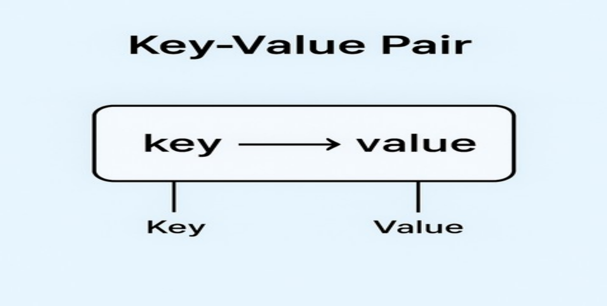
Mastering Maps in Dart

What is a Map in Dart?

A Map in Dart is a collection of key-value pairs, where each key is unique and associated with a specific value.

Think of it like a dictionary:

- A word is the key
- Its definition is the value



Why Use Maps?

1.Quick Data Lookup by Key

Instantly retrieve a value using its key — no need to search linearly like in lists.

2.Efficient Organization

Label and structure your data clearly with key-value mappings.

3.Flexible Data Types

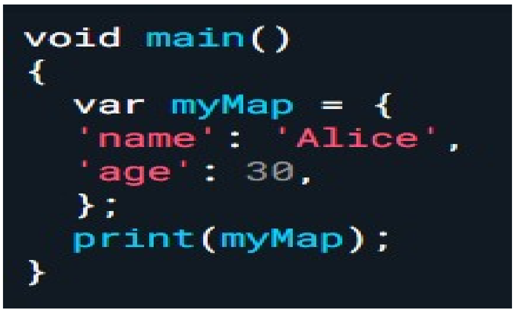
Store any value type — numbers, strings, booleans, lists, even other maps.

4.Essential for JSON Handling

Maps align perfectly with JSON data structures from APIs.

Map Declaration & Initialization

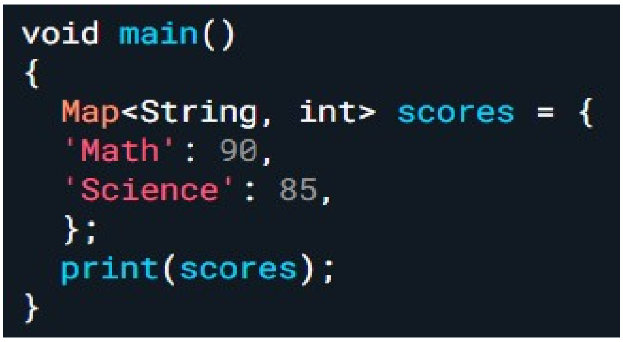
1.Map Literal



```
void main()
{
  var myMap = {
  'name' : 'Alice',
  'age': 30,
  };
  print(myMap);
}
```

- ✓ This is the simplest and most common way to create a map.
- ✓ The keys and values are inferred from what you provide.
- ✓ This creates a map with two entries.
- ✓ When you have a small map with known data at compile time.

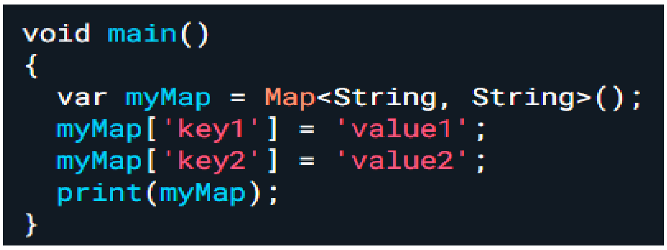
2. Typed Map Literal



```
void main()
{
  Map<String, int> scores = {
  'Math': 90,
  'Science': 85,
  };
  print(scores);
}
```

- ✓ The keys are Strings and the values are integers.
- ✓ This provides type safety, so you can't accidentally put non-integer values.
- ✓ When you want strict type checking and clarity about map contents.

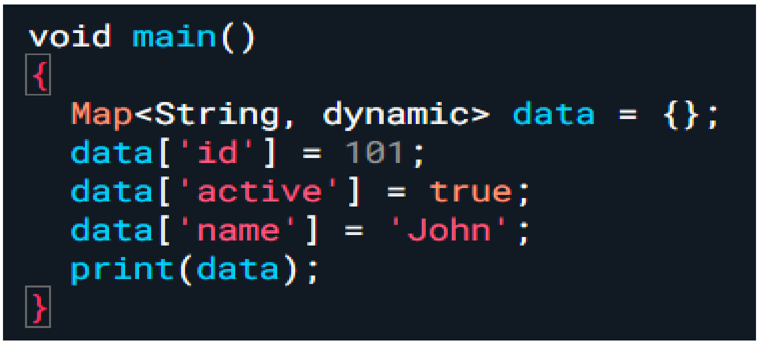
3. Empty Map with Constructor



```
void main()
{
  var myMap = Map<String, String>();
  myMap['key1'] = 'value1';
  myMap['key2'] = 'value2';
  print(myMap);
}
```

- ✓ Creates an empty map with specified key and value types.
- ✓ You add entries dynamically after creation.
- ✓ When you don’t have data at creation time, or you want to build the map step by step

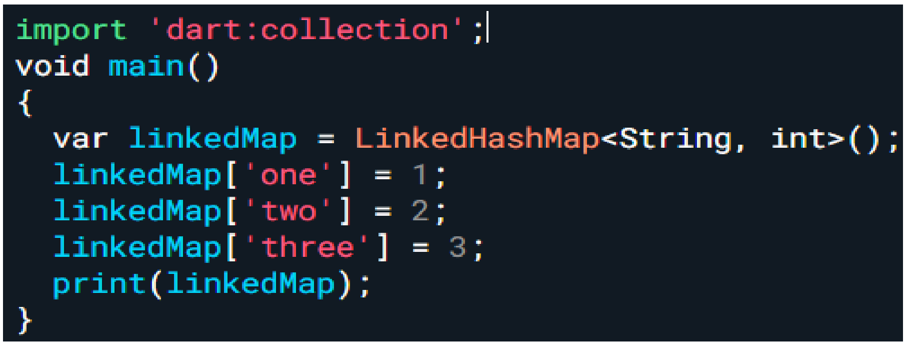
4. Map with Dynamic Values



```
void main()
{
  Map<String, dynamic> data = {};
  data['id'] = 101;
  data['active'] = true;
  data['name'] = 'John';
  print(data);
}
```

- ✓ Keys are Strings, but values can be any type (dynamic).
- ✓ Allows mixed types inside the same map.
- ✓ Very useful when working with JSON-like data or loosely typed info.

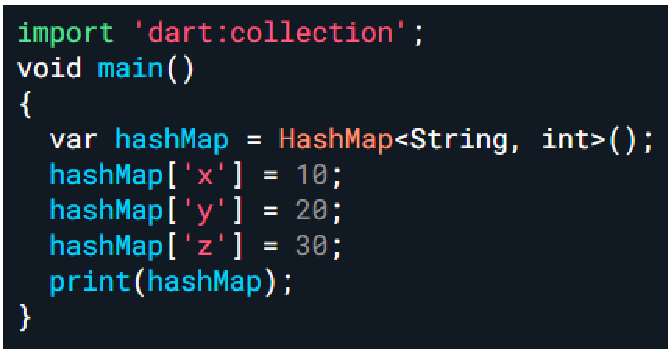
5. LinkedHashMap (Maintains insertion order — default behaviour of Dart maps)



```
import 'dart:collection';
void main()
{
  var linkedMap = LinkedHashMap<String, int>();
  linkedMap['one'] = 1;
  linkedMap['two'] = 2;
  linkedMap['three'] = 3;
  print(linkedMap);
}
```

- ✓ Maintains the order in which keys are inserted.
- ✓ Dart’s default Map implementation is actually a LinkedHashMap.
- ✓ Used when Order of keys matters (like menus, ordered data).

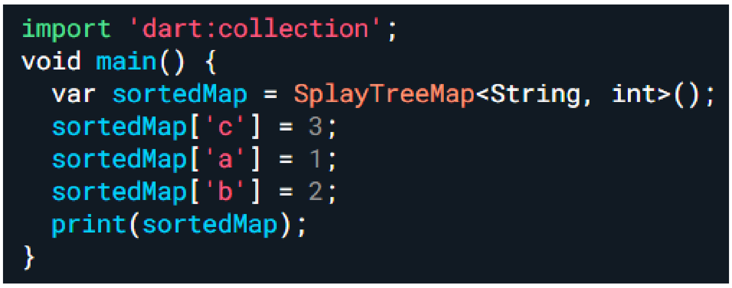
6. HashMap (No guaranteed order, fast lookup)



```dart
import 'dart:collection';
void main()
{
  var hashMap = HashMap<String, int>();
  hashMap['x'] = 10;
  hashMap['y'] = 20;
  hashMap['z'] = 30;
  print(hashMap);
}
```

- ✓ Uses a hash table for storage.
- ✓ Does not guarantee any specific order when iterating.
- ✓ Generally faster for lookups when order doesn’t matter.
- ✓ Use when: Performance is important and you don’t care about the order of keys.

7. SplayTreeMap (Sorted by key)



```dart
import 'dart:collection';
void main() {
  var sortedMap = SplayTreeMap<String, int>();
  sortedMap['c'] = 3;
  sortedMap['a'] = 1;
  sortedMap['b'] = 2;
  print(sortedMap);
}
```

- ✓ Automatically sorts the keys in ascending order.
- ✓ Uses a balanced tree internally.
- ✓ Iterating the map yields keys in sorted order.
- ✓ Use when: You need sorted key order without manually sorting**.**

Dart Map: Null Keys & Values

Yes, Dart Maps allow null keys and null values, as long as the key and value types are declared nullable (String? , int?) etc..

```
Map<String?, int?> nullableMap = {
  null: null,
  'one': 1,
  'two': null,
  null: 42, // Overwrites the previous null key
};

print(nullableMap);
// Output: {null: 42, one: 1, two: null}
```

SplayTreeMap does not allow null keys but allow null values

SplayTreeMap sorts keys to maintain order.

- Sorting requires **non-null, comparable keys**

```
import 'dart:collection';

void main()

{

  var map = SplayTreeMap<int, String?>();

  map[1] = 'Hello';

  map[2] = null; //       Null value is allowed

  map[null] = 'Oops'; //       throws an error

  print(map);

}
```

Properties of map

PROPERTY	DESCRIPTION
entries	returns an iterable of key-value pairs (`MapEntry<K, V>`)
isEmpty	`true` if the map has no key-value pairs
isNotEmpty	`true` if the map contains at least one entry
keys	Iterable of all keys in the map
values	Iterable of all values in the map
hashCode	Hash code of the map object
runtimeType	Returns the `Type` of the object
length	Number of key-value pairs

Modification methods of map

METHOD	DESCRIPTION
addAll(map)	Adds all key-value pairs from another map
addEntries(entries)	Adds a list of `MapEntry<K, V>`
clear()	Removes all entries
putIfAbsent(key, func)	Adds key with value from function if not already present
remove(key)	Removes entry with the given key
removeWhere((k, v))	Removes entries matching the predicate
update(key, (v) => ..., {ifAbsent})	Updates the value for a key
updateAll((k, v) => ...)	Updates all values using a function

Query methods of map

METHOD	DESCRIPTION
containsKey(key)	Returns true if the key exists
containsValue(value)	Returns true if the value exists

Iteration & Mapping

METHOD	DESCRIPTION
forEach((k, v))	Applies a function to each key-value pair
map((k, v) => MapEntry)	Creates a new map by transforming each entry

Type Casting & Conversion

METHOD	DESCRIPTION
cast<K2, V2>()	Casts the map to a different generic type
toString()	Returns a string representation of the map

Advanced

METHOD	DESCRIPTION
noSuchMethod()	Called when a non-existent method is accessed (from `Object`)

This is an advanced feature from Dart's Object class.

Normally, you don’t override this unless creating very dynamic or proxy objects.It catches calls to undefined methods.

```
class Person
{
  String name;

  Person(this.name);

  @override
  dynamic noSuchMethod(Invocation invocation)
  {
    var methodName = invocation.memberName.toString();
    return "Oops! '$methodName' doesn't exist on Person.";
  }
}

void main()
{

  dynamic p = Person('Alice');  // Declare as dynamic to allow
noSuchMethod to work

  print(p.name);          // Works fine
  print(p.sayHello());    // Calls noSuchMethod (no compile-time error now)
  print(p.age);           // Calls noSuchMethod
}
```

What is Invocation?

- ✓ When Dart calls noSuchMethod, it passes an Invocation object.

- ✓ This object contains details about the missing method or property call that triggered noSuchMethod.

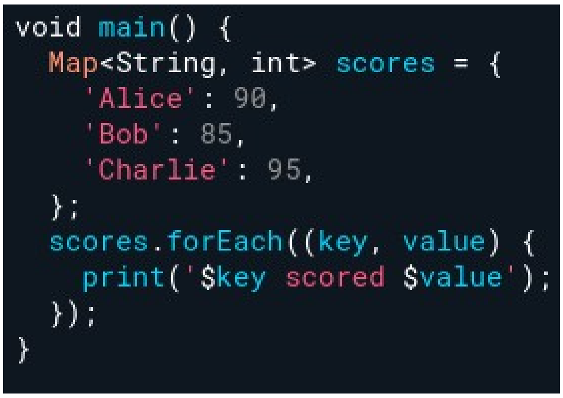
- ✓ It tells you what was called, with what arguments, etc.

memberName: The name of the method, getter, or setter that was called but doesn’t exist.

- Now calling sayHello() or accessing age will not cause compile errors, and instead call noSuchMethod at runtime.
- If you try the same with Person p = Person('Alice'); (non-dynamic), Dart will report errors at compile time, because it knows those methods don’t exist.
- Dart represents method and property names internally as Symbol objects.
- Instead of just storing a plain string, Dart uses Symbol to uniquely identify identifiers.

Map Iterations:

1.forEach – Quick looping, functional style



```
void main() {
  Map<String, int> scores = {
    'Alice': 90,
    'Bob': 85,
    'Charlie': 95,
  };
  scores.forEach((key, value) {
    print('$key scored $value');
  });
}
```

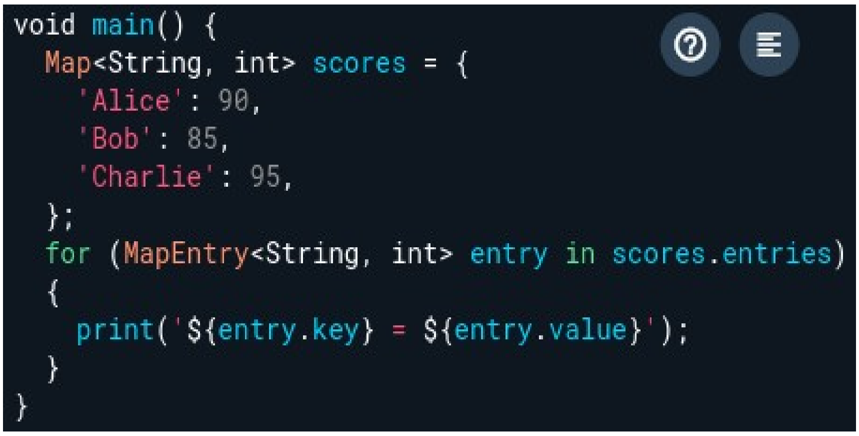
A concise, functional way to iterate over both keys and values.

Useful for simple operations like printing or basic processing.

Use Case:

Use forEach when you want a clean syntax to perform an operation on each key-value pair without needing to manage indexes or iterators.

2. for-in on entries – Access both keys and values



```dart
void main() {
  Map<String, int> scores = {
    'Alice': 90,
    'Bob': 85,
    'Charlie': 95,
  };
  for (MapEntry<String, int> entry in scores.entries)
  {
    print('${entry.key} = ${entry.value}');
  }
}
```

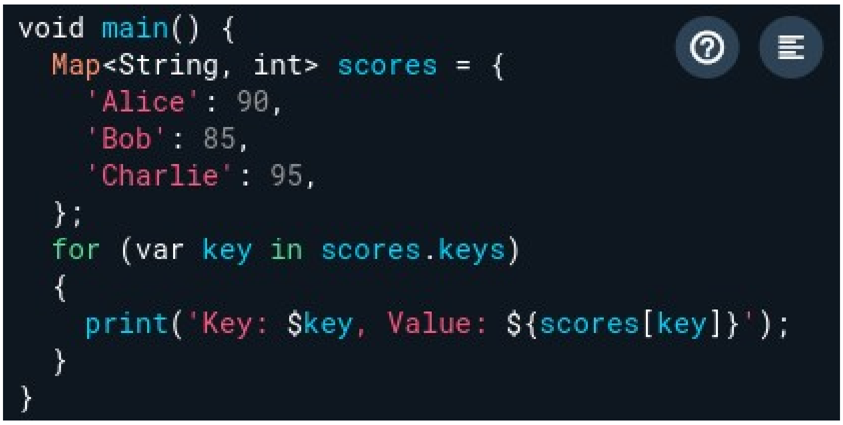
Iterates over the MapEntry objects in the map.

More readable than forEach when working in a traditional loop structure.

Use Case:

Choose this when you prefer a more explicit structure, especially for more complex operations that may span multiple lines.

3. Iterating over keys – When only keys or manual value access is needed



```dart
void main() {
  Map<String, int> scores = {
    'Alice': 90,
    'Bob': 85,
    'Charlie': 95,
  };
  for (var key in scores.keys)
  {
    print('Key: $key, Value: ${scores[key]}');
  }
}
```

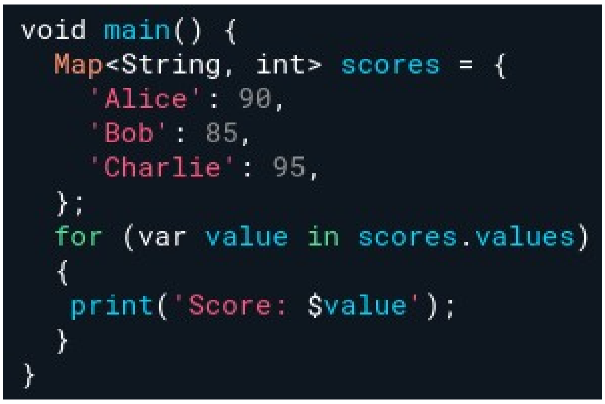
Loops through only the keys.

Use the key to manually access the corresponding value.

Use Case:

Use this when you're primarily focused on keys, or need custom logic for fetching or manipulating values.

4. Iterating over values – When only values are needed



```
void main() {
  Map<String, int> scores = {
    'Alice': 90,
    'Bob': 85,
    'Charlie': 95,
  };
  for (var value in scores.values)
  {
   print('Score: $value');
  }
}
```

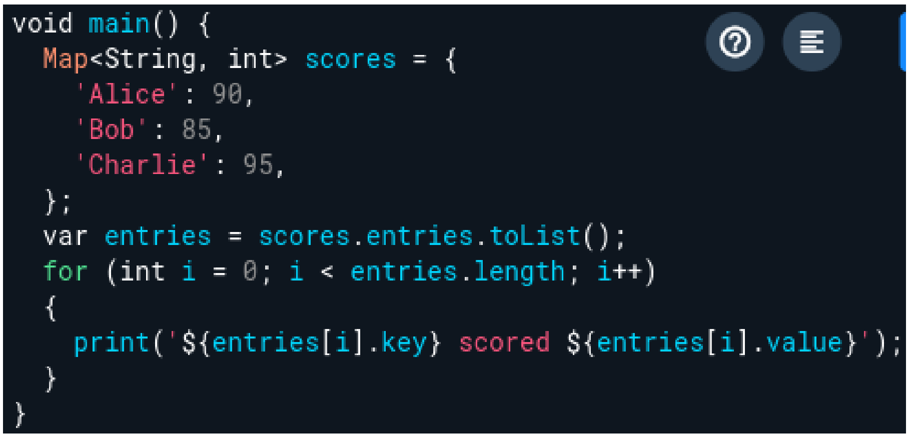
Iterates over just the values.

Does not provide access to keys.

Use Case:

Use this when you don’t care who the score belongs to, just need to work with the score data (e.g., calculating average, filtering values).

5. Classic for loop with entries.toList() – When index or position matters



```
void main() {
  Map<String, int> scores = {
    'Alice': 90,
    'Bob': 85,
    'Charlie': 95,
  };
  var entries = scores.entries.toList();
  for (int i = 0; i < entries.length; i++)
  {
    print('${entries[i].key} scored ${entries[i].value}');
  }
}
```

Converts entries to a list for indexed access.

Allows tracking of position, which is not possible with native map iteration.

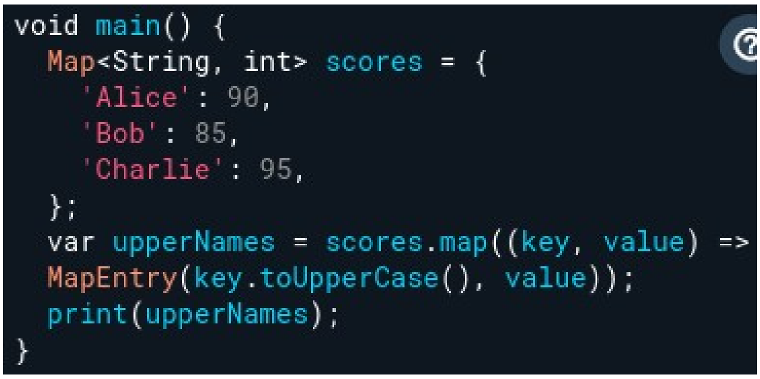
Use Case:

Dart Maps do NOT support indexing like lists do.

Converting the map entries or keys to a list — so you get indexed access to the entries

Use this when the order or position of items matters (e.g., displaying ranked results or applying operations based on index).

6. Using .map() – Transforming the map



```
void main() {
  Map<String, int> scores = {
    'Alice': 90,
    'Bob': 85,
    'Charlie': 95,
  };
  var upperNames = scores.map((key, value) =>
  MapEntry(key.toUpperCase(), value));
  print(upperNames);
}
```

Transforms the original map into a new one by applying a function to keys and/or values.

Does not mutate the original map.

Use Case:
Use .map() when you want to create a modified version of the map, such as changing the case of keys, transforming values, or filtering.

What is .entries in a Dart Map?

- ✓ An iterable collection of all MapEntry objects in a Map

- ✓ .entries is a property of the Map class.

- ✓ It returns an Iterable of MapEntry objects.

- ✓ Each MapEntry represents a single key-value pair in the map.

Think of .entries as a list of pairs, where each pair has a key and a value.It lets you loop through the map's contents with both keys and values together.

Time complexity

All standard iteration methods are linear time O(n) — they scale proportionally with the number of entries in the map.

The small differences like creating a list (toList()) might cost a little more, but it's still linear and usually negligible for small-to-medium maps.

Is Map better than List or Set?

None is universally better — it depends on your data structure needs.

Choose based on your access pattern, data uniqueness, and whether you need key-value association.

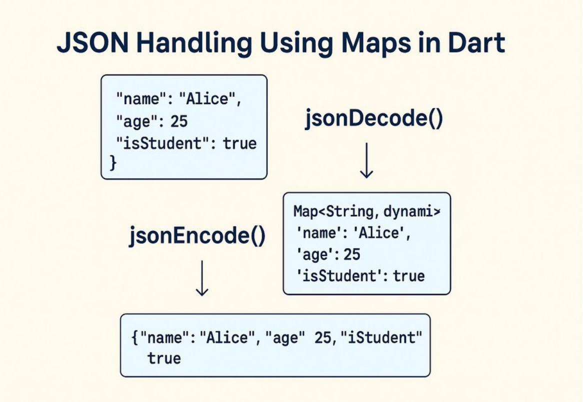
JSON Handling Using Maps in Dart

What is JSON?

- JSON stands for JavaScript Object Notation.
- It’s a lightweight data format used for storing and exchanging data between systems — especially between a client (like a mobile app) and a server (like an API).
- JSON is key-value based, and its structure closely resembles a Dart Map.
- That’s why Dart Maps are a perfect fit when dealing with JSON data.

Why is this important in Dart?

- In real-world Flutter/Dart apps, we often communicate with web APIs.
- These APIs usually return data in JSON format.
- To work with this data, we must convert JSON to Dart Map objects — and that’s where jsonDecode() and jsonEncode() come in.



```
 "name": "Alice",
 "age": 25
 "isStudent": true
}
```

```
Map<String, dynami>
 'name': 'Alice',
 'age': 25
 'isStudent': true
```

```
{"name": "Alice", "age"  25, "iStudent"
  true
```

Advanced Map Concepts

1.Map Equality Using MapEquality (from collection package)

- ✓ In Dart, comparing two maps with == checks reference, not content.
- ✓ Use MapEquality from the collection package to check content equality.

```
import 'package:collection/collection.dart';
  void main() {
 var map1 = {'name': 'Alice', 'age': 25};
 var map2 = {'name': 'Alice', 'age': 25};

 print(map1 == map2);//
 const equality = MapEquality();

 print(equality.equals(map1, map2));
}
```

2. Immutable Maps

- ✓ Prevents accidental modification.
- ✓ Useful for constant values like config or static data.
- ✓ Declaring a map as const ensures it's frozen at compile time — great for safety and optimization.

```
void main() {
 const config = {
  'theme': 'dark',
  'fontSize': 16,
 };
 // config['theme'] = 'light'; //      Error: Cannot modify a const map
 print(config);
}
```

3.Copying Maps in Dart: Reference vs Shallow vs Deep Copy

1. Reference Copy

- A reference copy means both variables point to the same memory location. Changing one affects the other
- Simple and fast
- Not safe when you want to preserve the original data

```
void main() {
  var original = {'name': 'Alice', 'age': 25};
  var copy = original; // Reference copy

  copy['name'] = 'Bob';

  print(original); //       Changed
  print(copy);
}
```

2. Shallow Copy

- Creates a new top-level map, but nested objects (maps/lists) are still shared by reference
- Fails for nested maps/lists — they are still shared

```
void main() {
  var original = {'name': 'Alice', 'age': 25};
  var copy = Map<String, dynamic>.from(original); // Shallow copy

  copy['name'] = 'Bob';

  print(original); //       Unchanged
  print(copy);
}

void main() {
  var original = {
    'user': {'name': 'Alice'}
  };
  var copy = Map<String, dynamic>.from(original); // Still shallow!

  copy['user']['name'] = 'Bob';

  print(original); //       Changed due to shared inner map
}
```

3. Deep Copy

- Creates a completely new copy, including all nested structures. No shared references.
- Safest for complex or nested maps
- Ensures complete separation of data

```
import 'dart:convert';
void main() {
  var original = {
    'user': {'name': 'Alice', 'age': 25}
  };

  // Deep copy using JSON
  var deepCopy = jsonDecode(jsonEncode(original));

  deepCopy['user']['name'] = 'Bob';

  print(original); //      Unchanged
  print(deepCopy);
}
```

Summary:

- Maps are key-value stores that offer fast, organized access to data — essential for many real-world Dart and Flutter applications.

- Dart offers multiple ways to create and manipulate maps — from simple literals to typed maps, and advanced types like HashMap and SplayTreeMap.

- You can efficiently iterate, query, update, and transform maps using built-in methods like forEach, update, map(), and entries.

- Understanding null safety, copying (reference vs deep), and immutability is crucial when working with complex or nested map data.

- Dart Maps integrate seamlessly with JSON handling, making them ideal for working with web APIs in real apps.

- Finally, with tools like MapEquality and const maps, you can write safer, cleaner, and more performant Dart code.