Asynchronous Programming

Class: Asynchronous Programming

Lesson Objectives

- Understand the non-blocking nature of Dart.
- Master the `Future` API and the `async`/`await` syntax.
- Handle errors effectively using `try-catch` and `.catchError`.
- Manage multiple asynchronous operations simultaneously.
- Grasp the basics of `Stream` for handling sequences of asynchronous events.

Topic 1: Futures, Async, and Await

I. High-level Overview

In Dart, operations that take time (like reading a file or downloading data) don't stop the rest of your code from running; instead, they return a "promise" that a result will be available later.

II. Definition

Future: An object representing a delayed computation. It has two states: Uncompleted (pending) and Completed (with a value or an error).

Async/Await: Syntactic sugar that allows you to write asynchronous code that looks and behaves like synchronous code, pausing execution only within the specific function until the Future completes.

III. Example

```
// Simulating a network request that takes 2 seconds
Future<String> fetchUserData() async {
```

```
  print('Fetching data...');

  // Pause this function for 2 seconds, but don't block the app
  await Future.delayed(Duration(seconds: 2));

  return 'User: John Doe';
}

void main() async {
  print('App Start');

  // Wait for the future to complete before moving to the print statement
  String user = await fetchUserData();

  print(user);
  print('App End');
}
```

IV. Real-world Simple Use Case

Login Screen: When a user clicks "Login," the app sends credentials to a server. The app shouldn't freeze while waiting for the server's response. Instead, it shows a loading spinner (awaiting the Future) and updates the UI once the server responds.

Topic 2: Future Chaining & Error Handling

I. High-level Overview

Sometimes you need to perform actions immediately after a task finishes without using `await`, or you need to handle failures gracefully to prevent the application from crashing.

II. Definition

Chaining (.then): A method to register a callback that triggers when a Future completes successfully.

Error Handling (.catchError / try-catch): Mechanisms to intercept exceptions thrown during asynchronous operations.

III. Example

```
Future<int> parseData(String data) async {
  if (data.isEmpty) throw Exception('Data is empty!');
  return data.length;
}

void main() {
  // Using .then and .catchError (Functional style)
  print('Starting operation...');

  parseData("")
      .then((length) ⇒ print('Length: $length'))
      .catchError((e) ⇒ print('Error caught: $e'))
      .whenComplete(() ⇒ print('Operation finished regardless of outcome.'));

  print('This prints BEFORE the error because we did not "await" above.');
}
```

IV. Real-world Simple Use Case

Analytics Logging: When a user performs an action, you might fire a network request to log this event (`analytics.logEvent()`). You don't need to pause the user's experience to wait for the log to succeed; you just fire it, chain a `.catchError` to log any failures silently to the console, and let the user continue.

Topic 3: Handling Multiple Tasks (Future.wait)

I. High-level Overview

This technique allows you to trigger multiple independent asynchronous tasks simultaneously and wait for all of them to finish before proceeding.

II. Definition

`Future.wait`**:** A static method that accepts a list of Futures and returns a new Future that completes with a list of results only when *all* provided Futures have completed.

III. Example

```
Future<String> fetchProfile() async ⇒ Future.delayed(Duration(seconds: 1), () ⇒ 'Profile Data');
Future<String> fetchSettings() async ⇒ Future.delayed(Duration(seconds: 2), () ⇒ 'Settings Data');

void main() async {
  print('Loading Dashboard...');
  final stopwatch = Stopwatch()..start();

  // Both run in parallel, effectively taking only as long as the slowest task (2s)
  final results = await Future.wait([fetchProfile(), fetchSettings()]);

  print('Loaded: ${results[0]} and ${results[1]}');
  print('Time elapsed: ${stopwatch.elapsed.inSeconds} seconds');
}
```

IV. Real-world Simple Use Case

App Initialization: When your app starts, you need to fetch the User Profile, App Configuration, and Notifications. Instead of fetching them one by one (taking 1s + 1s + 1s = 3s), you fetch them all at once (taking max(1s) = 1s), significantly reducing the loading screen time.

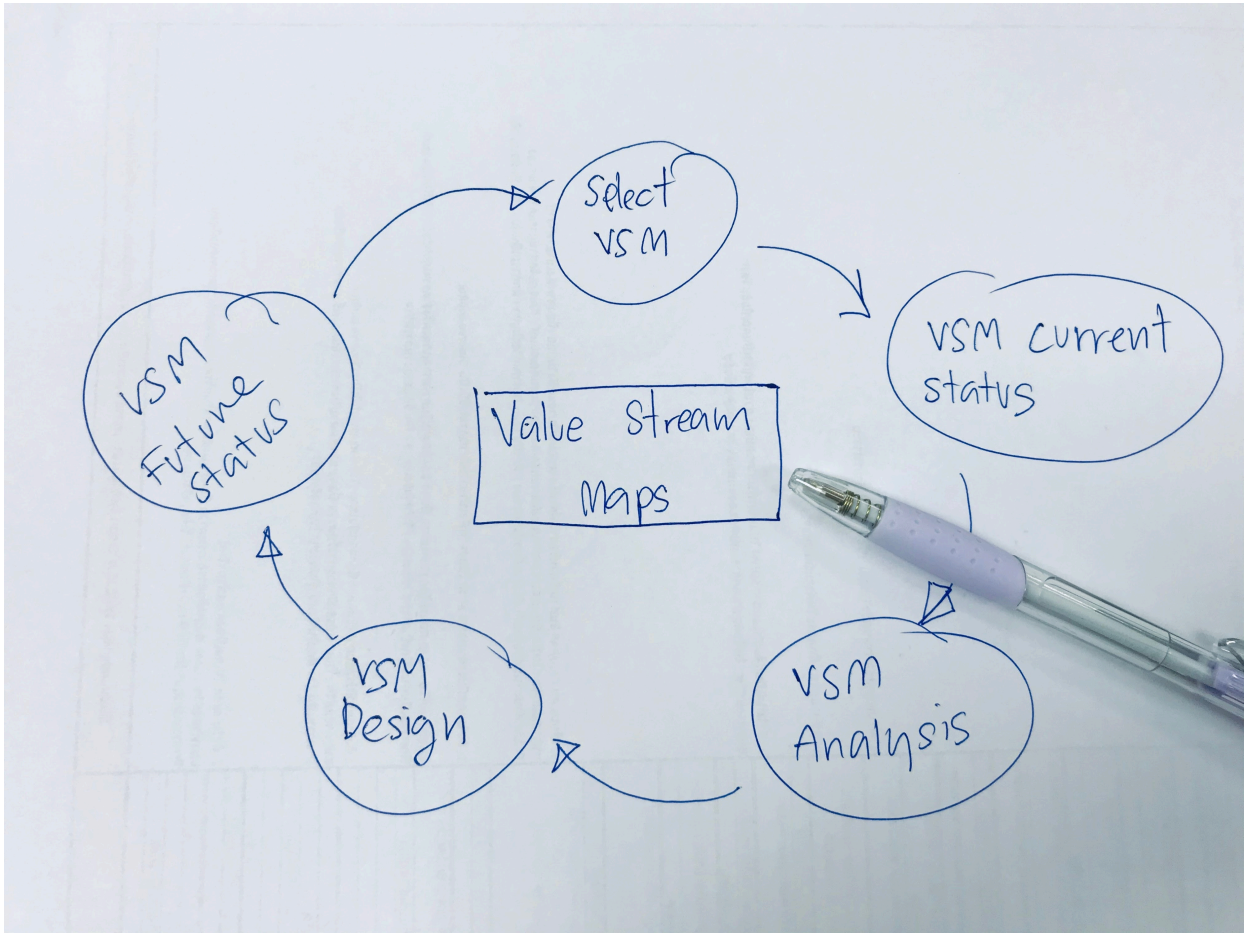
Topic 4: Streams Basics

I. High-level Overview

While a Future represents a *single* value delivered later, a Stream represents a *sequence* of values delivered over time.

II. Definition

Stream: A source of asynchronous data events. It is like an iterator that pushes data to you (the listener) whenever it arrives, rather than you asking for the next item.



III. Example

```
Stream<int> countSeconds(int max) async* {
  for (int i = 1; i <= max; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i; // 'yield' pushes data to the listener
  }
}

void main() {
```

```
  print('Timer started');

  // Listen to the stream
  countSeconds(3).listen(
    (time) ⇒ print('Tick: $time'),
    onDone: () ⇒ print('Timer finished!'),
  );
}
```

IV. Real-world Simple Use Case

Chat Application: In a chat app, you don't just receive one message. You open a connection (Stream) to the server, and every time a friend sends a message, the server pushes that new piece of data through the stream to your device to display immediately.

V. Problems for Learners

Problem 1: The Broken Fetch

Write a function fetchUsername that simulates a network delay of 2 seconds. Randomly, it should either return "DartMaster" or throw an exception "Server Error". Call this function in main using try-catch blocks to handle both scenarios.

Problem 2: The Race

Create three functions: runnerA, runnerB, and runnerC. Each waits for a random duration between 1 and 3 seconds and returns their name. Use Future.wait to start them all at once and print "Race Finished" only when all three are done.

Problem 3: The Countdown

Create a Stream that emits a number every second starting from 10 down to 1. When the stream emits 0, print "Blast off!". Listen to this stream in your main function.

VI. In-class Example: The Dashboard Simulator

This example simulates a real-world scenario where an application needs to fetch data from different services (User, Weather, News) with varying delays, handle potential errors, and utilize a continuous stream for a "Live Ticker."

```
import 'dart:async';
import 'dart:math';

// 1. Mock API Service
class ApiService {
  final Random _rng = Random();

  Future<String> getUserName() async {
    await Future.delayed(Duration(seconds: 1));
    return "Alice";
  }

  Future<int> getNotifications() async {
    await Future.delayed(Duration(seconds: 2));
    // Simulate random failure
    if (_rng.nextBool()) {
      throw Exception("Failed to fetch notifications");
    }
    return 5;
  }

  Future<String> getWeather() async {
    await Future.delayed(Duration(seconds: 3));
    return "Sunny, 25°C";
  }

  // Stream for a live stock ticker or news feed
  Stream<String> getLiveNewsTicker() async* {
    List<String> news = ["Market up", "Dart 3.0 released", "Flutter is awesom
e"];
    for (var item in news) {
      await Future.delayed(Duration(milliseconds: 500));
```

```
      yield item;
    }
  }
}

void main() async {
  final api = ApiService();
  print("--- Dashboard Loading ---");

  // 2. Handling Multiple Futures (some critical, some not)
  try {
    // We need User and Weather to show the main screen.
    // We run them in parallel.
    final criticalData = await Future.wait([
      api.getUserName(),
      api.getWeather()
    ]);

    print("Welcome back, ${criticalData[0]}!");
    print("Current Weather: ${criticalData[1]}");

  } catch (e) {
    print("Critical Error loading dashboard: $e");
    return; // Stop app if user/weather fails
  }

  // 3. Handling independent non-critical errors
  // We try to get notifications, but if it fails, we show 0 instead of crashing.
  int notifications = 0;
  try {
    notifications = await api.getNotifications();
  } catch (e) {
    print("Warning: Could not load notifications (Server error)");
  }
  print("You have $notifications unread notifications.");
```

```
  print("\n--- Connecting to Live Feed ---");

  // 4. Listening to a Stream
  // We 'await' the stream subscription to ensure we catch all events if needed,
  // or just let it run in background. Here we use 'await for' loop.
  await for (String newsItem in api.getLiveNewsTicker()) {
    print("BREAKING NEWS: $newsItem");
  }

  print("--- Dashboard Ready ---");
}
```

Summary

- **Futures** are for single async values (like a package arriving in the mail).
- **Async/Await** makes reading async code easier, avoiding "callback hell."
- **Future.wait** allows parallel execution, saving time.
- **Streams** are for continuous data flows (like a radio broadcast).
- Always handle errors (`try-catch` or `.catchError`) to prevent your app from crashing when the network or database fails.