Encapsulation

Module: OOP Advanced

Class: Encapsulation

Topics: Getters & Setters | Private Fields in Dart

1. Lesson Objectives

By the end of this session, students will be able to:

- Define **Encapsulation** and explain its importance in object-oriented design.
- Implement **private fields** in Dart using the underscore (`_`) syntax.
- Create **Getters and Setters** to control access to class properties.
- Apply validation logic within setters to protect object integrity.

2. Core Concepts: Encapsulation with Private Fields

Encapsulation is the practice of bundling data (variables) and methods (functions) that operate on that data into a single unit (class), while restricting direct access to some of an object's components.

Why is Encapsulation Important?

Encapsulation is often referred to as the "shield" of your code. It is crucial for three main reasons:

1. **Protection:** It prevents external code from accidentally corrupting the object's internal state (e.g., setting an `age` variable to -5).
2. **Flexibility (Maintainability):** You can change the internal implementation logic later (like renaming a private variable or changing a data type) without breaking the external code that uses your class, provided the public methods/getters remain the same.

3. **Simplification:** It hides complex implementation details, exposing only what is necessary for the user of the class to know.

Real-World Analogies

1. **A Car Dashboard:**
    - **Public Interface:** The steering wheel, gas pedal, and brake. You use these simple controls to drive.
    - **Private Implementation:** The fuel injection system, piston firing order, and combustion engine.
    - **Encapsulation:** You don't need to interact directly with the engine pistons to drive, and the dashboard prevents you from accidentally disconnecting the fuel line while driving.
2. **A Bank ATM:**
    - **Public Interface:** The keypad and screen.
    - **Private Implementation:** The internal cash vault and counting mechanism.
    - **Encapsulation:** You can request a withdrawal (using a public method), but you cannot physically reach inside the machine to change your balance or grab cash directly.

How Dart Handles Privacy

Unlike Java or C#, Dart does not have keywords like `private`, `public`, or `protected`.

- **Public:** By default, everything is public.
- **Private:** To make a member private to its **library** (file), prefix the name with an underscore (`_`).

Getters and Setters

We use special methods called **Getters** (`get`) and **Setters** (`set`) to read and write private fields. This allows us to add logic (like validation) before modifying data.

Lecture Code Example

```
class Employee {
  // Private field: Accessible only within this file/library
  String _name;
  double _salary;

  // Constructor
  Employee(this._name, this._salary);

  // --- GETTERS (Read Access) ---

  // Getter for name
  String get name => _name;

  // Getter for formatted salary
  String get salaryInfo => "Salary: \$${_salary.toStringAsFixed(2)}";

  // --- SETTERS (Write Access with Validation) ---

  // Setter for salary
  set salary(double newSalary) {
    if (newSalary < 0) {
      print("Error: Salary cannot be negative.");
    } else {
      _salary = newSalary;
      print("Salary updated to $_salary");
    }
  }
}

void main() {
  var emp = Employee("Alice", 50000);

  // Accessing via Getter
  print(emp.salaryInfo); // Output: Salary: $50000.00
  // Accessing via Setter (Valid)
  emp.salary = 55000; // Output: Salary updated to 55000.0

  // Accessing via Setter (Invalid)
  emp.salary = -100; // Output: Error: Salary cannot be negative.

  // NOTE: emp._salary would result in a compile error if accessed
  // from a different file, enforcing encapsulation.
}
```

3. In-Class Exercise

Scenario:

You are building a logic system for a Thermostat. The temperature should be private to prevent accidental extreme values.

Instructions:

1. Create a class named `Thermostat`.
2. Define a private field `_temperature` (double).
3. Create a constructor to initialize it.
4. Create a **Getter** `celsius` that returns the temperature.
5. Create a **Setter** `celsius` that:
   - Allows setting the temperature only if it is between -30 and 50 degrees.
   - Prints "Warning: Temperature out of range" if the value is invalid.
6. *Bonus:* Create a getter `fahrenheit` that converts the internal celsius value to fahrenheit ($C \times 9/5 + 32$).

4. Key Takeaways

1. **Control:** Encapsulation allows you to control **how** variables are accessed or modified.

2. **Validation:** Setters provide a specific place to validate data *before* it saves to the object (e.g., preventing negative age or salary).
3. **Read-Only:** By providing a Getter but **no** Setter, you make a property read-only from the outside.
4. **Abstraction:** The internal representation (private fields) can change without affecting external code that uses the public getters/setters.