By: Eng. Elias Al-Showaiter

[Dart Programming Language] - Lecture [13]

[Understanding Futures in Dart]

1. Introduction:

When we write code that interacts with the network (using online databases) the behavior of the code changes, in this lecture we will be introduced to the concept of future.

What is a Future?

A `Future` represents a potential value or error that will be available at some point in the future. It's a way to handle asynchronous operations, such as fetching data from the internet or reading files.

Key Concepts

1. **Asynchronous Operations**:
   - Operations that take time do not block the main thread, allowing other code to run while waiting for completion.
2. **Completing a Future**:
   - A `Future` can be completed with a value or an error. Once completed, it cannot change.
3. **States of a Future**:
   - **Uncompleted**: The operation is ongoing.
   - **Completed with a value**: The operation has completed successfully.
   - **Completed with an error**: The operation has failed.

Creating a Future

You can create a `Future` using the `Future` constructor method:

```
Future<String> fetchData() {
  return Future.delayed(Duration(seconds: 2), () => 'Dat
a fetched');
}
```

We created a function of type Future<String> called fetchData.

In this function we returned a string “Data fetched”, but this string was returned after two seconds, this is why we must use Future.

Using Futures

Handle the completion of a `Future` using the `then`, `catchError`, and `whenComplete` methods:

```
fetchData().then((data) {
  print(data); // Output: Data fetched
}).catchError((error) {
  print('Error: $error');
}).whenComplete(() {
  print('Operation completed');
});
```

When we use future functions we cannot assign the result of the function to a variable directly, we have two ways to handle this, the first is using .then method, which has a single parameter that is a lambda function, and this lambda function accepts a variable, this variable is the result of the function.

In the previous example we used .then with the function fetchData() , and called a lambda function with a variable called data, this variable contains the result of fetchData which is the string fetchData.

The second method of handling futures is to use the async/await keywords, this is usually a clearer way to handle futures.

```
Future<void> main() async {
  try {
    String data = await fetchData();
    print(data); // Output: Data fetched
  } catch (error) {
    print('Error: $error');
  } finally {
    print('Operation completed');}}
```

Using the same function, fetchData(), we notices that we added the keyword async after the main function, now instead of using the .then method, we can directly call the function fetchData and receive its results, we just have to use the keyword await before it.

A very common practice is to use async function inside a try catch method.

In this example, we'll create a function that simulates fetching user data from a database.

```
import 'dart:async';

// Simulating a user data fetch from an API
Future<String> fetchUserData() {
  return Future.delayed(Duration(seconds: 2), () {
    // Simulated user data
    return 'User: Ahmed Ali, Age: 30, Location: Yemen';
  });
}

// Main function to run the example
Future<void> main() async {
  print('Fetching user data...');

  try {
    // Wait for the fetchUserData Future to complete
    String userData = await fetchUserData();
    print(userData); // Output: User: Ahmed Ali, Age: 30
, Location: Yemen
  } catch (e) {
    print('Error fetching user data: $e');
  } finally {
    print('User data fetch operation completed.');
  }
}
```