OOP IN DART

Advanced Flutter R2
NAME: Tasneem Samy

Dart is an Object-Oriented Programming Language. Object-Oriented Programming is a way of creating computer programs using classes and objects. The main goal of OOP is to break down complex problems into smaller components. This helps to increase reusability and reduce code complexity. This makes the code easy to maintain, modify, and debug as well as reduces code repetition. It supports features like **Class, Object, Encapsulation, Inheritance, Polymorphism, and Abstraction.**

Abstract Classes:

Definition: A blueprint for creating related classes that share common characteristics but cannot be instantiated directly.

Purpose and Benefits:

- **Establish a Template:** Define a common foundation for related classes with shared characteristics. Subclasses inherit this blueprint, reducing code duplication and promoting consistency.
- **Enforce Implementation:** Specify essential methods that all subclasses must implement. This guarantees certain functionalities are available in all derived classes.
- **Model Abstract Concepts:** Represent abstract entities or ideas that cannot be directly instantiated but define the principles for concrete implementations.

Key Features and Usage:

- **Declaration:** Use the `abstract` keyword before the class name: `abstract class Shape {}`.
- **Abstract Methods**: Define method signatures without implementation (**`void draw();`**). Subclasses must provide their own implementations.
- **Concrete Methods**: Can also exist within an abstract class with their own implementations. These methods provide common functionality shared by all subclasses.
- **Instantiation Restriction:** Abstract classes themselves cannot be directly instantiated using the new keyword. Only subclasses can be created.

Examples and Use Cases:

- **Shape hierarchy**: Abstract class Shape with abstract methods draw() and area(). Concrete subclasses like Circle, Rectangle, and Triangle provide their own implementations for these methods.

```
abstract class Shape {
  void draw (); // Abstract method
}

class Circle extends Shape {
  @override
  void draw () {
    // Implementation for drawing a circle
  }
}
```

- **Validation logic:** Abstract class Validator with abstract methods for validating different data types (e.g., validateEmail, validateNumber). Concrete subclasses like EmailValidator and NumberValidator specialize the validation logic.

```
abstract class Validator {
  bool validateEmail (String email);
  bool validateNumber (String number);
}

class EmailValidator extends Validator {
  @override
  bool validateEmail (String email) {
    // Implement email validation logic here
  }

  @override
  bool validateNumber(String number) {
    // EmailValidator doesn't need to implement number validation
    return false;
  }
}

class NumberValidator extends Validator {
  @override
  bool validateEmail (String email) {
    // NumberValidator doesn't need to implement email validation
    return false;
  }

  @override
  bool validateNumber(String number) {
    // Implement number validation logic here
  }
}
```

- **Game entities**: Abstract class GameEntity with abstract methods for update and render. Concrete subclasses like Player, Enemy, and PowerUp inherit these methods and define their specific behaviors.

```
abstract class GameEntity {
  void update (double deltaTime);
  void render (Canvas canvas);
}
class Player extends GameEntity {
  @override
  void update (double deltaTime) {
    // Implement player-specific update logic (movement, actions, etc.)
  }

  @override
  void render (Canvas canvas) {
    // Draw the player's visual representation on the canvas
  }
}

class Enemy extends GameEntity {
  @override
  void update (double deltaTime) {
    // Implement enemy AI and movement logic
  }

  @override
  void render (Canvas canvas) {
    // Draw the enemy's visual representation on the canvas
  }
}

class PowerUp extends GameEntity {
  @override
  void update (double deltaTime) {
    // Implement power-up behavior (e.g., movement, duration)
  }

  @override
  void render (Canvas canvas) {
    // Draw the power-up's visual representation on the canvas
  }
}
```

Static Features:

Definition: Members associated with a class itself, rather than individual instances.

Purpose:

- Share data and behavior across all instances of a class.
- Create utility functions or constants not tied to specific objects.
- Improved Code Organization: Group related data and functions within the class itself, promoting modularity and clarity.
- Memory Efficiency: Shared static variables avoid redundant allocation in each object instance.
- Global Access: Convenient access to constants and utility functions from anywhere in the program.
- Singleton Implementation: Facilitates creation and management of a single instance for controlled access.

Examples:

1. Static Variables:

- Shared data across all instances of a class.
- Used for global information relevant to the class itself, like constants or configuration values.

```
class MathUtils {
  static const pi = 3.14159;
  static int factorial (int n) {
    // Implementation of factorial calculation
  }
}
```

2. Static Functions:

- Utility functions or calculations not tied to specific objects.
- Often used for helper methods or operations related to the class's functionality.

```
class StringUtils {
  static String capitalize (String str) {
    return str.substring (0, 1).toUpperCase() + str.substring(1);
  }
}
```

3. Singleton Pattern:

- Used to create and manage a single instance of a class throughout the application.
- Static members control creation and access to the singleton instance.

```
class Logger {
  static final Logger _instance = Logger._internal();
  factory Logger () => _instance;

  Logger._internal ();
```

```
  static void log (String message) {
    // Implement logging logic
  }
}
```

Important Considerations:

- Overuse of static methods can lead to tight coupling and make code less testable.
- Static variables should be carefully controlled to avoid unexpected side effects.
- Singleton pattern can introduce complexity and potentially violate Single Responsibility Principle if not used judiciously.

Encapsulation

Encapsulation is a fundamental principle of object-oriented programming, including Dart. It's the practice of bundling together data (attributes) and related behavior (methods) within a single unit - a class or object. This serves several crucial purposes:

1. Data Protection:

- Encapsulation gives you control over how data is accessed and modified.
- By using access modifiers like public, private, and protected, you can:
  - Restrict direct access: Keep sensitive data safe from accidental or unauthorized modification.
  - Enforce data integrity: Only authorized methods can manipulate data, upholding its consistency.
  - Promote controlled interaction: Provide public methods to safely access and modify data.

2. Modularity and Reusability:

- Encapsulated classes act as independent units, promoting modularity in your code.
- Changes within one class are less likely to break other parts of the program.
- You can reuse encapsulated classes without exposing their internal details, enhancing reusability.

3. Improved Maintainability:

- Encapsulation clarifies the responsibilities of each class, making code easier to understand and maintain.
- Internal implementation details are hidden, reducing distraction and focusing on exposed functionalities.
- Changes to internal implementation can be made without affecting existing clients, simplifying maintenance.

4. Key Techniques in Dart:

- **Access modifiers:** public, private, protected
- **Getters and setters:** Controlled access to private data through public methods
- **Constructors**: Initialize object state with proper values
- **Class design patterns:** Promote clean and efficient encapsulation strategies

5. Real-World Example

1. A **BankAccount class encapsulates** account balance (private) and withdrawal/deposit methods (public), protecting account data and ensuring controlled monetary operations.

```
class BankAccount {
  private int _balance = 0;
  void deposit(int amount) {
    if (amount > 0) {
      _balance += amount;
    } else {
      throw Exception("Invalid deposit amount");
    }
  }
  void withdraw(int amount) {
    if (amount <= _balance) {
      _balance -= amount;
    } else {
      throw Exception("Insufficient funds");
    }
  }
  int get balance => _balance;
}
```

- **Encapsulation in action:**
  - The _balance field is private, ensuring only authorized methods can access and modify it.
  - Public methods deposit and withdraw provide controlled access, enforcing data integrity and preventing invalid operations.

2. User Profile class encapsulation ensure:

- Private fields protect sensitive information like name, email, and password.
- Public getters provide controlled access to name and email.
- The setPassword method enforces password validation and security measures.
- The verifyPassword method allows secure authentication without exposing the password itself.

```
class UserProfile {
  private String _name, _email, _password;

  UserProfile (this._name, this._email, this._password);

  String get name => _name;
  String get email => _email;
```

```
  void setPassword (String newPassword) {
    // Apply password validation and hashing
    _password = newPassword;
  }
  bool verifyPassword (String enteredPassword) {
    // Compare with hashed password
  }
}
```

6. Benefits and Potential Drawbacks:

- **Benefits:** Improved data protection, modularity, maintainability, reusability
- **Drawbacks**: Over-encapsulation can lead to tightly coupled code and limited flexibility

Polymorphism

Polymorphism is a powerful concept in OOP, including Dart, that allows objects of different types to respond to the same method call in different ways. This flexibility and adaptability enhance code reusability and simplify complex behaviors.

Forms of Polymorphism in Dart:

- Inheritance-based: Subclasses inherit methods from parent classes and can override them to provide their own implementations. This enables specialization of behavior based on the specific subclass type.
- Interface-based: Classes implement shared interfaces that define method signatures and contracts. This allows objects of different types to be treated similarly based on their common interface, even if their implementations differ.

Benefits of Polymorphism:

- Code Reusability: You can define common functionality once in a base class or interface and reuse it across different subclasses or implementing classes.
- Reduced Redundancy: Avoids writing the same code for similar functionality in different classes.
- Flexibility and Adaptability: Makes code more dynamic and adaptable to different object types and situations.
- Simplifying Complex Behaviors: Enables elegant and concise handling of diverse scenarios with a single method call.

Examples:

1. User Management System:

That handles all aspects of creating, managing, and controlling user account.

Interface for User:

```
interface User {
  void login (String username, String password);
  List<String> getPermissions ();}
```

Concrete User Implementations:

```
class AdminUser implements User {
  @override
  void login (String username, String password) {
    // Perform admin-specific login logic
  }
  @override
  List<String> getPermissions() {
    return ['create_users', 'delete_users', 'manage_roles'];
  }
}
class RegularUser implements User {
  @override
  void login (String username, String password) {
    // Perform regular user login logic
  }
  @override
  List<String> getPermissions() {
    return ['view_profile', 'edit_profile'];  }
}
class GuestUser implements User {
  @override
  void login (String username) {
    // Perform guest user login (no password required)  }
  @override
  List<String> getPermissions() {
    return ['view_public_content']
}
}
```

Access Control Function:

```
void grantAccess(User user, String requiredPermission) {
  if (user.getPermissions().contains(requiredPermission)) {
    // Allow access
    print("Access granted for user ${user.runtimeType}");
  } else {
    // Deny access
    print("Access denied for user ${user.runtimeType}");
  }
}
```

Usage:

```
List<User> users = [
  AdminUser(),
  RegularUser(),
  GuestUser(),
];
for (User user in users) {
  grantAccess(user, "create_users"); // Only AdminUser will be granted access
}
```

2. Animal Sounds with Inheritance:

```
abstract class Animal {
  void makeSound();
}

class Dog extends Animal {
  @override
  void makeSound() {
    print("Woof!");
  }
}

class Cat extends Animal {
  @override
  void makeSound() {
```

```
    print("Meow!");
  }
}

void playWithAnimals(Animal animal) {
  animal.makeSound(); // Polymorphic call, will print "Woof!" for Dog or "Meow!" for Cat
}

Dog dog = Dog();
Cat cat = Cat();

playWithAnimals(dog); // Prints "Woof!"
playWithAnimals(cat); // Prints "Meow!"
```

Conclusion

Object-oriented programming (OOP) principles like abstraction, inheritance, and polymorphism empower developers in Dart to build robust, modular, and maintainable applications. By encapsulating data and behavior within objects, OOP promotes code organization and data protection. Inheritance allows for specialization and code reuse, while polymorphism enables flexible interaction with objects of different types. Understanding these concepts and leveraging their potential significantly improves the quality, efficiency, and scalability of Dart projects.