Dart

Functional Programming λ
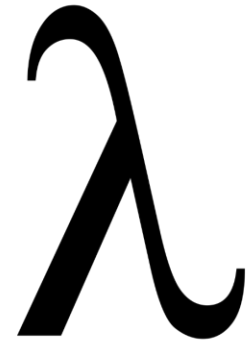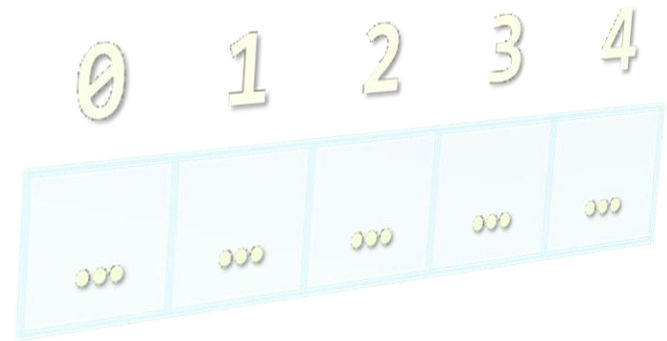
Table of Contents

Collections

0 1 2 3 4

List

- Dart has a List**<T>** type to declare list

```dart
List<String> colors = ["Red", "Green", "Blue"];
var names = ["Ali", "Ahmed", "Sara"];

const nums = [2, 3, 4];
var nullNums = List<int?>.filled(10, null);

colors.forEach((color) => print(color));
names.forEach((name) => print(name));
nums.forEach((num) => print(num));
nullNums.forEach((num) => print(num));
```

List Methods

```
const nums = [2, 3, 4];

nums.add(8);
nums.insert(0, 1);
nums.removeAt(2);
nums.remove(4);
nums.removeLast();
nums.removeRange(0, 2);
nums.removeWhere((num) => num > 3);
nums.removeRange(0, nums.length);
nums.addAll([1, 2, 3]);
nums.addAll([4, 5, 6]);
```

List destructuring

- List destructuring allows you to unpack or extract values from a list and assign them to variables in a clean and concise way

```
var fruits = ["Apple", "Banana", "Cherry", "Mango", "Orange"];

// Destructuring the list
// ... is used to unpack the remaining elements
var [firstFruit, secondFruit, thirdFruit, ...others] = fruits;

print("First fruit: $firstFruit");   // Output: First fruit: Apple
print("Second fruit: $secondFruit"); // Output: Second fruit: Banana
print("Third fruit: $thirdFruit");   // Output: Third fruit: Cherry
print("Others: $others");            // Output: Others: [Mango, Orange]
```

Spread operator (…)

- Spread operator (…) allows you to include all elements of one list inside another list
  - It "spreads" the elements of a list into a new list
  - The null-aware spread operator (…?) is used when the list you're spreading might be null

```
List<String> fruits = ["Apple", "Banana"];
List<String> vegetables = ["Carrot", "Broccoli"];

List<String> food = fruits + vegetables;
print(food); // Output: [Apple, Banana, Carrot, Broccoli]

food = [...fruits, ...vegetables];
print(food); // Output: [Apple, Banana, Carrot, Broccoli]
```

Set

- *Set is same as List but does not allow duplicates*

```
final Set<String> colors = {"red", "blue", "yellow"};
colors.add("pink");  // Adding a new element
 // Won't be added again because sets don't allow duplicates
colors.add("blue");
print(colors); // Output: {red, blue, yellow, pink}
```

Map

- Stores keys and associated values

```
Map<int, String> languages = {
    1: "Python",
    2: "Kotlin",
    3: "Java",
};

languages.forEach((key, value) {
    print("$key => $value");
});
```

Lambda

$$\lambda$$

Imperative vs. Declarative

Imperative Programming

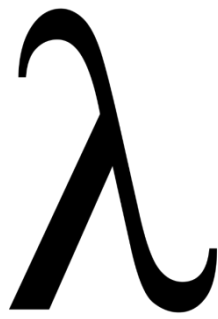- You tell the computer **how** to perform a task

Declarative Programming

- You tell the computer **what you want**, and you let the compiler (or runtime) figure out the best way to do it. This makes the code simpler and more concise
- Also known as **Functional Programming**
- **Declarative programming using Lambdas helps us to achieve KISS**

KEEP IT SHORT & SIMPLE

What is a Lambda?

- Lambda is an **anonymous function** that you can store in a variable, pass them as parameter, or return from other function. It has:
  - Parameters
  - A body

- It **don't have a name** (anonymous method)

- It **can be passed as a parameter** to other function:
  - As *code* to be executed by the receiving function

- Concise syntax:

```
(Parameters) => Body
```

Passing Lambda as a Parameter

- Lambda expression can be passed as a parameter to methods such as *forEach*, *filter* and *map* methods :

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
numbers.forEach((e) => print(e));
```

forEach - Calls a Lambda on Each Element of the list

- Left side of `=>` operator is a parameter variable

- Right side is the code to operate on the parameter and compute a result

- When using a lambda with a `List` the compiler can determine the parameter type

Lambda usage

- Allows working with collections in a **functional style**

```dart
bool isEven(int n) => n % 2 == 0;
void main() {
  // Range (1 to 10 inclusive)
  List<int> nums = List.generate(10, (i) => i + 1);
  // Version 1
  bool hasEvenNumber = nums.any((n) => n.isEven);
  // Verion 2
  hasEvenNumber = nums.any(isEven);

  // Version 3 - most compact
  hasEvenNumber = nums.any((n) => n % 2 == 0);
  print("Has even number: $hasEvenNumber");

  // Version 1
  List<int> evens = nums.where(isEven).toList();

  // Version 2
  evens = nums.where((n) => n % 2 == 0).toList();

  print("Even numbers: $evens");
}
```

14

Lambda usage

e.g. What's the average age of employees working in Doha?

```
List<Employee> employees = [
  Employee(name: "Sara Faleh", city: "Doha", age: 30),
  Employee(name: "Mariam Saleh", city: "Istanbul", age: 22),
  Employee(name: "Ali Al-Ali", city: "Doha", age: 24),
];

// Filtering employees in "Doha", mapping their ages,
// and calculating the average
double avgAge = employees
    .where((employee) => employee.city == "Doha")
    .map((employee) => employee.age)
    .reduce((a, b) => a + b) /
    employees.where((employee) => employee.city == "Doha").length;

print("Average age of employees in Doha: $avgAge");
```

Common operations on collections

Filter, Map, Reduce, and others
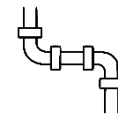
Common operations on collections

`.map`

o Applies a function to each list element

`.where(condition)`

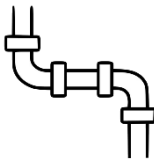o Returns a new list with the elements that satisfy the condition

`.find(condition)`

o Returns the first list element that satisfy the condition

`.reduce`

o Applies an accumulator function to each element of the list to reduce them to a single value

Operations Pipeline

- **A pipeline of operations:** a sequence of operations where the output of each operation becomes the input into the next

  - e.g., .where -> .map -> .toList

- Operations are either **Intermediate** or **Terminal**

- **Intermediate operations** produce a new list as output (e.g., map, filter, …)

- **Terminal operations** are the final operation in the pipeline (e.g., find, reduce, toList …)

  - Once a terminal operation is invoked then no further operations can be performed

Filter using .where 🔻
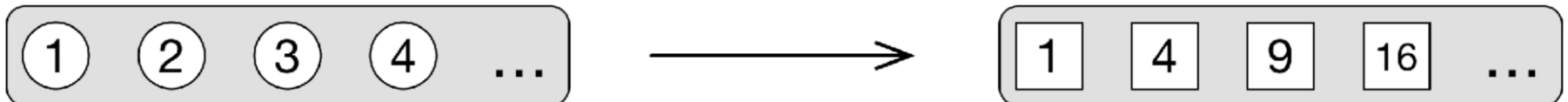
Keep elements that satisfy a condition

```
nums.where((n) => n % 2 == 0)
```
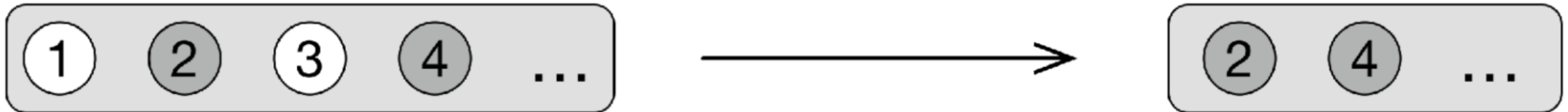

Map

Transform elements by applying a Lambda to each element

```
nums.map((n) => n * n)
```

Reduce

Apply an accumulator function to each element of the list to reduce them to a single value

```
// Imperative
var sum = 0;
for (var n in list)
    sum = sum + n;
```

```
//Declarative
var sum = nums.reduce( (acc, n) => acc + n );
```

Accumulation Variable

Accumulation Lambda

Collapse the multiple elements of a list into a single element

| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
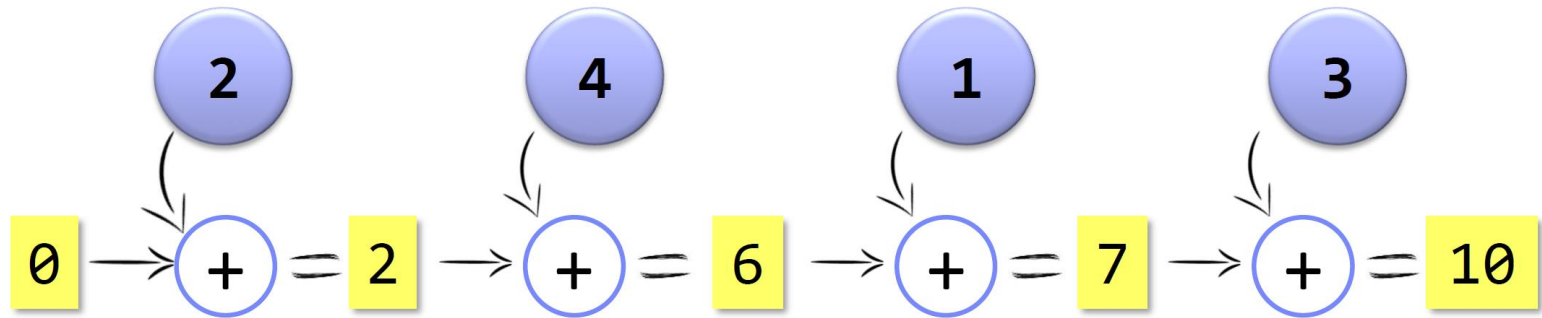
reduce → 36

Reduce

$.reduce$ { (acc, n) => acc + n }

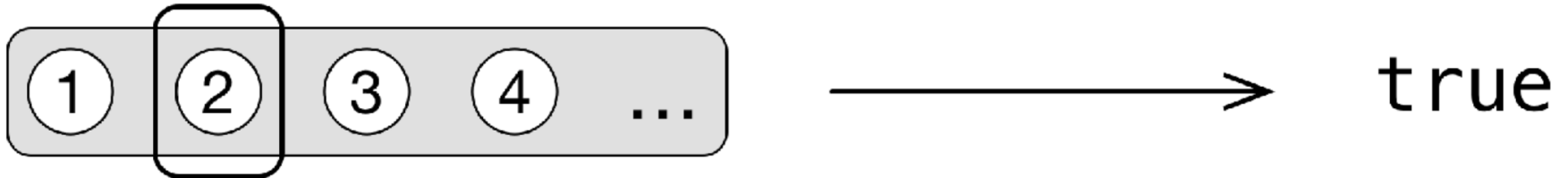Reduce is **terminal** operation that yields a single value

any and every

- **any** returns true if it finds an element that satisfies the lambda condition
- **every** returns false if it finds an element that fails the lambda condition

```
var hasEvenNumber = nums.any((n) => n % 2 == 0);
```


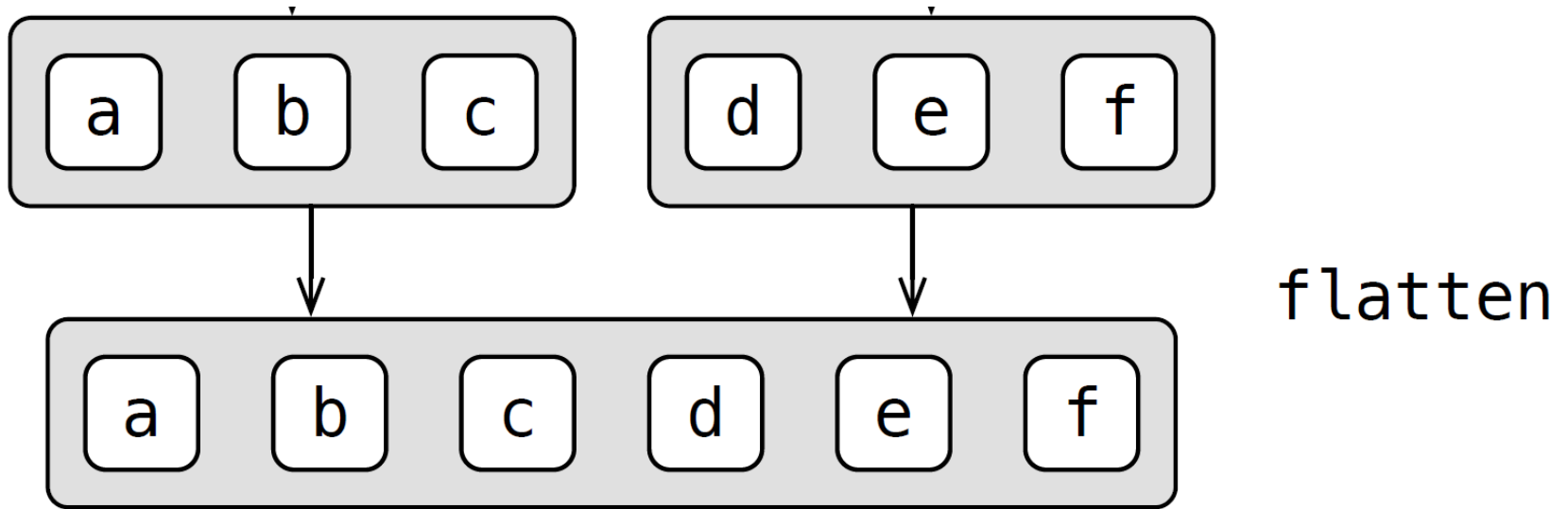# firstWhere

Return first element satisfying a condition

```
var firstEven = nums.firstWhere((n) => n % 2 == 0);
```
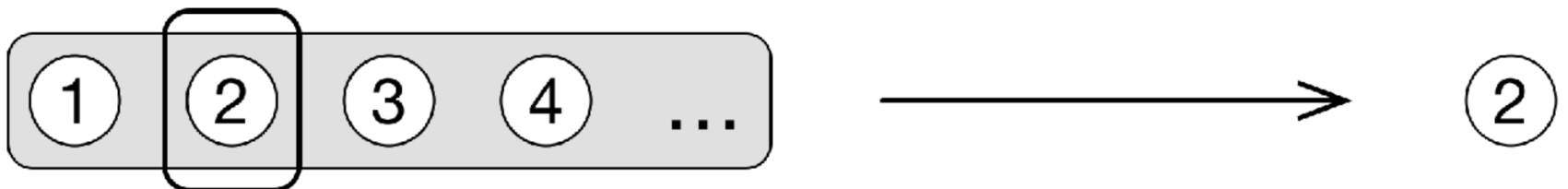
Expand

flatten

```
List<List<String>> listOfList = [
  ["a", "b", "c"],
  ["d", "e", "f"]
];

// Flattening the list of lists
List<String> singleList = listOfList.expand((list) => list).toList();

// Printing the result
print(singleList); // Output: [a, b, c, d, e, f]
```

expand

Do a map and flatten the results into 1 list

Each book has a list of authors. **expand** combines them to produce a single list of **all** authors

```dart
List<Book> books = [
  Book("Head First Dart", ["Dawn Griffiths", "David Griffiths"]),
  Book("Dart in Action", ["Dmitry Jemerov", "Svetlana Isakova"]),
];

// Flattening the list of authors
var authors = books.expand((book) => book.authors).toList();
print(authors);
```

Sort a List using Lambda

Sort strings by length (shortest to longest)

```
List<String> names = ["Farid", "Saleh", "Ali", "Sarah", "Samira", "Farida"];

var sorted = List.of(names)..sort((a, b) =>
                            a.length.compareTo(b.length));

// Without the cascade operator, you would have to
// do this in two steps:
// sorted = List.of(names);
// sorted.sort((a, b) => a.length.compareTo(b.length));

print(names);

print(">Sorted by length:");
print(sorted);
```

Record

- Records is a data structure that allows you to group **multiple values together** without needing to create a class

  - useful when you need to return or pass around multiple values from a function or when you want to combine values into a logical unit without a class

  - Supports Positional and Named Fields: Records can have both positional and named fields

  - Type Safety: Dart records are strongly typed, meaning the fields have specific types that must be followed

  - Immutability: Records are immutable; once created, you cannot change the values in them

Why Use Records?

- Convenient for returning multiple values:

  o Records are simpler than creating custom classes

  o No need for classes: You don't need to define a separate class for temporary or simple structures

- Type Safety: You have clear type constraints, reducing errors

- Readable code: Named fields improve readability and allow for clearer intent without the need for complex structures

- Efficient: Records are lightweight and immutable

Record Example

```dart
// Function returning coordinates as a record
(double, double) getCoordinates() {
  double latitude = 25.276987;
  double longitude = 51.520008;
  return (latitude, longitude);  // Return a record with two
positional fields
}

void main() {
  var coordinates = getCoordinates();

  print("Latitude: ${coordinates.$1}");
  print("Latitude: ${coordinates.$2}");

  // Extract the latitude and longitude from the record
  var (latitude, longitude) = getCoordinates();
  print("Latitude: $latitude");
  print("Longitude: $longitude");
}
```

Record with named fields

- Makes the code more readable and self-explanatory: Named fields make it clear what each value represents

```
// Function returning coordinates as a record with named fields
({double lat, double lon}) getCoordinates() {
  double latitude = 25.276987;
  double longitude = 51.520008;
  // Return a record with named fields
  return (lat: latitude, lon: longitude);
}

void main() {
  var coordinates = getCoordinates();

  // Extract the latitude and longitude from the record
  print("Latitude: ${coordinates.lat}");
  print("Longitude: ${coordinates.lon}");
}
```
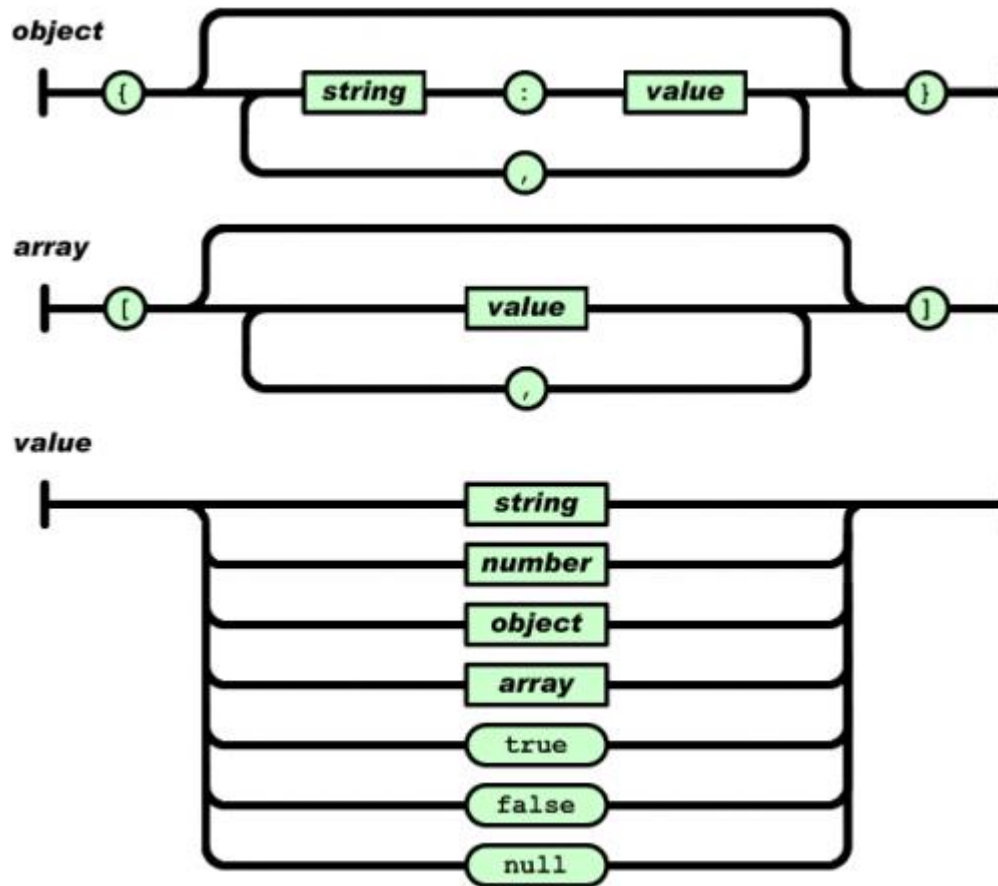
JSON Data Format

- **JSON** (JavaScript Object Notation) is a very popular lightweight data format to transform an object to a **text** form to ease storing and transporting data

  - o Encoding (aka serialization) turning a data structure into a string

  - o Decoding (aka deserialization) is the opposite process -> turning a string into a data structure

Serializing JSON manually using dart:convert

- Flutter has a built-in dart:convert library that includes a straightforward JSON encoder and decoder

```dart
import 'dart:convert';

void main() {
 var jsonString = '''
    {
      "name": "John Smith",
      "email": "john@dart.dev"
    }''';

  // Parse the JSON string into a Map
  final user = jsonDecode(jsonString) as Map<String, dynamic>;

  print('Hello, ${user['name']}!');
  print('We sent the verification link to ${user['email']}.');

  final userJsonString = jsonEncode(user);
  print(userJsonString);
}
```

Serializing JSON inside model classes

- Add two methods to the class:
  - ○ A Surah.fromJson() constructor, for constructing a new Surah instance from a map structure.
  - ○ A toJson() method, which converts a Surah instance into a map.

Surah

- id: int
- name: String
- englishName: String
- ayaCount: int
- type: String

```
// Convert a Surah object to a JSON map
Map<String, dynamic> toJson() => {
  'number': number,
  'arabicName': arabicName,
  'englishName': englishName,
  'verseCount': verseCount,
  'type': type,
};

// Convert a JSON map to a Surah object
Surah.fromJson(Map<String, dynamic> json) :
  number = json['number'],
  arabicName = json['arabicName'],
  englishName = json['englishName'],
  verseCount = json['verseCount'],
  type = json['type'];
```

@Serializable

- To use Json sterilization the class must be annotated with **@Serializable**

```kotlin
@Serializable
data class Surah (
    val id : Int,
    val name: String,
    val englishName : String,
    val ayaCount : Int,
    val type: String
)
```

Read JSON file

- Read a JSON file and convert its content to objects

```
val filePath = "data/surahs.json"
val fileContent = File(filePath).readText()

val surahs = Json.decodeFromString<List<Surah>>(fileContent)
```

- To utilize the **@Serializable** and **Json** class functionalities, ensure that you include the required dependencies in the **build.gradle** file of the module, as detailed in the documentation page.

Quick Tips!

You may use https://plugins.jetbrains.com/plugin/10054-generate-kotlin-data-classes-from-json Android Studio plugin to generate a Kotlin class from a json string!

Patterns Matching

- https://dart.dev/language/patterns

- https://happy-makadiya.medium.com/pattern-matching-in-dart-flutter-a-developers-playground-0f2143126acf

Summary

- To start thinking in the functional style ***avoid loops*** and instead use Lambdas

  - Widely used for list processing and GUI building to handle events

- A list can be processed in a pipeline

  - Typical pipeline operations are filter, map and reduce

- JSON is a very popular lightweight data format to **transform an object to a text form** to ease storing and transporting data

Resources

- JSON serialization

  - https://docs.flutter.dev/data-and-backend/serialization/json

  - https://codewithandrea.com/articles/parse-json-dart/