301 PROJECT

DART PROGRAMMING LANGUAGE FINAL VERSION

Prepared for: Dr. Asif Khan

Prepared by:

Abdullah Saad Alharbi - 2136600 (Team leader)
Faisal Hussain Al Ragheeb - 2136580
Monther Anas Mosa - 2142804
Saleh Mohammed Alsuli - 2136483
Faisal Fadel Othman - 2137119

Contents

Introduction

Dart is a general-purpose programming language developed by Google that is open source. It was initially introduced in 2011 and the stable 1.0 version was published in 2013.

Dart is an object-oriented, class-defined, garbage-collected programming language with C-style syntax. It includes interfaces, mixins, abstract classes, reified generics, and type inference. Dart may be compiled either native code or JavaScript. When compiled to JS, it can run in any modern web browser. When compiled to native code, it may generate mobile apps (Flutter framework) or command-line applications.

Dart was created to be simple to learn, familiar to programmers coming from languages such as Java or JavaScript, and suited for user interface development. Its goal is to integrate the greatest characteristics of traditional OOP languages with the most recent insights from current client and server-side languages.

Dart's main features are: optional typing, JIT and AOT compilation, segregated memory space, actors for concurrency, core libraries for collections, IO, and conversions, and so on. For mobile and web development, Dart includes a standard library as well as platform-specific libraries. The Pub package manager also provides a large number of open-source products.

Google uses Dart in Flutter for mobile development as well as online apps such as AdWords and AdSense. It competes with TypeScript and Kotlin for scalable web and mobile apps.

Domains

Dart is a popular programming language developed by Google. It is often used for building web, server, and mobile applications. Here are some suggested domains where Dart can be applied:

Mobile App Development:

Dart is widely used for developing cross-platform mobile applications. It can be used with the Flutter framework, which allows developers to create high-performance, visually attractive, and functionally rich mobile apps for both Android and iOS platforms.

Web Development:

Dart can be utilized in web development to build modern, responsive, and dynamic web applications. With frameworks like Angular Dart and Aqueduct, developers can create powerful and scalable web applications with ease.

Server-Side Development:

Dart can be used for server-side development, where it can be employed to create fast, scalable, and efficient server applications. Dart's asynchronous programming capabilities make it well-suited for handling concurrent operations and building robust backend systems.

Desktop Application Development:

With the help of the Flutter framework, Dart can be used to build desktop applications for various operating systems like Windows, macOS, and Linux. This allows developers to create visually appealing and highly functional desktop applications with a single codebase.

IoT Development:

Dart can be applied in the Internet of Things (IoT) domain, allowing developers to create applications for IoT devices and embedded systems. Its flexibility and performance make it a suitable language for developing applications that require real-time processing and connectivity.

Game Development:

While not as common as some other languages, Dart can be utilized for game development, especially with the help of the Flame game engine. Developers can create 2D mobile games and web-based games using Dart, taking advantage of its speed and simplicity.

Cross-Platform Development:

Dart's compatibility with Flutter allows developers to create applications that can run on multiple platforms, including mobile, web, and desktop, using a single codebase. This makes it easier to maintain and update applications across different platforms.

Programming Language Paradigm

Dart is a flexible programming language. It was developed by Google. It supports multiple programming paradigms. Some supported paradigms are Object-Oriented Programming, Imperative Programming, Statically Typed, Strongly Typed, and Reflective Programming.

Object-Oriented Programming:

OOP is one of the main paradigms in Dart. Dart is a class-based, object-oriented programming language. It is designed to assist programmers in creating modular and sustainable code. Dart allows programmers to create classes and objects. In addition to using inheritance and polymorphism in the program.

Imperative Programming:

Dart is primarily an imperative language, meaning it executes a sequence of statements to perform actions. Dart allows Programmers to write procedural code, defining functions, loops, and conditional statements to control the program.

Statically Typed:

Dart is Statically Typed. When declaring a variable in Dart, programmers must declare the data type, return values, and function parameters at the time of variable declaration or function definition.

Strongly Typed:

Dart is Strongly Typed. After declaring the Data Type of a variable in Dart, Programmers will not be able to change the data type without utilizing an explicit type conversion.

Reflective Programming:

Dart is not typically considered a reflective programming language, Reflection in Dart use libraries like 'dart:mirrors' for reflective functions, such as accessing object properties, calling methods, and examining class hierarchies, in addition to inspecting and manipulating the structure of objects and classes at runtime.

Language Evaluation Criteria

Readability

Dart is meant to be easy to understand and implement and read by other developers on the same project or even other developers who didn't work on it. yet readability is subjective, and developers may have other preferences. However, here is some main readability criteria:

Overall Simplicity:

Clear and Concise: Code should be clear and concise, avoiding unnecessary complexity.

[For example, Dart has the 'intl' package which provides the 'numberFormat' class that in code it can be used as: `int number = 345678;`

```
NumberFormat formatter = NumberFormat("#,###"); //, or . based on the formating

String formatted = formatter.format(number);
```

In this way the original number of 345678 will be saved in formatted as 345,678.]

Avoid Over-Engineering: Do not overcomplicate solutions; follow the principle of "Keep It Simple."

Orthogonality:

Consistent Semantics: Use consistent naming conventions and semantics throughout your code.

Avoid Ambiguity: Minimize ambiguity and ensure that the meaning of identifiers is clear and predictable.

Modularity: Encourage modular design and separate concerns, making it easier to understand individual components.

Control Statements:

Avoid Deep Nesting: Limit the depth of nested control structures (if statements, loops, etc.) to enhance readability.

Use Descriptive Conditionals: Make conditional expressions and loops descriptive and understandable.

Consistent Formatting: Maintain consistent indentation and formatting for control statements.

Data Types and Structures:

Use Descriptive Variable Names: Choose meaningful variable names that reflect the data they hold.

Avoid Obscure Data Types: Use data types that convey their purpose and meaning clearly.

Consistent Data Structure Usage: Use appropriate data structures (lists, maps, sets, etc.) for the specific requirements of the task.

Syntax Considerations:

Consistent Naming Conventions: Follow consistent naming conventions for variables, functions, and classes (e.g., camelCase or snake_case).

Avoid Magic Numbers and Strings: Replace magic numbers and strings with named constants or enums to make code more self-explanatory.

Proper Commenting: Include comments to explain complex or non-obvious code sections, but avoid excessive comments that clutter the code.

Writability

As mentioned in readability Dart is meant to be easy to implement and maintain clean and efficient code, yet it can be affected by team coding style and how the team may apply the implementation. However here is some main writability criteria:

Simplicity and Orthogonality:

Consistent Syntax: Dart's syntax is designed to be consistent and straightforward, making it easy to write code without unexpected surprises.

Minimize Boilerplate: Dart encourages the reduction of unnecessary boilerplate code, allowing developers to focus on the core logic.

Clear and Intuitive Semantics: Dart's semantics are designed to be clear and intuitive, helping developers write code that behaves as expected.

Reduced Complexity: Minimize language features that introduce unnecessary complexity and ensure that common tasks are straightforward.

Support for Abstraction:

Object-Oriented Features: Dart supports object-oriented programming, allowing developers to create reusable and abstracted code through classes, inheritance, and interfaces.

Functional Programming: Dart supports functional programming concepts like first-class functions and higher-order functions, enabling developers to write expressive and concise code.

Library and Package Support: Utilize Dart's package ecosystem and third-party libraries to leverage existing abstractions and reduce the need for reinventing the wheel.

Custom Abstractions: Create custom abstractions (e.g., classes, functions, and data structures) that encapsulate complexity and promote code reusability.

Expressivity:

Concise Syntax: Dart provides concise and expressive syntax, allowing developers to express complex ideas with minimal code.

[As an example, here in Dart it is possible to write code as

```
void main() { List<int> numbers = [1, 2, 3, 4, 5];
                int sumOfSquares = numbers.fold(0, (sum, number) => sum + number * number);
                print(sumOfSquares);}
```

but in java it's mandatory to write it as

```
public class Main {
   public static void main(String[] args) {
      int[] numbers = {1, 2, 3, 4, 5};
      int sumOfSquares = 0;
      for (int number : numbers) {
         sumOfSquares += number * number;}
      System.out.println(sumOfSquares);}}]
```

Language Features: Take advantage of Dart's language features, such as type inference (var) and cascading method calls (..), to write expressive and succinct code.

Use of Asynchronous Programming: Utilize Dart's async/await syntax to write asynchronous code that is more readable and maintainable.

Reliability

As an object-oriented language, DART uses great methods to ensure reliability, by using static typing, clean and easy-to-read syntax with a strong developer community, and many more methods, it assures compiler speed and reliability.

As a strong programming language, strong exception handling is needed and that's what DART has, an exception-handling system that allows developers and programmers to catch and handle errors during runtime.

Static typing means that variables are checked for type compatibility at compile time, DART uses static typing to seize any potential errors earlier in the development process.

Inspired by other programming languages like Java and JavaScript, DART has simple, clear, and concise syntax. The language design used makes the code easier to read and comprehend, this enhances the overall reliability of DART.

Cost

One of the most looked at criteria is cost, cost can be the leading factor for developers to choose a language solely because the cost is better than other languages. Cost is not just about the financials of the language, it also includes the time for new programmers to learn, the resources needed to run this language, the tools that the language uses, and the availability of a huge community to help and support.

DART has excellent community support, and the resources available from the community such as tutorials, forums, and documentation lead to cost-effective training.

IDEs used by DART are simple and free to use, the variety that the IDEs offer provides choices for developers to use.

As mentioned, DART has remarkable reliability which can help in reducing the cost further as debugging and maintenance are easier and that can lead to retained time, resources, and cost.

EBNF

Assigning Variables

<Flutter Code> → < Variable Declaration > {< Variable Declaration >}

<Variable Declaration> → <Type> <Identifier> "=" <Value> “;"

<Type> → "int" | "double" | "String" | "bool"

<Identifier> → <Letter> {<Letter> | "_" | <Digit>}

<Value> → < Number > | <String> | <Boolean>

<Number> → <Digit> {<Digit>} ["." <Digit> {<Digit>}]

<String> → '"' <Text> '"'

<Boolean> → (true | false)

<Letter> → (A | B | ... | Z | a | b | ... | z)

<Digit> → (0 | 1 | ... | 9)

<Text> → {<Character>}

<Character> → (<Letter> | <Digit> | <Special Character> | <Space>)

<Special Character> → ("_" | "-" | "." | ... (other allowed special characters))

< Space > blank space

Attribute grammar:

- <Variable Declaration> represents the declaration of a variable. It consists of a <Type>, followed by an <Identifier>, the assignment operator =, an <Expression> representing the initial value, and terminated with a semicolon “;”.
- <Type> represents the data type of the variable. This can be any valid type in Dart, such as int, double, String, bool, and others.
- <Identifier>: represents the name of the variable. It is a non-terminal symbol that can have associated attributes such as the variable's name or symbol table references.
- <Value> represents a literal value that can be assigned directly to the variable. It can be a <Number>, <String>, or <Boolean>. It is a non-terminal symbol that can have associated attributes related to the value's type or representation.
- <Number> represents a sequence of <Digit> characters, allowing you to assign numeric values to variables. It is a non-terminal symbol that can have associated attributes such as the numeric value itself or its type.
- <String> represents a string literal enclosed in double quotes. It can contain any characters except for double quotes. Boolean represents a Boolean literal. It is a non-terminal symbol that can have associated attributes such as the Boolean value or its type.
- <Boolean> represents a Boolean value, which can be either true or false.
- <Letter> represents letters (uppercase or lowercase) used in identifiers.
- <Digit> represents digits from 0 to 9.

Example:

int age = 25;

<Flutter Code> → < Variable Declaration >

<Variable Declaration> → <Type> <Identifier> "=" <Value> “;”

<Type> → "int" | "double" | "String" | "bool" <Identifier> "=" <Value> “;”

"int" <Identifier> "=" <Value> “;”

"int" <Identifier> → <Letter> {<Letter> | "_" | <Digit>} "=" <Value> “;”

"int" <Letter> → (A | B | ... | Z | a | b | ... | z) "=" <Value> “;”

"int" <Identifier> → a {<Letter> | "_" | <Digit>} "=" <Value> “;”

"int" <Identifier> → ag {<Letter> | "_" | <Digit>} "=" <Value> “;”

"int" <Identifier> → age {<Letter> | "_" | <Digit>} "=" <Value> “;”

"int age "=" <Value> → < Number > | <String> | <Boolean> “;”

"int age "=" <Number> → <Digit> {<Digit>} ["." <Digit> {<Digit>}] “;”

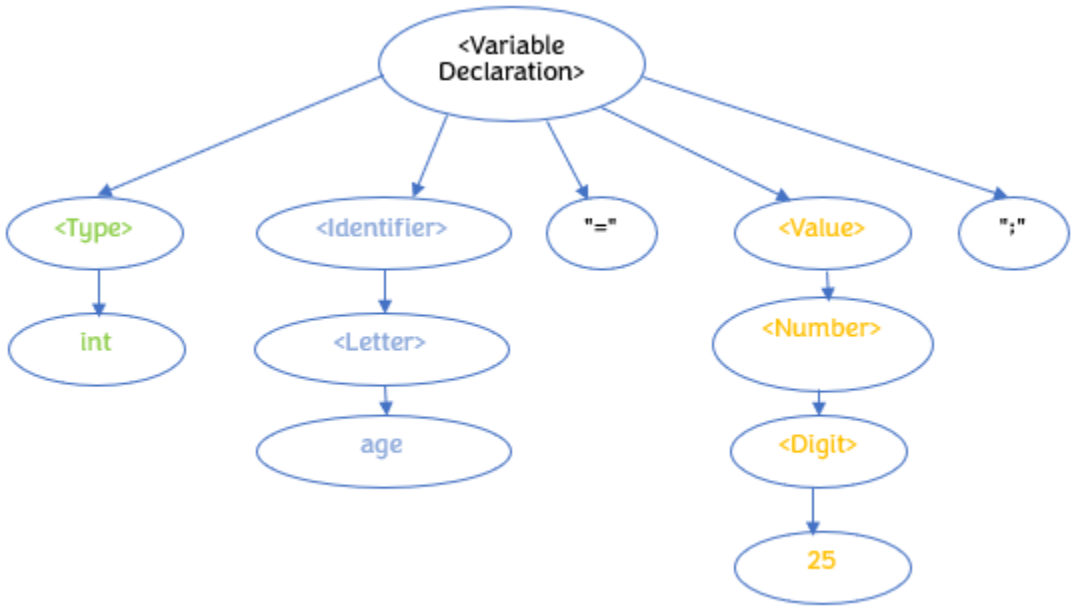
"int age "=" <Digit> → (0 | 1 | ... | 9) “;”

"int age "=" <Number> → 2 {<Digit>} ["." <Digit> {<Digit>}] “;”

"int age "=" <Number> → 25 {<Digit>} ["." <Digit> {<Digit>}] “;”

"int age "=" 25 “;”

Brace tree:



Writing Comment

<Comment> → {(<SingleLineComment> | <MultiLineComment>)}

<SingleLineComment> → "//" <Text> "\n"

<MultiLineComment> → "/*" <Text> "*/"

<Text> → {(< Char > | < Digit > | < Special Char >| < Space >)}

< Char > → (a | b | c | d | .... | Z)

< Digit > → (0 | 1 | 2 | ... | 9)

< Special Char > → (~ | ` | ! | .... | ?)

< Space > → blank space

Attribute grammar:

- <Comment> represents a comment in Dart code and can be either a <SingleLineComment> or a <MultiLineComment>.
- <SingleLineComment> represents a single-line comment and starts with //. It extends until the newline character \n is encountered.
- <MultiLineComment> represents a multi-line comment and starts with /* and ends with */. The comment can span across multiple lines.
- <Text> represents the content of the comment, which can consist of <Text> < Digit > < Special Char > < Space > except for newline characters.

Example:

//Hello Dart

<Comment> → {(<SingleLineComment> | <MultiLineComment>)}

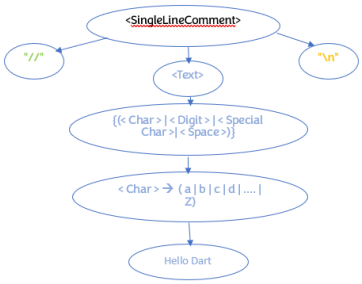
<SingleLineComment> → "//" <Text> "\n"

"//" <Text> → {(< Char > | < Digit > | < Special Char >| < Space >)} "\n"

"//" < Char > → (a | b | c | d | .... | Z) "\n"

"//" Hello Dart "\n"

Brace tree:



Types of Variables

DART programming languages uses many different types of variables of which, the normal variables like string, int, boolean, etc. as well as other special variables like

- Object: The superclass of all Dart classes except Null.
- Enum: The superclass of all enums.
- Future and Stream: Used in asynchrony support.
- Iterable: Used in for-in loops and in synchronous generator functions.

Furthermore the general type of variables can be categorized into 4 main categories

- ✓ Instance variables
- ✓ Static variables
- ✓ Local variables
- ✓ Final and Const variables

INSTANCE VARIABLE

Instance variables are declared within a class and are associated with an instance of the class.

STATIC VARIABLE

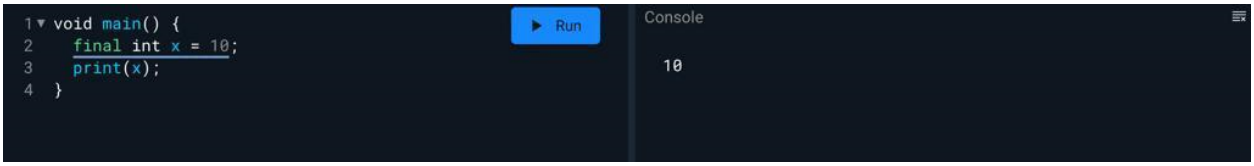
Static variables are declared using the static keyword within a class. They are associated with the class itself rather than with any particular instance.

LOCAL VARIABLES

Local variables are declared within methods, functions, or blocks.

FINAL and CONST VARIABLES

Final and const variables are used for values that cannot be changed once assigned. Only difference is that final runs at run time while const runs at compile time



LIFETIME

In DART, variables have different lifetimes depending on their scope.

- Instance variables are associated with an object and exist as long as the object exists.
- static variables are shared among all instances of a class and throughout the entire execution of the program.

STORAGE

variables are stored in different memory locations depending on their type and scope. Instance variables and static variables are stored on the heap, while local variables are stored on the stack.

Type of scope

Dart is considered a lexically (or statically) scoped language, which means that Dart's variables are determined statically at compiling. In other words, the scope of the variable is defined by the structure of the code and by its location in the code. Dart language combines runtime checks and static. It also ensures that an expression can only produce its own value type.

Ex:

```
Unknown x = 0;
```

In this example, we defined a variable without a valid Dart type. Therefore, If you assigned an unknown type to a variable Dart will show an error at compile time before running the code.

Ex:

```
int x = 1;
x = "1";
```

In this example, we defined x as an integer, then we tried to assign a string to an int variable. Therefore, Dart will also show an error at compile time before running the code.

Even though Dart is lexically scoped, it also has some characters of a dynamically scoped language.

Ex:

```
dynamic x = 0;
x= "it's a dynamic type";
x= false;
```

In this example, we defined x as dynamic which means that we can now assign any data type as we can do in a Dynamically scoped languages.

Global Scope:

When a variable is defined outside a function or a block it is considered a global scoped variable. It can be accessed within any function or block.

Ex:

```
var X = "global scope"
void main() {
}
void outsideFunction() {
}
```

Local Scope:

Variables declared inside a function is local to that function and can only be accessed inside that function, this means that you can define other variables in other functions with the same name.

Ex:

```
void main() {
  var X = " local scope to the main function";
}

void myFunction() {
}
```

Block Scope:

When a variable is defined inside a specific block enclosed within curly brackets {} it is considered a block scoped variable and can only be accessed within the block.

Ex:

```
void main() {

  {
    var blockVar = "block scoped insided the main function";
  }

}
```

Data types

Data types in Dart are static which means the developer can't just write the variable value without writing the data type, but it is possible under one condition that we will talk about.

All variables in Dart are objects, and in some built in types it is possible to use constructors for them. For example, Map() - to create a map object.

Data types in Dart:

Numbers:

int: integers are numbers without decimal point, for example here in Flutter using Dart we can declare int as: `var intNumber = 55;`

double: double is any number that has a decimal point no matter the memory space (in Dart there is no float), an example for double: `var doubleNumber = 5.5;`

num: it is also possible in Dart to declare a variable as 'num' which allows both int and double, for example: `num Number = 5;`

```
Number += 0.5;
```

String:

Strings in dart are a sequence of UTF-16 code units, Dart allows the use of single or double quotes to create a string (there is no character creation in Dart). An example of who to create a string in Dart:

```
var string1 = 'single quotes is allowed in Dart';

var string2 = "double quotes is allowed in Dart";
```

To handle any conflict that may occur while using (' or ") in the string; Dart use "\" to solve this issue.

Booleans:

bool: holds only two values either "true" or "false". Dart is type safety that will require checking values not a statement while using conditions.

Records:

Records in Dart are anonymous, immutable, aggregate type. It is used to combine multiple objects in a single one but it is fixed size and heterogeneous.

Records are also real values that you can declare using variables or nesting or even passing it to another function, to clarify the logic of the records:

```
var record = ('one', a: 1, b:2, 'two');

(int, int) swap((int, int) record){

        var (a,b) = record;

        return (b,a);}
```

in this code we will take the numbers in record and swap it

Lists:

List: an ordered group of objects, AKA array. Dart as the easy language it is, it allows you to create a list in this way: `var list = [1, 2, 3];`

It's not restricted to integer but it's the same way of creating a list using any data type value.

Sets:

Sets are an unordered but unique value, and to declare a set we can use:

```
var setTest = {'item1', 'item2', 'value 1'};
```

as shown, it is similar to list creation.

Maps:

Map: is the object in which associates keys to values without much care about the data type. The only condition is that the key can't be duplicated but the value could be used multiple times.

```
var age = {

        'John': '25',

        'Mark': '33',

        'Brok': '26'};
```

Runes:

Runes are used to manipulate Unicode to provide some characters like emojis and symbolic characters. For example, the hart character (♡) is \u2665.

Arrays

As mentioned arrays in Dart are considered a data type by the name of list. Lists are zero-based indexing where the first item in the list is indexed with 0 and list.length – 1 is the last item in the list.

Lists operations:

Listname.isEmpty – returns true or false

Listname.add('item') – adds an item to the list

Listname.addAll(['item1', 'item2']) – adds multiple items to the list

Listname.length – returns the length of the list

Listname.removeAt(index) – removes the item at the index

Listname.indexOf('item') – returns the index of the item

Listname.clear() – removes all items

List example:

```
var food = <String>[];

assert(food.isEmpty); #true

food.addAll(['fish', 'beef']);

food.add('bread');

assert(food.length == 4) #false answer is 3

food.removeAt(food.indexOf('fish'));

food.clear();
```

Control statements

control statements in Dart, including selection (conditional) and iterative (looping) statements.

Selection Statements:

1. **if Statement:**
    - The **if** statement is used for conditional execution of code.
    - Example:

if (condition)

{ // code to be executed if the condition is true }

else { // code to be executed if the condition is false }

2. **else-if Statement:**
    - Used to check multiple conditions.
    - Example:

if (condition1) { // code to be executed if condition1 is true }

else if (condition2) { // code to be executed if condition2 is true }

else { // code to be executed if all conditions are false }

3. **switch Statement:**
    - Used for multi-way branching based on the value of an expression.
    - Example:

switch (expression)

{ case value1: // code to be executed if expression == value1 break;

case value2: // code to be executed if expression == value2 break;

// ... default: // code to be executed if none of the cases match }

Iterative Statements:

1. **for Loop:**
    - Executes a block of code repeatedly for a specified number of times.
    - Example:

```
for (int i = 0; i < 5; i++)
{ // code to be executed in each iteration }
```

2. **while Loop:**
    - Repeats a block of code as long as a specified condition is true.
    - Example:

```
while (condition)
{ // code to be executed while the condition is true }
```

3. **do-while Loop:**
    - Similar to **while** loop, but the condition is evaluated after the block of code is executed, guaranteeing at least one execution.
    - Example:

```
do { // code to be executed at least once }
 while (condition);
```

4. **forEach Loop:**
    - Used for iterating over elements of collections like lists, sets, and maps.
    - Example:

```
List<int> numbers = [1, 2, 3, 4, 5];
 numbers.forEach((number) {
 // code to be executed for each element });
```

Example:

```
void main()
 { // Selection Statement (if-else)
 int x = 10;
```

```
 if (x > 0) {
print("Positive");
} else if (x < 0) {
print("Negative");
 } else {
 print("Zero");
} // Iterative Statement (for) for (int i = 0; i < 3; i++) {
print("Iteration $i"); } }
```

Purpose of the Example:

- **Illustration of Control Statements:**
  - Demonstrates how to use **if-else** for conditional branching based on a variable's value (**x** in this case).
  - Shows how a **for** loop works, iterating a specific number of times.
- **Educational Usage:**
  - Helps in understanding the syntax and usage of control statements in Dart.
  - Provides a basic structure for learning and experimenting with control flow in Dart programs.

This example acts as a simple demonstration to showcase the functionality of conditional statements (**if-else**) and loops (**for**) in Dart programming.

References

- *The Dart programming language*. (n.d.). Google Books. https://books.google.com.sa/books?hl=en&lr=&id=UHAlCwAAQBAJ&oi=fnd&pg=PT13&dq=dart+programming+language&ots=Pp6TdVCeFO&sig=BwBahATJXYXxD7v3aj1hKsyj3Uo&redir_esc=y#v=onepage&q=dart%20programming%20language&f=false
- *Dart for absolute beginners*. (n.d.). Google Books. https://books.google.com.sa/books?hl=en&lr=&id=EcvjAwAAQBAJ&oi=fnd&pg=PP3&dq=dart+programming+language&ots=eF469G0MhJ&sig=MPl2-WdDk0dqEBhScRwlh9sAMUY&redir_esc=y#v=onepage&q=dart%20programming%20language&f=false
- Badkar, A. (2023, July 14). *What is Dart Programming? Everything You Need to Get Started!* Simplilearn.com. https://www.simplilearn.com/what-is-dart-programming-article
- Azinar, A. (2023, July 19). Class hierarchy in Dart - ITNEXT. *Medium*. https://itnext.io/class-hierarchy-in-dart-ecacc28d0581