UNIVERSITY OF RWANDA - CST

COMPUTER ENGINEERING - YEAR 3

MOBILE APPLICATION DESIGN MODULE

ACADEMIC YEAR : 2025 - 2026

SEMESTER II

DATE : 12th Feb. 2026

Group Members : - 223026674

- 223026685

- 223027476

LAB 1

— o —

1. A function is a reusable block of code that performs a specific task. It helps organize code, avoid repetition, and makes program easier to understand and maintain.

We created welcomeMessage() to provide a standard greeting that can be called whenever needed in the school system. This promotes code reuse, consistency, and makes the program more modular. Instead of writing the same print statement multiple times throughout the program, I can just call this function whenever a welcome message is needed.

Functions also make debugging easier because if I need to change the welcome message, I only change it in one place rather than searching through the entire codebase.

---

2. Named parameters allow us to pass arguments to a function by explicitly specifying the parameter name followed by a colon and the value. In Dart, named parameters are enclosed in curly braces { }.

They are helpful because:

* READABILITY: The function call clearly shows which value corresponds to which parameter.

* FLEXIBILITY: Named parameters can be passed in any order. You don't have to remember the exact parameter sequence.

* SELF-DOCUMENTING: The code documents itself. Anyone reading the code immediately understands what each value represents.

* OPTIONAL WITH @required: We can make them required or optional. The @required annotation ensures important parameters are always provided.

* REDUCES ERRORS: Prevents bugs caused by mixing up parameter order.

3. Optional parameters allow a function to be called with fewer arguments than the function defines. In Dart, there are two types:

(*) OPTIONAL POSITIONAL PARAMETERS: Enclosed in square brackets [ ]. They must be placed at the end of the parameter list.
Example: void CreateTeacher (String name, [String? subject])

(*) OPTIONAL NAMED PARAMETERS: Enclosed in curly braces { }. They can be in any order and are optional by default.
Example: void CreateStudent ({String? name, int? age})

HOW THEY WORK:

- When an optional parameter is not provided, it defaults to null (unless a default value is specified.)
- We can provide default values: [String subject = "Not Assigned."]
- We must handle null values properly to avoid runtime errors.

In my code, I check if subject is null and print "Subject not assigned" as a fallback. This makes the function flexible - it works whether the teacher has a subject assigned or not.

PART 2.

4. A Constructor is a special method that is automatically called when an object is created. It initializes the object's properties and sets up the initial state of the object.

WHY CONSTRUCTORS ARE IMPORTANT:

(*) INITIALIZATION: They ensure objects are created in a valid state. Without constructors, we might forget to initialize properties.

(*) CONSISTENCY: Every object starts with properly initialized values.

(*) ENCAPSULATION: Constructors can enforce rules during object creation.

(*) CODE SIMPLICITY: Dart's constructor syntax (this.name, this.age) eliminates boilerplate code. No need to manually assign each parameter to a property.

(*) OVERLOADING: Dart supports multiple constructors (named constructors) for different initialization scenarios

Without constructors, we would have to create empty objects and then manually set each property, which is error-prone and verbose.

5- An object is an instance of a Class. It is a Concrete entity that exists in memory and has actual values for its properties.

OBJECT CREATION:

In Dart, we Create objects using the Constructor:

Student S1 = Student ("Anna", 20);

- 'Student' is the Class name (type)
- 'S1' is the object reference (Variable)
- 'Student ("Anna", 20)' Calls the Constructor.
- The 'new' keyword is optional in Dart.

OBJECT USAGE:

Once Created, we Can access the object's properties and methods:

- S1.name - gets the name property
- S1.age - gets the age property.

WHY OBJECTS MATTER:

- They model real-world entities (students, teachers, Courses)
- Each object has its own independent State.
- We Can Create multiple objects from the Same Class
- Objects encapsulate data and behavior together.

In my code, I Created one Student object and printed its details. This demonstrates the basic lifecycle: Class definition → Object Creation → property access.

PART 3

6. A class is a blueprint or template for Creating objects. It defines the structure (properties/variables) and behavior (methods/functions) that all objects of that Class will have.

KEY CHARACTERISTICS OF A CLASS

- BLUEPRINT: A Class itself doesn't occupy memory; it's a definition. Objects (instances) are Created from this blueprint.
- ENCAPSULATION: Classes bundle data and methods together.
- REUSABILITY: Once a Class is defined, we can Create many objects
- MODULARITY: Classes help organize Code into logical, self-Contained units that are easier to understand and maintain.
- ABSTRACTION: Classes hide Complex implementation details behind simple interfaces.

In my Person Class, I defined:

- Property: 'name' - what a person has
- Method: 'introduce()' - what a person can do.

This Person class can be used as a building block for more specific person types like Student, Teacher, etc.

7. Inheritance is an object-oriented programming concept where a class (child/sub class) acquires the properties and methods of another class (parent/super class). In Dart, we use the 'extends' keyword.

HOW STUDENT REUSES PERSON:

int age;
Student2 (String name, this.age): super (name);
}

* INHERITED MEMBERS: Student2 automatically gets the 'name' property and 'introduce()' method from Person without writing any code.
* CONSTRUCTOR CHAINING: 'super (name)' calls the Person constructor to initialize the inherited name property.
* EXTENSION: Student2 adds its own property 'age' that Person doesn't have. This makes Student2 a more specific version of Person.

BENEFITS OF INHERITANCE:

- CODE REUSE: Don't rewrite common functionality.
- IS-A RELATIONSHIP: Student2 IS-A Person (logical modeling)
- POLYMORPHISM: Student2 can be treated as a Person.
- MAINTAINABILITY: Changes in Person automatically apply to all children.

Without inheritance, I would have to duplicate the name property and introduce() method in every person — like class, leading to code repetition.

---

8. An interface is a contract that defines WHAT a class should do, but not HOW it should do it. It specifies method signatures without providing implementations.

PART 4.

KEY CHARACTERISTICS OF INTERFACES:

* ABSTRACTION: Interfaces represent pure abstraction — they only define behavior, not implementation.
* CONTRACT: Any class that implements an interface MUST provide implementations for ALL methods declared in the interface.
* MULTIPLE IMPLEMENTATION: In Dart, a class can implement multiple interfaces (unlike inheritance where a class can only extend one.)
* IN DART: There's no separate 'interface' keyword. Any class can serve as an interface, but abstract classes are commonly used for this purpose.

My registrable interface declares:
void registerCourse (String courseName);

This says, "Any registrable entity must be able to register for a course." It doesn't specify how registration happens — that's up to each class. Interfaces enable loose coupling and make code more flexible and testable. Different classes can implement the same interface in completely different ways.

9. Implementing an interface means a class commits to fulfilling the contract defined by that interface. In Dart, we use the 'implements' keyword.

HOW IMPLEMENTATION WORKS:

```
Class Student3 extends Person implement Registrable {
    @ override
    Void registerCourse (String courseName) {
        print ("$name is registered for $courseName");
    }
}
```

ENFORCING RULES:

* COMPULSORY IMPLEMENTATION: The Dart Compiler ENFORCES that ALL interface methods are implemented. If I omit registerCourse (), the code won't compile.
* @OVERRIDE ANNOTATION: This tells the compiler we're intentionally overriding/implementing a method. It's optional but recommended.
* TYPE SAFETY: Interfaces create a common type. We can write code that works with ANY Registrable object, regardless of its actual class.

WHY THIS IS POWERFUL:

- Standardization: All registrable entities follow the same pattern.
- Polymorphism: Different classes can be used interchangeably.
- Future-proof: New classes can easily implement existing interfaces
- Testing: Easy to create mock implementation for testing.

My Student3 class now has TWO identities: it IS-A Person (inheritance) and it CAN-BE-TREATED-AS Registrable (inheritance implementation.)

PART 5

10. A mixin is a way to reuse a class's code in multiple class hierarchies without using inheritance. It's a collection of properties and methods that can be "mixed into" other classes using the 'with' keyword.

KEY CHARACTERISTICS OF MIXINS:

* NO CONSTRUCTOR: Mixins cannot have constructors
* NO "IS-A" RELATIONSHIP: Unlike inheritance, mixins don't create a parent-child relationship. A class with mixin IS STILL the same type as its parent class.
* MULTIPLE MIXINS: A class can use multiple mixins: class MyClass with Mixin1, Mixin2, Mixin3
* HORIZONTAL REUSE: Inheritance is vertical (parent → child) while mixins are horizontal (can be added to unrelated classes).

My AttendanceMixin provides attendance tracking functionality that can be added to any class that needs it - Student, Teacher, Staff, etc. Without mixins, I would have to duplicate this code in every class or force unrelated classes to inherit from a common ancestor.

11. Mixins add behavior to classes through a process called "mixing in". When a class uses a mixin with the 'with' keyword, all the mixin's properties and methods become part of the class.

HOW MIXINS ADD BEHAVIOR:

* COMPOSITION OVER INHERITANCE: Instead of inheriting behavior, the class ~~becomes~~ composes the behavior from mixins
* CODE INJECTION: The Dart compiler effectively copies the mixin's code into the ~~the~~ class
* LINEARIZATION: If multiple mixins have conflicting methods, Dart uses a specific order (the last mixin wins).

IN MY CODE:

```
Student s4 = Student4 ("Daniel", 23);
s4.markAttendance();  // Behavior added by AttendanceMixin.
print (s4.attendanceCount);  // Property added by AttendanceMixin.
```

The Student4 class didn't define markAttendance() or attendanceCount. These entirely came from the mixin. Yet Student4 behaves as if it always had these capabilities.

BENEFITS OF THIS APPROACH:

- Single Responsibility: Each mixin handles one specific capability
- Flexibility: Add/Remove capabilities without changing inheritance.
- Reusability: Same mixin can be used by Student, Teacher, Course.
- Maintainability: Change behavior in one place, affects all users.

I marked attendance 3 times, and the mixin maintained the counter internally, demonstrating that mixins can maintain state too.

PART 6

12. A list is an ordered collection of objects where each element has a specific index (position). In Dart, lists are zero-indexed (first element at index 0).

KEY FEATURES OF LISTS

* ORDERED: Elements maintain the order they were added.
* INDEXED ACCESS: Elements can be accessed by their position. students[0] // Gets Alice.
* ITERABLE: Lists can be looped over using for loops, forEach, etc.
* DYNAMIC SIZE: Lists can grow and shrink (add, remove, insert).
* TYPED: List<Student> means this list can ONLY contain Student objects.

COMMON LIST OPERATIONS:

- students.length - Get number of elements
- students.add(newStudent) - Add to end
- students.removeAt(0) - Remove first element.
- students.contains(Student) - Check if exists.

WHY I USED A LIST:

I needed to store multiple Student objects. The order might matter (enrollment order, alphabetical, etc). Lists are perfect for this. The list allows me to process all students together, count them, filter them, sort them, etc.

In real school systems, lists are used everywhere: Class rosters, grade lists, attendance records.

13. A map is a collection of key-value pairs where each key is unique and maps to exactly one value. Think of it as a dictionary: you look up a word (key) to get its definition

KEY FEATURES OF MAPS:

* KEY-VALUE PAIRS: Each entry associates a key with a value.
* UNIQUE KEYS: No duplicate keys allowed. Adding with existing key overwrites the old value.
* FAST LOOKUP: Accessing by key is very fast, regardless of map size.
* FLEXIBLE KEY TYPES: Keys can be strings, numbers, objects, etc.

WHEN MAPS ARE USEFUL:

- When you need to look up data by a unique identifier.
  - Student ID → Student object.
  - Course Code → Course details.
  - Username → User profile.
- When order doesn't matter but fast access does.
- When you need to model real-world relationships.

WHY I USED A MAP:

In a real school system, students are identified by ID numbers, not their position in a list. A map accurately models this: "Give me the student with ID 5002" → Immediately get Bob.

With a list, I would have to search through all students to find ID 5002. With a map it's instant lookup.

Maps vs lists:

- List: Best order when matters or you need sequential access.
- Map: Best when you need to find things by a unique key.

PART 7.

14. An anonymous function (also called lambda or closure) is a function without a name. It's defined at the point where it's needed, rather than being declared separately with a name.

SYNTAX:

```
(parameters) {
    // function body
}
```

MY CODE:

```
students.forEach (Student) [
print ("Student : ${student.name}");
});
```

The forEach method takes a function as an argument. Instead of creating a named function elsewhere, I defined the behavior directly inside the forEach call. This function has no name → it's anonymous.

WHY USE ANONYMOUS FUNCTIONS:

* CONVENIENCE: Define behavior exactly where it's used
* READABILITY: The code reads naturally - "for each student, do this"
* ONE-TIME USE: Perfect for operations that don't need reuse.
* CALLBACKS: Commonly used in event handlers, timer callbacks, etc
* CLOSURES: Anonymous functions can access variables from their surrounding scope.

14. When NOT to use anonymous functions:

- If the same logic is needed in multiple places
- If the function is long/complex (hurts readability)
- If you need to test the function independently.

In my code, printing student names is a simple, one-time operation. An anonymous function is perfect for this use case.

---

15. An arrow function (=>) is a concise syntax for functions that contain a single expression. The arrow (=>) means "return the result of this expression".

SYNTAX COMPARISON:

```
//Regular function
void greetStudent (String name) {
    print ("Hello, $name! Welcome to class.");
}
//Arrow function
void greetStudent (String name) =>
    print ("Hello, $name! Welcome to class.");
```

KEY CHARACTERISTICS:

- IMPLICIT RETURN: The expression after => is automatically returned.
- NO CURLY BRACES: Eliminates {} and return keyword for single-line functions
- SINGLE EXPRESSION ONLY: Cannot contain multiple statements. If you need multiple lines, use regular function syntax.

WHY ARROW FUNCTIONS ARE SIMPLER

- LESS BOILERPLATE: Fewer characters to type, less visual clutter.
- CLEAR INTENT: The arrow visually indicates "input → output".
- FUNCTIONAL STYLE: Common in functional programming paradigms.
- CONSISTENCY: Works well with collection methods like map, where, reduce:

```
students.map((s) => s.name).toList()
```

In my code, greeting a student is a simple operation - print a message. The arrow function expresses this in the cleanest, most readable way.

Arrow functions embody the principle: "Say what you mean, as simply as possible."

PART 8

16. Asynchronous programming allows a program to perform time-consuming operations without blocking the main thread. In Dart, we use 'async' and 'await' keywords.

ASYNC FUNCTIONS:

* DECLARATION: Mark a function with 'async' keyword.
* RETURN TYPE: Always returns a Future. If you return T, the function returns Future<T>
* SUSPENSION: When an async function hits an 'await', it SUSPENDS execution and immediately returns an incomplete Future.
* RESUMPTION: When the awaited operation completes, the function resumes from where it left off.

WAITING:

```
await Future.delayed(Duration(seconds: 2));
```

This line:

- Creates a Future that will complete after 2 seconds
- 'await' tells Dart to pause this function until that Future completes.
- The thread is NOT blocked - other code can run during these 2 seconds.
- After 2 seconds, execution continues with the next line.

WHY THIS MATTERS:

In real apps, we frequently for:

- Network responses (API calls)
- Database queries
- File I/O operations
- User input.

Without async/await, the app would freeze during these operations.
With async/wait, the app remains responsive.

My loadStudents() simulates a database query that takes 2 seconds. The app doesn't freeze - it just waits patiently for the data.

---

17. Asynchronous programming is NOT optional in real application - it's ESSENTIAL. Here's why:

* USER EXPERIENCE: Imagine an app that freezes every time it loads data. User would uninstall it immediately. Async keeps the UI responsive while operations run in the background.
* REAL-WORLD EXPERIENCES:
  * FLUTTER APP:
    - Fetching weather data from internet -> Async
    - Reading photos from device storage -> Async
    - Saving data to database -> Async
    - Animations running smoothly -> Async.
  * WEB SERVER:
    - Handling hundreds of simultaneous users -> Async
    - Database queries -> Async
    - File uploads / downloads -> Async.

17. (*) RESOURCE EFFICIENCY: While waiting for one operation, the app can work on other tasks. One thread can handle many concurrent operations.

(*) CODE READABILITY: Before async/wait developers used callbacks, leading to "callback hell" (nested callbacks within callbacks).

```
// Callback hell (old style)
load User (user) {
    load Courses (user, (courses) {
        // deeply nested code
    });
});
}

// Async/wait (modern)
var user = await loadUser();
var courses = await loadCourses(user);
var grades = await loadGrades(courses);
```

In my code, await loadStudents() makes the program wait for 2 seconds, but the underlying thread is free. This is EXACTLY how flutter apps fetch data from internet without freezing the screen.

PART 9.

18. WHY MIXINS ARE USEFUL:

Mixins solve the "Cross-Cutting Concern" problem. Some functionality needs to be used across MULTIPLE, UNRELATED class hierarchies.

Real Example - School System:

- Student Class inherits from Person
- Teacher Class inherits from Person
- Course Class inherits from nothing (or AcademicEntity)

ALL of these need:

- Attendance tracking (AttendanceMixin)
- Notification sending (NotificationMixin)
- Logging (LoggerMixin)

Without mixins I would have to:

(*) Duplicate code in all three classes (terrible, violates DRY)

(*) Force Course to inherit from Person (makes NO logical sense)

(*) Use complex composition patterns (more boilerplate).

Mixins let me write the behavior ONCE and use it EVERYWHERE it's needed.

18. KEY DIFFERENCES: INHERITANCE Vs MIXINS

INHERITANCE:

- "IS-A" relationship
- Single parent only
- Creates parent-child chain
- Child is a specialized parent.
- Cannot be added to existing class
- Constructor required.

MIXINS:

- "HAS-A" or "CAN-DO" capability
- Multiple mixins allowed
- No hierarchy - horizontal
- Class remains same type
- Can be mixed into any class
- No constructor allowed.

ANALOGY:

- Inheritance is like FAMILY: You are born into it, defines what you ARE
- Mixins are like SKILLS: You can learn them, defines what you CAN do.

A Student IS-A Person (inheritance)
A Student can track attendance (mixin)
A Student CAN receive notifications (mixin)

You can't change who your parents are (single inheritance), but you can learn many skills (multiple mixins).

---

19. HOW I CREATED AND USED NOTIFICATIONMIXIN:

Step 1: DEFINE THE MIXIN

```
mixin NotificationMixin {
    void notifyRegistration (String studentName, String Course) {
        print ("Notification: $ StudentName has registered for $ Course");
    }
}
```

This mixin encapsulates notification functionality. It defines a method that sends a registration notification.

Step 2: APPLY THE MIXIN

```
Class FinalStudent extends Person
with AttendanceMixin, NotificationMixin
implements Registrable {
    // Class implementation
}
```

I used the 'with' keyword followed by the mixin names.
Notice the comma separation for multiple mixins.

STEP 3: USE THE MIXIN'S METHOD

```
@override
void registerCourse (String CourseName) {
    print ("$ name is registered for $ CourseName);
    notifyRegistration (name, CourseName); // ← Using mixin method
}
```

19. The FinalStudent class can now call notifyRegistration() as if it were its own method. No additional code needed in the class itself.

WHAT THIS ACHIEVES:

(*) SEPARATION OF CONCERNS:
- Student class handles student-specific logic
- AttendanceMixin handles attendance tracking
- NotificationMixin handles notification sending.

Each piece has one clear responsibility.

(*) REUSABILITY:
This NotificationMixin can now be used by Teacher, Course, Admin, or any other class that needs to send notifications.

(*) MAINTAINABILITY: If the notification format changes (email, SMS, push notification), I only need to change ONE file - the mixin. Every class that uses it automatically gets the updated behavior.

(*) COMPOSITION:
FinalStudent now has three sources of behavior:
- Person (inheritance) - base identity
- AttendanceMixin - can track attendance.
- NotificationMixin - can send notifications
- Registrable (interface) - contract to follow

This demonstrates the power of Dart's object-oriented features working together: inheritance FOR IS-A, mixins FOR CAN-DO, interfaces FOR MUST-DO.

---

20. Learning Dart is the foundation for understanding Flutter because FLUTTER IS DART. Every Flutter app is a Dart application.

When I build Flutter screens, I'm writing Dart classes and widgets. When I handle button taps, I'm writing Dart functions (often arrow functions). When I fetch data from an API, I'm using Dart's async/await. When I manage app state, I'm working with Dart collections and objects.

The concepts I learned in this lab directly translate to Flutter:
- Classes
- Constructors
- Inheritance
- Mixins
- Collections
- Async/await.

Without Dart, Flutter is just magic. With Dart, Flutter becomes logical. I now understand that Flutter isn't a separate language - it's a framework BUILT ON DART. Every line of Flutter code is Dart code, following the same rules I practiced in this lab.

This lab has given me the confidence to move from console applications to building real mobile apps with Flutter. The same skills apply.