Dart – Day10

Emp-id : 4781

1. Set

Set is a collection of unique, unordered elements. By default, Dart's Set is implemented using LinkedHashSet, so it preserves insertion order.

- → No duplicates allowed.
- → Can contain null.
- → Provides standard set operations: union, intersection, difference.
- → Inherits from Iterable, so all iterable methods are available.

Example:

```
void main()
{
  Set<int> nums = {1, 2, 3};
  nums.add(4);
  nums.remove(2);
  print(nums); // {1, 3, 4}
}
```

2. HashSet

HashSet is an implementation of Set based on hashing.

- → Fast lookup, addition, and removal: O(1) on average.
- → Unordered: does not preserve insertion order.

Example:

```
import 'dart:collection';

void main()
```

```
{
  HashSet<String> names = HashSet();
  names.add("Chandini");
  names.add("Sneha");
  names.add("Chandini"); // duplicate ignored
  print(names); // Order may vary
}
```

3. LinkedHashSet

LinkedHashSet is a hash-based set that preserves insertion order. This is actually the default Set in Dart.

- → Unique elements only.
- → Preserves order in which elements were added.
- → Slightly slower than HashSet due to order tracking.

Example:

```
import 'dart:collection';

void main()
{
  LinkedHashSet<int> nums = LinkedHashSet();
  nums.addAll([3, 1, 2]);
  print(nums); // {3, 1, 2} → preserves insertion order
}
```

4. SplayTreeSet

SplayTreeSet is a sorted set implemented as a self-balancing binary search tree. Elements are stored in sorted order automatically.

- → Unique elements only.
- → Always sorted.
- → Lookup, addition, and removal: O(log n).

Example:

```
import 'dart:collection';

void main()
{
  SplayTreeSet<int> nums = SplayTreeSet();
  nums.addAll([5, 1, 3]);
  print(nums); // {1, 3, 5} → automatically sorted
}
```

Differences Between Set, HashSet, LinkedHashSet, and SplayTreeSet

1. **Set vs HashSet**
    a. Set by default is a LinkedHashSet, preserves insertion order.
    b. HashSet does not preserve order, but is faster for lookups and modifications.
2. **Set vs LinkedHashSet**
    a. Default Set in Dart is already a LinkedHashSet.
    b. Both preserve insertion order.
    c. LinkedHashSet explicitly gives more control if you want to specify it.
3. **HashSet vs LinkedHashSet**
    a. HashSet is unordered → faster operations.
    b. LinkedHashSet preserves order → slightly slower.
4. **SplayTreeSet vs Others**
    a. Always sorted, unlike others.
    b. Slower than HashSet for add/remove (O(log n) instead of O(1)).
5. **All Sets**
    a. All enforce unique elements.
    b. All inherit from Set<E> / Iterable<E> so all iterable methods are available.

- ## Map

A Map in Dart is a collection of key-value pairs. Each key is unique, and values can be of any type. Maps are unordered by default (LinkedHashMap preserves insertion order).

Example:

```
void main()
{
  Map<String, int> scores = {"Alice": 90, "Bob": 85};
  print(scores); // {Alice: 90, Bob: 85}
}
```

- ## Empty Map

You can create an empty map using the literal {} or the Map constructor.

Example:

```
void main()
{
  var emptyMap1 = <String, int>{};
  var emptyMap2 = Map<String, int>();

  print(emptyMap1); // {}
  print(emptyMap2); // {}
}
```

- ## Map Creation by Literal

You can create a map using curly braces {} with key-value pairs.

Example:

```
void main()
{
  var fruits =
  {
    "apple": 3,
    "banana": 5,
    "mango": 2
  };
  print(fruits); // {apple: 3, banana: 5, mango: 2}
```

```
}
```

- ## Map Creation by Map Constructor

You can create a map using Map() and then add entries manually.

Example:

```
void main()
{
  var fruits = Map<String, int>();
  fruits["apple"] = 3;
  fruits["banana"] = 5;
  print(fruits); // {apple: 3, banana: 5}
}
```

- ## Map Creation by Map.from

Map.from() creates a new map from an existing map. Map.from gives runtime error for using different data types.

- Copies all key-value pairs.

Example:

```
void main()
{
  var original = {"a": 1, "b": 2};
  var copy = Map.from(original);
  print(copy); // {a: 1, b: 2}
}
```

- ## Map Creation by Map.of

Map.of() also creates a new map from another map, but allows type inference and is a bit safer in some situations. Map.of gives compiletime error for using different data types.

Example:

```
void main()
{
  var original = {"x": 10, "y": 20};
  var copy = Map.of(original);
  print(copy); // {x: 10, y: 20}
}
```

- ## Map Creation by Map.fromEntries

Map.fromEntries() creates a map from a list of MapEntry objects.

Example:

```
void main()
{
  var entries = [
    MapEntry("name", "Chandini"),
    MapEntry("age", 21)
  ];
  var map = Map.fromEntries(entries);
  print(map); // {name: Chandini, age: 21}
}
```

- ## Unmodifiable Map (Map.unmodifiable)

→ Created at runtime.
→ You cannot add, update, or remove keys/values.
→ But the original map (used to create it) can still be changed.
→ You cannot add, remove, or update entries once it's created.
→ If you try to modify it, Dart throws an UnsupportedError.

```
void main()
{
  var original = {"a": 1, "b": 2};
  var unmodifiableMap = Map.unmodifiable(original);

  print(unmodifiableMap); // {a: 1, b: 2}

  // Cannot modify unmodifiable map
  // unmodifiableMap["c"] = 3; // UnsupportedError

  // But changing the original will NOT affect the unmodifiable one
  original["a"] = 99;
  print(original);        // {a: 99, b: 2}
  print(unmodifiableMap); // {a: 1, b: 2} (unchanged)
}
```

- **Constant Map Literals(const {})**

→ Created at compile-time.
→ Must contain only compile-time constant values.
→ Cannot ever change, neither directly nor indirectly.
→ Must be declared with const.

```
void main()
{
  const constMap = {"x": 10, "y": 20};
  print(constMap); // {x: 10, y: 20}

  // Not allowed
  // constMap["z"] = 30; // Error: Unsupported operation
}
```