ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

📝 Lab Objectives:

By the end of this lab, students will:

- ✅ Understand Dart fundamentals (functions, OOP, inheritance, and async programming).
- ✅ Write Dart code that mirrors real-world Flutter app development.
- ✅ Develop reusable components similar to Flutter widgets.

📌 Task 1: Entry Point and Functions

Concept:

Every Dart program starts with the `main()` function, just like a Flutter app. The `main()` function initializes the app, similar to `void main() => runApp(MyApp());` in Flutter.

Task:

1. Create a **function** named `startApp()`.
2. Inside `startApp()`, print `"Welcome to Dart Lab!"`.
3. Call `startApp()` inside `main()`.
4. Add another function, `showAppInfo()`, which prints `"This is a Dart practice session"`.
5. Call `showAppInfo()` inside `main()`.

```dart
void main() {
  startApp();
  showAppInfo();
}

void startApp() {
  print("Welcome to Dart Lab!");
}

void showAppInfo() {
  print("This is a Dart practice session.");
}
```

📌 Task 2: Object-Oriented Programming (OOP) - Classes and Objects

Concept:

Flutter widgets are based on **classes**. In Dart, we create classes to define reusable UI components, like `Text` and `Button`.

Task:

1. Create a class called `Button`.
2. Add a property `label` (of type `String`).
3. Create a constructor that initializes `label`.
4. Add a **method** `click()`, which prints `"<label> button clicked!"`.
5. In `main()`, create an object of `Button`, set the label to `"Login"`, and call `click()`.

```dart
class Button {
  String label;

  Button(this.label);

  void click() {
    print("$label button clicked!");
  }
}

void main() {
  Button myButton = Button("Login");
  myButton.click();
}
```

📌 Task 3: Constructors & Named Parameters

Concept:

Flutter uses **named parameters** in widgets (e.g., `Text("Hello", style: TextStyle(fontSize: 20))`).

Task:

1. Create a class called `User`.
2. Define two properties: `name` (String) and `age` (int).
3. Create a constructor that uses **named parameters** with `required` keyword.
4. Add a method `showUser()` that prints the user's details.
5. In `main()`, create an object of `User` and call `showUser()`.

```dart
class User {
  String name;
  int age;

  User({required this.name, required this.age});

  void showUser() {
    print("User: $name, Age: $age");
  }
}

void main() {
  User user1 = User(name: "Ali", age: 25);
  user1.showUser();
}
```

📌 Task 4: Asynchronous Programming (Future, async-await)

Concept:

In Flutter, we use `Future` for tasks like API calls.

Task:

1. Create an **asynchronous function** called `fetchData()`.
2. Inside `fetchData()`, print `"Fetching data..."`.
3. Use `await Future.delayed(Duration(seconds: 2));` to simulate a delay.
4. After the delay, print `"Data retrieved!"`.
5. Call `fetchData()` in `main()` and print `"Executing other tasks..."` before it.

```dart
import 'dart:async';

Future<void> fetchData() async {
  print("Fetching data...");
  await Future.delayed(Duration(seconds: 2));
  print("Data retrieved!");
}

void main() {
  fetchData();
  print("Executing other tasks...");
}
```

📌 Task 5: Convert Map to JSON and Parse It Back

Concept:

Flutter apps frequently communicate with APIs, which send and receive **JSON** data. In Dart, we use `Map<String, dynamic>` to represent JSON data and the `dart:convert` library to convert between **Map** and **JSON format**.

Task:

1. Import the `dart:convert` library (`import 'dart:convert';`).
2. Create a **Map<String, dynamic>** named `product`, containing:
   - `"id"` (int)
   - `"name"` (String)
   - `"price"` (double)
3. Convert the **Map** to a **JSON string** using `jsonEncode()`.
4. Print the JSON string.
5. Convert the **JSON string** back to a **Map** using `jsonDecode()`.
6. Print the parsed Map.

```dart
import 'dart:convert';

void main() {
  Map<String, dynamic> product = {
    "id": 1,
    "name": "Laptop",
    "price": 1200.5
  };

  // Convert Map to JSON string
  String jsonString = jsonEncode(product);
  print("JSON String: $jsonString");

  // Convert JSON string back to Map
  Map<String, dynamic> parsedMap = jsonDecode(jsonString);
  print("Parsed Map: $parsedMap");
}
```

📌 Task 6: Creating a Simple Dart UI Model (Like Flutter Widgets)

Concept:

Flutter widgets are reusable classes.

Task:

1. Create a **class** `TextWidget` with a property `text`.
2. Define a **constructor** to initialize `text`.
3. Add a **method** `render()` that prints `"Rendering text: <text>"`.
4. In `main()`, create an object of `TextWidget`, set text to `"Welcome to Flutter"`, and call `render()`.

```dart
class TextWidget {
  String text;

  TextWidget(this.text);

  void render() {
    print("Rendering text: $text");
  }
}

void main() {
  TextWidget title = TextWidget("Welcome to Flutter");
  title.render();
}
```

ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

📌 Task 7: Inheritance + Method Overriding and Using super in Dart: Like Flutter Widgets Overriding build()

Concept:

In Flutter, widgets override the `build()` method to define UI. Similarly, in Dart, we use **method overriding** to change the behavior of a parent class method. Additionally, `super` is used to call a method from the parent class.

Task:

1. Create a **base class** `LabelWidget` with:
   - A method `display()` that prints `"Rendering label"`.
2. Create a **subclass** `BoldLabelWidget` that **extends** `LabelWidget`.
   - **Override** the `display()` method.
   - Inside the overridden method, **call** `super.display()` first.
   - Then print `"Label displayed in bold"`.
3. Create another **subclass** `FancyLabelWidget` that also **extends** `LabelWidget`.
   - **Override** `display()` to print `"Rendering a fancy label!"` (without calling `super.display()`).
4. In `main()`, create **objects** of `BoldLabelWidget` and `FancyLabelWidget`, then call `display()` on both.

```dart
class LabelWidget {
  void display() {
    print("Rendering label");
  }
}

class BoldLabelWidget extends LabelWidget {
  @override
  void display() {
    super.display();
    print("Label displayed in bold");
  }
}

class FancyLabelWidget extends LabelWidget {
  @override
  void display() {
    print("Rendering a fancy label!");
  }
}

void main() {
  BoldLabelWidget boldLabel = BoldLabelWidget();
  FancyLabelWidget fancyLabel = FancyLabelWidget();

  boldLabel.display();
  fancyLabel.display();
}
```

Department Of Computer Science
Program CET215 – Mobile Application Development
Lab 2
Instructor: Dr. Ahmed Elrefaiy
Eng. Shaymaa Mohamed, Eng. Yara Ahmed

ELSEWEDY UNIVERSITY OF TECHNOLOGY
- POLYTECHNIC OF EGYPT -

📌 Assignment: Object-Oriented Modelling for a To-Do App in Dart

📝 Objective:

Design and implement a To-Do App using Object-Oriented Programming (OOP) in Dart. The goal is to apply classes, constructors, inheritance, method overriding, lists, and map handling to create a simple task management system.

📖 Scenario: To-Do List App

A startup wants to develop a To-Do List Application where users can add, remove, complete, and list tasks. Each task has a title, description, priority level, and completion status. Some tasks may have due dates, while others do not.

Your task is to create a Dart program that models this system using OOP principles.

🔹 Requirements:

1 Create a `Task` class

- Properties:
  - `id` (int) → Unique identifier for each task.
  - `title` (String) → The task's name.
  - `description` (String) → A short description.
  - `priority` (String) → Can be `"Low"`, `"Medium"`, or `"High"`.
  - `isCompleted` (bool) → Default: `false`.
- Methods:
  - `markComplete()` → Marks the task as completed.
  - `displayTask()` → Prints the task details.

2 Create a `TaskWithDueDate` Class (Inheritance)

- Extends `Task` class and adds:
  - `dueDate` (DateTime) → Deadline for the task.
- Override `displayTask()` to include the due date.

3 Create a `TaskManager` Class

- Properties:
  - A **list** to store tasks.
- Methods:
  - `addTask(Task task)` → Adds a new task to the list.
  - `removeTask(int id)` → Removes a task by ID.
  - `listTasks()` → Displays all tasks.
  - `listPendingTasks()` → Displays only **incomplete** tasks.
  - `markTaskComplete(int id)` → Marks a task as completed.

4 Implement the Main Function

- Instantiate `TaskManager`.
- Add **at least 3 tasks** (including one with a due date).
- List all tasks.
- Mark a task as completed.
- List only pending tasks.