OOP Advanced Features

Module: OOP Advanced

Class: Advanced Features

Topics: Operator Overloading | Copy Constructors | Class Composition

1. Operator Overloading

Customizing Object Behavior

By default, operators like `==` and methods like `toString()` have default behaviors defined by the base `Object` class. Overloading allows us to define how these operators work for our specific custom classes.

The `toString()` Method

- **Default Behavior:** Returns "Instance of 'ClassName'".
- **Overriding:** We override this to provide a meaningful string representation of the object, which is crucial for debugging and logging.

The `==` Operator (Equality)

- **Reference Equality (Default):** Two variables are equal only if they point to the *exact same location* in memory.
- **Value Equality (Override):** We override `==` to check if two distinct objects contain the same *data*.
- **The `hashCode` Rule:** If you override `==`, you **must** also override `hashCode`. Equal objects must have the same hash code.

Code Example

```
class Point {
  Point(this.x, this.y);

  final int x;
  final int y;

    // 1. Overriding toString for better print output
  @override
  String toString() ⇒ 'Point($x, $y)';

  // 2. Overriding == for Value Equality
  @override
  bool operator ==(Object other) {
    if (identical(this, other)) return true;

    return other is Point &&
      other.x == x &&
      other.y == y;
  }

  // 3. Must override hashCode if == is overridden
  @override
  int get hashCode ⇒ x.hashCode ^ y.hashCode;
}

void main() {
  var p1 = Point(1, 2);
  var p2 = Point(1, 2);

  print(p1); // Output: Point(1, 2) instead of Instance of 'Point'
  print(p1 == p2); // Output: true (Value Equality)
}
```

2. Copy Constructor Concept (copyWith)

Cloning Objects & Immutability

In modern Dart, instead of a traditional "copy constructor," we typically use a `copyWith` instance method. This is essential for **Immutable** objects (where fields are `final`). Since you cannot change the fields of an existing object, you create a *new* object based on the old one, changing only specific properties.

- **Immutability:** Fields are `final` and cannot be changed after the object is created.
- **The `copyWith` Pattern:** A method that takes optional named parameters for every field. If a parameter is provided, it uses the new value; otherwise, it falls back to the existing value (`this.value`).

Why is this crucial for Flutter State Management?

In Flutter (and libraries like Bloc, Riverpod, or Redux), state is often immutable to ensure performance and predictability.

1. **Efficient Rebuilds:** Flutter needs to know *when* to rebuild the UI. Comparing memory addresses (Identity) is instant, while checking every field of a large object (Deep Equality) is slow.
2. **The Trigger:** If you mutate an object in place (`user.age = 26`), the object reference remains the same. The framework looks at the old and new state, sees the same memory reference, and assumes **nothing changed**, resulting in the UI not updating.
3. **The Solution:** Using `copyWith` creates a completely **new instance** (new memory reference). The framework sees `oldState != newState` and immediately knows to trigger a rebuild.

Code Example

```
class User {
  // Constructor requires all fields
  User({required this.name, required this.age});

  final String name;
```

```
  final int age;

  // Idiomatic 'copyWith' pattern
  // Parameters are nullable to allow 'null' to mean "don't change this field"
  User copyWith({
    String? name,
    int? age,
  }) {
    return User(
      name: name ?? this.name,
      age: age ?? this.age,
    );
  }

  @override
  String toString() ⇒ 'User(name: $name, age: $age)';
}

void main() {
  var original = User(name: "Alice", age: 25);

  // Create a copy, changing ONLY the name
  var copy = original.copyWith(name: "Bob");

  // Create a copy, changing ONLY the age
  var olderCopy = original.copyWith(age: 26);

  print(original);   // User(name: Alice, age: 25)
  print(copy);       // User(name: Bob, age: 25)
  print(olderCopy);  // User(name: Alice, age: 26)

  // Proof of different instances (Crucial for Flutter)
  print(original == copy); // false
}
```

3. Class Composition

"Has-A" Relationship

Composition is the design principle where a class contains objects of other classes as member variables.

Composition vs. Inheritance

- **Inheritance (Is-A):** `Car` *is a* `Vehicle`. Good for hierarchy.
- **Composition (Has-A):** `Car` *has an* `Engine`. Good for building complex objects from smaller, reusable components.

Why prefer Composition?

- **Flexibility:** You can easily swap components (e.g., change the `Engine` type inside a `Car`) at runtime.
- **Loose Coupling:** Changes in the component class don't ripple through a hierarchy as strictly as inheritance.

Code Example

```
// Component 1
class Engine {
  Engine(this.type);

  String type;

  void start() => print("$type engine starting...");
}

// Component 2
class Tires {
  Tires(this.size);

  int size;
}
```

```
// Composite Class
class Car {
  Car(this.model, this.engine, this.tires);

  String model;
  // Composition: Car HAS-A Engine and HAS-A Tires
  Engine engine;
  Tires tires;

  void startCar() {
    print("Checking system for $model...");
    engine.start(); // Delegating behavior to the component
  }
}

void main() {
  var v8 = Engine("V8");
  var offRoadTires = Tires(22);

  // Constructing object via composition
  var myCar = Car("Monster Truck", v8, offRoadTires);

  myCar.startCar();
}
```

4. Key Takeaways

1. **Operator Overloading:** Gives your objects natural behavior (like `==` for equality) and readable logging (`toString`).
2. **Reference vs. Value:** Without overriding `==`, two identical objects are considered different because they live in different memory addresses.
3. **Cloning:** Use the `copyWith` pattern to safely duplicate data. This is the standard for state management in Flutter, allowing the framework to detect state

changes efficiently by comparing object references.

4. **Composition over Inheritance:** Build complex objects by combining smaller, isolated classes. It creates more flexible and maintainable code structures.