* **Module 2: Dart Programming Essentials**

**Theory Assignments:**

1. **Explain the fundamental data types in Dart (int, double, String, List, Map, etc.) and their uses.**

Dart is a statically typed language, meaning every variable has a defined type. Understanding its fundamental data types is essential for writing clean, efficient, and bug-free code. Here's a breakdown of the core data types and their typical uses:

* int (Integer)
* **Definition**: Represents whole numbers without a decimal point.
* **Example**: int age = 30;
* **Use Case**: Counting items, indexing, loops, and any scenario where fractional values aren't needed.
* double (Floating Point)
* **Definition**: Represents numbers with decimal points.
* **Example**: double price = 19.99;
* **Use Case**: Financial calculations, measurements, percentages, or any value requiring precision.
* String
* **Definition**: A sequence of characters used to represent text.
* **Example**: String name = "Dart";
* **Use Case**: Displaying messages, storing user input, manipulating textual data.
* List (Array)
* **Definition**: An ordered collection of objects.
* **Example**: List fruits = ["apple", "banana", "cherry"];
* **Use Case**: Storing sequences of items, iterating over elements, managing collections.

Variants:

* List: List of integers
* List: Can hold mixed types
* Map (Dictionary)
* **Definition**: A collection of key-value pairs.
* **Example**: Map scores = {"Alice": 90, "Bob": 85};
* **Use Case**: Fast lookup by key, storing structured data, configuration settings.
* bool (Boolean)
* **Definition**: Represents true or false.
* **Example**: bool isLoggedIn = true;
* **Use Case**: Conditional logic, flags, toggles.
* dynamic
* **Definition**: A type that can hold any value. Type checking is deferred until runtime.
* **Example**: dynamic value = "Hello"; value = 42;
* **Use Case**: When type is unknown or flexible, but use cautiously to avoid runtime errors.
* null
* **Definition**: Represents the absence of a value.
* **Example**: String? name = null;
* **Use Case**: Optional values, uninitialized variables, error handling.
* Summary Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Description** | **Example** | **Common Use Case** |
| int | Whole numbers | int count = 5; | Loop counters, indexing |
| double | Decimal numbers | double temp = 36.6; | Calculations, measurements |
| String | Text | String city = "Paris"; | Messages, labels, user input |
| List | Ordered collection | List nums = [1,2,3]; | Arrays, sequences |
| Map | Key-value pairs | Map | Lookup tables, structured data |
| bool | True/false | bool isOpen = false; | Conditions, flags |
| dynamic | Flexible type | dynamic x = "hi"; x = 10; | Generic containers, loose typing |
| null | No value | String? name = null; | Optional fields, error states |

1. **Describe control structures in Dart with examples of if, else, for, while, and switch.**

Control structures in Dart allow you to manage the flow of your program based on conditions and loops. Here's a clear breakdown of the most commonly used control structures with examples:

* if and else

Used to execute code based on a condition.

**int score = 85;**

**if (score >= 90) {**

**print("Excellent");**

**} else if (score >= 75) {**

**print("Good");**

**} else {**

**print("Needs Improvement"); }**

* **Use Case**: Decision-making based on one or more conditions.
* for Loop

Used to repeat a block of code a specific number of times.

**for (int i = 0; i < 5; i++) {**

**print("Iteration $i");**

**}**

* **Use Case**: Iterating over arrays, performing repeated actions.
* while Loop

Repeats a block of code as long as the condition is true.

**int count = 0;**

**while (count < 3) {**

**print("Count is $count");**

**count++; }**

* **Use Case**: Looping when the number of iterations isn’t known in advance.
* do-while Loop

Similar to while, but guarantees the loop runs at least once.

**int count = 0;**

**do {**

**print("Count is $count");**

**count++;**

**} while (count < 3);**

* **Use Case**: When you want the loop to execute before checking the condition.
* switch Statement

Used to execute different code blocks based on the value of a variable.

**String grade = "B";**

**switch (grade) {**

**case "A": print("Excellent");**

**break;**

**case "B": print("Good");**

**break;**

**case "C": print("Average");**

**break;**

**default: print("Invalid grade"); }**

* **Use Case**: Cleaner alternative to multiple if-else statements when checking a single value.
* Summary Table

|  |  |  |
| --- | --- | --- |
| **Structure** | **Purpose** | **Example Keyword** |
| if/else | Conditional branching | if, else |
| for | Fixed iteration loop | for |
| while | Conditional loop (pre-check) | while |
| do-while | Conditional loop (post-check) | do, while |
| switch | Multi-branch based on value | switch, case |

1. **Explain object-oriented programming concepts in Dart, such as classes, inheritance, polymorphism, and interfaces.**

1. **Classes and Objects**

A **class** is a blueprint for creating objects. It encapsulates data (fields) and behavior (methods).

**class Car {**

**String brand;**

**int speed;**

**Car(this.brand, this.speed);**

**void accelerate() {**

**speed += 10;**

**print('$brand is now going at $speed km/h');**

**}**

**}**

**void main() {**

**Car myCar = Car('Tesla', 50);**

**myCar.accelerate(); // Tesla is now going at 60 km/h**

**}**

2. **Inheritance**

Inheritance allows one class to inherit properties and methods from another. Dart uses the extends keyword.

**class Vehicle {**

**void honk() => print('Beep beep!');**

**}**

**class Bike extends Vehicle {**

**void ride() => print('Riding the bike');**

**}**

**void main() {**

**Bike myBike = Bike();**

**myBike.honk(); // Inherited method**

**myBike.ride(); // Own method**

**}**

3. **Polymorphism**

Polymorphism means "many forms"—you can treat objects of different classes through a common interface or superclass.

**class Animal {**

**void speak() => print('Animal speaks');**

**}**

**class Dog extends Animal {**

**@override**

**void speak() => print('Dog barks');**

**}**

**class Cat extends Animal {**

**@override**

**void speak() => print('Cat meows');**

**}**

**void main() {**

**List animals = [Dog(), Cat()];**

**for (var animal in animals) {**

**animal.speak(); // Dog barks, Cat meows**

**}**

**}**

4. **Interfaces**

Dart doesn’t have a separate interface keyword. Instead, **every class can act as an interface**, and you implement it using implements.

**class Printable {**

**void printData();**

**}**

**class Document implements Printable {**

**@override**

**void printData() => print('Printing document...');**

**}**

**void main() {**

**Document doc = Document();**

**doc.printData(); // Printing document...**

**}**

**Note:** implements requires you to override all methods, while extends allows inheritance with optional overrides.

1. **Abstract Classes**

Used when you want to define a base class that **cannot be instantiated** directly.

**abstract class Shape {**

**void draw(); // Abstract method**

**}**

**class Circle extends Shape {**

**@override**

**void draw() => print('Drawing a circle');**

**}**

Would you like me to show how these concepts apply in a Flutter UI scenario? I can walk you through a widget-based example using inheritance or polymorphism.

1. **Describe asynchronous programming in Dart, including Future, async, await, and Stream.**

Sure! Dart’s asynchronous programming model is designed to handle tasks like I/O operations, network requests, and long-running computations without blocking the main execution thread. Here's a breakdown of the key concepts:

**1. Future**

A Future represents a potential value or error that will be available at some time in the future.

* It’s like a promise that something will be returned later.
* You can attach callbacks using .then() or handle errors with .catchError().

**Future<String> fetchData() {**

**return Future.delayed(Duration(seconds: 2), () => 'Data loaded');**

**}**

**2. async and await**

These keywords simplify working with Futures by allowing you to write asynchronous code that looks synchronous.

* async marks a function as asynchronous and returns a Future.
* await pauses execution until the Future completes.

**Future<void> loadData() async {**

**print('Loading...');**

**String data = await fetchData();**

**print(data); // Outputs: Data loaded**

**}**

**3. Stream**

A Stream is a sequence of asynchronous events. It’s useful when you expect multiple values over time (like user input, sensor data, or WebSocket messages).

* You can listen to a stream using .listen().
* Streams can be single-subscription or broadcast.

**Stream<int> countStream() async\* {**

**for (int i = 1; i <= 5; i++) {**

**await Future.delayed(Duration(seconds: 1));**

**yield i;**

**}**

**}**

**void main() {**

**countStream().listen((value) => print('Count: $value'));**

**}**

**Putting It All Together**

Dart’s asynchronous model is single-threaded but uses an event loop to manage tasks efficiently. Here's how the pieces fit:

|  |  |  |  |
| --- | --- | --- | --- |
| **Concept** | **Purpose** | **Returns** | **Usage Example** |
| Future | Represents a delayed result | Future | Future.delayed(...) |
| async | Marks a function as asynchronous | Future | Future myFunc() async |
| await | Waits for a Future to complete | T | var result = await myFuture |
| Stream | Emits multiple async values | Stream | stream.listen(...) |

Want to dive deeper into StreamController, error handling, or isolate-based concurrency in Dart?