `private` or `public`.
- Instead, you can prefix an identifier with an underscore `_` to make it `private`.
- In Dart, privacy is at the file level rather than the class level.

Example:

```dart
import 'dart:math' as math;

class Circle {
  double _radius = 1; // Private field
  Circle(this._radius);

  double findArea() => math.pi * _radius;
}
```

Using Private Fields:

```dart
import 'modifiers.dart';

void main() {
  var c1 = Circle(6);
  // print(c1._radius); // Error: '_radius' is private
  print('The area is ${c1.findArea()}');
}
```

🔄 Setter & Getter

Getter and Setter Methods:

- `Getter` and `setter` methods are used to `manipulate the data of class fields`.
- `Getter`: Used to `read or get the data` of the class field.
- `Setter`: Used to `set the data` of the class field to some variable.

Example:

```dart
import 'dart:math' as math;

class Circle {
  double _radius = 1;
  Circle(this._radius);

  // Setter: Sets the value of _radius
  set radius(double radius) => _radius = radius;

  // Getter: Retrieves the value of _radius
  double get radius => _radius;

  double findArea() => math.pi * _radius;
}
```

Using Setter & Getter:

```
import 'test_setter_getter.dart';

void main() {
  var c1 = Circle(6);
  c1.radius = 50; // Using the setter
  print(c1.radius); // Using the getter
  print('The area is ${c1.findArea()}');
}
```

🏛️ Inheritance

Definition:

- `Inheritance` in Dart is the process where `one class inherits` the `properties` and `characteristics` of `another class`.
- It allows the creation of a new class `child class` based on an existing class `parent class`.

Key Components:

- **`Parent Class:`** A class that is `inherited by another class` is known as the `parent` class.
- **`Child Class:`** A class that `inherits the properties and characteristics` of the `parent` class.

Example:

```
import 'dart:math' as math;

// Parent class: Circle
class Circle {
  double _radius = 1; // Private radius variable initialized to 1
  Circle(this._radius); // Constructor to set the radius
  double get radius => _radius; // Getter for radius
  // Method to calculate the area of the circle
  double findArea() => math.pi * _radius * radius;
}

// Child class: Cylinder (inherits from Circle)
class Cylinder extends Circle {
  double _height = 1; // Private height variable initialized to 1
  // Constructor to set radius and height, calls parent constructor using super
```

```
  Cylinder(double radius, this._height) : super(radius);
  // Setter and getter for height
  set height(double height) => _height = height;
  double get height => _height;
  // Override method to calculate the area of the cylinder
  @override
  double findArea() =>
      2 * math.pi * math.pow(_radius, 2) + 2 * math.pi * _radius * _height;
  // Method to calculate the volume of the cylinder
  double findVolume() => math.pi * math.pow(_radius, 2) * _height;
}

void main() {
  var c1 = Circle(6); // Creating a Circle instance with radius 6
  var cy1 = Cylinder(3, 4); // Creating a Cylinder instance with radius 3 and
height 4

  print(c1.findArea()); // Printing the area of the circle
  print(cy1.findArea()); // Printing the area of the cylinder
}
```

Explanation:

- In this example, we have two classes: `Circle` and `Cylinder`.
- `Cylinder` is a subclass of `Circle`, meaning it inherits all properties and methods from `Circle`.
- `Cylinder` extends `Circle` using `extends` keyword, indicating inheritance.
- The `Cylinder` class adds its own properties (`_height`) and methods (`findVolume()`).
- The `super` keyword is used in the constructor of `Cylinder` to call the constructor of the superclass (`Circle`).
- We override the `findArea()` method in `Cylinder` to provide a different implementation.
- In the `main()` function, we create instances of both classes and demonstrate calling their methods.

Polymorphism: 🎭

1. **Definition**:
   - Polymorphism means `objects` can `take on different forms` or `roles` in a program, like actors playing various characters in a play.
2. **Code Example**:

```
// Define a base class for employees
class Employee {
  void display() => print('Employee Class');
```

```
}

// Define subclasses for specific types of employees
class Doctor extends Employee {
  void display() => print('Doctor Class');
}

class Engineer extends Employee {
  void display() => print('Engineer Class');
}

void main() {
  Employee em1 = Employee();
  Doctor do1 = Doctor();
  Engineer en1 = Engineer();

  List<Employee> es = [em1, do1, en1];
  es.forEach((employee) => employee.display());
}
```

3. **Explanation**:
   - We have a base class `Employee` with a `display()` method.
   - Subclasses like `Doctor` and `Engineer` inherit from `Employee` and override the `display()` method.
   - In the `main()` function, we create instances of different types of employees and store them in a list.
   - We iterate through the list and call the `display()` method on each object.
   - Each object behaves differently based on its type, demonstrating polymorphism in action. 🚀

Abstract Class: 🖌️

1. **Definition**:
   - An abstract class is a `blueprint` for other classes, and it may contain `one or more abstract methods` (methods without implementation).
2. **Code Example**:

```
// Define an abstract class 'Person'
abstract class Person {
  void display(); // Abstract method without implementation
}
```

3. **Key Points**:

- **Abstract Method**:
  - An abstract method is a method without an implementation.
  - In Dart, if a class contains at least one abstract method, it must be declared as abstract using the `abstract` keyword.
- **Object Creation**:
  - Objects of an abstract class cannot be created directly.
  - However, the abstract class can be extended by other non-abstract classes.
- **Keyword Usage**:
  - The `abstract` keyword is used to declare an abstract class.
- **Implementation in Subclasses**:
  - Any class that extends an abstract class `must` provide implementations for all its abstract methods.

By defining abstract classes, we can create a hierarchy of classes with common behaviors and enforce certain methods to be implemented by subclasses. 🌟

Mixins: 🎭

1. **Definition**:
   - `Mixins` are a way to `reuse` code in multiple classes without using `inheritance`.
   - They are like `classes`, but they `cannot` be `instantiated` on their own.
2. **Code Example**:

```dart
// Define mixins for different behaviors
mixin Breathing {
  void swim() => print("Breathing");
}

mixin Walking {
  void walk() => print("Walking");
}

mixin Coding {
  void code() => print("print('Hello world!')");
}

// Classes using mixins to incorporate behaviors
class Human with Walking {}
class Developer with Walking, Coding {}
```

3. **Key Points**:
   - **Usage**:

        - Mixins are used to add functionalities to classes without using inheritance.
        - They allow code reuse across multiple classes.
    - **Syntax**:
        - Mixins are declared using the `mixin` keyword followed by the mixin name and its functionality.
        - They are then incorporated into classes using the `with` keyword, followed by the mixin name.
    - **Behavior Incorporation**:
        - Classes can incorporate multiple mixins to gain different behaviors.
        - Mixins are a way to achieve multiple inheritance-like behavior in Dart.

Mixins provide a flexible way to share code across multiple classes, promoting code reuse and maintainability. 🌟

Static: 🧊

1. **Definition**:
    - The `static` keyword is used in Dart to declare class-level variables and methods.
    - Static members belong to the class itself rather than instances of the class.
2. **Key Points**:
    - **Class Membership**:
        - Static variables and methods are associated with the class itself, not with individual instances of the class.
    - **Shared Among Instances**:
        - Static variables are shared among all instances of the class. There's only one copy for the entire class.
    - **Accessing Static Members**:
        - Static members can be accessed using the class name directly, without needing an instance of the class.
    - **Usage**:
        - Static methods are often used for utility functions or for operations that don't depend on instance data.
        - Static variables can be used to maintain state shared across all instances of a class.
3. **Code Example**:

```
class Student {
  // Declaring static variable
  static String stdBranch;
```

```
  String stdName;
  int rollNum;

  void showStdInfo() {
    print("Student's name is: ${stdName}");
    print("Student's roll number is: ${rollNum}");
  }
}

void main() {
  // Assigning value to static variable using class name
  Student.stdBranch = "Computer Science";

  // Creating instances of Student class
  Student std1 = Student();
  Student std2 = Student();

  // Assigning values to instance variables
  std1.stdName = "Ben Cutting";
  std1.rollNum = 90013;

  std2.stdName = "Peter Handscomb";
  std2.rollNum = 90014;

  // Accessing static variable
  print("Student's branch name is: ${Student.stdBranch}");

  // Calling instance method
  std1.showStdInfo();
  std2.showStdInfo();
}
```

Static members in Dart provide a way to maintain shared data and functionality at the class level. 🌟

Cascade Notation: 🌊

1. **Definition**:
   - Cascade notation, denoted by `..`, allows for performing a sequence of operations on the same object without repeating the object reference.
2. **Usage**:
   - **Chaining Operations**:
     - It's useful when you want to invoke multiple methods or set multiple properties on the same object in a sequence.

- **Avoiding Repetition**:
  - Cascade notation helps to avoid repeating the object reference for each method call or property assignment.

3. **Syntax**:
   - Cascade notation is represented by two dots `..` and is used to chain method calls or property assignments on the same object.
4. **Example**:

```
class Sample {
  var a;
  var b;

  void showA(x) {
    this.a = x;
  }

  void showB(y) {
    this.b = y;
  }

  void printValues() {
    print(this.a);
    print(this.b);
  }
}

void main() {
  Sample sampleOne = Sample();
  sampleOne.showA(2);
  sampleOne.showB(3);
  sampleOne.printValues();

  Sample sampleTwo = Sample();
  sampleTwo
    ..showA(2)
    ..showB(3)
    ..printValues();
}
```

In this example, the cascade notation `..` is used to call methods `showA()` and `showB()` as well as to print values without repeating the object reference `sampleTwo`. This improves code readability and conciseness. 🚀