Dart – Day5

Emp-id : 4781

- **Set**

A Set in Dart is an unordered collection of unique elements (no duplicates allowed).

Example:

1. void main()

```
{
  Set<String> languages = {'js', 'go', 'dart', 'java'};
  print(languages);   // {js, dart, go, java}  - Unordered output
}
```

2. void main()

```
{
 var batches= {'batch1', 'batch2', 'batch1'};
 print(batches); // {batch1, batch2}
}
```

- **Set Creation Methods**

1. Literal {}
Creates a Set directly with curly braces.

```
void main()
{
 var fruits = {'Apple', 'Banana', 'Mango'};
 print(fruits); // {Apple, Banana, Mango}
}
```

2. Set() Constructor
Creates an empty growable Set.

```
void main()
```

```
{
  var numbers = Set<int>();
  numbers.add(10);
  numbers.add(20);
  print(numbers); // {10, 20}
}
```

3. Set.from()

Creates a Set from another iterable (removes duplicates). When you want to convert any iterable into a set quickly. You don't care too much about strict typing, just want unique elements.

```
void main()
{
  var list = [1, 2, 2, 3];
  var set1 = Set.from(list);
  print(set1); // {1, 2, 3}

  var mixed = [1, "two", 3];
  var set2 = Set.from(mixed);
  print(set2); // {1, two, 3}  (allows mixed types)
}
```

4. Set.of()

Creates a Set from another collection. When you want a set but also want type safety. Compiler enforces that all elements are of the same type.

```
void main()
{
  var list = <int>[1, 2, 3];
  var set1 = Set.of(list);
  print(set1); // {1, 2, 3}

  // var mixed = [1, "two", 3];   // Error: type mismatch
  // var set2 = Set.of(mixed);    // Won't compile
}
```

5. Empty Set with type

Creates an empty Set with a specific type.

```
void main()
{
  Set<String> names = {};
  names.add("Chandini");
  print(names); // {Chandini}
}
```

6. Set.unmodifiable()
Creates a read-only Set (cannot be modified later).

```
void main()
{
  var fixedSet = Set.unmodifiable([1, 2, 3]);
  print(fixedSet); // {1, 2, 3}
  fixedSet.add(4);  // Error
}
```

- ## Set Methods in Dart

1. Adding & Removing

add(element) → Adds one element.

```
var s = {1, 2}; s.add(3); print(s); // {1, 2, 3}
```

addAll(iterable) → Adds multiple elements.

```
var s = {1}; s.addAll([2, 3]); print(s); // {1, 2, 3}
```

remove(element) → Removes specific element if present.

```
var s = {1, 2, 3}; s.remove(2); print(s); // {1, 3}
```

removeAll(iterable) → Removes all given elements.

```
var s = {1, 2, 3, 4}; s.removeAll([2, 3]); print(s); // {1, 4}
```

removeWhere(condition) → Removes elements matching a condition.

var s = {1, 2, 3, 4}; s.removeWhere((n) => n.isEven); print(s); // {1, 3}

retainAll(iterable) → Keeps only matching elements.

var s = {1, 2, 3}; s.retainAll([2, 3]); print(s); // {2, 3}

retainWhere(condition) → Keeps only elements matching condition.

var s = {1, 2, 3, 4}; s.retainWhere((n) => n.isEven); print(s); // {2, 4}

clear() → Removes all elements.

var s = {1, 2}; s.clear(); print(s); // {}

2. **Checking Elements**

contains(element) → Checks if element exists.

var s = {1, 2, 3}; print(s.contains(2)); // true

containsAll(iterable) → Checks if all given elements exist.

var s = {1, 2, 3}; print(s.containsAll([1, 3])); // true

elementAt(index) → Returns element at position (order not guaranteed).

var s = {10, 20, 30}; print(s.elementAt(1)); // 20

3. **Set Operations**

union(otherSet) → Combines two sets.

var a = {1, 2}; var b = {2, 3}; print(a.union(b)); // {1, 2, 3}

intersection(otherSet) → Common elements of two sets.

var a = {1, 2}; var b = {2, 3}; print(a.intersection(b)); // {2}

difference(otherSet) → Elements in first set but not in second.

var a = {1, 2, 3}; var b = {2}; print(a.difference(b)); // {1, 3}

4. **Properties**

isEmpty → Returns true if set has no elements.

var s = <dynamic>{}; print(s.isEmpty); // true

isNotEmpty → Returns true if set has elements.

var s = {1}; print(s.isNotEmpty); // true

length → Number of elements.

var s = {1, 2, 3}; print(s.length); // 3

first → First element (no fixed order).

var s = {10, 20, 30}; print(s.first); // 10

last → Last element (no fixed order).

var s = {10, 20, 30}; print(s.last); // 30

single → Returns the element if only one exists.

var s = {99}; print(s.single); // 99

5. **Iteration & Transformation**

forEach(action) → Runs action for each element.

var s = {1, 2, 3}; s.forEach((n) => print(n));

map(transform) → Returns new iterable with transformed values.

var s = {1, 2, 3}; print(s.map((n) => n * 2)); // (2, 4, 6)

where(condition) → Filters elements by condition.

var s = {1, 2, 3, 4}; print(s.where((n) => n.isEven)); // (2, 4)

any(condition) → Returns true if any element matches.

var s = {1, 2, 3}; print(s.any((n) => n > 2)); // true

every(condition) → Returns true if all elements match.

var s = {2, 4, 6}; print(s.every((n) => n.isEven)); // true

toList() → Converts set to list.

var s = {1, 2, 3}; print(s.toList()); // [1, 2, 3]

toSet() → Returns a new set from elements.

var s = {1, 2, 3}; print(s.toSet()); // {1, 2, 3}

- ## Extension Method

In Dart, an extension method allows you to add new methods or properties to existing classes without modifying them. It's like adding the new functionality to the class without changing it.

Example :

1. Extension Method for User-Defined Class

Can add methods to your own class without modifying it.

Example:

```
class Student
{
  String name;
  int age;

  Student(this.name, this.age);
}

// Extension method on Student
extension StudentExtension on Student {
  String get details => "$name is $age years old";
  void birthday() => age += 1;
}

void main()
{
  var s = Student("Chandini", 22);

  print(s.details); // Chandini is 22 years old
  s.birthday();
  print(s.details); // Chandini is 23 years old
}
```

Adds getter and method to a user-defined class.

2. Extension Method for Built-In Class

Can add methods or getters to built-in classes like String, int, etc.

Example:

```
extension NumExtension on int
{
  bool get isEvenNumber => this % 2 == 0; // getter added to int
  int square() => this * this;           // method added to int
}
```

```
void main()
{
  print(4.isEvenNumber); // true
  print(7.isEvenNumber); // false

  print(5.square());     // 25
  print(10.square());    // 100
}
```

Adds custom behavior to a built-in class.

- ## Merits of Extension Methods

1. **Add functionality without modifying class** – Works on built-in or third-party classes.
2. **Cleaner code** – Lets you call custom methods like they are part of the original class.
3. **Encapsulation** – Can keep logic related to a type in one place.
4. **Reusability** – Once created, the extension can be used across the project.

Example:

```
extension IntExtension on int
{
  bool get isEvenNumber => this % 2 == 0;
}
void main()
{
  print(4.isEvenNumber); // true
}
```

- ## Demerits of Extension Methods

1. **Name conflicts** – If a method with the same name exists in the class or another extension, it can cause ambiguity.
2. **Cannot access private members** – Extensions can only access public members of the class.
3. **Overuse can reduce readability** – Too many extensions can make code harder to follow.

4. **Not polymorphic** – Extensions don't override existing methods; they only add new ones.

Example (name conflict):

```
class A
{
  void show() => print("Class A");
}

extension AExtension on A
{
  void show() => print("Extension");
}

void main()
{
  var a = A();
  a.show();   // prints "Class A" (extension does NOT override)
}
```