

# Flutter & Dart Complete Guide

Prepared by Eng. Yousef Salah Kassem

DAFL 1  
Extensions & Async Programming

## Lecture 9



Chapter

09

## Table Of Content

- Extensions
- Asynchronous Programming
- Error Handling



# Extensions

# Extensions

In Dart, you can extend the functionality of a class by using extension. It is a new feature in Dart 2.7.0. It is similar to extension methods in C# and Kotlin. It is also similar to the concept of mixins in Dart.



# Extensions

## The Problem

It's not much of a solution to a problem, but a nice convenience. Extensions in Dart allow you to develop custom functionalities on any library or data structure, including your own. So how do we go about them?



# Extensions

## The Solution

There is a specific keyword in Dart used to create extensions: `extension`. When creating an extension we also need to specify on which data type it should apply. Let's say we want to make a few useful `DateTime` extension methods. We should create a separate file and create our date extensions inside:



```
// date_extensions.dart  
  
extension DateExtensions on DateTime {  
    // ...  
}
```

```
extension DateExtensions on DateTime {  
    bool isSameDay(DateTime other) {  
        return this.year == other.year &&  
            this.month == other.month &&  
            this.day == other.day;  
    }  
}
```

Click to Copy

# Extensions

Note: I've put `this` so you can see how the comparison happens. The IDE will most likely warn you: `Don't access members with this unless avoiding shadowing`.

We can use our new extension method like so (don't forget to import your dart extension file):

```
date1.isSameDate(date2)
```



# Extensions

The `this` in our extension method refers to the `date1` variable.

Aside from methods, we can also create getters, setters and operators as well. Here's a getter method that returns a date in a specific format:



```
extension DateExtensions on DateTime {  
    // ...  
    get formattedDate {  
        DateFormat format;  
        if (year != DateTime.now().year) {  
            format = DateFormat("E, d MMM yyyy");  
            // we can also use other extension methods inside  
        } else if (isToday()) {  
            return "Today";  
            // we can also use other extension methods inside  
        } else if (isYesterday()) {  
            return "Yesterday";  
        } else {  
            format = DateFormat("E, d MMM");  
        }  
  
        return format.format(this);  
    }  
}
```

# Extensions

Here we are extending the functionality of **String** class. We are adding a new method **capitalize** to the **String** class. We are using **extension** keyword to extend the functionality of **String** class.

```
void main(){  
    String name = "john";  
    print(name.capitalize());  
}  
  
extension StringExtension on String{  
    String capitalize(){  
        return "${this[0].toUpperCase()}${this.substring(1)}";  
    }  
}
```



# **Asynchronous Programming**

# Async Programming

**Asynchronous Programming** is a way of writing code that allows a program to do multiple tasks at the same time. Time consuming operations like fetching data from the internet, writing to a database, reading from a file, and downloading a file can be performed without blocking the main thread of execution.



# Sync Programming

In Synchronous programming, the program is executed line by line, one at a time. Synchronous operation means a task that needs to be solved before proceeding to the next one.



# Sync Programming

Here in this example, you can see that it will print line by line. Let's suppose **Second Big Operation** takes 3 seconds to load then **Third Operation** and **Last Operation** need to wait for 3 seconds. To solve this issue asynchronous programming is here.

```
void main() {  
    print("First Operation");  
    print("Second Big Operation");  
    print("Third Operation");  
    print("Last Operation");  
}
```



# Async Programming

In Asynchronous programming, program execution continues to the next line without waiting to complete other work. It simply means, **Don't wait**. It represents the task that doesn't need to solve before proceeding to the next one.

**Note:** Asynchronous Programming improves the responsiveness of the program.



# Async Programming

Here in this example, you can see that it will print **Second Big Operation** at last. It is taking 3 seconds to load and **Third Operation** and **Last Operation** don't need to wait for 3 seconds. This is the problem solved by Asynchronous Programming. A Future represents a value that is not yet available,

```
void main() {  
    print("First Operation");  
    Future.delayed(Duration(seconds:3), ()=>print('Second Big Operation'));  
    print("Third Operation");  
    print("Last Operation");  
}
```



# Async Programming

## Why We Need Asynchronous

- To Fetch Data From Internet,
- To Write Something to Database,
- To execute a long-time consuming task,
- To Read Data From File, and
- To Download File etc.



# Future

In dart, the Future represents a value or error that is not yet available. It is used to represent a potential value, or error, that will be available at some time in the future.



# Future

## How To Create Future In Dart

You can create a future in dart by using **Future** class. Here the function will return **Future<String>** after 5 seconds.

```
// function that returns a future
Future<String> getUsername() async {
  return Future.delayed(Duration(seconds: 2), () => 'Mark');
}
```



# Future

## How To Use Future In Dart

You can use future in dart by using **then()** method. Here the function will return **Future<String>** after 5 seconds.

```
// function that returns a future
Future<String> getUser_name() async {
  return Future.delayed(Duration(seconds: 2), () => 'Mark');
}

// main function
void main() {
  print("Start");
  getUser_name().then((value) => print(value));
  print("End");
}
```



# Future

**Future** represents the result of an asynchronous operation and can have 2 states.

## State Of Future

- Uncompleted
- Completed



# Future

- **Uncompleted**

When you call an asynchronous function, it returns to an uncompleted future. It means the future is waiting for the function asynchronous operation to finish or to throw an error.

- **Completed**

It can be completed with value or completed with error. **Future<int>** produces an int value, and **Future<String>** produces a String value. If the future doesn't produce any value, then the type of future is **Future<void>**.



```
void main() {  
    print("Start");  
    getData();  
    print("End");  
}  
  
void getData() async{  
    String data = await middleFunction();  
    print(data);  
}  
  
Future<String> middleFunction(){  
    return Future.delayed(Duration(seconds:5), ()=> "Hello");  
}
```

Start  
End  
Hello

# Async & Await

- **Async/await** is a feature in Dart that allows us to write asynchronous code that looks and behaves like synchronous code, making it easier to read.
- When a function is marked **async**, it signifies that it will carry out some work that could take some time and will return a Future object that wraps the result of that work.
- The **await** keyword, on the other hand, allows you to delay the execution of an async function until the awaited Future has finished. This enables us to create code that appears to be synchronous but is actually asynchronous.
- The **async** and **await** keywords both provide a declarative way to define an asynchronous function and use their results. You can use the **async** keyword before a function body to make it asynchronous. You can use the **await** keyword to get the completed result of an asynchronous expression.



# Async & Await

## Important Concept

- To define an Asynchronous function, add `async` before the function body.
- The `await` keyword work only in the `async` function.



# Async & Await

```
void main() {  
    print("Start");  
    getData();  
    print("End");  
}  
  
void getData() {  
    String data = middleFunction();  
    print(data);  
}  
  
Future<String> middleFunction(){  
    return Future.delayed(Duration(seconds:5), ()=> "Hello");  
}
```



# Async & Await

```
void main() {  
    print("Start");  
    getData();  
    print("End");  
}  
  
void getData() async{  
    String data = await middleFunction();  
    print(data);  
}  
  
Future<String> middleFunction(){  
    return Future.delayed(Duration(seconds:5), ()=> "Hello");  
}
```



# Streams

In Dart, a stream represents a sequence of asynchronous events. It's a way to handle data that comes in over time, rather than all at once, the stream tells you that there is an event when it is ready. An 'Event' can either be a 'data' event, an 'error' event, or a 'done' event. Streams are a fundamental concept in Dart for handling asynchronous programming, allowing developers to process data as it becomes available, without blocking the execution flow.

Dart provides two types of streams: single subscription or broadcast



|       | Single Value                   | Zero or more values            |
|-------|--------------------------------|--------------------------------|
| Sync  | <code>int</code>               | <code>Iterator</code>          |
| Async | <code>Future&lt;int&gt;</code> | <code>Stream&lt;int&gt;</code> |

# Single Subscription

**Single-subscription streams:** These are streams where only one listener can listen to the stream during its lifetime. Once data is emitted or the stream is closed, it cannot be listened to again. Examples include **Stream** and **Future**.



```
import 'dart:async';

void main() {
  // Create a stream controller
  StreamController<int> controller = StreamController<int>();

  // Create a stream
  Stream<int> stream = controller.stream;

  // Listen to the stream
  stream.listen(
    (int data) {
      print('Data: $data');
    },
    onError: (error) {
      print('Error: $error');
    },
    onDone: () {
      print('Stream closed');
    },
  );

  // Add data to the stream
  controller.add(1);
  controller.add(2);
  controller.add(3);

  // Close the stream
  controller.close();
}
```

# Broadcast Subscription

These are streams where multiple listeners can listen to the stream during its lifetime. Data emitted by the stream will be received by all active listeners.

This is useful when you want to broadcast events to multiple parts of your application simultaneously. To create a broadcast stream simply include ***StreamController.broadcast()*** on an existing single subscription stream.



```
import 'dart:async';

// Initializing a stream controller for a broadcast stream
StreamController<String> controller = StreamController<String>.broadcast();

// Creating a new broadcast stream through the controller
Stream<String> stream = controller.stream;

void main() {
  // Setting up a subscriber to listen for any events sent on the stream
  StreamSubscription<String> subscriber1 = stream.listen((String data) {
    print('Subscriber1 on Data: ${data}');
  }, onError: (error) {
    print('Subscriber1 on Error: ${error}');
  }, onDone: () {
    print('Subscriber1: Stream closed');
  });

  // Setting up another subscriber to listen for any events sent on the stream
  StreamSubscription<String> subscriber2 = stream.listen((String data) {
    print('Subscriber2 on Data: ${data}');
  }, onError: (error) {
    print('Subscriber2 on Error: ${error}');
  }, onDone: () {
    print('Subscriber2: Stream closed');
  });

  // Adding a data event to the stream with the controller
  controller.sink.add('Hello!');

  // Adding an error event to the stream with the controller
  controller.addError('Hi!');

  // Closing the stream with the controller
  controller.close();
}
```

# Error Handling

# Error Handling

You can handle errors in the dart async function by using try-catch. You can write try-catch code the same way you write synchronous code.

Note: We cannot perform an asynchronous operation from a synchronous function.



```
main() {  
    print("Start");  
    getData();  
    print("End");  
}  
  
void getData() async{  
    try{  
        String data = await middleFunction();  
        print(data);  
    }catch(err){  
        print("Some error $err");  
    }  
}  
  
Future<String> middleFunction(){  
    return Future.delayed(Duration(seconds:5), ()=> "Hello");  
}
```

Start  
End  
Hello

# **What is Next ?**



**Thank You**