Objectives

- Working with Functions
- Working with Library
- Working with Packages

Functions

Introduction

A function is a reusable block of code designed to perform a specific task. Functions are fundamental to writing efficient and well-structured programs. By following the DRY (Don't Repeat Yourself) principle, functions help eliminate code duplication and promote cleaner code. They also make it easier to break down complex problems into smaller, manageable parts, resulting in code that is more organized, easier to understand, and simpler to maintain.

Creating Functions

We create function by following those steps:

1. Identify the code to reuse.
2. Choose a function name.
3. Decide what parameters it needs.
4. Define what it should return.

```
sayHello() {
  print("Hello, user!");
}

void main() {
  sayHello();
}
```

We can explicitly specify the return type of a function. If we don’t, it defaults to `dynamic`. If the function returns nothing, we use `void`.

```
void sayHello() {
  print("Hello, user!");
}
```

```
void main() {
  sayHello();
}
```

Function with Parameters

We can make a function accept arguments by adding parameters in parentheses after the method name in the definition. Parameters act as placeholders for the values (arguments) we pass when calling the method.
We can also return a value from the function using the `return` keyword, followed by the value or variable we want to return.

```
void main() {

  print(add_two_numbers(4, 5));
}
int add_two_numbers(num1, num2){
    int result = num1 + num2;
    return result;
}
```

We can explicitly specify the types of the parameters that a function takes. This ensures the function only accepts arguments of the specified types. If we don't, the parameter types default to `dynamic`

```
void main() {

  print(add_two_numbers(4, 5));
}
int add_two_numbers(int num1, int num2){
    int result = num1 + num2;
    return result;
}
```

Default Parameters

We can make parameters optional by assigning them default values. If no argument is provided when calling the function, the default value will be used instead. Dart provides two ways to define optional parameters: optional positional and optional named.

Optional Positional Parameters `[]`

Use square brackets to define parameters that are optional and rely on their position in the argument list.

```
void sayHello([String name = "user"]) {
  print("Hello $name");
}

void main() {
  sayHello();             // Output: Hello user
  sayHello("Mohamed");    // Output: Hello Mohamed
}
```

We must pass arguments in order. We can't skip a parameter in the middle.

Optional Named Parameters `{}`

Use curly braces to define **named parameters**. This allows arguments to be passed by name, in any order. We can also assign default values to make them optional.

```
void sayHello({String name = "user"}) {
  print("Hello $name");
}

void main() {
  sayHello();                  // Output: Hello user
  sayHello(name: "Mohamed");   // Output: Hello Mohamed
}
```

Named parameters are better because they are more readable and flexible, especially when our function has many parameters or when we want to skip some.

Variable Scope

Scope defines the region within a program where a particular variable or function can be accessed and used. Dart mainly distinguishes between global scope and local scope.

Global scope

Global scope refers to the outermost level of a Dart file. Any variable, constant, or function defined outside of any class, function, or block is considered to be in the global scope. These elements can be accessed from anywhere in the file, including inside functions or classes.

```
String globalMessage = "This is global";

void printMessage() {
  print(globalMessage); // Accessible here
}
```

Local scope

Local scope limits the visibility of a variable or constant to the specific block or function where it is declared. Variables defined inside functions, loops, or conditionals are local variables, and they are not accessible outside of that block.

```
void printSum(int a, int b) {
  int result = a + b; // Local variable
  print(result);
}

void main() {
  printSum(3, 5);
  // print(result); // Error: 'result' is not defined in this scope
}
```

Arbitrary Number of Arguments

Unlike some other languages, Dart doesn't support variadic functions in the traditional sense. However, we can achieve similar behavior by using a `List` to accept an arbitrary number of positional arguments, or a `Map` to accept an arbitrary number of key-value arguments.
Example 1

```
void main(){
sayHelloToAll(['Mohamed', 'Ali', 'Ahmed']);
}
void sayHelloToAll(List<String> names) {
  for (var name in names) {
    print('Hello $name');
```

```
  }
}
```

Example 2

```
void main(){
greetPeople({
  'name1': 'Mohamed',
  'name2': 'Ahmed',
  'name3': 'Ali'
});
}
void greetPeople(Map<String, String> names) {
  names.forEach((key, value) => print('Hello $value'));
}
```

Recursive Functions

Recursive functions are special function that have ability to call theirself untill a condition (that we call base state) is valid.
let's suppose we want to create a function that calculates the factorial of numbers
we know that:

- 0! is equal to 1
- 1! is equal to 1 = 1*0!
- 2! is equal to 2*1 = 2*1!
- 3! is equal to 3*2*1 = 3*2!
- 4! is equal to 4*3*2*1 = 4*3!
- 5! is equal to 5*4*3*2*1 = 5*4!

with that in mind, we can set the base condition as if n == 0 we return 1,else we return n multiplied by the factorial of n-1 and so on

```
void main(){
 print(factorial(5));

}
int factorial(int n){
    if (n == 0) return 1;
```

```
    return n * factorial(n - 1);
}
```

Anonymous Functions (Lambdas)

Anonymous functions, often referred to as lambdas, are functions without a name. They're useful when you need short, one-time-use functionality without creating a full function definition. In Dart, anonymous functions can be assigned to variables, passed as arguments, or stored in collections.
We can create lambda functions using the arrow (`=>`) notation. This syntax is useful for short functions that return a single expression.

```
 void main() {
  var square = (int n) => n * n;
  print(square(4)); // Output: 16
}
```

Passing Functions as Arguments

We can pass functions as arguments just like variables. This allows us to reuse behavior dynamically.

```
void greet(String name) {
  print("Hello, $name!");
}

void runCallback(Function(String) callback, String value) {
  callback(value);
}

void main() {
  runCallback(greet, "Dart");
  runCallback((String name)=>print("Hello, $name"),"Ali");
}
```

To make a function accept another function as a parameter, we should specify the parameter type as `Function(type1, ...)` , where the parentheses contain the types of arguments the passed function takes.
We can pass a function as an argument, or use a lambda directly instead.

Returning Functions

We can return a function from another function, just like returning a variable. This is useful for selecting behavior dynamically.

```
int add(int a, int b) => a + b;

int multiply(int a, int b) => a * b;

Function(int, int) getOperation(String type) {
  if (type == "add") {
    return add;
  } else if (type == "multiply") {
    return multiply;
  } else {
    throw Exception("Unknown operation");
  }
}

void main() {
  var operation = getOperation("multiply");
  print(operation(3, 4));
}
```

To return a function from another function, we should set the return type to `Function(type1,...)` with the appropriate parameter types specified inside the parentheses.

Functional Programming Concepts

Functional programming breaks problems into smaller sub-problems, each solved by pure functions.
Functional programming cover the following 5 concepts

Pure functions :

These functions respect the following :

- They always produce the same output for same arguments irrespective of anything else.
- They have no side-effects i.e. they do not modify any arguments or local/global variables or input/output.

- They have immutability. The pure function's only result is the value it returns. They are deterministic.

Recursive function :

Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves until they reach the base case.

First-Class functions and Higher-Order function :

First-class functions are dealt with as first-class variable. The first class variables can be passed to functions as parameter, can be returned from functions or stored in data structures. Higher order functions are the functions that take other functions as arguments and they can also return functions.

Referential transparency :

In functional programs variables, once defined don't change their value throughout the program. Functional programs do not have assignment statements. If we have to store some value, we define new variables instead. This eliminates any chances of side effects because any variable can be replaced with its actual value at any point of execution. State of any variable is constant at any instant.

Variables are Immutable :

In functional programming, we can't modify a variable after it's been initialized. We can create new variables but we can't modify existing variables, and this really helps to maintain state throughout the runtime of a program. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.

Libraries

Introduction

In Dart, libraries are used to organize and reuse code by grouping related functions, classes, variables, and constants. Libraries help maintain clean, modular, and scalable codebases, and also prevent name conflicts through the use of namespaces.

Built-in Libraries

Dart includes a wide range of core libraries that come with the SDK. These libraries provide useful features like math operations, date and time handling, collections, I/O, and more.
We can use built-in libraries by importing them with the `import` keyword.

```
import 'dart:math';

void main() {
  print(sqrt(25));
}
```

Creating Custom Libraries

To create our own Dart library, we start by defining a `.dart` file (e.g., `my_library.dart`) and placing our functions, classes, or variables inside it. We can then import this file anywhere in our project using `import 'my_library.dart';`.
We can explicitly declare our library at the top of the file for better documentation and tooling support:

```
library my_library;

String greet(String name) => 'Hello, $name!';
```

When implementing a large library with a significant amount of code, we can split it across multiple files using Dart's `part` and `part of` directives.

Main Library File

We begin by creating the main library file and explicitly specifying the parts it contains.

```
library my_library;
part 'helpers.dart';
part 'calculations.dart';

const String libraryName = 'MyLibrary';
```

Part Files

Each part file should include the `part of` keyword followed by the library name.

```
part of my_library;

String greet(String name) => 'Welcome to $libraryName, $name!';
```

Using `export`

We can use the `export` keyword to create a library that combines multiple libraries into one. Instead of using `part` and `part of`, which are meant for splitting a single library into multiple files, `export` allows us to reuse and re-expose entire libraries, making it ideal for building package interfaces or modular designs

Create individual feature libraries:

We start by creating a small library that groups together related features and functions, making them easier to manage and reuse.
math_ops

```
library math_ops;

int add(int a, int b) => a + b;
int subtract(int a, int b) => a - b;
```

string_ops

```
library string_ops;

String capitalize(String text) => text.toUpperCase();
String reverse(String text) => text.split('').reversed.join();
```

Create a main library that exports them:

Then, we create a main library that exports and combines all these smaller libraries, providing a single entry point for accessing their features.

```
library my_library;

export 'math_operations.dart';
export 'string_operations.dart';
export 'src/advanced_calculations.dart' show AdvancedCalculator;
```

Usage in other files:

Finally, we can use our libraries by simply importing the main library, which gives access to all the combined features

```
import 'my_library.dart';

void main() {
  print(add(2, 3)); // 5
  print(capitalize('hello')); // HELLO
  final calc = AdvancedCalculator();
}
```

Controlling Visibility with `show` and `hide`

When using `export` or `import`, Dart allows fine-grained control over what gets exposed by using the `show` and `hide` keywords.

- `show` is used to include only specific elements.
- `hide` is used to exclude specific elements while including everything else.

We can also use the `as` keyword to assign an alias to a library when importing it, making it easier to reference and avoid naming conflicts.

```
import 'math_utils.dart' as math;
import 'string_utils.dart' as string;

void main() {
  print(math.calculate(2, 3));
  print(string.calculate("2", "3"));
}
```

Packages

Introduction

In Dart, **packages** are collections of libraries, code, resources, and assets bundled together to provide reusable functionality. Packages help organize large projects, share code between apps, and distribute solutions across different teams or publicly via repositories like pub.dev.

Built-in Packages

Dart provides a set of core packages that come bundled with the SDK, offering essential features like asynchronous programming, HTTP servers, I/O, and more.
We can use built-in packages by importing them with the `import` keyword.
Example using the `http` package from Dart's SDK:

```
import 'dart:convert';
import 'dart:io';

void main() async {
  var url = Uri.parse('https://example.com');
  var response = await HttpClient().getUrl(url);
  print('Request sent to $url');
}
```

Installing Third-Party Packages

Dart has a central repository called pub.dev where developers can find thousands of third-party packages.
To use a third-party package in a project:

1. Open the `pubspec.yaml` file.
2. Under the `dependencies:` section, add the package name and version.
3. Run `dart pub get` or use your IDE's built-in tools to fetch the package.
   Example: To add the `http` package for making network requests:
   pubspec.yaml

```
dependencies:
  http: ^1.2.0
```

Then run

```
dart pub get
```

Now you can import and use the package in your Dart files:

```
import 'package:http/http.dart' as http;

void main() async {
  var response = await http.get(Uri.parse('https://example.com'));
  print('Response status: ${response.statusCode}');
}
```

Dart Project

We can see that to use third-party packages, we need a `pubspec.yaml` file. To generate this file, we must create a Dart project instead of working with standalone files.
We can create a Dart project using the following command:

```
dart create project_name
```

Creating Custom Packages

Sometimes, we may want to create our own packages either to reuse code across multiple projects or to share them with the community.

Steps to create a Dart package:

1. We start by creating a new project:

```
`dart create -t package-simple my_package`
```

This generates a basic structure:

```
my_package/
├── lib/
│   └── my_package.dart
├── pubspec.yaml
└── test/
```

2. We write our library code inside the `lib/` directory.
   Example `lib/my_package.dart`:

```
library my_package;

String greet(String name) => 'Hello, $name!';
```

3. We define the package details inside `pubspec.yaml`:

```
name: my_package
description: A simple greeting package
version: 0.0.1
environment:
  sdk: '>=3.0.0 <4.0.0'
```

4. To use our custom package in a project, we reference it in the `pubspec.yaml` file:

```
dependencies:
  my_package:
    path: ../my_package
```

Then we import and use it:

```
import 'package:my_package/my_package.dart';

void main() {
  print(greet('World'));
}
```

If we want to share our package publicly, we can publish it to pub.dev.
First, we validate the package and then run:

```
dart pub publish
```

Tasks

Task 1

Create a Dart function that tests whether a given number is a prime number or not.

Task 2

Write a Dart function that converts a decimal number to its binary representation using recursion.

Task 3

Create a library that contains the following functions:

- **Function** to find the largest element in an list.
- **Function** to find the smallest element.
- **Function** to calculate the average of all the elements in the list.

Then, create a program file that prompts the user to enter the number of elements in the List, then enter the elements of the list and uses the previously defined functions to find and display the largest element, smallest element, and the average of the list elements.