RTB | RWANDA TVET BOARD

RQFL EVEL 5

SWDMA501

TRAINEE'S

MANUAL

RTB | RWANDA TVET BOARD

KOICA Korea International Cooperation Agency TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate V in Software Development, specifically for the module "**SWDMA501: Mobile Application Development** ".

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) Guiding Policies and Directives

Under Financial and Technical Support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMANA Asher Emmanuel

Production Team

Authoring and Review

HABANABAKIZE Jerome

TUYISINGIZE Leopord

NDAGIJIMANA Enock

Validation

MUKASHYAKA Christine

MUZIMANGANYE Jean Pierre

Conception, Adaptation and Editorial

works HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

NTEZIMANA Aimable

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

SHYAKA Emmanuel

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

AAB: Android App Bundle

API: Application Programming

Interface **APK:** Android Package Kit

CLI: Command Line Interface

E2E: End-to-End

IDE: Integrated Development

Environment **iOS:** iPhone Operating

System

IoT: Internet of Things

IPA: iOS package App Store

KOICA: Korean International Cooperation

Agency **OOP:** Object Oriented Programming

ORM: Object-Relational Mapping

OS: Operating System

PWA: Progressive Web Apps

RTB: Rwanda TVET Board

SDK: Software Development Kit

SPA: Single Page Applications

TQUM Project: TVET Quality Management Project

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in software development specifically for the module of **"Mobile Application Development".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labour market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

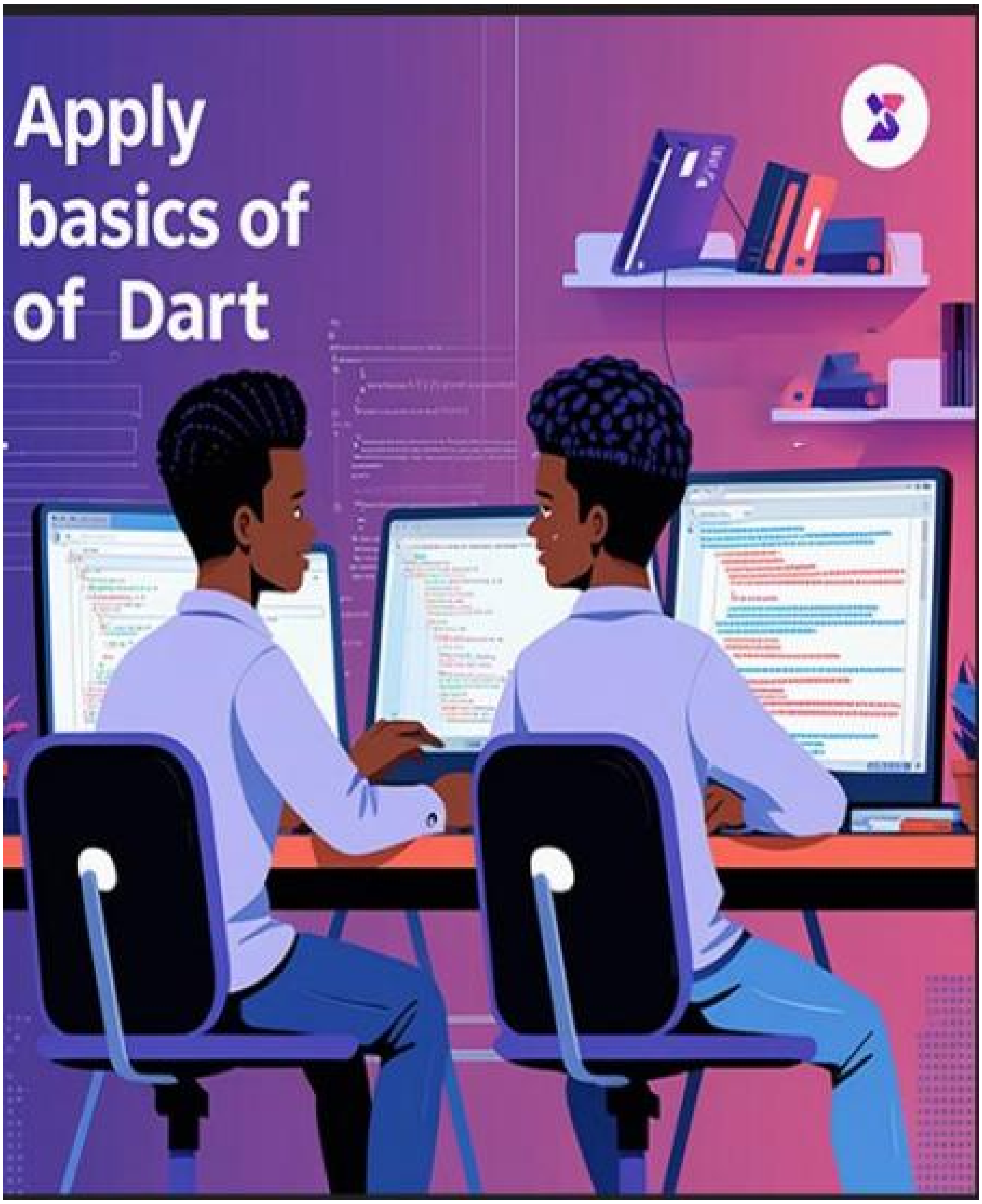
Learning Outcome 1: Apply Basics of Dart

Learning Outcome 2: Implement UI Designs

Learning Outcome 3: Integrate Backend Functionality

Learning Outcome 4: Publish Application

Learning Outcome 1: Apply Basics of Dart



Indicative Contents

1.1 Preparation of Development Environment

1.2 Applying the Dart Concept

1.3 Applying Object Oriented Programming (OOP)

1.4 Using Dart Libraries and Packages

Key Competencies for Learning Outcome 1: Apply Basics of Dart

Knowledge	Skills	Attitudes
● Description of development environment. ● Description of dart concept. ● Description of Object Oriented Programming (OOP) in Dart ● Description of dart libraries and packages	● Installing of key tools (Windows and Apple) ● Applying the dart concept ● Applying Object oriented programming (OOP) in Dart. ● Using dart libraries and packages	● Having Teamwork ability ● Having Passion for Learning ● Having good Collaboration and Communication ● Being attentive for using dart libraries and packages.

Duration: 20 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly the development of environment as used in dart language 2. Describe effectively the dart concept used in dart language.
3. Apply correctly the dart concept as applied in mobile application.
4. Install properly the key tools as applied in Mobile application.
5. Describe correctly the Object-Oriented Programming (OOP) used in dart language. 6. Apply correctly the Object-oriented programming (OOP) in Dart language. 7. Use properly dart libraries and packages as applied in mobile application development.

Resources

Equipment	Tools	Materials
● Computer	● VS Code ● Android Studio ● Dart SDK	● Third party libraries ● Internet ● Electricity

Indicative content 1.1: Preparation of Development Environment

Duration: 5 hrs

Theoretical Activity 1.1.1: Description of Dart concepts

Tasks:

1. You are requested to answer the following questions:
   i. Describe the following key terms:
      ✔ Dart language
      ✔ Data type
      ✔ Variable
      ✔ Native apps
      ✔ Cross platform
   ii. Describe dart features and characteristics
   iii. Identify use cases of dart language.

iv. Differentiate control flow structure and function

2. Write your findings to flipchart/papers
3. Present the findings to the whole class or trainer
4. For more clarification, read the key readings 1.1.1. In addition, ask questions w here necessary.

Key readings 1.1.1: Description of dart concepts

✔ **Introduction to Dart**

Dart is a modern, object-oriented programming language developed by Google. It is specifically designed for building high-performance, cross platform applications. Dart offers a clean syntax, strong typing, and a wide range of built in libraries, making it a powerful tool for app development.

Dart features and characteristics

✔ **Flutter Framework: Dart** is the primary language used for developing mobile applications with Flutter. Flutter is a UI toolkit developed by Google that allows you to create natively compiled applications for mobile, web, and desktop from a single codebase. Flutter has gained popularity for its hot reload feature, expressive UI, and excellent performance.

✔ **Cross-Platform Development:** Dart, in conjunction with Flutter, enables you to build cross-platform applications.

This means you can write code once and run it on both iOS and Android platforms, saving time and effort compared to developing separate codebases for each platform.

✔ **Fast Development:** Dart's hot reload feature in Flutter allows developers to instantly see the effect of the changes made in the code without restarting the entire application. This significantly speeds up the development process and makes it easier to experiment with different UI designs and features.

✔ **Strongly Typed Language:** Dart is a statically typed language, which means that it helps catch errors during development before the code is executed. This can lead to more reliable and maintainable code, especially in larger projects.

✔ **Asynchronous Programming**: Dart has built-in support for asynchronous programming, making it well-suited for tasks that involve handling multiple operations simultaneously. This is particularly useful for developing responsive and efficient applications, especially in scenarios like web development.

✔ **Growing Ecosystem:** Dart's ecosystem is growing, with an increasing number of packages and libraries available through the Dart Package Manager (pub.dev). This can save developers time by leveraging existing solutions for common tasks.

✔ **Google's Support:** Dart is backed by Google, and the company actively uses it in various projects, including some high-profile ones. This support provides a level of confidence in the language's stability and future development.

✔ **Server -Side Development:** While Dart is widely known for client-side development with Flutter, it can also be used for server-side development. The Dart SDK includes libraries for building server-side applications, making it a versatile language for full-stack development.

Dart Framework

Dart has several frameworks that cater to different aspects of development, especially for building web, mobile, and server-side applications. **Here's an overview of popular Dart frameworks:**

1. Flutter
2. Aqueduct
3. Shelf
4. Angel
5. Jaguar
6. GetX

Description of listed dart frameworks

✔ **Flutter**

Purpose: Cross-platform mobile, web, and desktop applications.

Features:

· Rich set of pre-designed widgets for building modern UIs.

· Hot-reload for quick code changes.

· Strong community and ecosystem.

· Allows building apps for Android, iOS, web, macOS, Windows, and Linux. · Good performance due to Dart's ahead-of-time (AOT) compilation. · Use Case: Ideal for building visually attractive and highly performant mobile apps.

Examples: Google Pay, Alibaba, and Reflect.

✔ **Aqueduct (deprecated)**

Purpose: Web framework for building RESTful APIs.

Features:

· Built-in ORM for interacting with databases.

· Dependency injection and middleware support.

· Testing utilities for API development.

· Use Case: Used for backend web services and REST API development in Dart, but currently deprecated. For similar purposes, developers might opt for more general-purpose solutions like `shelf`.

✔ **Shelf**

Purpose: Simple web server framework.

Features:

· Lightweight and flexible.

· Provides a minimalistic structure for handling HTTP requests. · Middleware support for request/response manipulation.

· Use Case: Ideal for creating small web servers, microservices, and API servers with custom routing and middleware.

✔ **Angel**

Purpose: Full-stack server framework.

Features:

· Includes an ORM, templating engine, and real-time capabilities. · Supports GraphQL, WebSockets, and REST.

· Modular architecture for easy plugin management.

· Use Case: Useful for building both server-side applications and APIs with built-in support for multiple protocols like HTTP, GraphQL, etc.

✔ **Jaguar**

Purpose: Dart framework for building APIs.

Features:

· Route handling, middleware support, and ORM for database interaction. · Built for speed and performance.

· Strongly focused on RESTful API design.

· Use Case: Fast web framework designed for building APIs and backend services.

✔ **GetX (for Flutter)**

Purpose: State management, navigation, and dependency injection for Flutter.

Features:

· Powerful state management and easy navigation system.

· Minimal boilerplate code.

· Lightweight, and highly performant.

· Use Case: Commonly used in Flutter projects for managing state, routes, and dependencies.

Note that: These frameworks allow Dart to cover both front-end and back-end development, making it versatile for a wide range of applications. **Use cases of dart language**

✔ **Cross-Platform Mobile App Development (Flutter)**

Use Case: Dart is the primary language for Flutter, Google's popular framework for building cross-platform mobile applications.

Example: Building apps for both Android and iOS with a single codebase, such as Google Pay or Reflectly.

✔ **Web Development**

Use Case: Dart can be compiled to JavaScript, making it suitable for building

client-side web applications.
Example: Progressive web apps (PWAs) and Single Page Applications (SPAs) using Dart’s Flutter for Web.

- ✔ **Desktop Applications (Flutter Desktop)**

  Use Case: With Flutter, Dart can also be used to develop desktop applications for Windows, macOS, and Linux.
  Example: Creating desktop versions of mobile apps using the same Dart codebase.

- ✔ **Backend Development (Web Servers and APIs)**

  Use Case: Dart has server-side frameworks like Shelf, Angel, and Jaguar that can be used to build web servers and RESTful APIs.
  Example: Developing microservices or APIs for mobile applications.

- ✔ **Game Development**

  Use Case: Dart can be used in game development, often in conjunction with Flutter for casual games or simple 2D games.
  Example: Creating mobile games with Flame, a 2D game engine for Flutter.

- ✔ **IoT Development**

  Use Case: Dart's scalability and cross-platform support make it suitable for Internet of Things (IoT) projects where mobile or web apps interact with hardware.
  Example: Building control apps for smart home devices.

- ✔ **Command-Line Tools**

  Use Case: Dart can be used to create command-line tools or scripts for automation tasks, with Dart’s strong package ecosystem aiding in CLI development.
  Example: Writing CLI tools for task automation, file processing, or package management.

- ✔ **Full-Stack Development**

  Use Case: Dart can be used for full-stack development, handling both front-end (with Flutter for Web or Mobile) and back-end (with Shelf or Angel). **Example**: Creating full web and mobile applications with a Dart-powered backend and frontend.

- ✔ **Real-Time Applications**

  Use Case: Dart can be used to build real-time applications such as chat applications, especially when using WebSockets.
  Example: Developing real-time messaging systems or collaborative apps using Dart and Flutter.

- ✔ **Embedded Systems**

  Use Case: While not as common as other languages in this domain, Dart's performance and small footprint can be utilized in embedded systems when combined with Flutter.

Example: Developing companion apps that interact with embedded hardware.

2. Description of key terms

Data type

Data type refer to the classification or categorization of data items based on the kind of value they hold and the operations that can be performed on them. In programming, data types specify what type of data a variable can store, such as numbers, text, or more complex structures like lists or maps. They help the compiler or interpreter understand how to handle and process the data correctly.

Example: int, double, String, Boolean, Lists, Map, Set

Variables

In Dart, **variables** are containers that store data values which can be referenced and manipulated throughout a program. A variable has a name, a type (explicit or inferred), and can hold different types of data such as numbers, strings, or objects. Dart is statically typed, meaning every variable must have a type, but the type can be inferred or explicitly defined.

Control flow Structures

Control flow structures are used to determine the flow of execution in a program based on conditions or repetitive tasks. These include conditional statements, loops, and branching.

Functions

A **function** in Dart is a block of code that performs a specific task. Functions help to make code modular, reusable, and organized. Dart supports both named and anonymous functions.

Native Apps

Native Apps refer to applications developed specifically for a particular operating system or platform using the native programming languages and tools provided by that platform. These apps are designed to run directly on the operating system and take full advantage of its features and functionalities.

Cross Platform

Cross-Platform refers to the development of applications that can run on multiple operating systems or platforms using a single codebase. This approach allows developers to write the app once and deploy it across different platforms, such as iOS and Android.

Practical Activity 1.1.2: Installing key tools (Windows and Apple)

Task:

1: Read Key reading 1.1.2.

2: Referring to the key reading 1.1.2, You are requested to go to the computer lab to install key tools (android studio and vs code).

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 1.1.2: Installing key tools (Visual studio code, Android studio and Dart SDK)

✔ **How to Install Android studio on Windows?**

Step 1: Setting up Android tools and emulator for android devices.

The first step is to download and install Android Studio. To do this, navigate to the official page of **Android Studio** and click on '**Download Android Studio**'.

After accepting the license agreements, you are good to go! Click on the final **Download** button to start downloading.

After the download is complete, let's move on to the next step, i.e. installation. Under '**Components** ', make sure that both Android Studio and Android Virtual Device are checked, and only then proceed. The Android Virtual Device is an essential tool for running various types and sizes of android emulators to test your flutter project. Henceforth, click on '**Next**'.

Select the directory you would want your file to be installed in. It is recommended to select some other path apart from the system drive. Once done, click on '**Next**'.

Step 2: Install android studio

Finally, click on '**Install**'. Wait for a couple of seconds for the installation to complete. Check the box beside '**Launch Android Studio**'. Click on '**Finish**'. Wait for Android Studio to launch on your computer. On the home screen, click **Next > Custom > Next**.

For the Java Development kit location in the next step, it is recommended to keep the default path it requires, to avoid the hassle. In the next step, choose the UI appearance you'd like for Android Studio. Click '**Next**'.

This next step is a bit important. Remember to check the required boxes exactly as shown below. If kits have already been installed, you can ignore those and move on. Click '**Next**'. Set your desired folder for Android SDK.

With that done, click on '**Finish**'. Android Studio will now install all the necessary android tools required for the execution of your flutter projects. This may take a significant time – it's better to wait!

Now, we are ready to create and build flutter projects on Android Studio and run it on a real or a virtual Android device (emulator).

Step 3: Set SDK as an environment variable, for global access.

Now, open Command Prompt terminal and run 'flutter doctor' again. If you have

installed Android SDK in the default directory suggested by Android Studio, there wouldn't be any problem that would appear. Nevertheless, if you have installed it in a non-default directory, flutter would not be able to detect it in your system. To help it able to do that, you guessed it…we would be assigning it as an environment variable, **giving global access**.

As discussed earlier in Step 4, go to environment variables and click '**New**', and do the following (as recommended by flutter doctor). Click '**OK**'.

Step 4: Accept required Android Licenses.
On the Command Prompt terminal, type in:

flutter doctor --android-licenses

as suggested by flutter doctor. Hit Enter. To review licenses, type '**y**' for Yes.

You'll see a couple of repeated prompts that look like this:

Accept? (y/N):

Type '**y**' whenever asked for.

Finally, after all the license agreements have been accepted, you should see a message that looks something like this:

All SDK package licenses accepted

Step 5: Setup Android Emulator.

You have the option to choose between an Android Device or an Android Emulator to build your application on. It depends totally on you.

For setting up Android Emulator, you need to go through the following steps: · Open **Android Studio.**

· On the topmost menu bar, click on **Tools > SDK Manager.**

· Verify whether you have the latest SDK installed. Remember to install the latest stable version too by checking on the box to the left. In my case, it is '**Android 9.0 (Pie)**'. *You can even uncheck the latest version (if not stable), to not only save space but also run all your applications on the stable version itself.*

· Under the '**SDK Tools**' tab, don't forget to check Google USB Driver to later connect a real Android Device. With that, click '**Apply**'. Click '**OK**' to start SDK installation. This might take a couple of minutes to complete. After the setup is done, click on '**Finish**'. Your setup is now complete!

· To have a first look at your Android Emulator, open Android Studio. Go to **Tools > AVD Manager**. A dialog box appears.

· Click on **'Create Virtual Device…'**, select a device and its dimensions according to your preference, select a system image and lastly, under all default settings,

click on 'Finish'. Click on the ' ' button to fire up your emulator.

There you go! You now have a fully functional flutter framework with devices/emulators to build your beautiful apps on.

Installation of virtual studio code on windows

Step1. Download: Get the installer from https://code.visualstudio.com.

Step2. Accept License: Agree to the terms.

Step3. Choose Location: Pick an installation folder (default is fine).

Step4. Select Options: (Recommended)

✔ Add "Open with Code" to context menu

✔ Add to PATH for terminal access

Step5. Install: Click Install and wait for it to finish.

Step6. Launch & Verify: Open VS Code and confirm by typing code in the command prompt.

Points to Remember

- **Dart** is a modern, object-oriented programming language developed by Google.
- Dart Features and Characteristics
  - Flutter Framework
  - Cross-Platform Development
  - Object-Oriented Programming

- There are different Use Cases of Dart:
  - ✔ Mobile Apps
  - ✔ Web Apps
  - ✔ Desktop Apps
- Dart Frameworks:
  - ✔ **Flutter:** for building mobile, web, and desktop apps with Dart.
  - ✔ **Angel:** Full-stack server-side framework supporting HTTP, GraphQL, and WebSockets.
  - ✔ **Shelf:** Lightweight web server for creating HTTP APIs and microservices.
- Description of the following key terms:
  - ✔ **Data Type:** Defines the type of data a variable can hold.
  - ✔ **Variables:** Containers for storing data values.
  - ✔ **Control Flow Structures:** Used for decision-making and looping .
  - ✔ **Functions:** Blocks of code that perform specific tasks.
  - ✔ **Native Apps:** Applications specifically built for a single platform (e.g., iOS or Android) using native tools.
  - ✔ **Cross-Platform:** Apps that run on multiple platforms from a single codebase (e.g., Flutter apps for both iOS and Android).
- While installing Android studio pass through the following steps:
  - ✔ Download android studio setup.
  - ✔ Install android studio setup.
  - ✔ Set SDK
  - ✔ Accept required android licenses.
  - ✔ Set up Android emulator.

- **While installing Visual Studio code pass through the following steps:**
  - ✔ Download Visual studio code setup.
  - ✔ Accept license.
  - ✔ Choose location.
  - ✔ Select option.
  - ✔ Install Visual studio code setup.
  - ✔ Launch and verify.

Application of learning 1.1.

The **Free Company** assigned you to create a cross-platform mobile application using Flutter and Dart. To ensure the application works properly on Windows or macOS, you need to set up a development environment on one of those operating systems.

As mobile developer you are requested to help the company to install the key tools (Visual studio code and Android studio).

Indicative content 1.2: Applying the Dart Concept

Duration: 5 hrs

Practical activity 1.2.1: Performing variable declaration

Tasks:

1: Read Key reading 1.2.1

2: With referring to the key reading 1.2.1, You are requested to go to the computer lab to p erform a program using variable declaration in Dart programming language. 3: Present your work to the trainer/ your colleagues.
4: Ask for clarification if any.

Key readings 1.2.1.: Performing variable declaration

Here are the steps to implement variable declaration, data types, and naming conventions in Dart:

Step 1: Variable Declaration

✔ **Using var:**

- Use var when you want the Dart analyser to infer the type based on the value assigned.
- Example:

✔ **Using final:**

· Use final for variables that should only be set once.
· Example:

✔ **Using const:**

· Use const for compile-time constants that cannot change.
· Example:

Step 2: Data Types

✔ **Numbers:**

· Declare integers and doubles.
· Example:

✔ **Strings:**

· Declare strings using single, double, or triple quotes.
· Example:

✔ **Booleans:**

· Declare boolean values.
· Example:

✔ **Lists:**

· Declare lists of various types.
· Example:

✔ **Maps:**

· Declare maps with key-value pairs.

· Example:

✔ **Sets:**

· Declare sets of unique items.

· Example:

✔ **Dynamic:**

· Use dynamic for variables that can change type.

· Example:

✔ **Null:**

· Declare nullable variables using ?.

· Example:

✔ **Type Inference:**

· Use var, final, or const to let the type be inferred.

· Example:

Step 3: Naming Conventions

✔ **General Guidelines:**

· Use **Camel Case** for variables, methods, and parameters. o
Example: myVariable, calculateSum().

· Use **Pascal Case** for class and enum names.

o Example: MyClass, AnimalType.

· Use **Snake Case** for constants (less common).

o Example: MAX_LENGTH, API_KEY.

✔ **Naming Variables:**

· Use **descriptive names**.
o Example: userName, totalPrice.
· Avoid abbreviations.
o Example: Use customerAccount instead of custAcct. ✔

Naming Methods:

· Use **verb phrases** for method names.
o Example: fetchData(), calculateTotal().
· For boolean methods, prefix with is, has, or can.
o Example: isEmpty(), hasPermission().

✔ **Naming Classes and Enums:**

· Use nouns for class names.
o Example: Car, UserProfile.
· Enum names should use Pascal case; values in uppercase snake case.

o Example:

✔ **Naming Constants:**

· Use uppercase letters with underscores for constants.
o Example: MAX_CONNECTIONS, DEFAULT_TIMEOUT.

✔ **File and Directory Naming:**

· Use lowercase with underscores for file names.
o Example: user_profile.dart, data_service.dart.

✔ **Package Naming:**

· Use all lowercase letters for package names.
o Example: my_awesome_package.

Practical Activity 1.2.2: Applying control flow structures.

Task:

1. Read Key reading 1.2.2
2. Referring to the key reading 1.2.2, You are requested to go to the computer lab to apply control flow structures (conditional statements, sequence switch statement and iterative statements) in dart.
3. Present your work to the trainer/ your colleagues.
4. Ask for clarification if any.

Key readings 1.2.2: Applying control flow structures

Here’s a brief overview of control flow structures in Dart:

Conditional Statements

1.if Statement: Executes a block of code if a specified condition is true.

```
if (condition) {
 // Code to execute if condition is true
}
```

Example:

2.else Statement: Executes a block of code if the condition in the if statement is false.

```
if (condition) {
 // Code for true branch
} else {
 // Code for false branch
}
```

Example:

3.else if Statement: Allows for multiple conditions to be checked in sequence.

```
if (condition1) {
 // Code for condition1
} else if (condition2) {
 // Code for condition2
} else {
 // Code if neither condition is true
}
```

Example:

Sequence switch statements

switch Statement: Evaluates an expression and executes code based on matching cases.

```
switch (value) {
case case1:
// Code for case1
break;
case case2:
// Code for case2
break;
default:
// Code if no cases match
}
```

Example:

Iterating statements

Iterating statements, often referred to as loops, are fundamental control flow structures in programming that allow you to repeatedly execute a block of code as long as a specified condition is true.

In Dart, there are several types of iterating statements:

1. for Loop: Repeats a block of code a specified number of times.

Syntax:

```
for (initialization; condition; increment) {
 // Code to execute on each iteration
}
```

Example:

2. while Loop: Repeats a block of code as long as a specified condition is true. Syntax:

```
Initialization;
while (condition) {
 // Code to execute
}
```

Example:

3.do-while Loop: Similar to the while loop, but it guarantees that the block of code is executed at least once.

Syntax:

```
Initialization;
do {
 // Code to execute
} while (condition);
```

for-in Loop: Used to iterate over elements in a collection (like a list or set).

```
for (var item in collection) {
```

```
// Code to execute for each item
}
```

Example:

4.break: Exits the nearest enclosing loop or switch statement.

Example:

```
for (int i = 0; i < 10; i++) {
if (i == 5) break; // Exit the loop when i equals 5
}
```

5.continue: Skips the current iteration of a loop and proceeds to the next iteration.

Example:

```
for (int i = 0; i < 10; i++) {
if (i % 2 == 0) continue; // Skip even numbers
// Code for odd numbers
}
```

Practical Activity 1.2.3: Performing functions in dart

Task:

1: Read Key reading 1.2.3

2: Referring to the key reading 1.2.3, You are requested to go to the computer lab to perform functions in dart.

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 1.2.3: Performing functions in dart

Here are the implementation steps for using built-in functions, declaring your own functions, specifying parameters and return types, and calling functions in Dart:

Step 1: Using Built-in Functions

1. Printing Output:

· Use the **print()** function to display output in the console.

2. Mathematical Operations:

· Import the dart: math library to access mathematical functions.

3. String Manipulation:

· Use built-in string functions to manipulate text.

4. Converting Data Types:

· Convert strings to numbers and vice versa.

Step 2: Declaring Functions

Syntax for Declaring Functions:

Example:

Step 3: Perform parameters and return types

· Specify the parameter types and the return type in your function declaration.

Example:

Step 4: Calling functions

· Invoke your functions by passing the necessary arguments.

Example:

Points to Remember

· **While performing variable declaration, pass through the following steps:**

✔ Using var, Using final and Using const.

✔ Using datatypes.

✔ Naming variable

· **While applying control flow structures pass through the following steps:** ✔ Use Conditional Statements:Implement if, else to control flow based on conditions.

✔ Sequence switch statements

✔ Iterating statements: Use for loop, while loop, and do-while loop, break and continue.

· **While Performing functions pass through the following steps:**

✔ Using built-in functions

✔ Declaring functions

✔ Perform parameters and return types.

✔ Calling functions.

Application of learning 1.2.

DevSolutions Inc needs to create a simple mobile application that helps employees manage their tasks. The app will allow users to add tasks, mark them as complete, and view their progress. As mobile developer you are requested to perform variable declaration, apply control flow structures and functions in dart.

Duration: 5 hrs

Theoretical Activity 1.3.1: Description of Object-Oriented Programming (OOP)

Tasks:

1: You are requested to answer the following questions:

i. Explain the following terms:

1. Classes
2. Objects
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Abstraction

2: Write your findings to flipchart/papers

3: Present the findings to the whole class or trainer

4: For more clarification, read the key readings 1.3.1. In addition, ask questions whe re necessary.

Key readings 1.3.1: Description of Object-Oriented Programming (OOP)

This will breakdown of each of these concepts in dart:

1. Classes

A **class** is a blueprint or template for creating objects (instances). It defines a set of attributes (variables) and behaviors (methods) that the objects created from the class will have. In Dart, classes provide structure and a way to model real world entities.

Example:

```
class Car {
String model;
int year;
// Constructor
Car(this.model, this.year);
// Method
void displayInfo() {
print('Model: $model, Year: $year');
```

```
}
}
```

2. Objects

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created. Objects hold the data (attributes) and allow access to methods defined in the class. **Example**:

```
void main() {
Car myCar = Car('Tesla', 2022); // Object of class Car
myCar.displayInfo(); // Accessing a method through the
object }
```

3. Inheritance

Inheritance allows one class (child or subclass) to inherit the properties (attributes and methods) of another class (parent or superclass). It helps in code reuse and organizing classes hierarchically. In Dart, the extends keyword is used for inheritance.

Example:

```
class Vehicle {
void start() {
print('Vehicle started');
 }
}

class Car extends Vehicle {
// Car inherits the start method from Vehicle
void honk() {
print('Car is honking');
 }
}

void main() {
 Car myCar = Car();
 myCar.start(); // Inherited from Vehicle
 myCar.honk(); // Defined in Car
}
```

4. Polymorphism

Polymorphism means "many forms." In object-oriented programming, it allows a single interface to represent different data types or classes. This means that a method or property can behave differently based on the object that is invoking it. In Dart, polymorphism is achieved through method overriding.

Example:

```
class Animal {
 void sound() {
 print('Animal makes a sound');
 }
}
```

```
class Dog extends Animal {
 @override
 void sound() {
 print('Dog barks');
 }
}

class Cat extends Animal {
 @override
 void sound() {
 print('Cat meows');
 }
}

void main() {
 Animal myDog = Dog();
 Animal myCat = Cat();

 myDog.sound(); // Dog barks
 myCat.sound(); // Cat meows
}
```

5. Encapsulation

Encapsulation is the practice of hiding the internal details or data of an object from the outside world and only exposing necessary parts via methods. This is typically done using access modifiers like private (by prefixing _ to variable names in Dart) to restrict direct access.

Example:

```
class BankAccount {
 double _balance; // Private variable

 BankAccount(this._balance);

 // Public method to get balance (read-only access)
```

```
 double getBalance() {
 return _balance;
 }

 // Public method to deposit money (controlled access)
 void deposit(double amount) {
 if (amount > 0) {
 _balance += amount;
 }
 }
```

}

6. Abstraction

Abstraction is the concept of hiding complex details and showing only the essential features of an object. In Dart, this can be achieved using abstract classes and interfaces. An abstract class can define methods without an implementation, forcing subclasses to provide the concrete implementation.

Example:

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

class Circle extends Shape {
 @override
 void draw() {
 print('Drawing a circle');
 }
}

class Square extends Shape {
 @override
 void draw() {
 print('Drawing a square');
 }
}

void main() {
 Shape myCircle = Circle();
 Shape mySquare = Square();

 myCircle.draw(); // Drawing a circle
```

```
 mySquare.draw(); // Drawing a square
}
```

Note That:

· **Classes** define the structure of objects.
· **Objects** are instances of classes.
· **Inheritance** enables classes to inherit properties and behaviors from other classes.
· **Polymorphism** allows objects of different types to be treated as instances of the same class through a common interface.
· **Encapsulation** protects data by controlling access to it via methods. · **Abstraction** hides unnecessary details and exposes only the essential features.

Practical Activity 1.3.2: Applying Object Oriented Programming in Dart

OOP
Task:

1: Read Key reading 1.3.2

2: Referring to the key reading 1.3.2, You are requested to go to the computer lab to apply the OOP in dart programming language.

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 1.3.2 Applying Object Oriented Programming in Dart

There are the steps to implement Object-Oriented Programming (OOP) concepts in Dart, including classes and objects, encapsulation, inheritance, polymorphism, and abstraction:

Step 1: Creating classes and objects

1. Create a Class:

· Create a blueprint using the class keyword.

· Define properties and methods within the class.

Example:

2. Create an Object:

· Instantiate an object (instance) of the class using the new keyword or directly.

Example:

Step 2: Creating Encapsulation

1. Bundle Data and Methods:

· Use private variables (by prefixing with _) and public methods to access them.

Example:

Step 3: Creating Inheritance

1. Create a Base Class:

· Define a class that will serve as a parent class.

Example:

2. Create a Derived Class:

· Use the extends keyword to inherit from the base class.

Example:

3. Use Inherited Methods:

Step 4: Creating Polymorphism

1. Define Methods in Base Class:

· Create a method in the base class that can be overridden.

Example:

2.Override in Derived Classes:

3. Use Polymorphism:

Step 5:Creating abstract

1. Define an Abstract Class:

· Use the abstract keyword to create a class that cannot be instantiated directly.

Example:

2. Implement the Abstract Class:

· Create subclasses that implement the abstract methods.

Example:

3. Use Abstraction:

Points to Remember

· Description of Object-Oriented Programming (OOP) terms:

- ✔ **Class** is a blueprint for creating object.
- ✔ **Object** is an instance of class.
- ✔ **Encapsulation**Bundle data and methods, restricting access.
- ✔ **Inheritance** is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and methods from another class

- ✔ **Polymorphism**: Polymorphism is a core principle of object-oriented programming that allows methods to do different things based on the object that it is acting upon.
- ✔ **Abstraction**is a fundamental concept in object-oriented programming (OOP) that involves hiding complex implementation details and exposing only the necessary parts of an object or system.

· While apply Object Oriented Programming, pass through the following steps:

- ✔ Create class and object.
- ✔ Create Inheritance
- ✔ Create Polymorphism
- ✔ Create Encapsulation
- ✔ Create Abstraction

Application of learning 1.3.

Tech Innovations Inc manages different types of employees (e.g., full-time, part-time, and interns) and their specific attributes and behaviours. The system will allow HR personnel to add employees, calculate salaries, and generate reports based on employee types. As mobile developer, you are requested to create classes and objects, inheritance, polymorphism, encapsulation, and abstraction.

Indicative

content 1.4: Using Dart Libraries and Packages

Duration: 5 hrs

Theoretical Activity 1.4.1: Description of dart libraries and packages

Tasks:

1: You are requested to answer the following questions:

a. Discuss the following dart libraries and packagies:

b. Importing and using libraries

c. Exploring built-in Dart libraries

d. Managing dependencies with pub (Dart's package manager)

e. Using external packages for enhanced functionality

2: Involve trainees in presentation of their findings.

3: Provide expert view and clarifies ideas by using didactic materials.

4: Address any questions or concerns.

5: Ask trainees to read the key reading 1.4.1 in the trainee manual.

Key readings 1.4.1: Description of dart libraries and packages

✔ **Importing and using libraries**

Use the **import** keyword to bring in built-in or third-party libraries.

Example:

```
import 'dart:math';
void main() {
 print(pi); // Prints the value of Pi from the 'dart:math' library.
}
```

✔ **Exploring built-in Dart libraries**

· dart:core: Core library for basic Dart functionality (automatically imported). · dart:async: For asynchronous programming, such as using *Future* and *Stream*. · dart:convert: For encoding and decoding JSON, UTF-8, and other conversions

✔ **Managing dependencies with pub (Dart's package manager)**

pubspec.yaml: This file is where dependencies and other project details are listed.

Example: Adding a Dependency:

```
dependencies:
 http: ^0.13.0
```

This adds the http package to your Dart project, which can be used for making

HTTP requests. To install the dependencies, run dart pub get in your project directory.

✔ **Using external packages for enhanced functionality**

1. Find a package on pub.dev (e.g., http for HTTP requests).

2. Add the package to your pubspec.yaml file:

```
dependencies:
 http: ^0.13.0
```

3. Install the package by running dart pub get.

4. Import the package into your code:

```
import 'package:http/http.dart' as http;
```

5. Use the package to add functionality to your app:

```
var response = await http.get(Uri.parse('https://example.com'));
print(response.body);
```

Practical Activity 1.4.2: Performing dart libraries and packages

Task:
in OOP
1: Read Key reading 1.4.2
OOP
2: Referring to the key reading 1.4.2, You are requested to go to the computer lab to perfor m dart libraries and packages

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 1.4.2Performingdart libraries and packages

You are now in the computer lab. Your task is to import built-in Dart libraries and external packages into your project. Follow the steps below:

Step 1: Importing Built-In Libraries

Dart comes with several built-in libraries. Let's import one of the most common ones, such as **dart: math** for mathematical functions.

Example:

```
import 'dart:math';
void main() {
 var randomNumber = Random().nextInt(100); // Generates a random
number between 0 and 99
 print('Random number: $randomNumber');
}
```

Step 2: Importing External Packages

To use external packages, you need to add them to your project. Here's how you do it:

1. Open your **pubspec.yaml** file.
2. Add the external package (for example, http for making HTTP requests). **pubspec.yaml:**
```
dependencies:
 http: ^0.14.0
```
3. Run **flutter pub get** or **dart pub get** to install the package.
4. Now, you can import the package in your Dart/Flutter file.

Example:

```
import 'package:http/http.dart' as http;

void main() async {
 var response = await
http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'));
 print('Response status: ${response.statusCode}');
 print('Response body: ${response.body}');
}
```

Task Instructions:

I. Open the **pubspec.yaml** file in your project and add an external package of your choice **(e.g., http, provider).**

II. Import a built-in Dart library **(dart:math, dart:io, etc**.) and demonstrate how to use its functionality.

III. Use the external package you added in a basic program (e.g., make a simple HTTP request using the http package).

Points to Remember

Description of using dart libraries and packages

- ✔ Importing and using libraries
- ✔ Exploring built-in Dart libraries
- ✔ Managing dependencies with pub (Dart's package manager)
- ✔ Using external packages for enhanced functionality

· **While Performing dart libraries and packages, pass through the following steps:**

- ✔ Importing Built-In Libraries
- ✔ Importing External Packages

Application of learning 1.4.

WeatherWise Solutions is a tech start-up dedicated to providing accurate and timely weather information to users. Their goal is to develop an application that aggregates weather data from various sources, allowing users to access it easily for planning their activities. As a mobile developer, you are requested to set up a dart project for importing built-in libraries and adding external packages to fetch data from an API and store the results locally.

Learning

outcome 1 end assessment

Theoretical assessment:

I. Circle the letter corresponding to the correct answer:

Question 1: What type of programming paradigm does Dart primarily support?

A) Functional programming

B) Object-oriented programming

C) Procedural programming

D) Logic programming

Question 2: Which of the following features is NOT a characteristic of Dart?

A) Strongly typed

B) Just-in-time (JIT) compilation

C) Automatic memory management

D) Lack of null safety

Question 3: Dart Frameworks

Which framework is commonly used with Dart for building web applications?

A) Angular

B) React

C) Flutter

D) Vue.js

Question 4: Dart is primarily used for which of the following applications?

A) Desktop applications only

B) Mobile applications only

C) Web and mobile applications

D) Game development only

Question 5: Which feature of Dart enhances its performance for mobile applications?

A) Dart VM

B) Hot reload

C) Dynamic typing

D) Synchronous programming

Question 6: What is the primary framework for building cross-platform mobile applications using Dart?

A) AngularDart

B) Flutter

C) DartPad

D) Express.js

II. Answer by TRUE to the correct statements and FALSE to the wrong statements. 1) Dart supports both static and dynamic data types, allowing for flexibility in variable declaration.

2) In Dart, variables must be declared with a type before they can be used...........

3) Dart provides standard control flow structures like if, else, for, and switch............ 4) In Dart, functions are first-class citizens, meaning they can be assigned to variables and passed as arguments.

5) Dart can only be used for developing web applications and is not suitable for native app development.

III. Fill the empty by place by using the correct world listed in blacket.

(Class, Encapsulation, Polymorphism, Inheritance, Abstraction, Object).

A)In Dart, a blueprint for creating objects is called a ___________.

B)___________ allows a class to inherit properties and methods from another class, promoting code reusability.

C)___________ is the ability of different classes to be treated as instances of the same class through a common interface.

D)___________ is a principle that restricts access to certain components of an object, allowing only controlled access through methods.

E)___________ is the concept of hiding complex implementation details and exposing only the necessary features of an object.

IV. Match the following terms to their description and write letters in empty place of answers.

Answers	**Term**	**Description**
1……….	1) Using built-in functions	A) Defining a function with specific inputs and outputs.
2………	2) Declaring functions	B) The act of executing a function's code.
3……..	3) Parameters and return types	C) Classification that specifies which type of value a variable can hold.
4………	4) Calling functions	D) Specifying the types of inputs and outputs in a function.
		C) Utilizing functions provided by Dart's libraries.

Practical Assessment

SmartHome Innovations is a technology company focused on developing smart home solutions that allow users to control and monitor their home devices through a centralized application. As a mobile developer you are requested to set up key development tools, apply dart concepts, apply Object-Oriented Programming (OOP), perform dart libraries and packages.

References

https://chatgpt.com/g/g-RGr8YRENd-chart-gpt-3
https://stackoverflow.com/
https://www.geeksforgeeks.org/what-is-ide/
https://www.w3schools.com/
Rose, R. (2022). Flutter and Dart Cookbook 1st Edition. Calfornia: O'Reilly Media.
Thornton, E. (2020). Flutter For Beginners: A Genius Guide to Flutter App Development. KDP: Amazon Digital Services LLC.
Tutorials Point. (2019). Flutter. Retrieved from Tutorials Point (I) Pvt.

Ltd: .

Indicative Contents

2.1 Preparation of Flutter Environment

2.2 Applying Flutter's Widget System

2.3 Implementation of State Management

2.4Using Pre-Designed Widgets

Key Competencies for Learning Outcome 2: Implement UI designs

Knowledge	Skills	Attitudes
● Description of flutter environment ● Description of flutter's widget system. ● Description of state management. ● Description of pre designed widgets	● Preparing Flutter Environment ● Applying flutter's widgets design. ● Implementing state management ● Using pre-designed widgets	● Having a teamwork spirit. ● Being creative in Optimizing Performance ● Being adaptive. ● Having Time management ability ● Being critical thinker ● Being Innovative in Unique Design Solutions ● Having Passion for Learning

Duration: 30 hrs
Learning outcome 2 objectives: By the end of the learning outcome, the trainees will be able to: 1. Describe correctly flutter framework as applied in software development 2. Describe correctly widgets as used in flutter software development. 3. Describe correctly state management as used in flutter software development. 4. Install correctly Flutter SDK, IDE (Android Studio) and Configuration of development environment as used in mobile application development. 5. Create properly a new Flutter project according to development environment. 6. Develop correctly flutter's widget system based on flutter functionalities. 7. Implement state management based on flutter functionalities. 8. Use correctly pre-designed widgets as required in flutter framework.
Resources

Equipment	Tools	Materials
● Computer (Windows, MacOS) ● Mobile Phone (Android, iPhone)	● VS Code ● Android Studio ● Xcode ● Emulators ● Flutter SDK. ● FlutterFlow ● Figma	● Flutter Icons ● Internet

Indicative content 2.1: Preparation of flutter environment

Duration: 7 hrs

Theoretical Activity 2.1.1: Introduction to Flutter framework

Tasks:

Step 1: You are requested to answer the following questions:

I. Define the following terms:

a. Flutter

b. Widgets

c. State management

d. core widgets

e. Widget Tree

ii. What are purposes of flutter framework?

iii. Identify features of flutter framework.

iv. Differentiate types of widgets

v. Differentiate widget Lifecycle from packages used in flutter framework.

vi. What are methods and Libraries used in flutter state management?

Step 2: Involve trainees in presentation of their findings

Step 3: Provide expert view and clarifies ideas by using Flutter Widgets.

Step 4: Address any questions or concerns.

Step 5: Ask trainees to read the key reading 2.1.1 in the trainee manual.

Key readings 2.1.1Introduction to Flutter Framework

Definition

Flutter is an open-source UI software development kit (SDK) created by Google for building natively compiled applications across multiple platforms, including mobile (iOS, Android), web, and desktop (Windows, macOS, Linux) from a single codebase. It uses the Dart programming language to create fast, expressive, and flexible user interfaces.

Purpose

The primary purpose of Flutter is to enable developers to create high performance, visually attractive applications that work seamlessly across different platforms using a single codebase. This minimizes development effort

and allows for faster development cycles while ensuring a native-like experience on each platform.

Features of Flutter

1. **Cross-platform Development**: Write once and deploy to multiple platforms (iOS, Android, web, desktop).
2. **Hot Reload**: Instantly view the effects of changes in the code without restarting the app. This speeds up the development process.
3. **Single Codebase**: One codebase for multiple platforms reduces redundancy and maintenance overhead.
4. **Customizable Widgets**: Flutter provides a rich library of customizable widgets that allow for the creation of complex UIs.
5. **Native Performance**: Flutter compiles to ARM or Intel machine code, ensuring high performance and native-like speed on mobile platforms.

6. **Expressive UI**: With Flutter’s flexible layout and powerful compositing capabilities, developers can easily create beautiful, responsive UIs. 7. **Dart Language**: Powered by Dart, a fast and modern programming language with strong support for async programming and concise syntax. 8. **Rich Ecosystem**: Flutter comes with an extensive package ecosystem, providing tools for networking, database access, state management, and more. 9. **Declarative UI**: UI is built declaratively in Flutter, which makes code more readable and easier to manage.

10. **Support for Animation**: Flutter has built-in support for creating complex animations with ease.

11. **Open Source**: Being open-source, Flutter is supported by a large and active community, along with regular updates and contributions from Google. ✔

Widgets in Flutter

Definition

Widgets are the core building blocks of a Flutter application. They describe the structure, layout, and behavior of UI elements. Every visual component in Flutter is a widget, including text, images, buttons, layouts, and even the entire screen. Flutter uses a composition-based approach, meaning complex UIs are created by nesting smaller widgets.

Types of Widgets

Widgets in Flutter are broadly categorized into two types:

1. **Stateless Widgets**:

- o **Definition**: Stateless widgets do not store any state information. Their appearance and behavior are based solely on the input properties passed to them. They are immutable, meaning once they are built, they cannot be changed.
- o **Example**: Text, Icon, and Image widgets are stateless.

- o **Use Case**: For UI elements that do not require any dynamic change, such as static text or images.

```
class MyStatelessWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
 return Text("Hello, Flutter!");
 }
}
```

2. **Stateful Widgets**:

- o **Definition**: Stateful widgets are dynamic and can change over time in response to user interaction or other events. They have an associated state object that holds information that can be updated.
- o **Example**: Checkbox, Slider, and Forms are examples of stateful widgets. o **Use Case**: For UI elements that need to update or redraw based on user interaction or external data (e.g., a button click changing the UI).

```
class MyStatefulWidget extends StatefulWidget {
 @override
 _MyStatefulWidgetState createState() =>
_MyStatefulWidgetState(); }

class _MyStatefulWidgetState extends State<MyStatefulWidget>
{  String text = "Hello!";

 @override
 Widget build(BuildContext context) {
 return Column(
 children: [
 Text(text),
 ElevatedButton(
 onPressed: () {
 setState(() {
```

```
 text = "Button Pressed!";
 });

child: Text("Press me"),
)
],
 );
```

```
 }
}
```

Widget Lifecycle

The lifecycle of a widget is especially important for **StatefulWidgets**, as it defines the stages of a widget's existence from its creation to its removal.

The key stages of the widget lifecycle are:

1. **createState()**:
   - This is the first method called when a stateful widget is created. It creates an instance of the widget's state object (State).

```
@override
_MyWidgetState createState() => _MyWidgetState();
```

2. **initState()**:
   - This method is called once, immediately after the widget is inserted into the widget tree. It's used for any initial setup, such as subscribing to a service or initializing data.

```
@override
void initState() {
 super.initState();
 // Initialization code here
}
```

3. **build()**:
   - The build() method is called whenever the widget needs to be rendered. This method can be called multiple times during the widget's lifecycle as a result of state changes or updates from the framework.

```
@override
Widget build(BuildContext context) {
 return Text("Building UI");
```

```
}
```

4. **didUpdateWidget()**:
   - Called whenever the widget is rebuilt due to changes in the parent widget. This method provides an opportunity to respond to these changes, such as updating dependent data.

```
@override
void didUpdateWidget(MyWidget oldWidget) {
 super.didUpdateWidget(oldWidget);
 // Handle changes
}
```

5. **setState()**:
   - This method is called to inform Flutter that the widget's state has changed and it needs to rebuild. It triggers the build() method again to update the UI.

```
void updateText() {
 setState(() {
 text = "New Text!";
 });
}
```

6. **deactivate()**:
   - This is called when the widget is removed from the widget tree but may be reinserted before being disposed of. It’s used to release resources or cancel subscriptions temporarily.

```
@override
void deactivate() {
 super.deactivate();
 // Clean up or pause resources
}
```

7. **dispose()**:
   - This method is called when the widget is permanently removed from the tree and will not be rebuilt. It is used for cleanup, such as releasing resources or canceling subscriptions.

```
@override
void dispose() {
 // Clean up resources
 super.dispose();
}
```

✔ **State Management:**

Definition: State management refers to the way an application handles and maintains the state of its components, such as data and UI elements, across different interactions, user inputs, and events. It ensures that the UI reflects the current state of the application and changes dynamically in response to user actions or external events.

Packages:

Definition: A package is a bundle of code that is shared and distributed. It typically includes one or more libraries, tools, assets, and other resources that you can import into your project to add functionality. Packages simplify the reuse of code and are often maintained by the community or organizations.

Purpose: Packages provide additional features or tools that help in development, such as state management, networking, or UI components.

Example (Flutter):

- **GetX**, **Provider**, **Riverpod**, **Bloc** are all **packages** that include tools to manage state.
- **npm** packages in JavaScript like **axios** for HTTP requests or **lodash** for utility functions.

Libraries:

Definition: A library is a collection of pre-written code that provides specific functionality, such as a utility function, framework features, or an entire architecture. Libraries can be a part of a package but focus solely on the code and functionality.

Purpose: Libraries are used to perform common tasks without the need to write code from scratch. They are often more focused than packages and are typically part of a larger package ecosystem.

Example:

o **Redux** is a **library** that helps manage application state in JavaScript. o **RxJS** is a reactive programming library in Angular for handling asynchronous data.

o **Lodash** is a JavaScript utility library to simplify working with arrays, numbers, objects, etc.

Methods:

· **Definition:** Methods refer to specific functions or techniques used within code to achieve a particular goal. They could be part of a class, object, or even a standalone function in programming languages. In the context of state management, methods are specific techniques or patterns used to manage state.

· **Purpose:** Methods perform actions or operations. In state management, methods might refer to how you update the state, manage side effects, or handle user input.

· **Example:**

○ In **Redux**, methods like dispatch() send actions to reducers. ○ In **GetX**, methods like update() are used to notify the UI of state changes.

○ In a **StatefulWidget** in Flutter, methods like setState() are used to update the widget's state.

Note That:

Packages: are comprehensive bundles that can include multiple libraries and tools.

Libraries: are collections of code that focus on specific functionality.

Methods: are individual functions or techniques used to implement the logic.

Practical Activity 2.1.2: Preparation of flutter environment

Task:

1: Read Key reading 2.1.2

2: Referring to the key reading 2.1.2, You are requested to go to the computer lab to prepare flutter environment and create new flutter project.

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 2.1.2 Preparation of flutter environment:

1. Install Flutter SDK

· Go to the official Flutter SDK website and follow the steps for your platform (Windows, macOS, or Linux).

For Windows:

1. Download the latest Flutter SDK.

2. Extract the file to your desired location (e.g., C:\src\flutter).

3. Add the Flutter bin directory to your system's PATH.

- Right-click on "This PC" > Properties > Advanced system settings > Environment Variables.

- Under "System Variables," select Path, click "Edit," and add the flutter\bin path.

For macOS:

1. Download the latest Flutter SDK from here.

2. Extract it and update your .bash_profile or .zshrc to include the flutter/bin path:

```
export PATH="$PATH:`pwd`/flutter/bin"
```

For Linux:

1. Download the Flutter SDK from here.

2. Extract it and add the flutter/bin directory to your PATH.

2. Install Android Studio for all platform or Xcode for macOS

· **Android Studio:**

1. Download and install Android Studio from the official

site

2. During installation, ensure you install the Android SDK, Android SDK Platform-Tools, and Android Virtual Device (AVD).

3. Open Android Studio and go to "Configure" > "Plugins" and install the **Flutter** plugin.

· **X
c
o
d
e**

**(
f
or macOS only):**

1. Install Xcode from the App Store.

2. Open Xcode and go to "Preferences" > "Locations" > Command Line Tools, and select the version of Xcode you're using.

3. Configure Flutter Development Environment

· **For Android:**

1. Open Android Studio and configure the AVD (Android Virtual Device) for running Flutter apps.

2. Open the terminal and run:

```
flutter doctor
```

· This command checks if your environment is set up correctly. Follow any prompts to fix missing dependencies.

· **For iOS (macOS only):**

1. Ensure Xcode is installed and configured.

2. Run:

```
flutter doctor
```

4. Create a New Flutter Project

1. Open Android Studio.

2. Go to **File > New > New Flutter Project**.

3. Choose **Flutter Application** and click **Next**.

4. Set the **Project Name** (e.g., my_flutter_app), choose the Flutter SDK path, and click **Next**.

5. Configure the package name and platform, then click **Finish**.

6. Flutter create project name(create new project using command line Interface)

To create a new Flutter project using the command line interface (CLI), follow these steps:

I. Open your terminal or command prompt.

II. Navigate to the directory where you want to create the new Flutter project.

III. Run the following command:

```
flutter create project_name
```

Replace project_name with the name you want to give your project.

For example:

```
flutter create my_motivation_app
```

This command will generate a new Flutter project with the default directory structure, and you can start developing from there. After it's created, you can navigate into your new project folder:

```
cd my_motivation_app
```

5. Create a New Activity Named HomeActivity with a Text Widget

1. Open the lib/main.dart file in your new project.

This is the default program and runs in external Emulator (Mobile Phone)

You can test your Flutter app on mobile devices using two primary methods: via **USB Debugging** (for a physical device) and using an **Emulator**. Here's how to set up both:

1. Testing on a Physical Device via USB Debugging

Step 1: Enable USB Debugging

1. **On Android:**

- Open **Settings** on your Android device.
- Go to **About Phone** and tap on **Build number** seven times to enable Developer Mode.
- Go back to **Settings**>**Developer options** and turn on **USB Debugging**. 2. **On iOS**

(for macOS users):

- Connect your device to your Mac.
- Trust your computer on your iPhone.
- Enable **developer mode** if prompted.

Step 2: Connect Your Device

· Connect your Android or iOS device to your computer via USB.

Step 3: Verify Device Connection

· Open the terminal or command prompt and run:

```
flutter devices
```

This will list connected devices. If your device appears in the list, you're ready to run the app.

Step 4: Run the App

· In Android Studio:

o Click the green **Run** button, and select your connected device as the target.

· Via the terminal:

o Run:

```
flutter run
```

· Your app should launch on the connected physical device.

2. Testing on an Emulator

Step 1: Create an Android Emulator or Use iOS Simulator

· **Android Studio (Android Emulator):**

1. Open Android Studio.

2. Go to **AVD Manager** (Android Virtual Device Manager) from the toolbar (click the device icon).

3. Click **Create Virtual Device** and choose a device model. 4. Select a system

image (e.g., Android 11 or above), and click **Finish**.

5. Once created, launch the emulator by clicking the green play icon next to the emulator in AVD Manager.

· **Xcode (iOS Simulator - macOS only):**

1. Open Xcode.

2. Go to **Xcode**>**Preferences**>**Components** to install available simulators.

3. Once installed, launch the simulator from **Xcode**>**Open Developer Tool**>**Simulator**.

Step 2: Run the App

· **In Android Studio:**

o Choose the emulator as the target device, then click the **Run** button. · **Via**

Terminal:

o Start the emulator using:

```
flutter emulators --launch <emulator_id>
```

o Run the app:

```
flutter run
```

· The app will launch on the emulator or iOS simulator.

These are the two ways to test your Flutter app: on a physical device using USB debugging or on an emulator.

2. Replace the code with the following to create a new HomeActivity:

```
import 'package:flutter/material.dart';

void main() {

 runApp(MyApp());

}

class MyApp extends StatelessWidget {

 @override

 Widget build(BuildContext context) {
```

```
 return MaterialApp(

 title: 'Flutter Demo',

 home: HomeActivity(),

 );

 }}

class HomeActivity extends StatelessWidget {

 @override

 Widget build(BuildContext context) {

 return Scaffold(

 appBar: AppBar(

 title: Text('Home Activity'),

 ),

 body: Center(

 child: Text('Hello, HomeActivity!'),

 ),

 );

 }}
```

6. Run and Test the Project

1. **In Android Studio:**

○ Select your device (emulator or connected physical device). ○ Click

the green "Run" button in Android Studio to run the app. 2. **In the**

terminal:

○ Navigate to the project directory and run:

```
flutter run
```

Once you have done this, the app should run, displaying "Hello, HomeActivity!" on the screen.

Points to Remember

✔ **Description of Flutter framework**

- Flutter is an open-source UI software development kit created by Google. It enables developers to build natively compiled applications for mobile, web, and desktop from a single codebase.
- Flutter's important characteristics include Single Codebase, Rich UI Components, High Performance, Hot Reload, Flexible Layouts, State Management Options.
- The primary purpose of Flutter is to enable developers to create high quality, natively compiled applications for multiple platforms using a single codebase. · Lifecycle is especially important for **StatefulWidgets**, as it defines the stages of a widget's existence from its creation to its removal.
- Widgets in flutter are broadly categorized into two types: Stateless and Stateful Widgets

✔ **Preparation of flutter environment**

- **While Installing Flutter SDK pass through the following steps:**

  Download the latest Flutter SDK.

  Extract the file to your desired location (e.g., C:\src\flutter).

  Add the Flutter bin directory to your system's PATH.

- **While creating new flutter project pass though the following steps:**

  Open Android Studio

  Go to **File > New > New Flutter Project**.

Choose **Flutter Application** and click **Next**.

Set the **Project Name** (e.g., my_flutter_app), choose the Flutter SDK path, and click **Next**.

Configure the package name and platform, then click **Finish**

Application of learning 2.1.

KUPONI LTD is a software development company located in Kicukiro district, focusing on developing cross-platform mobile applications. They have a project aimed at tracking the fitness of gym members and intend to use Flutter technology for its development. As the lead developer at KUPONI LTD, your role is to help the company prepare the Flutter environment and create a new Flutter project for this initiative.

Indicative content 2.2: Applying flutter’s widget system

Duration: 8 hrs

Theoretical Activity 2.2.1 Description of flutter widget's system

Tasks:

1: You are requested to answer the following questions:

I. Differentiate stateful and stateless widget

II. Explain the concept of widget composition in Flutter. How can you combine different widgets to create complex UI layouts?

III. What are the primary differences between the Row and Column widgets? IV. What is the purpose of the Expanded widget?

V. How can you customize the appearance of text in Flutter, including font, size, color, and alignment

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.2.1. In addition, ask questions where necessary.

Key readings 2.2.1:Description of flutter widget's system

✔ **Stateful and Stateless Widgets**

Stateless Widgets: These are immutable and do not store any state. They are built once and their properties are final. Use Stateless widgets when the UI does not change based on user interaction or other events. For example, a simple text label or an icon.

o **Example**:

```
class MyStatelessWidget extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
 return Text('Hello, Flutter!');
 }
}
```

Stateful Widgets: These can change their state during the app's lifetime. They can rebuild their UI when the internal state changes. Use Stateful widgets for interactive components like forms, buttons, and other elements that require a

dynamic UI.

Example:

```
class MyStatefulWidget extends StatefulWidget {
 @override
 _MyStatefulWidgetState createState() =>
_MyStatefulWidgetState(); }
```

```
class _MyStatefulWidgetState extends State<MyStatefulWidget>
{  int counter = 0;

 void _incrementCounter() {
 setState(() {
 counter++;
 });
 }

 @override
 Widget build(BuildContext context) {
 return Column(
 children: [
 Text('Counter: $counter'),
 ElevatedButton(
 onPressed: _incrementCounter,
 child: Text('Increment'),
 ),
 ],
 );
 }
}
```

✔ **Widget Tree and Hierarchy**

Parent-child Relationships: In Flutter, widgets are organized in a tree structure, where each widget can have a parent and may have multiple children. The top level widget is the root of the tree, often a MaterialApp or CupertinoApp.

Widget Composition: You can create complex UIs by combining simple widgets. For instance, you can use Rows and Columns to arrange widgets in a grid-like layout.

Row and Column

Row: Arranges its children in a horizontal manner. It takes up the minimum height required by its children.