



Dart

Functional Programming

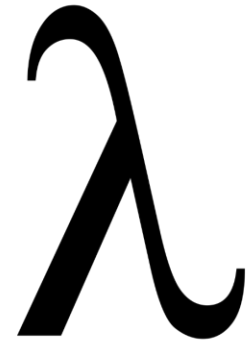
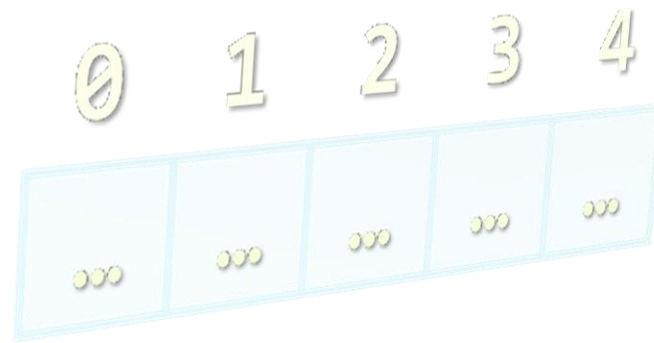


Table of Contents

1. Collections
2. Lambda
3. Common operations on collections
4. Records
5. JSON
6. Pattern Matching

Collections



List

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

```
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

A large, stylized black lambda symbol (λ) centered on the page. The symbol is a cursive-style character with a thick stroke, featuring a small hook at the top and a curved tail at the bottom.

Imperative vs. Declarative

Imperative Programming

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

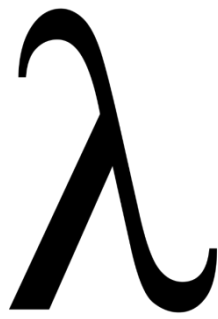
Declarative Programming

- 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 **I**T **S**HORT & **S**IMPLE



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) \Rightarrow 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**

```
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);
    // Version 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");
}
```

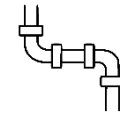
Lambda usage

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

- Applies a function to each list element

.where(condition)

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

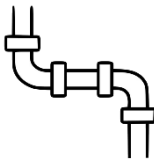
.firstWhere(cond: on)

- Returns the first list element that satisfy the condition

.reduce

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

Operations Pipeline

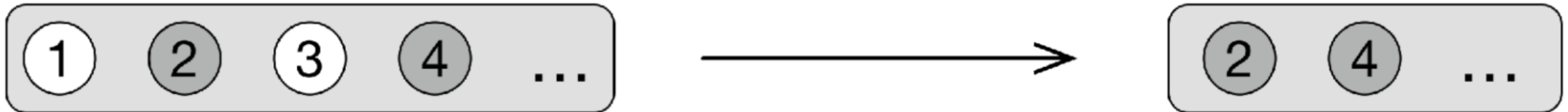


- **A pipeline of operations:** a sequence of operations where the output of each operation becomes the input into the next
 - e.g., `.where -> .map -> .toList`
- Operations are either **Intermediate** or **Terminal**
- **Intermediate operations** produce a new list as output (e.g., `map`, `filter`, ...)
- **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

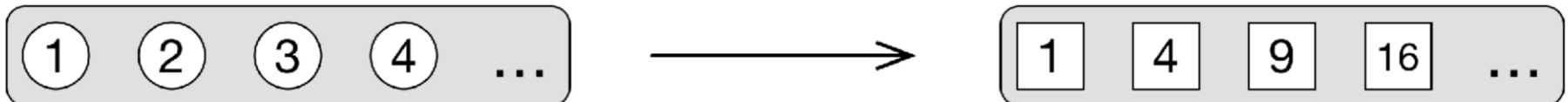
```
nums.where((n) => n % 2 == 0)
```



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

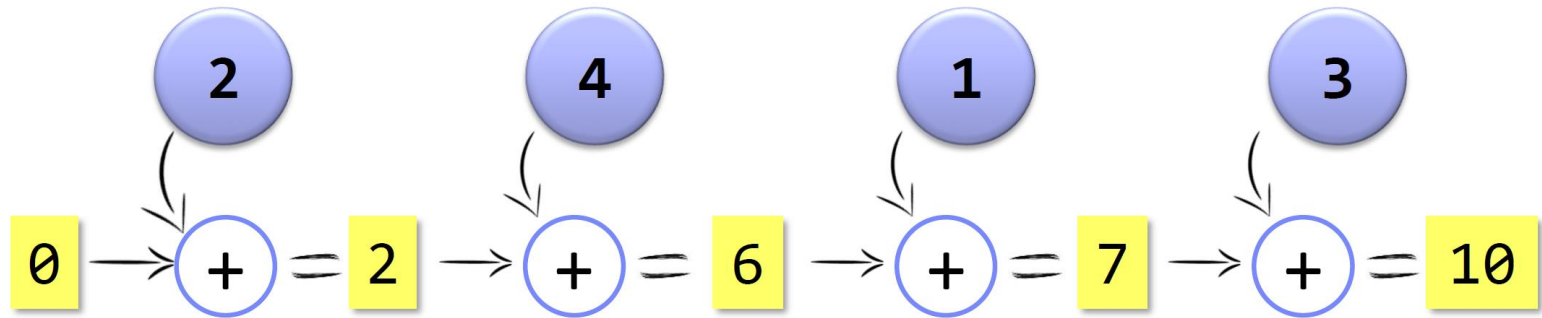
Accumulation
Lambda

Collapse the multiple elements of a list into a single element



36

Reduce



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

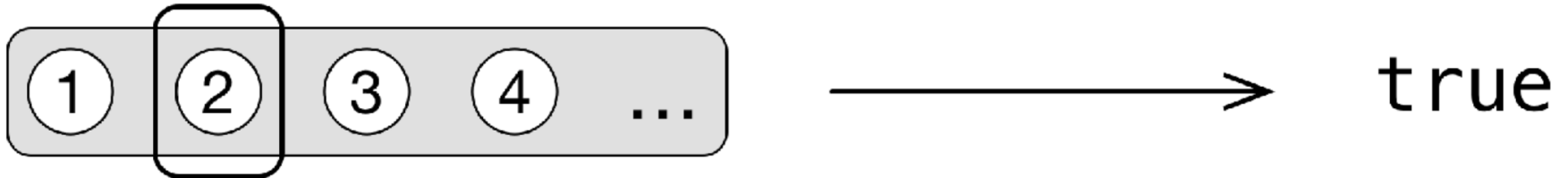
Reduce is **terminal** operation that yields a single value

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

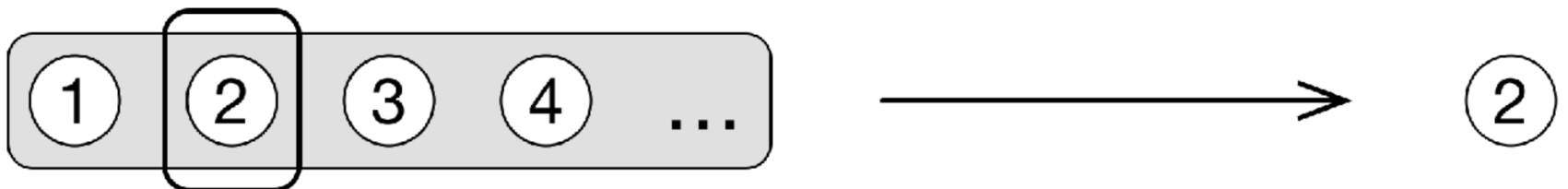
```
var hasEvenNumber = nums.any((n) => n % 2 == 0);
```



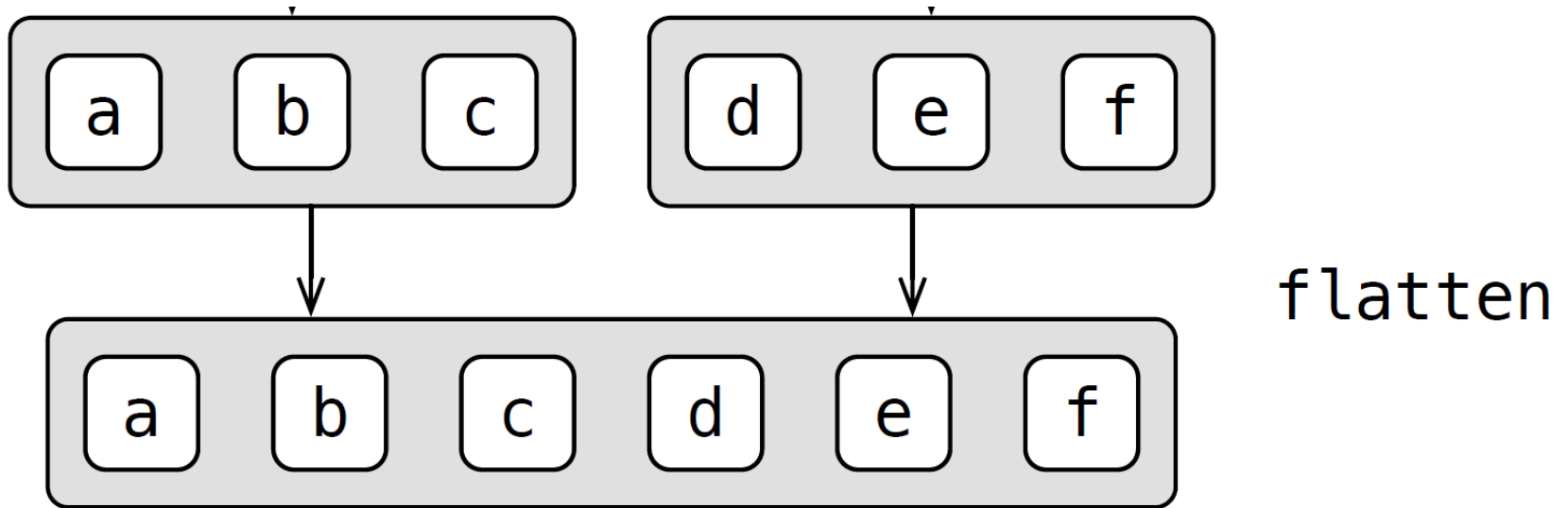
firstWhere

Return first element satisfying a condition

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



Expand



```
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

```
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);  
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);
```

Records

```
var (latitude, longitude) =  
    (25.276987, 51.520008);
```

Records

- A Record is a data structure that allows you to group **multiple values together** without needing to create a class
 - Records are comma-delimited field lists enclosed in parentheses
 - Records can contain both named and positional fields, like argument lists in a function
 - 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
 - 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:
 - Records are simpler than creating custom classes
 - 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

```
// 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");
}
```

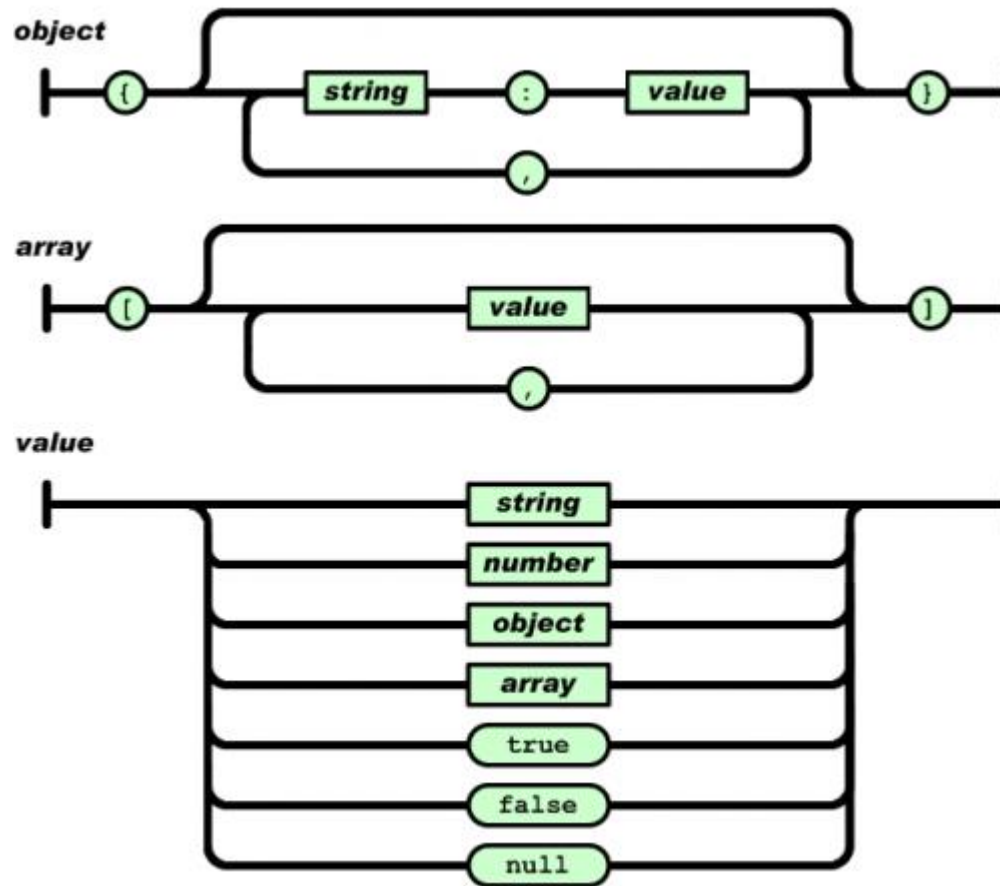
Record with named fields

- 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 long}) getCoordinates() {
    double latitude = 25.276987;
    double longitude = 51.520008;
    // Return a record with named fields
    return (lat: latitude, long: longitude);
}

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

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



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

```
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

```
// 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'];
```

Surah

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

Serializing JSON using a code generation library

- Package [json_serializable](#) can be used to auto-generate the implementation of `fromJson` and `toJson`
 - Simply annotate the class with **@JsonSerializable()**

```
/// An annotation for the code generator to know that this class needs the
/// JSON serialization logic to be generated.
@JsonSerializable()
class User {
  String name;
  String email;
  User(this.name, this.email);

  /// A necessary factory constructor for creating a new User instance
  /// from a map. Pass the map to the generated `_$_UserFromJson()` constructor.
  /// The constructor is named after the source class, in this case, User.
  factory User.fromJson(Map<String, dynamic> json) => _$_UserFromJson(json);

  /// `toJson` implementation simply calls the private, generated
  /// helper method `_$_UserToJson`.
  Map<String, dynamic> toJson() => _$_UserToJson(this);
}
```

json_serializable dependencies

- To utilize [json_serializable](#) package, ensure that you include the required dependencies in **pubspec.yaml**

```
dev_dependencies:  
  build_runner: ^2.4.9  
  json_annotation: ^4.9.0  
  json_serializable: ^6.8.0
```

- Run the build_runner to generate the .g.dart files:

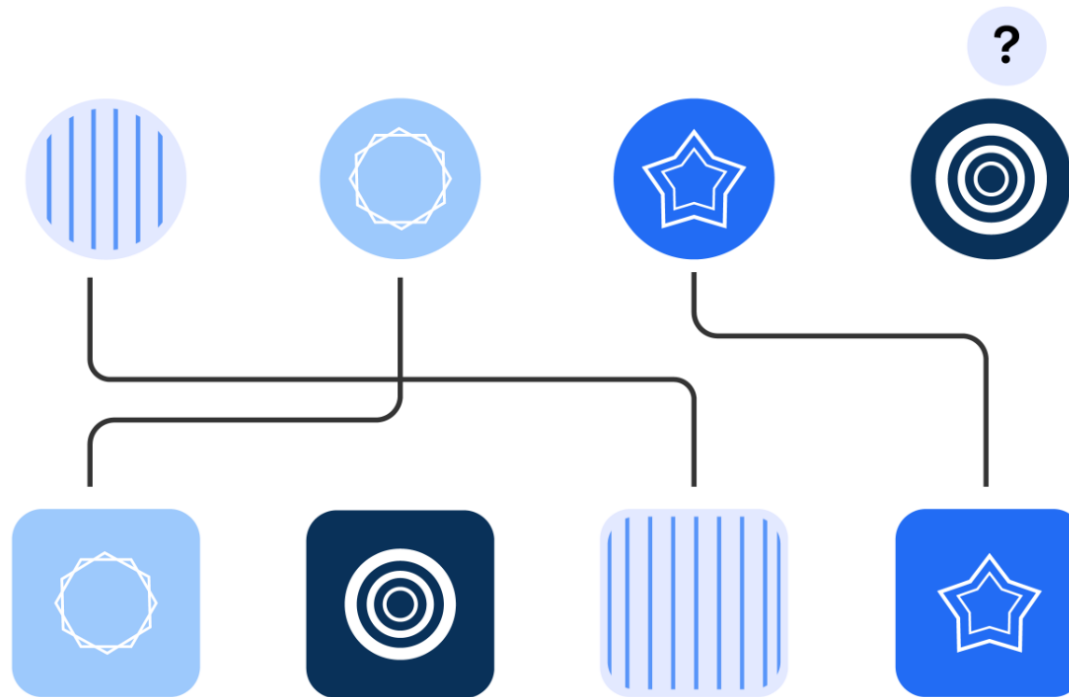
```
dart run build_runner build --delete-conflicting-outputs
```

Read JSON file

- Read a JSON file and convert its content to objects

```
final filePath = "data/surahs.json"  
// Read the content of the file at the given path as a string  
final fileContent = File(filePath).readAsStringSync();  
  
// Parse the JSON content into a list of dynamic objects  
final List<dynamic> jsonList = jsonDecode(fileContent);  
  
// Convert each dynamic object into a Surah instance using fromJson  
final surahs = jsonList.map((json) => Surah.fromJson(json)).toList();
```

Pattern Matching



What are patterns?

- A pattern defines a specific **shape** that that app data may match
 - They are used to check whether a piece of data has the form you expect => this is called **pattern matching**
 - If it does, then optionally use the pattern to extract portions of the data into new variables => this is called **destructuring** (i.e., break a value into its **constituent parts**)
- Patterns enable you to break down complex data structures (e.g., objects, records, list) directly within control flow statements like switch, if-case, and loops to match specific types, values, or structures
 - simplifying data handling, type checking, data validation and extraction of values

Key Benefits of Patterns

Patterns provide a **more intuitive, concise and simpler** way for matching and deconstructing data structures to extract specific parts from it. Resulting in:

- **Cleaner code:** reduces the need for manual type checks and value extraction
- **Expressive control flow:** by combining pattern matching with switch, if, and for loops, you can build expressive and flexible control flows to handle different cases based on the type or structure of data
- **Simplified way to match or extract specific parts of a complex data structures** including lists, records and objects

Usage scenarios

- **Switch statements** for matching and deconstructing a complex data (list, map, object, record)
- **If-case statements** for conditional pattern matching
- **For loops** to destructure elements while iterating over collections
- **Deconstructing a complex data (list, map, object, record)** to extract specific fields easily

Constant patterns

- Use constant patterns to match a value against a constant (such as a specific number, string, or enum value)
 - Allows you to check if a variable holds a specific constant value and execute logic based on that match

```
String handleUserRole(UserRole role) {  
    // Use a switch expression and constant patterns to return a  
    // message based on the role  
    return switch (role) {  
        UserRole.admin => 'Welcome, Admin! You have full access.',  
        UserRole.user  => 'Welcome, User! You have limited access.',  
        UserRole.guest  => 'Welcome, Guest! You can only view public content.',  
    };  
}
```

Relational patterns

- Use relational patterns to test how a value compares to constants or ranges of values
 - E.g., instead of writing complex if-else blocks, you can use relational patterns to test if a value is less than, greater than, or within a certain range, which makes the code more concise and readable.

```
String getLetterGrade(double grade) {  
    // Use a switch expression with relational patterns to map  
    // the grade to a letter grade  
    return switch (grade) {  
        >= 90          => 'A',  
        >= 85 && < 90 => 'B+',  
        >= 80 && < 85 => 'B',  
        >= 75 && < 80 => 'C+',  
        >= 70 && < 75 => 'C',  
        >= 65 && < 70 => 'D+',  
        >= 60 && < 65 => 'D',  
        < 60          => 'F',  
        _            => 'Invalid grade'  
    };  
}
```

According to [QU grading policy](#)

Type Patterns: Matching Data Types

- Type patterns allow you to **check the type of a variable** and deconstruct it in a single operation
 - Allows for easier type-specific handling without writing complex if-else chains
 - E.g., differentiate between different user roles and handle them appropriately

```
String getUserScreen(User user) {  
    // Using A switch expression to determine the screen based on the user type  
    return switch (user) {  
        Admin admin => AdminDashboard(admin),  
        VerifiedUser verifiedUser => UserHome(verifiedUser),  
        Guest guest => WelcomeScreen(),  
    };  
}
```

Destructuring Patterns: Deconstructing Records

- Patterns help destructure records to access their values in a concise way

```
void printCoordinates((double, double) point) {  
    // record (double, double) is deconstructed directly  
    // within the switch statement to access lat and long  
    switch (point) {  
        case (var lat, var long):  
            print('lat: $lat, long: $long');  
    }  
}
```

```
void main() {  
    var point = (10.553, 21.562);  
    printCoordinates(point);  
}
```

Destructuring Patterns: Deconstructing Lists

- List patterns allow you to match specific elements in a list and extract them

```
void processList(List<int> numbers) {  
    switch (numbers) {  
        /* The list pattern [var first, var second, ...] is used to extract  
        the first two elements of the list while ignoring the rest. */  
        case [var first, var second, ...]:  
            print('First: $first, Second: $second');  
            break;  
        default:  
            print('List does not match the expected pattern.');    }  
}  
  
void main() {  
    var myList = [10, 20, 30, 40];  
    processList(myList); // Output: First: 10, Second: 20  
}
```

Patterns in if-case statement

- You can use patterns inside if statements for more complex matching logic, making conditional checks easier and more expressive

```
double getArea(Shape shape) {  
    /* Using pattern matching to calculate area based on shape type  
       This if pattern allows you to check the structure of the data  
       and extract relevant values in the same expression.  
    */  
    if (shape case Circle(:var radius)) {  
        return math.pi * radius * radius;  
    } else if (shape case Rectangle(:var width, :var height)) {  
        return width * height;  
    } else {  
        throw Exception('Unknown shape');  
    }  
}
```

Patterns with for Loops

- Patterns can also be applied in for loops to destructure and process each element in a collection.

```
void printScores(List<(String , int )> students) {  
    /*  
    Each record in the list of students is destructured in the for loop,  
    allowing you to directly access name and score without needing to access  
    each record's individual fields manually */  
  
    for (var (name, score) in students) {  
        print('Student: $name, Score: $score');  
    }  
}  
  
void main() {  
    var students = [('Ali', 85), ('Fatima', 90), ('Ahmed', 78)];  
    printScores(students);  
}
```


Validating incoming JSON

- JSON data typically comes from an external source over the network. You need to validate it first to confirm its structure before deconstructing it
 - Can apply patterns to validate and extract data from JSON objects in a clean and readable way

// Without patterns, validation is verbose:

```
if (json is Map<String, Object> &&
    json.length == 1 &&
    json.containsKey('user')) {
    var user = json['user'];
    if (user is List<Object> &&
        user.length == 2 &&
        user[0] is String &&
        user[1] is int) {
        var name = user[0] as String;
        var age = user[1] as int;
        print('User $name is $age years old.');
```

A pattern in an if-case statement can achieve the same JSON validation in a **more declarative**, and **much less verbose** way.

//With Pattern: less verbose method of validating

```
if (json case {'user': [String name, int age]}) {
    print('User $name is $age years old.');
```

Pattern Matching Exhaustive Checking

- Exhaustive checking ensures that when you use a switch statement with patterns, all possible cases are covered
 - This guarantees that no case is left unhandled, making your code more reliable
 - E.g., ensuring that all possible payment types are handled in the switch expression

```
String processPayment(PaymentMethod paymentMethod) {  
  return switch(paymentMethod) {  
    CreditCard(:var cardNumber) => 'Payment with card number: $cardNumber',  
    PayPal(:var email) => 'PayPal payment with email: $email',  
    BankTransfer(:var bankAccount) => 'Bank transfer to account: $bankAccount',  
  };  
}
```

- Dart compiler knows that PaymentMethod has exactly three possible types: CreditCard, PayPal, and BankTransfer because it's a sealed class.

- This allows the switch expression to be exhaustive.

- Sealed class that can only be extended by classes defined in the same library

```
sealed class PaymentMethod {}  
class CreditCard extends PaymentMethod {  
  final String cardNumber;  
  CreditCard(this.cardNumber);  
}  
class PayPal extends PaymentMethod {  
  final String email;  
  PayPal(this.email);  
}  
class BankTransfer extends PaymentMethod {  
  final String bankAccount;  
  BankTransfer(this.bankAccount);  
}
```

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
- Patterns are used to check whether a piece of data conforms to particular **shape** (i.e., **pattern matching**)
 - and then optionally use the pattern for **destructuring** to extract portions of the data into new variables

Resources

- Drat Collections
 - <https://dart.dev/language/collections>
- JSON serialization
 - <https://docs.flutter.dev/data-and-backend/serialization/json>
 - <https://codewithandrea.com/articles/parse-json-dart/>
 - JSON serialization package
https://pub.dev/packages/json_serializable
- Records
 - <https://dart.dev/language/records>
- Patterns
 - <https://dart.dev/language/patterns> & [YouTube video](#)