Objectives

- Advanced Dart Features
- Asynchronous Programming
- Working with Files

Advanced Dart Features

Enums

Enums (short for "enumerations") are special types that allow us to define a fixed set of constant values.
Instead of using strings or numbers to represent choices or states, Dart provides `enum` to group related constant values under one type.

Defining an Enum

We use the `enum` keyword followed by the enum name and a list of constant values separated by commas.

```
enum Weather {
  sunny,
  rainy,
  cloudy,
  snowy,
}

void main() {
  Weather today = Weather.sunny;
  print(today); // Output: Weather.sunny
}
```

- Here, `Weather` is the enum, and its values are `sunny`, `rainy`, `cloudy`, and `snowy`.

Using Enums

We can use enums in control structures like `switch` to perform different actions based on the value.

```
void describeWeather(Weather weather) {
  switch (weather) {
    case Weather.sunny:
      print('It\'s sunny today!');
      break;
    case Weather.rainy:
      print('Don\'t forget your umbrella.');
      break;
    case Weather.cloudy:
      print('It might rain later.');
      break;
    case Weather.snowy:
      print('Time for snowball fights!');
      break;
  }
}

void main() {
  describeWeather(Weather.snowy);
}
```

In the newest Dart versions, enums are no longer just simple lists of choices they became much more powerful.
Starting from Dart 2.17, enums can now have:

- Fields (variables)
- Methods
- Constructors

This means enums are able to hold data for each choice and perform actions using methods. You can think of enums now almost like mini-classes where each choice (called a value) can have its own data and behavior.

The constructor in an enum is used to initialize the fields for each choice.
It is automatically called when you create or use an enum value.

```
enum Weather {
  sunny(30),
  rainy(18),
  cloudy(22),
```

```
  snowy(-5);

  final int temperature;

  const Weather(this.temperature);

  void describe() {
    print('Typical temperature: $temperature°C');
  }
}

void main() {
  Weather today = Weather.rainy;
  today.describe(); // Output: Typical temperature: 18°C
}
```

Generics

Generics allow us to write flexible, reusable code by using types as parameters.
Instead of writing separate classes or functions for each type (like `int`, `String`, etc.), we can use generic type placeholders.

Generic Classes

We can create a class that works with any type by using `<T>`, where `T` is a placeholder for the type.

```
class Box<T> {
  T content;

  Box(this.content);

  void showContent() {
    print('Content: $content');
  }
}

void main() {
  Box<String> stringBox = Box('Hello');
  stringBox.showContent(); // Output: Content: Hello
```

```
  Box<int> intBox = Box(123);
  intBox.showContent(); // Output: Content: 123
}
```

- `T` can be any type Dart will replace it with the type you specify when you create the object.

Generic Functions

Functions can also be generic using the same `<T>` syntax.

```
T pickFirst<T>(List<T> items) {
  return items[0];
}

void main() {
  var numbers = [1, 2, 3];
  var names = ['Ali', 'Sara', 'John'];

  print(pickFirst(numbers)); // Output: 1
  print(pickFirst(names));   // Output: Ali
}
```

To create a generic function, we start with `<T>`, which means the function's return value can be any type. In the function `pickFirst`, `<T>` after the function name tells Dart that this is a generic function.
The function takes a parameter `List<T> items`, meaning it expects a list of elements where all items are of type `T`. For example, if `T` is `String`, it would be a `List<String>`, and if `T` is `int`, it would be a `List<int>`.
Inside the function, `return items[0];` simply returns the first element of the list (at index 0). Since the list holds elements of type `T`, the function's return type is also `T`. This allows the same function to work for lists of any type without rewriting it multiple times.

Constraints on Generics

We can restrict generic types using extends.

```
class Animal {
  void makeSound() => print('Some generic sound');
}
```

```
class Dog extends Animal {
  @override
  void makeSound() => print('Bark!');
}

class AnimalCage<T extends Animal> {
  T animal;

  AnimalCage(this.animal);

  void hearAnimal() {
    animal.makeSound();
  }
}

void main() {
  var dogCage = AnimalCage(Dog());
  dogCage.hearAnimal(); // Output: Bark!
}
```

- Here, `T extends Animal` ensures only types that are `Animal` or its subclasses can be used.

Extensions

Extensions let us add new functionality to existing classes without modifying them.
It's very useful when we want to enhance Dart's built-in types like `String`, `List`, or even our own classes.

Creating an Extension

We use the `extension` keyword to create extensions.

```
extension StringExtension on String {
  String get reversed {
    return split('').reversed.join();
  }

  bool get isPalindrome {
    return this == split('').reversed.join();
  }
```

```
}

void main() {
  String word = 'level';
  print(word.reversed);      // Output: level
  print(word.isPalindrome);  // Output: true
}
```

- Here, we added two properties `reversed` and `isPalindrome` to any `String` object without modifying the original String class.

Cascade Notation

The cascade operator (`..`) allows us to perform multiple operations on the same object without repeating its name.

Example

Without cascade:

```
var person = Person();
person.name = 'Alice';
person.age = 30;
person.greet();
```

Using cascade:

```
var person = Person()
  ..name = 'Alice'
  ..age = 30
  ..greet();
```

- With `..`, we can chain multiple operations on `person` in a clean way.

Typedef

In Dart, we use `typedef` to create a custom name for a function type, Instead of writing long function signatures every time, we can define a `typedef` once and reuse it easily.

Simple Typedef

```
typedef IntOperation = int Function(int a, int b);

int add(int a, int b) => a + b;
int multiply(int a, int b) => a * b;

void performOperation(IntOperation operation, int x, int y) {
  print(operation(x, y));
}

void main() {
  performOperation(add, 2, 3);       // Output: 5
  performOperation(multiply, 2, 3);  // Output: 6
}
```

- `IntOperation` is an alias for a function that takes two `int` and returns an `int`.

Asynchronous Programming

Introduction

In Dart, asynchronous programming is used to handle tasks that take time to complete, like fetching data from the internet, reading files, or waiting for a user action without blocking the rest of the program.
Instead of freezing the whole app until the task finishes, Dart uses futures, async/await, and streams to deal with these operations smoothly.

Futures

A Future represents a value that will be available at some point in the future.
It acts like a promise that the program will eventually get a result (or an error).

Creating a Future

We can create a Future that completes after some delay using `Future.delayed`:

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

```
  });
}
void main() {
  fetchData().then((data) {
    print(data); // Output after 2 seconds: Data fetched
  });
}
```

- Here, `fetchData` returns a Future that completes after 2 seconds and returns a string `'Data fetched'`.
- We use `.then()` to tell Dart what to do once the future is completed.
- We use `.catchError()` to tell Dart what to do if the future raised error.
- We use `.whenComplete()` : to tell Dart what to do when the `Future` completes, regardless of whether it was with a value or an error.

Async and Await

When working with asynchronous operations, writing readable and easy-to-follow code is important, Dart helps us with two powerful keywords: `async` and `await`.
We mark a function with the `async` keyword to tell Dart that the function will perform asynchronous work, an `async` function automatically returns a `Future`, even if you don't explicitly write `Future` in the return type.
Inside an `async` function, we can use the `await` keyword to pause the execution of the function until a `Future` is completed.

Example

```
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2));
  return 'Data fetched';
}

void main() async {
  print('Fetching data...');
  String data = await fetchData();
  print(data);
}
```

- `fetchData` waits 2 seconds before returning the data.
- In `main`, we use `await fetchData();` to pause until the data is ready.
- This way, the code looks synchronous but still runs asynchronously under the hood.

Handling Errors

Just like regular code, asynchronous code can fail.
We can handle errors using `.catchError()` or a `try-catch` block with `await`.

Using try-catch

```
Future<String> fetchData() async {
  throw Exception('Failed to fetch data');
}

void main() async {
  try {
    String data = await fetchData();
    print(data);
  } catch (e) {
    print('Error: $e');
  }
}
```

- Here, if `fetchData` throws an exception, the error is caught and printed.
- `try-catch` makes error handling with `await` very clean and familiar.

Streams

While a Future gives us one result, a Stream gives us multiple values over time.
We can think of a stream as a sequence of asynchronous events, like receiving multiple chat messages or sensor updates.

Listening to a Stream

To create a function that returns a stream, we use `async*` instead of `async`, and we explicitly specify the return type as `Stream<type>`.
Inside the function, we emit a sequence of asynchronous events using `yield`.
We can listen to a stream and handle its events using either `await for` or `stream.listen()`.

```
Stream<int> countStream() async* {
  for (int i = 1; i <= 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}

void main() async {
  // with await for
  await for (int number in countStream()) {
    print(number);
  }
  // with stream.listen
   final stream = countStream(5);
  stream.listen(
    (value) => print('Value: $value'),
    onError: (error) => print('Error: $error'),
    onDone: () => print('Stream completed!'),
  );
}
```

- `countStream` is an asynchronous generator that emits numbers 1 to 5, one every second.
- `await for` is used to listen to the stream and wait for each value.

`stream.listen()` is often better because it uses a non-blocking approach it listens to incoming events and handles them as they arrive without waiting for all of them to complete.
On the other hand, `await for` will wait until all events in the sequence are received and processed before continuing, which can block further code execution until the entire stream is finished

Working with Files

Introduction

Working with files is a fundamental part of programming, files allows us to store data permanently, read configuration files, save logs, or even build entire applications that depend on persistent storage.
Dart provides powerful tools for file management through its `dart:io` library.
Using the `File` class, we can create, read, write, modify, rename, and delete files with ease.
In this section, we will focus on three main operations:

- Reading from files
- Writing to files
- Managing files (deleting and renaming)

Opening Files

Before we can interact with a file, we must first open it.
We usually create a `File` object by providing the file's path, then to work with file contents, we combine it with methods like `readAsString()`, `readAsLines()`, `openRead()`, and `openWrite()`.
Dart automatically opens and closes the file internally when we read or write. but when using streams, we should manually close when needed (especially while writing).

```
import 'dart:io';
void main(){
var file = File('example.txt'); // Created file Object
}
```

Reading Files

Dart provides multiple ways to **read** data from a file:

- `readAsString()` : Reads the entire file as one single `String`.
- `readAsLines()` : Reads the file and returns all lines as a `List<String>`.
- `openRead()` : Opens the file as a stream, allowing us to process large files line-by-line.

Reading the entire file as a String

```
import 'dart:io';

void main() async {
  var file = File('example.txt');
  String contents = await file.readAsString();

  print(contents);
}
```

Reading line by line

```dart
import 'dart:io';

void main() async {
  var file = File('example.txt');
  List<String> lines = await file.readAsLines();

  for (var line in lines) {
    print(line);
  }
}
```

Streaming a large file line-by-line

```dart
import 'dart:io';
import 'dart:convert';

void main() async {
  var file = File('example.txt');

  var lines = file.openRead()
      .transform(utf8.decoder)
      .transform(LineSplitter());

  await for (var line in lines) {
    print(line);
  }
}
```

- `openRead()` allows us to **stream** data.
- Very useful when dealing with **large files** without loading everything into memory.

Writing to Files

To write to a file, we usually use:

- `writeAsString()` : To overwrite or create a new file.
- `openWrite()` : To append data or write using streams.

Overwriting a file

```
import 'dart:io';

void main() async {
  var file = File('output.txt');
  await file.writeAsString('Hello, world!');
}
```

- If the file exists, its contents will be replaced.
- If the file does not exist, Dart will create it.

Appending to a file

```
import 'dart:io';

void main() async {
  var file = File('output.txt');
  var sink = file.openWrite(mode: FileMode.append);

  sink.writeln('Another line');
  await sink.close();
}
```

- Using `FileMode.append` allows us to add new data without deleting existing data.
- `writeln()` writes a line followed by a newline character.
- Remember to call `close()` when using `openWrite()` to save and release resources.

Checking File Existence and Metadata

Before interacting with a file, it is often good to check its status:

```
import 'dart:io';

void main() async {
  var file = File('output.txt');
```

```
  print(await file.exists());  // true if file exists
  print(await file.length());  // file size in bytes

  var directory = Directory('my_folder');
  print(await directory.exists());  // true if directory exists
}
```

- `exists()` checks if the file or directory actually exists.
- `length()` gives us the size of a file in bytes.

Deleting and Renaming Files

Dart allows us to rename and delete files easily.

Renaming a file

```
import 'dart:io';

void main() async {
  var file = File('old.txt');
  await file.rename('new.txt');
}
```

- `rename()` changes the file's name or moves it to a different path.

Deleting a file

```
import 'dart:io';

void main() async {
  var file = File('unwanted.txt');
  await file.delete();
}
```

- `delete()` removes the file permanently.

Task

Using everything we've learned during this course, create a Ruby project that solves a problem you face in your daily life.
Think about tasks you often repeat, information you want to organize, or anything you wish could be automated then build a solution using Dart!