🧪 Section 5 – Asynchronous Programming

# 🎯 Learning Goals

By the end of this section, you will be able to:

* Use **Futures** for asynchronous tasks
* Simplify asynchronous code using **async/await**
* Work with **Streams** for continuous asynchronous data
* Execute heavy computations in parallel using **Isolates**

# 🔹 1. Futures

## 💡 Example

void main() {

print('Fetching data...');

Future<String> fetchData() {

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

}

fetchData().then((data) {

print(data);

}).catchError((error) {

print('Error: $error');

});

print('Request sent.');

}

Explanation

* Future.delayed() simulates a delay like a network call.
* .then() runs when the Future completes successfully.
* .catchError() handles exceptions.
* Note how "Request sent." appears before the result — showing that the operation runs **asynchronously**.

### 🧩 Exercise 1

Create a function downloadFile() that waits 3 seconds before returning "File downloaded".

* Use Future.delayed() to simulate the delay.
* Handle success and errors using .then() and .catchError().
* Print "Download started" before and "Download complete" after.

# 🔹 2. Async and Await

## 💡 Example

Future<String> fetchUser() async {

print('Getting user info...');

await Future.delayed(Duration(seconds: 2));

return 'User: Alice';

}

void main() async {

print('Start');

String user = await fetchUser();

print(user);

print('End');

}

Explanation

* async marks a function that can use await.
* await pauses execution until the Future completes.
* This syntax makes asynchronous code easier to read and write.

### 🧩 Exercise 2

Write an asynchronous function getWeather() that:

1. Waits for 2 seconds and returns "Sunny 25°C".
2. In main(), print "Fetching weather...", call await getWeather(), and print the result.
3. Finally, print "Weather check completed."

# 🔹 3. Streams

## 💡 Example

void main() {

Stream<int> countStream() async\* {

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

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

yield i;

}

}

countStream().listen(

(data) => print('Received: $data'),

onDone: () => print('Stream closed.'),

);

}

Explanation

* async\* defines a **stream generator** that emits multiple values over time.
* yield sends a value to the stream.
* listen() is used to react to each emitted value.
* Streams are ideal for continuous data (messages, progress, sensors, etc.).

### 🧩 Exercise 3

1. Create a Stream<String> named messageStream() that yields 3 messages, each after 1 second.
2. In main(), listen to the stream and print each message.
3. When the stream ends, print "All messages received."

# 🔹 4. Isolates

## 💡 Example

import 'dart:isolate';

void heavyComputation(SendPort sendPort) {

int sum = 0;

for (int i = 0; i < 100000000; i++) {

sum += i;

}

sendPort.send(sum);

}

void main() async {

ReceivePort receivePort = ReceivePort();

await Isolate.spawn(heavyComputation, receivePort.sendPort);

print('Waiting for result...');

int result = await receivePort.first;

print('Sum from isolate: $result');

}

Explanation

* **Isolates** are independent threads of execution with their own memory.
* Communication happens through SendPort and ReceivePort.
* They’re useful for CPU-heavy computations that would otherwise block the main thread.

### 🧩 Exercise 4

1. Create an isolate that computes the factorial of a number (e.g., 6).
2. Send the result back to the main isolate.
3. Print "Calculation done in isolate" and then display the factorial result.

# 🏁 Summary Checklist

By completing this section, you can now:  
✅ Use **Futures** to manage asynchronous operations  
✅ Write clean async code using **async/await**  
✅ Handle multiple asynchronous events with **Streams**  
✅ Use **Isolates** for parallel processing