Objectives

- Working with String Lists and Maps
- Comparison and Logical Operators
- Conditional Statements
- Working with Loops

Working with String Lists and Maps

Strings

We use strings to store text values like names, messages, or any kind of written data. Strings are created by wrapping text in either double quotes (`"`) or single quotes (`'`):

```
var name = "Alice";
var greeting = 'Hello, world!';
```

Dart provides many built-in methods to work with and manipulate strings easily.

Accessing Characters in a String

In Dart, we can access individual characters in a string using index numbers. String indices start at 0:

```
var word = "Dart";
print(word[0]);  // Output: D
print(word[1]);  // Output: a
print(word[3]);  // Output: t
```

If we try to access an index that's out of bounds, Dart throws a `RangeError`:

```
var word = "Dart";
print(word[10]); // This will throw an error
```

Accessing Substrings

We can get substrings using the `.substring()` method:

```
`var word = "Programming";
print(word.substring(0, 5)); // Output: Progr (from index 0 to 4)`
```

String Methods

`.length`

Returns the number of characters in the string:

```
var name = "Alice";
print(name.length); // Output: 5
```

`.toUpperCase()` and `.toLowerCase()`

Convert a string to all uppercase or lowercase:

```
print("hello".toUpperCase()); // Output: HELLO
print("WORLD".toLowerCase()); // Output: world
```

`.trim()`

Removes extra whitespace from the beginning and end:

```
var input = "   hello   "; print(input.trim()); // Output: hello
```

`.contains()`

Checks if a string contains a certain substring:

```
var sentence = "Dart is fun";
print(sentence.contains("fun"));    // Output: true
print(sentence.contains("boring")); // Output: false
```

`+` or interpolation

Adds two strings together:

```
var greeting = "Hello, " + "Alice";
print(greeting); // Output: Hello, Alice
```

We can also use interpolation:

```
var name = "Bob";
print("Hello, $name"); // Output: Hello, Bob
```

`.replaceAll()`

Replaces parts of a string:

```
var text = "I like Java";
print(text.replaceAll("Java", "Dart")); // Output: I like Dart
```

`.split()`

Splits a string into a list, based on a character or space:

```
var words = "apple,banana,orange";
print(words.split(",")); // Output: [apple, banana, orange]
```

Lists

We use lists to store ordered collections of values. These values can be of the same type or different types strings, numbers, booleans, or even other lists. Lists are written with square brackets `[]`, and values are separated by commas:

```
var colors = ["red", "green", "blue"];
var mixed = ["hello", 42, true];
```

We can also create a list explicitly:

```
List<int> numbers = [1, 2, 3, 4, 5];
```

Accessing Elements in a List

Just like strings, we access list elements using index numbers starting from 0:

```
var colors = ["red", "green", "blue"];
print(colors[0]); // Output: red
```

```
print(colors[1]); // Output: green
print(colors[2]); // Output: blue`
```

Accessing an index that doesn't exist throws a `RangeError`:

```
print(colors[10]); // This will throw an error
```

Accessing Sublists

We use `.sublist(start, end)` to extract a portion of the list:

```
var numbers = [10, 20, 30, 40, 50];
print(numbers.sublist(1, 4)); // Output: [20, 30, 40]
print(numbers.sublist(0, 2)); // Output: [10, 20]
```

List Methods

`.length`

Returns the number of elements:

```
var colors = ["red", "green", "blue"];
print(colors.length); // Output: 3
```

`.add(value)`

Adds a value at the end:

```
var colors = ["red", "green", "blue"];
colors.add("yellow");
print(colors); // Output: ["red", "green", "blue", yellow]
```

`.insert(index, value)`

Inserts a value at the given index:

```
colors.insert(2, "black");
print(colors); // Output: [red, green, black, blue]
```

`.removeLast()`

Removes and returns the last element:

```
var lastColor = colors.removeLast();
print(lastColor); // Output: blue
```

`.remove(value)`

Removes the first occurrence of a value:

```
var arr = [1, 2, 3, 2, 4];
arr.remove(2);
print(arr); // Output: [1, 3, 2, 4]
```

`.removeAt(index)`

Removes the element at the given index:

```
var arr = [1, 2, 3, 4];
arr.removeAt(1);
print(arr); // Output: [1, 3, 4]
```

`.toSet().toList()`

Removes duplicates:

```
var arr = [1, 2, 2, 3];
arr = arr.toSet().toList();
print(arr); // Output: [1, 2, 3]
```

`.contains(value)`

Checks if the list contains a value:

```
var colors = ["red", "green", "blue"];
print(colors.contains("green")); // Output: true
```

`.reversed.toList()`

Reverses the list:

```
var colors = ["red", "green", "blue"];
print(colors.reversed.toList()); // Output: [blue, green, red]
```

`.sort()`

Sorts the list (only works for comparable elements):

```
var numbers = [3, 1, 5, 2];
numbers.sort();
print(numbers); // Output: [1, 2, 3, 5]
```

`.join()`

Joins elements into a single string:

```
var colors = ["red", "green", "blue"];
print(colors.join(", ")); // Output: red, green, blue
```

Sets

A **set** is an unordered collection of unique values. Sets are useful when we want to ensure that no duplicates are stored. In Dart, sets are written using curly braces `{}` or created with the `Set` constructor.

```
var colors = {"red", "green", "blue"};
var numbers = {1, 2, 3, 4, 5};
```

We can also create a set explicitly:

```
Set<String> names = {"Alice", "Bob", "Charlie"};
```

Adding Elements

We use `.add()` to add a single element. Duplicates are ignored:

```
var colors = {"red", "green", "blue"};
colors.add("yellow");
```

```
print(colors); // Output: {red, green, blue, yellow}
colors.add("red"); // Already exists
print(colors);     // Output: {red, green, blue, yellow}`
```

Removing Elements

We use `.remove(value)` to remove a specific element:

```
var colors = {"red", "green", "blue"};
colors.remove("green");
print(colors); // Output: {red, blue, yellow}
```

Set Methods

`.length`

Returns the number of elements:

```
var colors = {"red", "green", "blue"};
print(colors.length); // Output: 3
```

`.contains(value)`

Checks if a value is in the set:

```
var colors = {"red", "green", "blue"};
print(colors.contains("blue"));  // Output: true
print(colors.contains("orange")); // Output: false
```

`.clear()`

Removes all elements:

```
var colors = {"red", "green", "blue"};
colors.clear();
print(colors); // Output: {}
```

Set Operations

Dart supports mathematical set operations using methods like union, intersection, and difference.

Union

Combines two sets, removing duplicates:

```
var a = {1, 2, 3};
var b = {3, 4, 5};
print(a.union(b)); // Output: {1, 2, 3, 4, 5}
```

Intersection

Finds common elements:

```
var a = {1, 2, 3};
var b = {3, 4, 5};
print(a.intersection(b)); // Output: {3}
```

Difference

Finds elements in one set that aren’t in the other:

```
var a = {1, 2, 3};
var b = {3, 4, 5};
print(a.difference(b)); // Output: {1, 2} print(b.difference(a)); // Output: {4, 5}
```

Converting Between Lists and Sets

We can convert a list to a set to remove duplicates:

```
var list = [1, 2, 2, 3];
var uniqueSet = list.toSet();
print(uniqueSet); // Output: {1, 2, 3}
```

And convert a set back to a list:

```
var uniqueList = uniqueSet.toList();
print(uniqueList); // Output: [1, 2, 3]
```

Maps

Maps are collections of key-value pairs. Each key must be unique, and every key maps to a specific value. In Dart, maps are written using curly braces `{}` with keys and values separated by colons.

```
var person = {   "name": "Alice",   "age": 30,   "isStudent": false };
```

We can also define the type explicitly:

```
Map<String, String> capitals = {   "France": "Paris",   "Japan": "Tokyo" };
```

Accessing Values

We se the key in square brackets to get a value:

```
var person = {   "name": "Alice",   "age": 30,   "isStudent": false };
print(person["name"]); // Output: Alice
print(person["age"]);  // Output: 30`
```

Accessing a non-existent key returns `null`:

```
var person = {   "name": "Alice",   "age": 30,   "isStudent": false };
print(person["height"]); // Output: null
```

Adding and Updating Values

We can add a new key-value pair or update an existing one:

```
person["email"] = "alice@example.com"; // Add
person["age"] = 31;                    // Update
```

We can also use .addAll() to add multiple entries:

```
person.addAll({"city": "London"});
```

Removing Entries

We use `.remove(key)` to delete a key-value pair:

```
var person = {   "name": "Alice",   "age": 30,   "isStudent": false };
person.remove("isStudent");
print(person); // Output: {name: Alice, age: 31, city: London}
```

Map Methods

`.keys` and `.values`

Returns all keys and values:

```
var person = {   "name": "Alice",   "age": 31 };
print(person.keys);   // Output: (name, age)
print(person.values); // Output: (Alice, 31)
```

`.length`

Returns the number of key-value pairs:

```
print(person.length); // Output: 3
```

`.containsKey(key)`

Checks if the map contains a specific key:

```
var person = {   "name": "Alice",   "age": 31 };
print(person.containsKey("name"));  // Output: true
print(person.containsKey("salary")); // Output: false
```

`.containsValue(value)`

Checks if the map contains a specific value:

```
var person = {   "name": "Alice",   "age": 31, "city": "London" };
print(person.containsValue("London")); // Output: true
```

`.isEmpty` and `.isNotEmpty`

Check if the map is empty:

```
var person = {   "name": "Alice",   "age": 31, "city": "London" };
print(person.isEmpty);    // Output: false
print(person.isNotEmpty); // Output: true
```

`.clear()`

Removes all entries from the map:

```
var person = {   "name": "Alice",   "age": 31, "city": "London" };
person.clear();
print(person); // Output: {}
```

Merging Maps

We use the spread operator or `.addAll()` to combine maps:

```
var defaults = {"font": "Arial", "size": 12};
var settings = {"color": "red"};
var merged = {...defaults, ...settings}; // => {font: Arial, size: 14, color : red}
```

To update the original map we use `addAll`:

```
defaults.addAll(settings);
```

Comparison and Logical Operators

Comparison operators help us compare values or variables with each other and create conditions based on the results. These comparisons always return a boolean value either `true` or `false`.
Dart provides us with the following comparison operators:

- `>` (**Greater than**): Checks if the first operand is greater than the second.

```
5 > 4   # => true
3 > 10  # => false
```

- `<` (**Less than**): Checks if the first operand is less than the second.

```
4 < 5   # => true
7 < 3   # => false
```

- `==` (**Equal to**): Checks if both operands are equal.

```
4 == 4   # => true
"hi" == "hello"  # => false
```

- `!=` (**Not equal to**): Checks if the operands are not equal.

```
1 != 0   # => true
5 != 5   # => false
```

- `>=` (**Greater than or equal to**): Checks if the first operand is greater than or equal to the second.

```
2 >= 2   # => true
3 >= 5   # => false
```

- `<=` (**Less than or equal to**): Checks if the first operand is less than or equal to the second.

```
2 <= 3   # => true
4 <= 1   # => false
```

Logical Operators

Dart provides logical operators to combine multiple conditions and build more complex expressions. The result of a logical operation is always a boolean value either true or false.

`||` (OR)

- Returns `true` if **at least one** of the conditions is true.
- Returns `false` only if **all** the conditions are false.

```
print(true || false)   # => true
print(false || false)  # => false
```

`&&` (AND)

- Returns `true` only if **all** the conditions are true.
- Returns `false` if **any** of the conditions is false.

```
print(true && true)    # => true
print(true && false)   # => false
```

`!` (NOT)

- Reverses the logical state of the condition.
- If the condition is `true`, `!` makes it `false`, and vice versa.

```
print(!true)   # => false
print(!false)  # => true
```

Truth Table for Logical Operators:

A	B	A && B	A | B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Conditional Statements

Conditional statements are fundamental programming constructs that enable our code to make decisions and execute different actions based on specific conditions. They control program flow by evaluating whether certain criteria are met, allowing for dynamic behavior that responds to different situations, inputs, or values.

`if`, `else`, and `else if` Statement

Single Condition with `if`

When we need to execute code only when a condition is true:

```
if (condition){
    // Code executes only if condition evaluates to true
```

```
}
```

Example:

```
void main(){
var age = 18;
if (age >= 18){
    print("You are an adult.");
}
}
```

Alternative Path with `if-else`

When we want to handle both true and false cases differently:

```
if (condition){
  // Code for true case
}else{
  // Code for false case
}
```

Example

```
void main(){
var is_raining = false;
if (is_raining){
  print("Bring an umbrella.");
}else{
  print("No umbrella needed today.");
}
}
```

Multiple Conditions with `else if`

When we need to check several possible conditions in sequence:

```
if (condition1){
  // First case
}else if (condition2){
  // Second case
}else{
  // Default case
}
```

Example:

```
void main(){
var core = 85;
if (score >= 90){
  print("Excellent performance");
}else if (score >= 80){
  print("Good job");
}else
  print("Room for improvement");
}
}
```

The `switch`, `case` Statement

The `switch` statement in Dart helps us execute different code blocks based on the value of a single variable. It's a cleaner alternative to writing multiple `if-else` statements and makes our code more readable and organized, especially when we are comparing the same value against multiple options.

```
switch (variable) {
  case value1:
    // Code runs if variable == value1
    break;
  case value2:
    // Code runs if variable == value2
    break;
  default:
    // Code runs if no case matches
}
```

Example:

```
void main(){
String day = "Monday";

switch (day) {
  case "Monday":
    print("Start of the week");
    break;
  case "Friday":
    print("Almost the weekend!");
    break;
  case "Saturday":
  case "Sunday":
    print("It’s the weekend!");
    break;
  default:
    print("Just another weekday");
}
}
```

If we want to combine multiple cases in Dart that is, have them all run the same block of code we can list them one after another without using `break` until the shared block is written.

```
switch (variable) {
  case value1:
  case value2:
  case value3:
    // Shared code for value1, value2, and value3
    break;
}
```

Example

```
void main(){
String day = "Sunday";

switch (day) {
  case "Saturday":
  case "Sunday":
```

```
    print("It’s the weekend!");
    break;
  default:
    print("It’s a weekday.");
}
}
```

Loops

Loops allow us to repeat a block of code multiple times either a set number of times or until a certain condition is met. This helps us avoid repetition and makes our programs more efficient.

`while` Loop

We use a `while` loop when we want to repeat something as long as a condition is true.

```
void main(){
int i = 0;

while (i < 3) {
  print('i is $i');
  i++;
}
}
```

This loop keeps running while `i` is less than 3.

`do-while` Loop

A `do-while` loop is like a `while` loop, but it **runs at least once**, because the condition is checked **after** the first run.

```
void main(){
int i = 0;

do {
  print('i is $i');
  i++;
} while (i < 2);
}
```

`for` Loop

We use a `for` loop when we know how many times we want to repeat something.

```
void main(){
for (int i = 0; i < 5; i++) {
  print('Count: $i');
}
}
```

This prints numbers from 0 to 4.
`i++` increases the counter each time.

`for-in` Loop

We use a `for-in` loop to go through each element in a collection like a list.

```
void main(){
List<String> fruits = ['apple', 'banana', 'cherry'];

for (var fruit in fruits) {
  print(fruit);
}
}
```

This prints each fruit in the list.

`break` and `continue`

- `break` : exits the loop completely when condition is met.
- `continue` : skips to the next iteration when condition is met

Example 1

```
void main(){
for (int i = 0; i < 5; i++) {
  if (i == 3) break;
  print(i);  // Stops printing at 3
```

```
}
}
```

Example 2

```
void main(){
for (int i = 0; i < 5; i++) {
  if (i == 2) continue;
  print(i);  // Skips printing 2
}
}
```

Tasks

Task1

- Create string variable with name `word` and value `"Developer"`
  - Print the first and last characters using indexing`
  - Convert to uppercase and print`
- Create string variable named `phrase` with this text `"I love Python"`
  - Replace "Python" with "Dart" and print the result
- Create string variable named `sentence` with this text `"Learning Dart is fun"`
  - Print true if "Dart" is in the sentence
  - Print the number of characters in the sentence`

Task 2

- Create an list of colors: `["red", "green", "blue"]`
  - print last and first element
  - Add "yellow" to the end of the list
  - Insert "black" at the beginning
  - Remove and print the last element
  - Check if "green" exists in the list
- Create a new list with only unique values from `[1, 2, 2, 3, 4, 4, 5]`
  - Remove the second item
  - Insert 25 at index 2`
- Sort the list `[5, 3, 8, 1]` in ascending order

Task 3

- Create a map representing a person with name, age, and city
  - Print the name
  - Add a new key-value pair for occupation
  - Print all the keys of the map
  - Check if the map contains a key :age
  - Merge with another map `{ "hobbies": ["reading", "coding"] }`
  - Delete the city information from the map

Task 4

- Write an if-else statement that checks if a number is even or odd
- Create a grading system using else if
- Write a program that recommends clothing based on temperature:
  - Above 30°C: "Wear shorts"
  - 20-30°C: "T-shirt weather"
  - Below 20°C: "Bring a jacket"

Class interval	Letter grade	Grade point	Remarks
80-100	A+	4.00	First Class
75-79	A	3.75	First Class
70-74	A−	3.50	First Class
65-69	B+	3.25	First Class
60-64	B	3.00	First Class
55-59	B−	2.75	Second Class
50-54	C+	2.50	Second Class
45-49	C	2.25	Second Class Upper
40-44	D	2.00	Third Class
0-39	F	0.00	Fail

Task5

Write a C program that reads three numbers, **a**, **b**, and **c**, from the user, and then solves the quadratic equation:

$$a^2 + bx + c = 0$$

The program should handle the following cases:

1. When the discriminant b2−4ac is positive, there are two real and distinct roots.
2. When the discriminant is zero, there is one real root .
3. When the discriminant is negative, the equation has no real solutions.

Task 6

- Create a program that asks the user to enter a number and calculates its factorial using a `while` loop.
- Ask the user for two numbers and display all prime numbers between them using a `for` loop.