Dart Session 3

Topics Covered:

1. Quick recap - quiz
2. Synchronous vs Asynchronous Code
3. Futures
4. Async & Await
5. Understanding Delays
6. Real Code Examples

Quick recap – through a short quiz

Synchronous vs. Asynchronous Code

Synchronous:

Synchronous Execution: Tasks run sequentially, one after another, where each operation must fully complete before the next one begins, creating a single linear flow.

The Single Thread: In Dart/Flutter, the UI and main logic run on a single line of execution (the main thread).

Impact in Code: If one task takes 5 seconds (e.g., fetchData()), the entire single thread stops. The UI becomes unresponsive.

Asynchronous:

Asynchronous Execution: Non-blocking approach where tasks can start and run independently, without waiting for previous operation to fully complete.

Common asynchronous operations:

- Fetching data over a network.
- Writing to a database.
- Reading data from a file.

The goal of Asynchronous: Delegate the task to a background worker and return control to the Main Thread immediately.

Solution: Async, await and future

Analogy to further explain the concept:

Dart Concept	Analogy Role	Explanation
Synchronous	**Making the Pizza Yourself**	You decide to make the pizza from scratch. You start chopping, kneading, and waiting for the dough to rise. **You cannot do *anything* else** (like watching TV or playing a game) until the entire process is complete. **This is blocking.**
Future	**The Order Ticket/Receipt**	When you call the pizza place, you immediately get an order number or receipt. This receipt is your **Future**. It doesn't have the pizza yet, but it's a **promise** that a pizza will be delivered later.
`async`	**The Pizza Place/Delivery Service**	The delivery service is the function marked `async`. It takes your request and immediately gives you the Future (the receipt), promising to return the result later.
`await`	**The Doorbell Ring**	You are watching TV (the rest of your application is running). When you use `await` before the delivery call, you are saying, **"I will stop watching TV right here and go to the door the moment I hear the doorbell ring."**
Non-Blocking	**Watching TV While Waiting**	You start watching TV (the rest of your app keeps running) immediately after placing the order. Your overall process wasn't stopped by the slow task.

What is a future:

A future (lower case "f") is an instance of the Future (capitalized "F") class. A future represents the result of an asynchronous operation, and can have two states: uncompleted or completed.

- Uncompleted: When you call an asynchronous function, it returns an uncompleted future. That future is waiting for the function's asynchronous operation to finish or to throw an error.
- Completed: If the asynchronous operation succeeds, the future completes with a value. Otherwise, it completes with an error.

A future of type Future<T> completes with a value of type T. For example, a future with type Future<String> produces a string value. If a future doesn't produce a usable value, then the future's type is Future<void>.

Example:

```dart
Future<void> fetchUserOrder() {
  // Imagine that this function is fetching user info from the service
database.
  return Future.delayed(const Duration(seconds: 2), () => print('Large Latte'));
}

void main() {
  fetchUserOrder();
  print('Fetching user order...');
}
```

Run

Question:
Which will print first: "Large Latte" or "Fetching user order..."

Output:

Now we have seen what futures are, but how do you use the results of asynchronous functions?

We can get results with the async and await keywords.

The Solution: async and await

Need for these tools: The goal is to **pause** our function until the Future completes, but **without blocking** the entire application.

async Keyword:

- Placed on a function to mark it as asynchronous
- It ensures the function must return a future

await Keyword:

- it pauses the current async function until the future completes
- It makes asynchronous code look synchronous, but its non blocking.

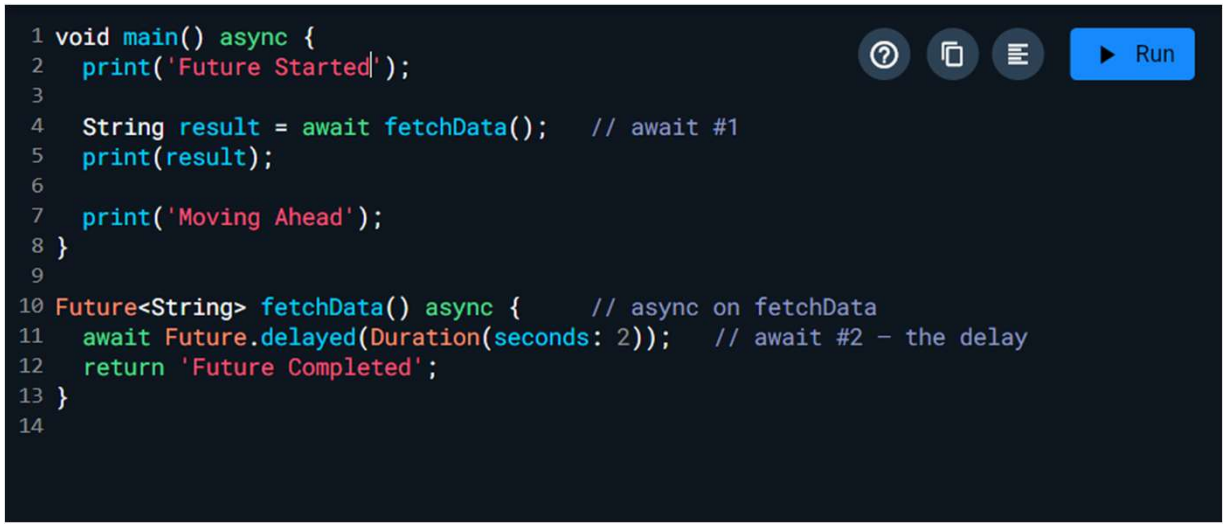
Remember these two basic guidelines when using async and await:

- To define an async function, add async before the function body:
- The await keyword works only in async functions.

Example:

Now we will be looking at 3 versions of a code. Try predicting the output, after which you also code it once to understand how it works.

Version 1: Normal async (both awaits present)



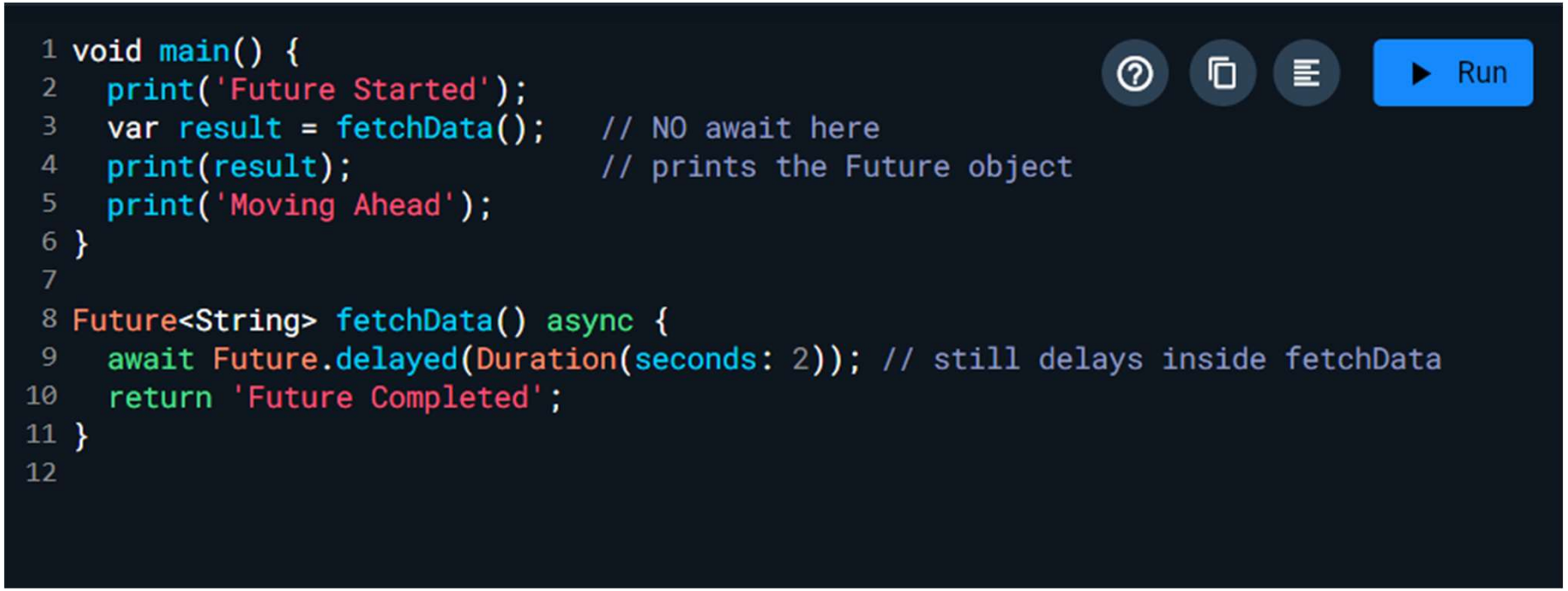

```dart
void main() async {
  print('Future Started');

  String result = await fetchData();   // await #1
  print(result);

  print('Moving Ahead');
}

Future<String> fetchData() async {     // async on fetchData
  await Future.delayed(Duration(seconds: 2));   // await #2 - the delay
  return 'Future Completed';
}
```

Output:

```
Future Started
Future Completed
Moving Ahead
```

Note: There is a 2 second pause between future started and future completed

Version 2: Remove the await in main() (keep the await inside fetchData())




```dart
void main() {
  print('Future Started');
  var result = fetchData();   // NO await here
  print(result);              // prints the Future object
  print('Moving Ahead');
}

Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2)); // still delays inside fetchData
  return 'Future Completed';
}
```

Output:

```
Future Started.
Instance of '_Future<String>'
Moving Ahead
```

Step-by-step explanation:

- main() starts and prints Future Started
- main() calls fetchData(). Because fetchData() is async, it **immediately returns a Future object** to the caller while it continues running asynchronously.
- **Because main() did not await, it receives that Future object in result and print(result) prints the Future object representation (Instance of 'Future<String>').**
- main() continues and prints Moving Ahead — it does not wait for fetchData() to finish.
- Meanwhile fetchData() is running in the background: it hits await Future.delayed(...) and actually pauses for 2 seconds. After 2 seconds it completes and returns the string internally — but no code in main() prints that result because main() never awaited or attached a callback.

Conclusion: await in main() is what tells main() to wait for the Future's final value. Removing it means main() keeps running immediately and only gets a Future object, not the resolved value.

Version 3: Keep await in main(), but remove the await inside fetchData()


```dart
void main() async {
  print('Future Started');
  String result = await fetchData();   // main awaits
  print(result);
  print('Moving Ahead');
}

Future<String> fetchData() async {
  Future.delayed(Duration(seconds: 2)); // started, but NOT awaited
  return 'Future Completed';            // returned immediately
}
```

Output:

```
Future Started
Future Completed
Moving Ahead
```

Step-by-step explanation

- main() starts and prints Future Started.
- main() calls fetchData() and awaits it. Because main() used await, it will pause until the Future returned by fetchData() completes.
- **Inside fetchData(), we started Future.delayed(...) but did not await it. That means Future.delayed is scheduled, but fetchData() does NOT wait for it — it proceeds immediately to the next line.**
- fetchData() then return 'Future Completed**'. Because fetchData() is async, returning a plain value wraps it in an already-completed Future<String> and that Future completes immediately with 'Future Completed'.**
- main() receives that immediate value (because it awaited), prints Future Completed, then prints Moving Ahead.
- The Future.delayed is still running in the background but nothing is waiting on it — its completion is ignored.

Conclusion: If you forget to await an internal Future, the calling function can still await the callee, but the callee may return immediately before the internal work completes.

Version 4: Remove both awaits (no await in main and no await in fetchData())

```dart
void main() {
  print('Future Started');
  var result = fetchData();   // no await
  print(result);              // prints Future object
  print('Moving Ahead');
}

Future<String> fetchData() async {
  Future.delayed(Duration(seconds: 2)); // fire-and-forget delay
  return 'Future Completed';            // returns immediately
}

```

Output:

```
Future Started
Instance of '_Future<String>'
Moving Ahead
```

Step-by-step explanation

- main() prints Future Started
- main() calls fetchData() and gets back a Future object immediately (because fetchData() is async and returns a wrapped value right away).
- print(result) prints the Future object (not the final string).
- main() prints Moving Ahead.
- fetchData() started Future.delayed, but because it was not awaited, fetchData() returned immediately with 'Future Completed' wrapped in a completed Future. The delayed Future continues and then completes after 2 seconds — but nothing uses it.

Problem To Solve:

Problem Statement: You are making tea.

Write a Dart program that performs these 3 tasks:

- Boil water
  - Takes 3 seconds
  - Returns "Water boiled"
- Add tea leaves
  - Takes 2 seconds
  - Returns "Tea leaves added"
- Add sugar
  - Takes 1 second
  - Returns "Sugar added"

Requirements
Each task must be written as a Future<String> function.
Each function must use await Future.delayed(...).
In main(), call these functions in order using await, so the steps happen one after another, like real tea-making.

Solution:

```dart
Future<void> main() async {
  String step1 = await boilWater();
  print(step1);
  String step2 = await addTeaLeaves();
  print(step2);
  String step3 = await addSugar();
  print(step3);
  print("Tea is ready");
}

Future<String> boilWater() async {
  await Future.delayed(Duration(seconds: 3));
  return "Water boiled";
}

Future<String> addTeaLeaves() async {
  await Future.delayed(Duration(seconds: 2));
  return "Tea leaves added";
}

Future<String> addSugar() async {
  await Future.delayed(Duration(seconds: 1));
  return "Sugar added";
}
```

Output:

```
[Running] dart "c:\Users\Siddhart
Water boiled
Tea leaves added
Sugar added
Tea is ready
```

Future Concepts:

1. Handling Errors
2. Making a chat using sockets from the dart:io package
3. Introduction to flutter