Objectives :

- Learn About How Computers Work
- Intoduction to Programming Languages
- Intoduction to Dart
- User Input and Output

How Computers Work

We all have computers, and we use them for a variety of purposes, including watching videos, playing games, performing mathematical calculations, communicating with friends, and many other applications. But the fundamental question is: how do these devices actually work?

The answer lies in their electrical nature. Computers are essentially electrical devices that perform all calculations using electrical signals. The central processing unit (CPU), often referred to as the "brain" of the computer, executes these calculations. To temporarily store data while the computer is in operation, it relies on memory, specifically Random Access Memory (RAM).

Crucially, all information within a computer is represented by electrical signals. This includes data stored in memory and the instructions that the CPU executes. These electrical signals exist in one of two distinct states: presence (typically represented by the digit '1') or absence (represented by '0').

With this binary representation in mind, we can understand that CPU instructions are essentially sequences of 1s and 0s. This sequence of binary digits is known as machine code, which is the most fundamental level of programming language that the CPU can directly understand.

Introduction to Programming Languages

Programming languages are tools that were developed to facilitate communication with computers. Instead of writing instructions directly in binary code, which can become incredibly cumbersome for large programs, we can use programming languages with their more user-friendly syntax. This simplifies the coding process, making it easier to read, understand, and debug code.

However, computers cannot directly understand the syntax of these high-level programming languages. To bridge this gap, we use a program called a compiler. The compiler first checks the code for any errors and then translates it into machine code, a low-level language consisting of binary instructions that the computer can execute.

Intoduction to Dart

Introduction

Dart is a modern, object-oriented programming language developed by Google, primarily used for building fast and scalable applications across web, mobile, desktop, and server platforms. It is known for its clean and easy-to-read syntax, which is familiar to developers coming from languages like Java, JavaScript, or C#. Dart is the language behind Flutter, a popular UI toolkit for building natively compiled applications from a single codebase.

Dart source code is typically written in files that end with the `.dart` extension. These files contain the logic and structure of Dart programs, which can range from simple scripts to complex applications.

One of Dart's strengths is its flexible compilation model. Dart code can be run directly using Just-In-Time (JIT) compilation, which is useful during development for faster iterations. This is done using the `dart run` command. For production, Dart offers Ahead-Of-Time (AOT) compilation, which compiles Dart code into optimized machine code. This results in faster startup times and better performance. Dart also supports compiling to JavaScript using the `dart compile js` command, making it suitable for web development. Other compilation targets include executables (`exe`), snapshots (`aot-snapshot` , `jit-snapshot`), and kernel intermediate formats.

Dart SDK

The Dart SDK (Software Development Kit) is a collection of tools and libraries that allow us to write, compile, and run Dart applications. An SDK in general provides everything necessary for developing in a specific language, and in the case of Dart, it includes the Dart VM (virtual machine), core libraries, a package manager (`pub`), and command-line tools such as `dart` and `dartfmt` .

The main role of the Dart SDK is to serve as the backbone of our Dart development. It enables us to write code, format it, analyze it for errors, and compile it into native machine code or JavaScript (for web apps). Whether we're building a Flutter mobile app, a web application, or a command-line utility, the Dart SDK is essential.

To get started with Dart, we need to install the Dart SDK on our system. The official guide for installation is available on the Dart website at: https://dart.dev/get-dart

Once installed, we can verify the setup by running `dart --version` in the terminal or command prompt.

Running our First Program

Once we have the Dart SDK installed, we're ready to run our first Dart program! Dart programs typically begin with a special function called `main()`. This function serves as the entry point of every Dart application, meaning it's the first function that runs when our program starts.

```
void main() {
  print('Hello, world!');
}
```

In this code, the `main` function uses the `print()` function to display "Hello, world!" on the console. Every Dart program must have a `main()` function even if it does nothing because that's where execution begins.

To run this program, we save it in a file with a `.dart` extension, such as `hello.dart`. Then, we open our terminal, navigate to the directory where the file is located, and run the following command:

```
dart run hello.dart
```

This runs the file using Dart's Just-In-Time (JIT) compilation, which is ideal for development and testing.

If we want to compile our Dart program into a native executable (so it can run without needing the Dart SDK), we can use Ahead-Of-Time (AOT) compilation:

```
dart compile exe hello.dart
```

This creates an executable file that we can run directly using:

```
./hello
```

Variables

Variables are fundamental building blocks in Dart programs they act like labeled containers that store data. These containers can hold different kinds of information such as numbers, text (strings), true/false values (booleans), and even more complex data structures.
Dart is a **statically typed** language, but it supports **type inference**. This means we can either explicitly declare the type of a variable or let Dart infer it based on the assigned value. This provides the benefits of both safety and flexibility

Creating Variables

In Dart, we can create variables in two main ways: by using the `var` keyword or by explicitly specifying the type. Both approaches are valid and serve different purposes depending on how strict or clear we want our code to be.
Using `var`, Dart automatically infers the type based on the value assigned:

```
var city = 'Algiers';
```

If we want to be more explicit or ensure type safety, we can declare the type ourselves:

```
String country = 'Algeria';
int population = 43000000;
```

Once a variable is assigned a type, we cannot change its type later in the program.

Null Safety in Dart

Dart includes a powerful feature called **null safety**. It helps us avoid one of the most common bugs in programming: null reference errors (also known as "null pointer exceptions"). In Dart, variables are **non-nullable** by default, meaning they **must have a value** when declared.
If we want a variable to be nullable that is, able to hold either a value or `null` we add a `?` after the type:

```
int? score;
```

Data Types

Numbers

Dart supports both integers and floating-point numbers, which are used for numeric operations, measurements, and calculations.

```
int age = 30;
double price = 19.99;
```

If we want a variable that can hold either an `int` or a `double`, we can use the `num` type. It’s a superclass for both `int` and `double` and is useful when we're working with numeric values in a more general way.

```
num temperature = 36.6;
```

```
temperature = 37;
```

We can perform arithmetic operations using the following operators:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (modulus or remainder)
- `~/` (integer division)
- `**` (exponentiation using the `pow` function from `dart:math`)

```
int total = 10 + 5;
double half = 20 / 2.0;
```

we can use num to declare variable

Strings

Strings are sequences of characters enclosed in either single (`'`) or double (`"`) quotes. We use them to store and display text data.

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

We can combine (concatenate) strings using the `+` operator, or embed values using **string interpolation**:

```
String firstName = "Alice";
String lastName = "Joly";
String greet = "Hello, ${firstName + " " + lastName}!";
```

Dart strings are full-featured objects with many useful methods like `.length`, `.toUpperCase()`, `.toLowerCase()`, `.contains()`, etc.

Booleans

The `bool` type represents logical values: `true` or `false`. These are commonly used in conditions, comparisons, and control flow.

```
bool isLoggedIn = true;
bool hasPermission = false;
```

We use booleans with `if` statements, loops, and logical operators (`&&` , `||` , `!`).

Lists

Lists are ordered collections that hold multiple values in a single variable. Each element is accessed by its index, starting at 0.
When we creating List we should indicate the types of values inside it

```
List<String> colors = ["red", "green", "blue"];
print(colors[0]);
```

Dart lists can be fixed or growable, and we can use methods like `.add()` , `.remove()` , `.length` , and `.forEach()` .
we can also create List of mixed value by using dynamic.

```
List<dynamic> mixed = [1, "hello", true];
```

Maps

Maps are collections of key-value pairs, similar to dictionaries in other languages. Each key must be unique, and we use it to retrieve or update the value.
When we creating List we should indicate the types of the keys and values inside it

```
Map<String, int> scores = {   "Ali": 90,   "Sara": 85 };
print(scores["Ali"]);
```

Maps can use strings, numbers, or even symbols as keys.

Sets

Sets are unordered collections of unique values. They're useful when we want to store items without duplication.

```
Set<int> uniqueNumbers = {1, 2, 3};
uniqueNumbers.add(2);
```

Sets support operations like union, intersection, and difference.

The `dynamic` and `Object` Types

If we want to declare a variable that can hold any type, we can use `dynamic`. But this disables type checking, so we lose safety:

```
dynamic something = 'Hello';
something = 42;
```

Constants

In Dart, **constants** are values that never change. We declare them using the `const` or `final` keywords:

- Use `final` when the value is set at runtime but won’t change afterward.
- Use `const` when the value is known at compile time and is truly constant.

```
const double pi = 3.14159;
final int maxUsers = 100;
const int defaultTimeout = 30;
```

Type Conversion

Dart provides ways to convert between data types, which is especially useful when dealing with user input, APIs, or string manipulation.

Convert to Integer

We use `int.parse()` to convert strings to integers:

```
int value = int.parse("42");
```

Trying to convert a non-numeric string will throw an error, so we can use `tryParse()` for safer conversion:

```
int? value = int.tryParse("hello");
```

Convert to Double

To convert strings to floating-point numbers, we use `double.parse()`:

```
double price = double.parse("3.14");  // => 3.14
```

Again, `double.tryParse()` is available for safe conversion.

Convert to String

Any object can be converted to a string using `.toString()`:

```
String message = 100.toString();
String status = true.toString();
```

Comments

Comments in Dart help us document our code. They're ignored during execution and serve as helpful notes for ourselves or other developers.

Single-line Comments

Use `//` to write single-line comments:

```
// This is a single-line comment
print("Hello, world!");  // This prints a greeting
```

Multi-line Comments

Use `/* ... */` for multi-line comments:

```
/*
This is a multi-line comment.
It spans multiple lines.
Useful for explaining complex logic.
*/
```

User Input and Output

In most programs, interacting with the user is essential. Dart provides straightforward ways to display information (output) and receive input from the user.

Output

In Dart, we can use `print` and `stdout.write` to show text on the screen.

- `print` adds a newline after the output.
- `stdout.write` keeps the cursor on the same line.

```
print("Hello!");  // => Hello!
stdout.write("Hi"); // => Hi
stdout.write(" there!");  // => Hi there!
```

Escape Characters

Escape characters start with a backslash (`\`) and are used to include special characters inside strings.

Escape Character	Meaning	Example
`\n`	Newline	`"Line 1\nLine 2"`
`\t`	Tab space	`"Name:\tAlice"`
`\\`	Backslash	`"C:\\Users\\Alice"`
`\"`	Double quote	`"He said, \"Hello!\""`
`\'`	Single quote	`'It\'s fine.'`

```
print("She said, \"Welcome!\"\nLet’s start learning Dart.\n");
```

Output:

```
She said, "Welcome!"
Let’s start learning Dart.
```

Input

To read input from the user in Dart, we use the `stdin.readLineSync()` function. However, this function is part of the `dart:io` library, so we must import it first.

```
import 'dart:io';
void main(){
stdout.write("What's your name? ");
String? name = stdin.readLineSync();
print("Nice to meet you, $name!");
}
```

Note that `stdin.readLineSync()` captures the newline (`\n`) when the user presses Enter, and it returns the input as a `String?` , so we use `!` to tell Dart that we are sure the value is not null. Alternatively, we can use `??` or `if` checks to safely handle nulls.

```
String name = stdin.readLineSync()!;
String user = stdin.readLineSync() ?? "Guest"; // if null we will get "Guest" as
default value
```

Removing the Newline

If you want to remove the newline that comes with user input, you can use the `trim()` method:

```
import 'dart:io';
void main(){
stdout.write("What's your name? ");
String? name = stdin.readLineSync()?.trim();
print("Nice to meet you, $name!");
}
```

Getting Numeric Input

All input from `stdin.readLineSync()` is treated as a `String?` by default. To get a number, we need to convert it using `int.parse()` for integers or `double.parse()` for floats:

```
import 'dart:io';
void main(){
stdout.write("Enter your age: ");
String? ageInput = stdin.readLineSync();
int age = int.parse(ageInput!);
print("In 5 years, you'll be ${age + 5} years old!");
}
```

To handle errors gracefully, it's a good idea to use `tryParse()` for safe parsing:

```
import 'dart:io';
void main(){
stdout.write("Enter your age: ");
String? ageInput = stdin.readLineSync();
int? age = int.tryParse(ageInput!);
print("In 5 years, you’ll be ${age + 5} years old!");
}
```

Tasks

Task 1

Write a program that reads the radius of a circle from the user and then displays its surface area.

Task 2

Develop a temperature converter that convert from celsius to fahrenheit