Comments in Dart

In Dart, comments are lines of code that are ignored by the compiler and are used to provide explanations or add notes within the code. Comments are not executed or considered as part of the program's functionality. They exist solely for developers to document and explain their code to make it more understandable and maintainable.

In Dart, comments are lines of code that are ignored by the compiler and are used to provide explanations or add notes within the code. Comments are not executed or considered as part of the program's functionality. They exist solely for developers to document and explain their code to make it more understandable and maintainable.

Dart supports two types of comments:

1. **Single-line comments:** These comments start with two forward slashes (//) and continue until the end of the line. Everything after the slashes is ignored by the compiler. For example:

```
// This is a single-line comment in Dart
```

2. **Multi-line comments:** These comments start with a forward slash followed by an asterisk (/*) and end with an asterisk followed by a forward slash (*/). They can span multiple lines. For example:

```
/*
This is a multi-line comment
in Dart that can span
across multiple lines
*/
```

Comments are useful for documenting the code, explaining the purpose of functions or variables, providing instructions, or temporarily disabling code without deleting it. They help other developers (including yourself) understand the code's intent and make it easier to maintain and debug in the future.

How the dart Code compiled and executed

Certainly! Here's a simplified explanation of how Dart code is compiled and executed:

1. **Writing Dart code**: Developers write Dart code using a text editor or an IDE, just like writing a story or a recipe.

2. **Dart Compiler**: Dart code needs to be converted into a language that computers understand. Dart has a special program called the Dart Compiler that does this conversion. It takes the Dart code and transforms it into a format that can be executed.

3. **Running Dart code**: Once the Dart code is converted, it can be run on different platforms:

    a. On the computer: The converted Dart code can be run directly on a computer using a program called the Dart Virtual Machine (VM). It takes the converted code and makes it work on your computer.

    b. In web browsers: If you want to run Dart code in a web browser, it goes through an extra step. The Dart Compiler can convert Dart code into a language called JavaScript, which is understood by web browsers. This allows the Dart code to work in browsers just like any other website or web application.

    c. Mobile and desktop apps: Dart can also be used to build mobile apps and desktop applications. The Dart code is converted into a language that works specifically on mobile devices (like smartphones) or desktop computers. This conversion makes the app run smoothly and perform well on those devices.

Debugging: When working on Dart code, developers may need to find and fix errors or problems. Dart provides tools that help developers find these issues. It's like using a magnifying glass to examine the code closely and understand what's happening step by step.

Variables in Dart Language

In Dart, variables are used to store and manipulate data within a program. A variable is like a labeled container that holds a value, and this value can change during the execution of the program.

Here are some important points about variables in Dart:

Declaration: To create a variable, you need to declare its name and optionally specify its type. Dart supports both static typing and type inference. Static typing means you explicitly declare the type of the variable, such as int, double, String, etc. Type inference allows Dart to automatically determine the type based on the assigned value.

Syntax: The general syntax to declare a variable in Dart is: type variableName = initialValue; Here, type represents the data type of the variable, variableName is the chosen name for the variable, and initialValue is an optional value assigned to the variable at the time of declaration.

Example:

```
int age = 25;  // Declaration of an integer variable named 'age' with an initial value of 25
double price = 9.99;  // Declaration of a double variable named 'price' with an initial value of 9.99
String name = "John";  // Declaration of a string variable named 'name' with an initial value of "John"
```

Changing variable values: Once a variable is declared, its value can be changed by assigning a new value using the variable name. The new value must be compatible with the variable's data type.

```
age = 30; // Changing the value of the 'age' variable to 30
name = "Jane"; // Changing the value of the 'name' variable to "Jane"
```

Variable naming rules: Dart has some rules for naming variables:

Variable names can contain letters, digits, underscores, and dollar signs.
They must start with a letter or an underscore or a $ sign.
Variable names are case-sensitive.
It's recommended to use descriptive names that indicate the purpose of the variable.

Variables play a crucial role in storing and manipulating data during the execution of a Dart program. They allow you to store information, perform calculations, and create dynamic and interactive applications.

More About Variables

Here's some information about variables in Dart:

Variable Declaration:

You can declare variables using the var keyword, which allows Dart to infer the type based on the assigned value. For example:

var message = 'Hello, Dart!';

Alternatively, you can explicitly specify the type using the type annotation syntax. For example:

String name = 'Alice';

Variable Naming:

Variable names in Dart must start with a letter or underscore and can contain letters, digits, or underscores.
Dart is case-sensitive, so **“myVariable”** and **"myvariable”** are considered different variables.

Data Types and Type Inference:

Dart is a statically typed language, which means variables have types that are checked at compile-time.
Dart also supports type inference, where the type is automatically determined based on the assigned value. This allows you to omit the type annotation in certain cases.

Constants:

You can declare constants using the **final** or **const** keyword.
final variables can be assigned only once and their values can be determined at runtime.

const variables are compile-time constants whose values must be known at compile-time.

Variable Mutability:

Variables declared with the **var** or **final** keywords are ***immutable***, meaning their values cannot be changed once assigned.

Variables declared with the **var** or **final** keywords can still ***reference objects*** whose internal state can change.

To declare a mutable variable, you can use the var or type annotation with the late modifier.

Dynamic Typing:

Dart also supports the **dynamic** type, which enables dynamic typing. Variables of type **dynamic** can hold values of any type, and their type can change at runtime.

Here's an example that demonstrates variable usage in Dart:

```
void main() {
  var greeting = 'Hello, Dart!';
  int age = 25;
  final double piValue = 3.14;

  print(greeting);   // Output: Hello, Dart!
  print(age);        // Output: 25
  print(piValue);    // Output: 3.14

  age = 30;
  print(age);        // Output: 30

  // piValue = 3.14159;  // Error: final variables cannot be reassigned
}
```

These are some key aspects of variables in Dart. They allow you to store, manipulate, and reference values in your Dart programs.

Datatypes in Dart

A data type is a classification or categorization of data in programming languages. It determines the kind of values that a variable can hold, the operations that can be performed on those values, and the amount of memory allocated for storing them. Data types define the characteristics and behavior of data within a program.

In other words, In computer programming, a data type is an attribute of data that tells the compiler or interpreter how to treat the data. It specifies the size and type of values that can be stored in a particular variable or object. The data type also determines what operations can be performed on the data, such as arithmetic operations or comparisons.

In Dart, there are several built-in data types available for storing different kinds of values. Here are the commonly used data types in Dart:

Numbers:

Dart supports both integer and floating-point numbers. The int type represents integers, and the double type represents floating-point numbers.

int: Represents integers (whole numbers) such as -3, 0, 42.
double: Represents floating-point numbers with decimal places, such as 3.14, -0.5.

Strings:

A string is a sequence of characters. In Dart, you can create a string by enclosing a sequence of characters in single or double quotes.

String: Represents a sequence of characters enclosed in single quotes ('') or double quotes ("").

Booleans:

A boolean is a data type that has two possible values: true and false. In Dart, the bool type represents boolean values.

bool: Represents a boolean value, which can be either true or false.

Lists:

A list is an ordered collection of objects. In Dart, you can create a list using the List class.

List: Represents an ordered collection of objects. Lists in Dart can contain elements of any type.

Maps:

A map is an unordered collection of key-value pairs. In Dart, you can create a map using the Map class.

Map: Represents a collection of key-value pairs. Maps associate values (the elements) with keys (unique identifiers).

Sets:

A set is an unordered collection of unique objects. In Dart, you can create a set using the Set class.

Set: Represents an unordered collection of unique elements. Sets do not allow duplicate values.

Runes:

A rune represents a Unicode code point. In Dart, you can create a rune using the **\uXXXX** syntax, where **XXXX** is the hexadecimal representation of the code point.

Runes: Represents a sequence of Unicode character codes. It is used to work with special characters and emojis.

Symbols:

A symbol represents an identifier. In Dart, you can create a symbol using the # symbol followed by the identifier.

Symbol: Represents an identifier that is used to access metadata or to identify methods or properties dynamically.

These are the main and basic data types available in Dart. Each data type has its own purpose and usage.

However, Dart also supports other data types such as DateTime for representing dates and times, Duration for representing time intervals, and Function for representing functions.

Operators in dart language

The operators are special symbols that are used to carry out certain operations on the operands. The Dart has numerous built-in operators which can be used to carry out different functions, for example, '+' is used to add two operands. Operators are meant to carry operations on one or two operands.

An expression is a special kind of statement that evaluates to a value. Every expression is composed of −

Operands − Represents the data
Operator − Defines how the operands will be processed to produce a value.

Consider the following expression – "2 + 3". In this expression, 2 and 3 are **operands** and the symbol "+" (plus) is the **operator**.

In this chapter, we will discuss the operators that are available in Dart.

- Arithmetic Operators
- Relational Operators
- Type Test Operators
- Bitwise Operators
- Assignment Operators
- Logical Operators
- Conditional Operator
- Cascade Notation Operator

Arithmetic Operators

Arithmetic operators are a type of operator in Dart that are used to perform mathematical operations on numeric values. There are several arithmetic operators in Dart, which include:

1. **Addition Operator (+):** The addition operator is used to add two values together.

   int x = 5;

```
int y = 10;
int result = x + y;
print(result); // Output: 15
```

2. **Subtraction Operator (-):** The subtraction operator is used to subtract one value from another.

```
int x = 10;
int y = 5;
int result = x - y;
print(result); // Output: 5
```

3. **Multiplication Operator (*):** The multiplication operator is used to multiply two values together.

```
int x = 5;
int y = 10;
int result = x * y;
print(result); // Output: 50
```

4. **Division Operator (/):** The division operator is used to divide one value by another.

```
double x = 10;
double y = 2;
double result = x / y;
print(result); // Output: 5.0
```

5. **Modulo Operator (%):** The modulo operator is used to return the remainder of a division operation.

```
int x = 10;
int y = 3;
int result = x % y;
print(result); // Output: 1
```

These arithmetic operators can be used with variables or literal values to perform mathematical operations in your Dart code.

Relational Operators

Relational operators are a type of operator in Dart that are used to compare two values and return a Boolean value (true or false) based on the comparison. There are several relational operators in Dart, which include:

1. **Equal to Operator (==):** The equal to operator is used to check if two values are equal.

   ```
   int x = 5;
   int y = 5;
   bool result = x == y;
   print(result); // Output: true
   ```

2. **Not equal to Operator (!=):** The not equal to operator is used to check if two values are not equal.

   ```
   int x = 5;
   int y = 10;
   bool result = x != y;
   print(result); // Output: true
   ```

3. **Greater than Operator (>):** The greater than operator is used to check if one value is greater than another. For example:

   ```
   int x = 10;
   int y = 5;
   bool result = x > y;
   print(result); // Output: true
   ```

4. **Less than Operator (<):** The less than operator is used to check if one value is less than another.

   ```
   int x = 5;
   int y = 10;
   bool result = x < y;
   print(result); // Output: true
   ```

5. **Greater than or equal to Operator (>=):** The greater than or equal to operator is used to check if one value is greater than or equal to another.

```
int x = 10;
int y = 10;
bool result = x >= y;
print(result); // Output: true
```

6. **Less than or equal to Operator (<=):** The less than or equal to operator is used to check if one value is less than or equal to another.

```
int x = 5;
int y = 10;
bool result = x <= y;
print(result); // Output: true
```

These relational operators can be used with variables or literal values to perform comparisons in your Dart code.

Type Test Operators

In Dart, type test operators are used to check the type of a value or variable at runtime. These operators return a Boolean value, true or false, based on whether the value or variable is of a particular type. There are three types of type test operators in Dart:

1. **is Operator:** The is operator is used to check if an object is an instance of a particular class.

```
String a = 'GFG';
// Using is to compare
print(a is String);
```

2. **Is ! Operator:** he is not operator is the negation of the is operator, and it is used to check if an object is not an instance of a particular class.

```
double b = 3.3;
// Using is! to compare
print(b is !int);
```

3. **as Operator:** The as operator is used to cast an object to a particular type.

   For example, let's say we have a variable called myVar of type dynamic, which means it can hold any type of value:
   dynamic myVar = 'Hello';

   We can check if this variable is a string using the is operator, and then cast it to a String using the as operator:

   ```
   if (myVar is String) {
     String myString = myVar as String;
     print('myString is $myString');
   }
   ```

   Here, the is operator checks if **myVar** is a string, and if it is, the as operator casts **myVar** to a **String** and assigns it to the **myString** variable. Finally, the value of myString is printed to the console.

   I hope this explanation helps you understand the Type Test Operators in Dart better.

Bitwise Operators

Bitwise operators are a type of operator in Dart that are used to manipulate the bits (0s and 1s) in binary representations of integers. These operators perform operations on the individual bits of the numbers, rather than on the values of the numbers themselves. There are several bitwise operators in Dart, which include:

Operator Symbol	Operator Name	Operator Description

&	Bitwise AND	The bitwise AND operator is used to perform a bitwise AND operation on two values. It returns a value in which each bit is set to 1 only if both corresponding bits of the two values are also 1.
		
		int x = 5;
		int y = 3;

		int result = x & y; print(result); // Output: 1

|	Bitwise OR	The bitwise OR operator is used to perform a bitwise OR operation on two values. It returns a value in which each bit is set to 1 if at least one corresponding bit of the two values is 1.
		
		int x = 5;
		int y = 3;
		int result = x | y;
		print(result); // Output: 7

^	Bitwise XOR	The bitwise XOR operator is used to perform a bitwise XOR operation on two values. It returns a value in which each bit is set to 1 only if the corresponding bits of the two values are different.
		
		int x = 5;
		int y = 3;
		int result = x ^ y;
		print(result); // Output: 6

~	Bitwise NOT	The bitwise NOT operator is used to perform a bitwise NOT operation on a value. It returns a value in which each bit is flipped (i.e., 0s become 1s and 1s become 0s).
		
		int x = 5;
		int result = ~x;
		print(result); // Output: -6

<<	Left Shift	The bitwise left shift operator is used to shift the bits of a value to the left by a specified number of positions.
		
		int x = 5;
		int result = x << 2;
		print(result); // Output: 20

>>	Right Shift	The bitwise right shift operator is used to shift the bits of a value to the right by a specified number of positions. int x = 20;

		int result = x >> 2; print(result); // Output: 5

Assignment Operators

Assignment operators are a type of operator in Dart that are used to assign a value to a variable. They combine the assignment operator (=) with another operator to perform an operation and then assign the result to the variable. This can be a convenient shorthand for performing an operation and assigning the result to a variable in a single statement.

There are several assignment operators in Dart, which include:

Operator Symbol	Operator Name	Operator Description

=	Assignment Operator (=)	The bitwise AND operator is used to perform a bitwise AND operation on two values. It returns a value in which each bit is set to 1 only if both corresponding bits of the two values are also 1.
		
		int x = 5;
		int y = 3;
		int result = x & y;
		print(result); // Output: 1

??=	Null-aware assignment operator (??=)	??= is a Null-aware assignment operator in Dart. It is used to assign a value to a variable only if the variable is null. If the variable already has a value, the operator will not modify it. The syntax of the ??= operator is: variable ??= value; Here, variable is the variable you want to assign a value to, and value is the value you want to assign to the variable if the variable is null.

		We will explain it further. After this table

+=	Compound Addition Operator (+=)	The compound addition operator (+=) is used to add a value to a variable and then assign the result to the variable.
		
		int x = 5;
		x += 2; // equivalent to x = x + 2;
		print(x); // Output: 7

-=	Compound Subtraction Operator (-=)	The compound subtraction operator (-=) is used to subtract a value from a variable and then assign the result to the variable.
		
		int x = 5;
		x -= 2; // equivalent to x = x - 2;
		print(x); // Output: 3

*=	Compound Multiplication Operator (*=)	The compound multiplication operator (*=) is used to multiply a variable by a value and then assign the result to the variable.
		
		int x = 5;
		x *= 2; // equivalent to x = x * 2;
		print(x); // Output: 10

/=	Compound Division Operator (/=)	The compound division operator (/=) is used to divide a variable by a value and then assign the result to the variable.
		
		double x = 10.0;
		x /= 2; // equivalent to x = x / 2;
		print(x); // Output: 5.0

%=	Compound Modulo Operator (%=)	The compound modulo operator (%=) is used to calculate the remainder when a variable is divided by a value and then assign the result to the variable.
		
		int x = 10;
		x %= 3; // equivalent to x = x % 3;
		print(x); // Output: 1

Null Aware Operator

Null-aware operators are a type of operator in Dart that help you handle null values in your code. These operators provide a concise way to check if a value is null before using it, without having to write verbose null-checking code.

The most commonly used null-aware operators are the **?.** and **??** operators.

1.
The **?.** operator is called the null-aware access operator. It allows you to access a property or call a method on an object only if the object is not null. If the object is null, the entire expression evaluates to null.

For example, consider the following code:

```
String name;
int length = name?.length;
```

In this code, the **?.** operator is used to access the **length** property of the name string only if **name** is not null. If **name** is null, the entire expression evaluates to null, and the value of **length** will be null.

2.
The **??** operator is called the null-aware coalescing operator. It allows you to provide a default value to use if a value is null.

For example, consider the following code:

```
String name;
String greeting = 'Hello, ${name ?? 'Guest'}!';
```

In this code, the ?? operator is used to provide a default value of 'Guest' if name is null. If name is not null, the value of name is used in the greeting.

These null-aware operators can help you write more concise and readable code, especially when dealing with null values.

Logical Operators

Logical operators are a type of operator in Dart that are used to perform logical operations on Boolean values. These operators return a Boolean value (true or false) based on the result of the logical operation. There are three logical operators in Dart:

1. **Logical AND Operator (&&):** The logical AND operator is used to perform a logical AND operation on two Boolean values. It returns true if both Boolean values are true, and false otherwise.

   ```
   bool x = true;
   bool y = false;
   print(x && y); // Output: false
   ```

2. **Logical OR Operator (||):** The logical OR operator is used to perform a logical OR operation on two Boolean values. It returns true if at least one of the Boolean values is true, and false otherwise.

   ```
   bool x = true;
   bool y = false;
   print(x || y); // Output: true
   ```

3. **Logical NOT Operator (!):** The logical NOT operator is used to perform a logical NOT operation on a Boolean value. It returns true if the Boolean value is false, and false if the Boolean value is true.

   ```
   bool x = true;
   print(!x); // Output: false
   ```

These logical operators can be used to perform logical operations on Boolean values in your Dart code. Logical operators are commonly used in conditional statements (if, while, for, etc.) to check if a particular condition is true or false.

Conditional Operator

The conditional operator is a shorthand way of expressing a simple if-else statement in Dart. It is also called the ternary operator because it takes three operands.

The syntax of the conditional operator is:

condition ? expression1 : expression2

Here, condition is a Boolean expression. If the condition is true, **expression1** is evaluated and its value is returned. Otherwise, **expression2** is evaluated and its value is returned.

For example:

```
int x = 10;
int y = 5;
int max = x > y ? x : y;
```

In this example, the conditional operator is used to assign the maximum value of **“x”** and **“y”** to the max variable. If **“x”** is greater than **“y”**, the value of **“x”** is assigned to max. Otherwise, the value of “**y”** is assigned to max.

The conditional operator can be useful when you need to choose between two values based on a condition in a single statement. However, it is important to use it judiciously, as overly complex expressions can be difficult to read and understand.

expersion1 ?? expersion2

If expersion1 is non-null it returns its value; else returns expression2 value.

Cascade Notation Operator

This class of operators allows you to perform a sequence of operations on the same element. It allows you to perform multiple methods on the same object.

Or

Cascade notation operator (also called the cascade operator) is a syntactic sugar in Dart that allows you to perform a sequence of operations on the same object, without having to repeat the object reference for each operation. The cascade notation operator consists of two dots (..) and it is used to chain a series of method calls or property accesses on an object.

The syntax of the cascade notation operator is:

object..method1()..method2()..property = value;
Here, **object** is the object that you want to perform the operations on, method1 and **method2** are the methods that you want to call on the object, and property is the property of the object that you want to set the value for.

For example:

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

  void sayHello() {
    print('Hello, my name is $name and I am $age years old.');
  }
}

Person john = Person()
  ..name = 'John'
  ..age = 25
  ..sayHello();
```

In this example, the cascade notation operator is used to create a **Person** object and set its **name** and **age** properties in a single statement. Then, the **sayHello()** method is called on the same object to print a message to the console.

The output of this code will be:

Hello, my name is John and I am 25 years old.

The cascade notation operator can be useful when you want to perform a series of operations on the same object, without having to repeat the object reference for each operation. However, it is important to use it judiciously, as overly complex expressions can be difficult to read and understand.

Now Assignment Time Check Assignment file.
Assignment Submission Process also mentioned in the assignment file.

String & Properties

In Dart, a String is a data type used to represent a sequence of characters. It is used to store and manipulate textual data, such as names, sentences, or any other kind of textual information. You can use either single or double quotes to create a string.

Dart provides a rich set of methods and operators to work with strings.

1. **Declaration and Initialization**: You can declare and initialize a string variable using single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or `"""`) for multiline strings.

```
var string = "I love Pakistan";
var string1 = 'Freelance Bahawalpur is a great platform for upgrading skills';
```

Both the strings above when running on a Dart editor will work perfectly.

Multiline String

A multiline string, also known as a multiline text or a block of text, refers to a string that spans multiple lines of code. In Dart, multiline strings are created using triple quotes (''' or """).

Unlike regular strings enclosed in single quotes (') or double quotes (''), multiline strings preserve line breaks and allow you to write text across multiple lines without the need for escape characters. This is particularly useful when you want to define long strings or preserve the formatting of the text.
Here's an example of a multiline string in Dart:

```
String poem = '''
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
''';
```

In this example, the variable poem contains a multiline string that consists of four lines. The triple quotes (''') allow you to include line breaks and preserve the exact structure of the text.

Multiline strings are commonly used for:

1. Writing long paragraphs or blocks of text.
2. Including formatted text, such as code snippets, HTML, or JSON.
3. Creating multiline comments or documentation.
4. Defining templates or multiline string literals for specific purposes.

It's worth noting that multiline strings can also be created using the String Concatenation (+) at the end of each line to continue the string onto the next line. However, triple quotes provide a more convenient and readable way to define multiline strings in Dart.

```
String poem = 'Roses are red, ' +
  'Violets are blue, ' +
  'Sugar is sweet, ' +
  'And so are you.';
```

While the above code achieves the same result as the multiline string example, it requires explicit string concatenation using the + operator and is less readable compared to using triple quotes.

Multiline strings in Dart are flexible and allow you to include line breaks and preserve the formatting of the text, making them useful for a variety of scenarios.

2. Concatenation: Strings can be concatenated using the `+` operator or by using the `${}` syntax for string interpolation.

String concatenation refers to the process of combining or joining multiple strings together to create a single string. In Dart, you can use the + operator or the string interpolation ($) syntax to concatenate strings.

Here are two common approaches to string concatenation in Dart:

Using the + operator:

```
String firstName = 'John';
String lastName = 'Doe';
```

```
String fullName = firstName + ' ' + lastName;
```

In this example, the + operator is used to concatenate the firstName, a space, and the lastName into the fullName string.

Using string interpolation ($):

```
String interpolatedFullName = '$firstName $lastName';
```

In this example, the $ syntax allows you to embed variables directly within the string. The variables firstName and lastName are interpolated and combined with the space character to form the fullName string.

Both approaches achieve the same result. String concatenation allows you to combine strings, variables, and literal text to create dynamic and meaningful strings.

It's important to note that string concatenation can also be used with **other types of values**, such as numbers, by converting them to strings before concatenation:

```
int age = 25;
String message = 'I am ' + age.toString() + ' years old.';
```

In this example, the age integer is converted to a string using the toString() method before concatenation.

String concatenation is a fundamental operation when working with strings, and it allows you to build dynamic strings by combining various components.

Expression inside String

In Dart, you can include expressions inside a string by using string interpolation. String interpolation allows you to embed expressions, variables, or even function calls within a string literal to produce dynamic results. It is denoted by the $ symbol followed by the expression within curly braces {}.

Here's an example to illustrate the usage of string interpolation:

```
String name = 'Ali';
int age = 30;
```

```
String message = 'My name is ${name} and I am ${age} years old.';
String message = 'My name is $name and I am $age years old.';
```

In both the examples, the variables `name` and `age` are embedded within the string using string interpolation. The expressions `$name` and `$age` are replaced with the corresponding values at runtime, resulting in the final string: `'My name is Alice and I am 30 years old.'`.

You can also perform operations or function calls within the expression:

```
int a = 5;
int b = 10;
String result = 'The sum of $a and $b is ${a + b}.';
```

In this case, the expression `${a + b}` within the string calculates the sum of `a` and `b` and includes the result within the string. The final value of `result` would be `'The sum of 5 and 10 is 15.'`.

String interpolation is a powerful feature that allows you to create dynamic strings by evaluating expressions and incorporating their values directly into the string. It provides a convenient way to construct complex strings based on the current state of your program.

Raw String

In Dart, a raw string is a string literal that allows you to include special characters, such as escape sequences (\), without interpreting them. It is denoted by the r prefix before the opening quote (' or ").

Here's an example to illustrate the usage of a raw string:

String rawString = r'This is a raw string.\n';

In this example, the string `rawString` is a raw string literal. The `r` prefix indicates that the string should be treated as a raw string. Therefore, the escape sequence `\n` is not interpreted as a newline character but is treated as a literal backslash followed by the characters 'n'. The resulting value of `rawString` would be `'This is a raw string.\\n'`.

Raw strings are commonly used when working with regular expressions, file paths, or any string that contains a large number of backslashes or escape sequences. They allow you to simplify the string and avoid the need for excessive escaping of special characters.

Here's an example of using a raw string with a regular expression pattern:

RegExp pattern = RegExp(r'\d+');

In this example, the regular expression pattern `'\d+'` is specified as a raw string using the `r` prefix. This allows the pattern to be written without additional escaping of the backslash, as the backslash is treated as a literal character rather than an escape character.

Raw strings provide a convenient way to represent strings that contain special characters without the need for extensive escaping. They are particularly useful in scenarios where the inclusion of backslashes or escape sequences is common or where readability and maintainability are important.

String Properties

In Dart, the String class provides several properties (also known as getter methods) that allow you to retrieve information or perform operations on strings. Here are some commonly used properties of the String class:

1. **runtimeType:** property is a getter method available on all objects, including strings. It returns a Type object representing the runtime type of the string instance.

   ```
   String message = 'Hello, World!';
   Type type = message.runtimeType;
   print(type); // Output: String
   ```

2. **length**: Returns the number of characters in the string.

   ```
   String message = 'Hello, World!';
   int messageLength = message.length; // 13
   ```

3. **isEmpty and isNotEmpty**: Indicate whether the string is empty or not.

   ```
   String emptyString = '';
   bool isEmpty = emptyString.isEmpty; // true
   bool isNotEmpty = emptyString.isNotEmpty; // false
   ```

4. **runes**: Returns an iterable of Unicode code points (runes) in the string.

   ```
   String greeting = '👋 Hello!';
   Iterable<int> greetingRunes = greeting.runes; // [128075, 32, 72, 101, 108, 108, 111, 33]
   ```

5. **codeUnitAt(index)**: Returns the Unicode code unit of the character at the specified index.

   ```
   String word = 'Dart';
   int codeUnit = word.codeUnitAt(0); // 68
   ```

6. **runes.last**: Returns the Unicode code point (rune) of the last character in the string.

   ```
   String word = 'Dart';
   int lastRune = word.runes.last; // 116 (the code point for 't')
   ```

7. **hashCode**: Returns the hash code of the string.
   ```
   String word = 'Dart';
   int hashCode = word.hashCode; // -1428961390
   ```

These are just a few examples of properties available in the String class. Dart's String class provides additional properties and methods for manipulating and working with strings. You can refer to the official Dart documentation for a comprehensive list of properties and methods available for strings: https://api.dart.dev/stable/dart-core/String-class.html

String Methods

In Dart, the String class provides a variety of methods that allow you to manipulate and work with strings. Here are some commonly used methods of the String class:

1. **toUpperCase() and toLowerCase()**: Returns a new string with all characters converted to uppercase or lowercase, respectively.

   ```
   String message = 'Hello, World!';
   String upperCaseMessage = message.toUpperCase(); // "HELLO, WORLD!"
   String lowerCaseMessage = message.toLowerCase(); // "hello, world!"
   ```

2. **substring(startIndex, endIndex)**: Returns a new string containing the characters from startIndex to endIndex-1.

   ```
   String message = 'Hello, World!';
   String substring = message.substring(7, 12); // "World"
   ```

3. **startsWith(prefix) and endsWith(suffix)**: Returns true if the string starts or ends with the specified prefix or suffix, respectively.

   ```
   String word = 'Hello';
   bool startsWithH = word.startsWith('H'); // true
   bool endsWithO = word.endsWith('o'); // false
   ```

4. **contains(substring)**: Returns true if the string contains the specified substring.

   ```
   String sentence = 'The quick brown fox';
   bool containsFox = sentence.contains('fox'); // true
   bool containsDog = sentence.contains('dog'); // false
   ```

5. **replaceAll(oldSubstring, newSubstring)**: Returns a new string with all occurrences of oldSubstring replaced by newSubstring.

   ```
   String sentence = 'I love apples. Apples are delicious.';
   String newSentence = sentence.replaceAll('apples', 'oranges');
   // "I love oranges. Oranges are delicious."
   ```

6. **trim() and trimLeft()/trimRight()**: Returns a new string with leading/trailing whitespace characters removed, or both.

   ```
   String text = '   Hello, World!   ';
   String trimmedText = text.trim(); // "Hello, World!"
   String leftTrimmedText = text.trimLeft(); // "Hello, World!   "
   String rightTrimmedText = text.trimRight(); // "   Hello, World!"
   ```

7. **padLeft() and padRight()**: methods are used to pad a string with specified characters on the left or right side, respectively, until it reaches a desired length.

   `1. padLeft(width, [paddingCharacter])`: Pads the string on the left side with the specified `paddingCharacter` (default is space) until the resulting string has a length of `width`.

In this example, the padLeft() method is used to pad the string number with zeros on the left side until the resulting string has a length of 5. The output is "00042".

`2. padRight(width, [paddingCharacter])`: Pads the string on the right side with the specified `paddingCharacter` (default is space) until the resulting string has a length of `width`.

```
String word = 'Dart';
String paddedWord = word.padRight(10, '.');
print(paddedWord); // Output: "Dart......"
```

In this example, the padRight() method is used to pad the string word with dots on the right side until the resulting string has a length of 10. The output is "Dart.....".

```
String number = '42';
String paddedNumber = number.padLeft(5, '0');
print(paddedNumber); // Output: "00042"
```

Both padLeft() and padRight() return a new string, leaving the original string unchanged. If the original string is already equal to or longer than the desired width, no padding is applied, and the original string is returned as-is.

These methods are useful for aligning or formatting strings to a specific width by adding padding characters. They are commonly used in scenarios where you need to display text in a fixed-width column or format numerical values with leading zeros.

8. **split()**: method is commonly used to split a string into a list of substrings based on a specified delimiter. In Dart, the split() method is called on a string and takes a delimiter as a parameter. It splits the string at each occurrence of the delimiter and returns a list of substrings.

```
String sentence = 'I love apples and bananas';
List<String> words = sentence.split(' ');
print(words); // Output: ["I", "love", "apples", "and", "bananas"]
```

In this example, the `split()` method is used to split the string `sentence` into a list of words based on the space delimiter. The resulting list contains individual words: `["I", "love", "apples", "and", "bananas"]`.
The split() method can be used with various delimiters, such as spaces, commas, hyphens, or any other character or sequence of characters. It's important to note that the delimiter itself is not included in the resulting substrings.

If the string does not contain the specified delimiter, the split() method will return a list with a single element, which is the original string itself.

String phrase = 'Hello, World!';
List<String> parts = phrase.split(',');
print(parts); // Output: ["Hello", " World!"]

In this example, the split() method splits the string phrase at the comma ,. The resulting list contains two substrings: ["Hello", " World!"].

The split() method is a versatile tool for breaking a string into smaller parts based on a specific delimiter. It allows you to process and manipulate the resulting substrings individually.

These are just a few examples of methods available in the String class. Dart's String class provides many more methods for manipulating, searching, replacing, splitting, and working with strings. You can refer to the official Dart documentation for a comprehensive list of methods available for strings:
https://api.dart.dev/stable/dart-core/String-class.html

Conditional expression

A conditional expression, also known as a ternary operator, is a concise way to write an if-else statement in many programming languages, including Dart. It allows you to evaluate a condition and choose one of two expressions based on the result of that condition.

The syntax of a conditional expression in Dart is as follows:

condition ? expression1 : expression2

Here's how it works:

- The **condition** is a boolean expression that is evaluated.
- If the condition is true, **expression1** is evaluated and becomes the result of the entire conditional expression.
- If the condition is false, **expression2** is evaluated and becomes the result of the entire conditional expression.

Here's an example to demonstrate the usage of a conditional expression:

```
int number = 7;
String result = number % 2 == 0 ? 'Even' : 'Odd';
print(result); // Output: "Odd"
```

In this example, the condition number % 2 == 0 checks whether the number is divisible by 2 (i.e., even). If the condition is true, the expression 'Even' is chosen as the result. If the condition is false, the expression 'Odd' is chosen as the result. In this case, since the number is 7 (which is odd), the output is "Odd".

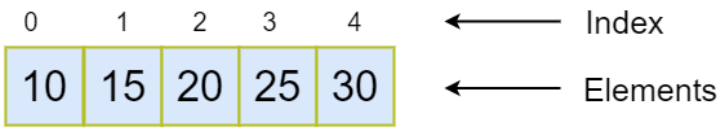
Conditional expressions are useful for writing concise code when you have simple if-else logic. However, it's important to use them judiciously to maintain code readability. In more complex scenarios or when there are multiple conditions to evaluate, it's often better to use if-else statements for better clarity and maintainability.

```
var is_login = false;
var userResult = is_login ? "User found" : "USer Not Found";
  print(userResult);
```

List in Dart

In Dart, a list is an ordered collection of objects. It is similar to an array in other programming languages. A list allows you to store multiple values in a single variable and access them by their index.

Let's understand the graphical representation of the list -



Index: Each element has its index number that tells the element position in the list. The index number is used to access the particular element from the list, such as list_name[index]. The list indexing starts from 0 to length-1 where length denotes the numbers of the element present in the list. For example, - The length of the above list is 4.

Elements: The List elements refers to the actual value or dart object stored in the given list.

Dart provides two types of lists: fixed-length lists and growable lists.

1. Fixed Length List
2. Growable List

First, we will discuss the regular Dart list and how we can create it, then we will briefly discuss a fixed length list or a growable list.

Regular Dart list

```dart
// Dynamic  List
var Lstitems = []; // empty list
print(Lstitems);

var items = ["Dart", "Flutter", "Mobile App", "IOS App", 7845, 21.5]; // dynamic list
print(items);
print(items.runtimeType); // check the list type

items.add("Apple"); // add items to the empty list
items.add("Mango");
items.add("Laptop");
items.add(786);

print(items);
```

1. Fixed Length List

Fixed-length lists: In Dart, you can create a fixed-length list by specifying the size of the list when declaring it. Once created, the size of the list cannot be changed. Here's an example of creating a fixed-length list:

you can create a fixed-length list using the **List<T>.filled()** constructor.

```
var numbers = List.filled(3, 0);
print(numbers);

// Access via indexs
print(numbers[0]);
print(numbers[1]);
print(numbers[2]);
```

We use the **List<int>.filled(3, 0)** constructor to create a fixed-length list called numbers with a length of **3**. The filled() constructor takes two arguments: the length of the list and the initial value to fill the list with (in this case, 0). Then, we assign values to the list as before and print them.

You can also check its type.

```
var numbers = List.filled(3, 0);
print(numbers);
print(numbers.runtimeType);
```

2. Growable List

The list is declared without specifying size is known as a Growable list. Growable lists, as the name suggests, can grow or shrink dynamically. You can add or remove elements from a growable list.

Here's an example of creating a growable list:

```dart
// Growable list
var groceryItems = ["apple", "mango", "grapes", "orange"]; //String type  List
var ItemsNumbers = [1,2,3]; // Int type list
var Items = ["dart", "python", "Flutter", 123, 456, 879]; // dynamic list

print(groceryItems);
print(groceryItems.runtimeType);

print(ItemsNumbers);
print(ItemsNumbers.runtimeType);

print(Items);
print(Items.runtimeType);
```

```dart
var groceryItems = ["apple", "mango", "grapes", "orange"]; //String type  List

print(groceryItems);
print(groceryItems.runtimeType);

groceryItems.add("PineApple");
groceryItems.remove("apple");
```

Both **fixed-length** and growable lists can store objects of any type. In the examples above, **int** and **String** were used as the types of elements, but you can use other types as well.

Lists in Dart have many useful methods and properties to manipulate and access the elements. You can iterate over the elements, find their length, sort them, and perform various other operations.

List Properties

Dart list provides several properties that allow you to access and work with the elements of the list.
Here are some commonly used properties:

1. **length**: Returns the number of elements in the list.

```dart
// Dart List Properties
// length : Returns the number of elements in the list.

List<int> numbers = [1, 2, 3, 4, 5];
print(numbers.length);   // Output: 5
```

2. **first**: Returns the first element of the list.

```dart
// Dart List Properties
// first :  Returns the first element of the list.

List<int> numbers = [1, 2, 3, 4, 5];
print(numbers.first); // Output: 1
```

3. **last**: Returns the last element of the list.

```dart
// Dart List Properties

// last
List<int> numbers = [1, 2, 3, 4, 5];
print(numbers.last); // Output: 5
```

4. **isEmpty**: Returns **true** if the list is empty, otherwise false.

```dart
// Dart List Properties
// isEmpty: Returns true if the list is empty, otherwise false.
List<int> numbers = [];
print(numbers.isEmpty); // Output: true
```

5. **isNotEmpty**: Returns **true** if the list is not empty, otherwise **false**.

```dart
// Dart List Properties
// isNotEmpty: Returns true if the list is not empty, otherwise false.
List<int> numbers = [1, 2, 3];
print(numbers.isNotEmpty); // Output: true
```

6. **reversed**: Returns an iterable of the list elements in reverse order.

```dart
// Dart List Properties
// reversed: Returns an iterable of the list elements in reverse order.
List<int> numbers = [1, 2, 3, 4, 5];
var reversedList = numbers.reversed.toList();
print(reversedList); // Output: [5, 4, 3, 2, 1]
```

These are just a few examples of properties available for regular Dart lists. You can refer to the Dart documentation for the List class to explore more properties and methods that can be used with lists: [List class documentation]

Dart List Methods

Dart provides a wide range of methods for manipulating and working with lists. Here are some commonly used methods available for Dart lists:

1. **add()**: Adds an element to the end of the list.

```dart
// Dart List Properties
// add(): Adds an element to the end of the list.
List<int> numbers = [1, 2, 3];
numbers.add(4); // [1, 2, 3, 4]
```

2. **addAll()**: Adds all elements from another iterable to the end of the list.

```dart
// Dart List Properties
// addAll(): Adds all elements from another iterable to the end of the list.
List<int> numbers = [1, 2, 3];
numbers.addAll([4, 5]); // [1, 2, 3, 4, 5]
```

3. insert(): Inserts an element at a specified index in the list.

```dart
// Dart List Properties
// insert(): Inserts an element at a specified index in the list.
List<String> fruits = ['apple', 'banana', 'cherry'];
fruits.insert(1, 'orange');  // ['apple', 'orange', 'banana', 'cherry']
```

4. **remove()**: Removes the first occurrence of a specific element from the list.

```dart
// Dart List Properties
// remove(): Removes the first occurrence of a specific element from the list.
List<int> numbers = [1, 2, 3, 4, 5];
numbers.remove(3); // [1, 2, 4, 5]
```

5. **removeAt()**: Removes the element at a specified index from the list.

```dart
// Dart List Properties
// removeAt(): Removes the element at a specified index from the list.
List<String> fruits = ['apple', 'banana', 'cherry'];
fruits.removeAt(1); // ['apple', 'cherry']
```

6. **removeLast()**: Removes the last element from the list.

```dart
// Dart List Properties
// removeLast(): Removes the last element from the list.
List<int> numbers = [1, 2, 3];
numbers.removeLast(); // [1, 2]
```

7. **sort()**: Sorts the elements of the list in ascending order.

```dart
// Dart List Properties
// sort(): Sorts the elements of the list in ascending order.
List<int> numbers = [3, 1, 4, 2, 5];
numbers.sort(); // [1, 2, 3, 4, 5]
```

8. **indexOf()**: Returns the index of the first occurrence of a specified element in the list. Returns -1 if the element is not found.

```dart
// Dart List Properties
List<String> fruits = ['apple', 'banana', 'cherry'];
int index = fruits.indexOf('banana'); // 1
print(index);
```

9. **contains()**: Checks if the list contains a specific element. Returns true if the element is found, false otherwise.

```dart
// Dart List Properties
List<int> numbers = [1, 2, 3, 4, 5];
bool containsThree = numbers.contains(3); // true
print(containsThree);
```

10. **forEach()**: Executes a function on each element of the list.

```dart
// Dart List Properties
// forEach(): Executes a function on each element of the list.
List<String> fruits = ['apple', 'banana', 'cherry'];
fruits.forEach((fruit) {
  print(fruit);
});
```

11. **sublist()**: Returns a new list containing a portion of the original list specified by start and end indices.

```dart
// Dart List Properties
List<int> numbers = [1, 2, 3, 4, 5];
List<int> subList = numbers.sublist(1, 3); // [2, 3]
print(subList);
```

These are just a few examples of the many methods available for Dart lists. Dart's list class provides various other useful methods for manipulating, accessing, and modifying lists. You can refer to the Dart documentation for a complete list of list methods and their detailed explanations

Dart List Assignment

Dart List Assignment : Download

SET in Dart

In Dart, a Set is an unordered collection of unique elements. It is similar to a mathematical set, where each element can only appear once. Sets are useful when you want to store a collection of items without any duplicates and don't require the elements to be in a specific order.

Here's an example of creating and using a Set in Dart:

```dart
// How to create a set
var users = {"Arslan", "Ibraheem", "Adnan"};
print(users);
print(users.runtimeType);
```

```dart
// This one is an type object because we add numeric and string values
var users = {1,2, "Arslan", "Ibraheem", "Adnan"};
print(users);
print(users.runtimeType);
```

You can also create SET using Set keyword

```dart
// You can also create SET using Set keyword
Set counting = {"One", "two", "three"};
print(counting);
```

As we said, duplicate values are not stored in SET. Here's is an example

```
// Duplicate values are not stored
var users = {1, 1, 2, "Arslan", "Ibraheem", "Adnan"};
print(users);
// output {1, 2, Arslan, Ibraheem, Adnan}
```

you can create an empty set using the Set constructor without providing any initial elements. Here's how you can create an empty set:

```
// Empty SET
var emptySet = Set();
print(emptySet); // Output: {}
```

In the example above, we create an empty set called emptySet of type Set<String>. Since no initial elements are provided, the set is empty. The curly braces {} represent an empty set when printed.

Alternatively, you can also use a set literal to create an empty set:

```
// Use SET literal to create empty Set
var emptySet = <String>{};
print(emptySet);
print(emptySet.runtimeType);
```

```dart
// This one is also empty set of type int
Set<int> emptySet = {};
print(emptySet);
print(emptySet.runtimeType);
```

In this case, the empty curly braces {} create an empty set without explicitly invoking the Set constructor. The type of the set is inferred as Set<String> based on the context.

Both approaches will create an empty set that you can later populate with elements using the add method or other set operations.

Now we can see how to add items to an empty set in Dart, you can use the add method. Here's an example:

```dart
// How to add new items into the empty Set
Set<String> emptySet = Set<String>();

// Adding items to the set
emptySet.add("apple");
emptySet.add("banana");
emptySet.add("orange");

print(emptySet); // Output: {apple, banana, orange}
```

In the example above, we create an empty set called `emptySet` using the `Set<String>()` constructor. Then, we use the `add` method to add items to the set. Each item is specified as an argument to the `add` method.

After adding the items, we print the set, and the output will show the added elements: `{apple, banana, orange}`.

You can add as many items as you need to the set using the `add` method. Remember that sets in Dart only allow unique elements, so duplicate items will not be added.

Alternatively, you can also add multiple items to the set at once using the addAll method, which takes an iterable of items as an argument. Here's an example:

```dart
// Create Empty Set
Set<String> emptySet = Set<String>();

// Adding multiple items to the set
emptySet.addAll(["apple", "banana", "orange"]);

print(emptySet); // Output: {apple, banana, orange}
```

In this example, we use the `addAll` method to add multiple items to the set in a single operation. The items are specified as an iterable (in this case, a list) inside square brackets `[]`.

Both the `add` and `addAll` methods allow you to add items to an empty set or an existing set.

SET Properties

Here are some common properties and methods available for Dart sets, along with examples:

1. **length**: Returns the number of elements in the set.

```dart
// Length Property
Set<String> fruits = {'apple', 'banana', 'orange'};
print(fruits.length); // Output: 3
```

2. **first**: It is used to get the first element in the given set.

```dart
// first Property
Set<String> fruits = {'apple', 'banana', 'orange'};
print(fruits.first); // Output: apple
```

3. **last**: It is used to get the last element in the given set.

```dart
// first Property
Set<String> fruits = {'apple', 'banana', 'orange'};
print(fruits.last); // Output: orange
```

4. **isEmpty:** If the set does not contain any element, it returns true.

```dart
// isEmpty Property
Set<String> fruits = {'apple', 'banana', 'orange'};
print(fruits.isEmpty); // Output: False
```

5. **isNotEmpty**: If the set contains at least one element, it returns true.

```dart
// isEmpty Property
Set<String> fruits = {'apple', 'banana', 'orange'};
print(fruits.isNotEmpty); // Output: True
```

6. **Hashcode:** It is used to get the hash code for the corresponding object.

```dart
// isEmpty Property
Set<String> fruits = {'apple', 'banana', 'orange'};
print(fruits.hashCode); // Output: 333557973
```

SET Methods

Here are some of the commonly used Dart Set methods along with examples:

1. **add(element)** - Adds an element to the set.

```dart
Set<int> mySet = Set();

// Adding elements to the set
mySet.add(1);
print(mySet);
```

2. **addAll(elements)** - Adds multiple elements to the set.

```dart
Set<int> mySet = Set();

mySet.addAll([2, 3, 4]);
print(mySet);
```

3. **remove(element)** - Removes an element from the set.

```dart
// Removing an element from the set
mySet.remove(2);
print(mySet);  // Output: {1, 3, 4}
```

4. **contains(element)** - Checks if the set contains a specific element.

```dart
// Checking if the set contains a specific element
bool containsElement = mySet.contains(3);
print(containsElement);  // Output: true
```

5. **clear()** - Removes all elements from the set.

```
// Clearing the set
mySet.clear();
print(mySet);  // Output: {}
```

6. **forEach(callback)** - Applies a callback function to each element of the set.

```
// Applying a callback function to each element of the set
mySet.forEach((element) {
  print(element * 2);  // Output: 2, 6, 8
});
```

7. **toList()** - Converts the set to a list.

```
// Converting the set to a list
List<int> myList = mySet.toList();
print(myList);  // Output: [1, 3, 4]
```

More Important SET Methods

1. **Union**: The union of two sets includes all the elements present in either set. To find the union of two sets, you can use the union method.

```dart
// Union Method
Set<int> set1 = {1, 2, 3};
Set<int> set2 = {3, 4, 5};

Set<int> unionSet = set1.union(set2);
print(unionSet);  // Output: {1, 2, 3, 4, 5}
```

2. **Intersection**: The intersection of two sets contains only the elements that are common to both sets. Dart doesn't provide a direct intersection method, but you can achieve the intersection by using the where method and the contains method.

```dart
// Intersection
Set<int> set1 = {1, 2, 3};
Set<int> set2 = {3, 4, 5};

Set<int> intersectionSet = set1.where((element) => set2.contains(element)).toSet();
print(intersectionSet);  // Output: {3}
```

This Example code demonstrates how to find the intersection of two sets in the Dart programming language.

Here's a step-by-step explanation:

1. Two sets, set1 and set2, are defined. set1 contains the elements {1, 2, 3}, while set2 contains the elements {3, 4, 5}. Sets are unordered collections of unique elements, so duplicate elements are automatically removed.

2. The where method is called on set1. This method takes a predicate function as an argument and returns a new iterable containing only the elements that satisfy the condition specified in the predicate function.

3. In this case, the predicate function is (element) => set2.contains(element), which checks if each element in set1 is also

present in set2. If the condition is true, the element is included in the resulting iterable.

4. The toSet() method is called on the resulting iterable to convert it back into a set. This step is optional since the where method already returns an iterable, but calling toSet() ensures that the final result is a set.

5. Finally, the intersectionSet is printed, which should contain the intersection of set1 and set2. In this case, the output will be {3} since 3 is the only element present in both sets.

3. **Subtracting**: Subtracting one set from another removes the elements that are present in the second set from the first set. You can use the difference method to perform set subtraction.

```dart
// Subtracting
Set<int> set1 = {1, 2, 3};
Set<int> set2 = {3, 4, 5};

Set<int> subtractedSet = set1.difference(set2);
print(subtractedSet);  // Output: {1, 2}
```

These methods work on sets of any type, not just integers. Simply replace int with the appropriate type when using sets of different elements.

Map in Dart

In Dart, a Map is a collection of key-value pairs. It is also known as an associative array or a dictionary in other programming languages. The keys in a map are unique, and each key is associated with a specific value. Maps are useful for organizing and accessing data based on keys.

Dart Map is an object that stores data in the form of a key-value pair. Each value is associated with its key, and it is used to access its corresponding value. Both keys and values can be any type. In Dart Map, each key must be unique, but the same value can occur multiple times. The Map representation is quite similar to Python Dictionary. The Map can be declared by using curly braces {} ,and each key-value pair is separated by the commas(,). The value of the key can be accessed by using a square bracket([]).

Declaring a Dart Map

To declare a map in Dart, you have a few different options. Here are three common ways to declare a map:

1. **Using a map literal:**

```
Map<String, int> studentGrades = {
'Computer': 85,
'Physic': 90,
'Math': 95,
};
```

In this example, we declare a map called studentGrades that maps String keys to int values. The map is initialized with key-value pairs using curly braces {}. Each key-value pair is separated by a colon : and multiple pairs are separated by commas ,

2. **Using the Map constructor:**

```
Map<String, int> studentGrades = Map<String, int>();
print(studentGrades);
print(studentGrades.runtimeType);
```

Here, we declare a map called studentGrades using the Map constructor. The type parameters <String, int> specify that the keys will be of type String and the values will be of type int. This constructor creates an empty map that can be populated later.

3. **Using the Map literal constructor:**

```
// 3rd Method
Map<String, int> studentGrades = {'CS': 85, 'PHY': 90, 'MTH': 95};
print(studentGrades);
print(studentGrades.runtimeType);
```

This is similar to the first approach using a map literal, but we explicitly use the Map constructor with the key-value pairs inside curly braces {}. The type parameters <String, int> specify the types of keys and values in the map.

In all these cases, you can replace String with any other appropriate key type and int with any value type you need for your map.

Create Empty Map using Map Constructor

To create an empty map using the Map constructor in Dart, you can simply call the constructor without any arguments.

Here's how you can do it:

```
Map<String, int> emptyMap = Map<String, int>();

print(emptyMap);  // Output: {}
print(emptyMap.length);  // Output: 0
```

In this example, we declare a variable emptyMap and initialize it as an empty map using the Map constructor. The type parameters <String, int> specify that the keys will be of type String and the values will be of type int.

When you print emptyMap, it will display an empty map {}. The length property of the map will be 0, indicating that there are no key-value pairs in the map.

Add Items to Map

You can add items to a Map using the square bracket ([]) operator. Here's an example of how you can add items to a Map:

```
// Create an empty map
var myMap = {};

// Add items to the map
myMap['key1'] = 'value1';
myMap['key2'] = 'value2';
myMap['key3'] = 'value3';

// Print the map
print(myMap);
```

In this example, we start with an empty map called **myMap**. We then use the square bracket operator to assign values to specific keys. The key-value pairs are

separated by an equal sign (=), where the left side represents the key, and the right side represents the value. Finally, we print the map, which will output:

```
{key1: value1, key2: value2, key3: value3}
```

You can also initialize a map with items directly using a map literal syntax. Here's an example:

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

print(myMap);
```

This code achieves the same result as the previous example.

Access Map elements

To access elements in a Dart Map, you can use the square bracket ([]) operator with the key of the desired element.

Here's an example:

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

print(myMap);

// // Access individual elements
print(myMap['key1']);
print(myMap['key2']);
print(myMap['key3']);
```

In this example, we have a **Map** called "**myMap**" with three key-value pairs. To access individual elements, we use the square bracket operator with the corresponding key. The values are assigned to variables **value1** and **value2**, and we print them to the console.

If you try to access a key that doesn't exist in the map, it will return **null**. You can also use the "**containsKey()**" method to check if a key exists in the map before accessing its value. Here's an example:

```
// method to check if a key exists in the map or not
if (myMap.containsKey('key1')) {
  var value = myMap['key1'];
  print(value); // Output: value1
} else {
  print('Key not found');
}

if (myMap.containsKey('key4')) {
  var value = myMap['key4'];
  print(value); // This block won't execute
} else {
  print('Key not found'); // Output: Key not found
}
```

In this example, we check if the key **'key1'** exists in the map using the **containsKey()** method. If it exists, we access the value and print it. If the key doesn't exist, we print a message indicating that the key was not found.

Map Properties

Here are some commonly used properties of the Map class in Dart:

1. `length`: Returns the number of key-value pairs in the map.
2. `keys`: Returns an iterable containing all the keys in the map.
3. `values`: Returns an iterable containing all the values in the map.
4. `isEmpty`: Returns `true` if the map is empty (contains no key-value pairs), otherwise returns `false`.
5. `isNotEmpty`: Returns `true` if the map is not empty, otherwise returns `false`.

Let's go through each property of the Map class in Dart one by one, with examples illustrating their usage.

1. **length**: This property returns the number of key-value pairs in the map.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

print('Length: ${myMap.length}');
```

2. **keys**: This property returns an iterable containing all the keys in the map.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

print('Keys:');
for (var key in myMap.keys) {
  print(key);
}
```

3. **values**: This property returns an iterable containing all the values in the map.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

print('Values:');
for (var value in myMap.values) {
  print(value);
}
```

4. **isEmpty**: This property returns true if the map is empty (contains no key-value pairs), otherwise it returns false.

```
var emptyMap = {};

var nonEmptyMap = {
  'key1': 'value1',
};

print('Is empty? ${emptyMap.isEmpty}'); // Output: Is empty? true
print('Is empty? ${nonEmptyMap.isEmpty}'); // Output: Is empty? false
```

5. **isNotEmpty**: This property returns true if the map is not empty, otherwise it returns false.

```dart
// isNotEmpty
var emptyMap = {};

var nonEmptyMap = {
  'key1': 'value1',
};

print('Is not empty? ${emptyMap.isNotEmpty}'); // Output: Is not empty? false
print('Is not empty? ${nonEmptyMap.isNotEmpty}'); // Output: Is not empty? true
```

I hope these explanations and examples clarify the usage of each property of the Map class in Dart.

Map Methods

Here is a comprehensive list of methods available for the Dart Map class, along with their descriptions:

1. **containsKey(Object key)**: Returns true if the map contains the specified key.

```dart
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

var keyExists = myMap.containsKey('key2');
print(keyExists); // Output: true
```

In this example, we use the **containsKey()** method to check if the map **myMap** contains the key **'key2'**. It returns **true** because the key is present in the map.

2. **containsValue(Object value)**: Returns true if the map contains the specified value.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

var valueExists = myMap.containsValue('value2');
print(valueExists); // Output: true
```

Here, we use the **containsValue()** method to check if the map **myMap** contains the value **'value2'**. It returns **true** because the value is present in the map.

3. **remove(Object key):** Removes the entry with the specified key from the map and returns the associated value.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

var removedValue = myMap.remove('key2');
print(removedValue); // Output: value2

print(myMap);
```

In this example, we use the remove() method to remove the entry with the key 'key2' from the map myMap. The method returns the associated value, which is 'value2'. We then print the removed value and the updated map after the removal.

4. **addAll(Map<K, V> other)**: Adds all key-value pairs from the other map into the current map.

```
var map1 = {'key1': 'value1'};
var map2 = {'key2': 'value2', 'key3': 'value3'};

map1.addAll(map2);
print(map1);
```

Here, we use the addAll() method to add all key-value pairs from map2 into map1. After calling addAll(), map1 contains all the key-value pairs from both maps.

5. **clear()**: Removes all key-value pairs from the map.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

myMap.clear();
print(myMap); // Output: {}
```

In this example, we use the clear() method to remove all key-value pairs from the map myMap. After calling clear(), the map becomes empty.

6. **forEach(void action(K key, V value))**: Applies the specified function to each key-value pair in the map.

```
var myMap = {
  'key1': 'value1',
  'key2': 'value2',
  'key3': 'value3',
};

myMap.forEach((key, value) {
  print('Key: $key, Value: $value');
});
```

In this example, we define an anonymous function (key, value), which is the action function for forEach. Inside the function, we print the key and value for each key-value pair.

More Precisely Explain about how forEach Work?

The forEach method of the Map class allows you to apply a specified function to each key-value pair in the map.

Here's the syntax

void forEach(void action(K key, V value))

The action function takes two parameters: key and value. For each key-value pair in the map, the action function is called and provided with the corresponding key and value.

Control Flow Statement

A control flow statement is a feature of programming languages that allows you to control the execution order and flow of statements in a program. It provides mechanisms to conditionally execute code blocks, loop over code segments multiple times, and handle different cases or scenarios based on specific conditions.

Control flow statements help you direct the flow of execution in your code based on certain conditions or requirements. They allow you to make decisions, repeat code, and handle different scenarios, thus making your programs more flexible and capable of responding to different situations.

Common control flow statements in programming languages, including Dart, include:

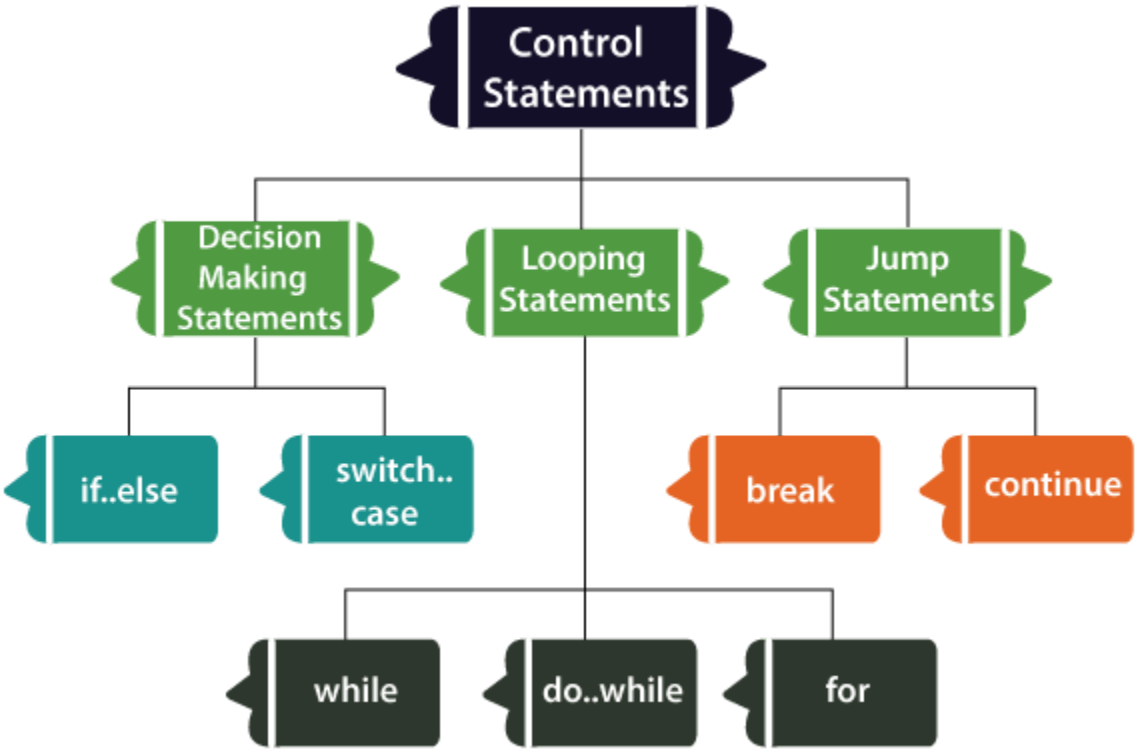
1. **Conditional Statements:**
    - if statement
    - if-else statement
    - switch statement
2. **Looping Statements:**
    - for loop
    - for in loop
    - while loop
    - do-while loop
    - forEach loop
3. **Jump Statements:**
    - break statement
    - continue statement
    - return statement

These statements give you the ability to structure your code's logic, control program flow, and handle different scenarios based on conditions or iterations. They provide mechanisms to make decisions, iterate over sequences, exit or skip code blocks, and more.

By utilizing control flow statements effectively, you can create programs that respond dynamically to different situations, make decisions based on certain conditions, and execute code in a structured and organized manner.

Decision-Making Statements

The Decision-making statements allow us to determine which statement to execute based on the test expression at runtime. Decision-making statements are also known as the Selection statements. In the Dart program, a single or multiple test expression (or condition) can exist, which evaluates Boolean TRUE and FALSE. These results of the expression/condition helps to decide which block of statement (s) will execute if the given condition is TRUE or FALSE.



Dart if Statements

In Dart, the if statement is used for conditional execution of code blocks. It allows you to execute a block of code only if a specified condition is true. If the condition evaluates to true, the code block within the if statement is executed.

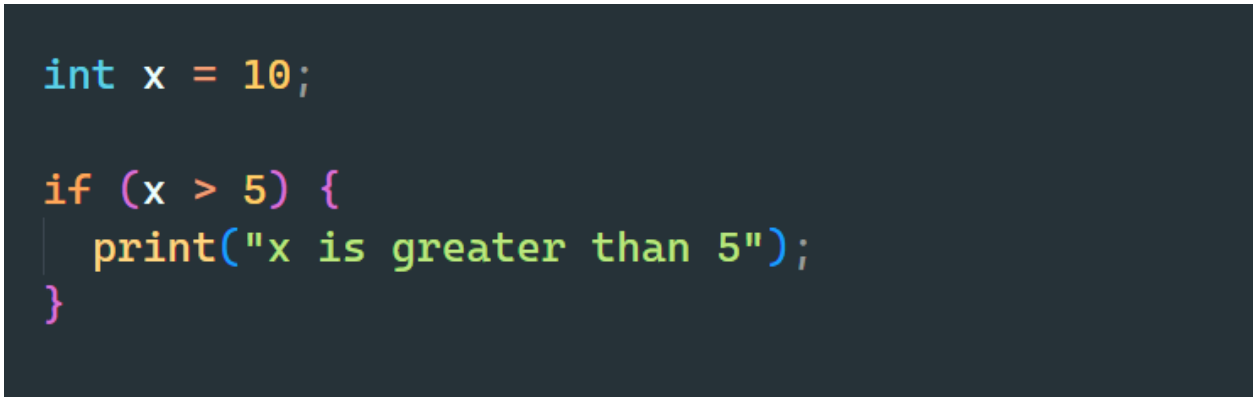
Here's the syntax for the "**if**" statement:

```
if (condition) {
  // Code to be executed if the condition is true
}
```

condition: A Boolean expression that determines whether the code block should be executed. It can be any expression that evaluates to either true or false.

Code to be executed: The block of code to execute if the condition is true. It can contain one or more statements.

Example:



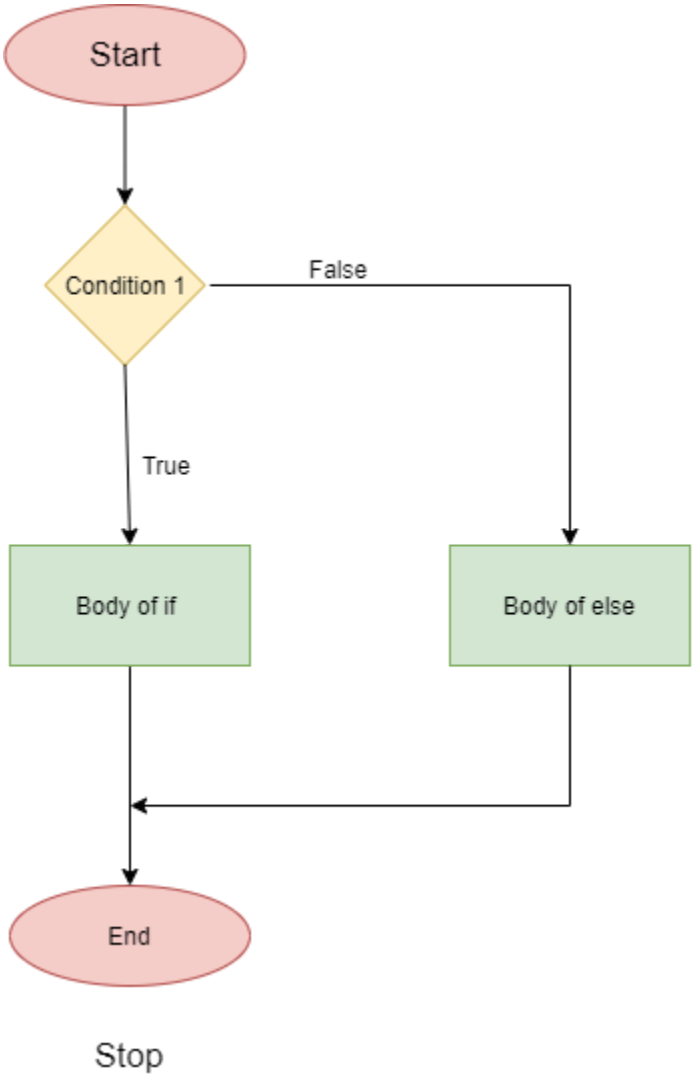
In this example, the value of **x** is 10. The condition **x > 5** evaluates to true, so the code block within the if statement is executed, and it prints "**x is greater than 5**" to the console.

If the value of x were, for example, 3, the condition x > 5 would evaluate to false. In that case nothing prints on the screen.

Dart if-else Statement

The if-else statement allows you to execute different code blocks based on a condition. It provides an alternative block of code to execute if the condition in the if statement evaluates to false.

Dart if...else Statement Flow Diagram



Stop

Here's the syntax for the if-else statement:

```
if (condition) {
  // Code to be executed if the condition is true
} else {
  // Code to be executed if the condition is false
}
```

condition: A Boolean expression that determines whether the code block should be executed. It can be any expression that evaluates to either true or false.

Code to be executed if the condition is true: The block of code to execute if the condition is true. It can contain one or more statements.

Code to be executed if the condition is false: The block of code to execute if the condition is false. It can also contain one or more statements.

Example:

```
int age = 20;

if (age >= 18) {
  print("You are an adult");
} else {
  print("You are not an adult");
}
```

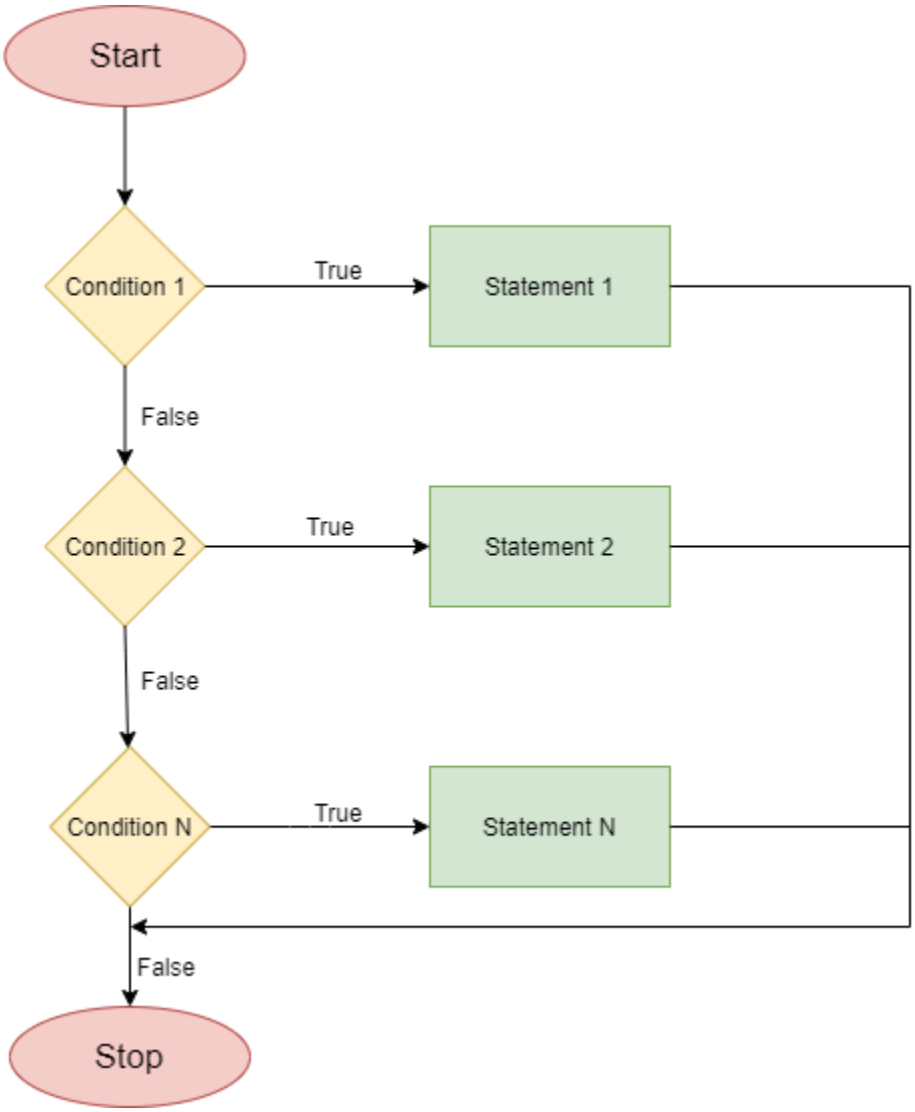
In this example, the value of **age** is 20. The condition **age >= 18** evaluates to true, so the code block within the **if statement** is executed, and it prints "You are an adult" to the console.

If the value of **age** were, for example, 15, the condition **age >= 18** would evaluate to false. In that case, the code block within the **else statement** would be executed, and it would print "You are not an adult" to the console.
The **if-else** statement allows you to handle two possible outcomes based on a condition. It is useful when you want to execute different blocks of code depending on whether a condition is true or false.

Dart if else-if Statement

The **if-else if** statement allows you to check multiple conditions and execute different code blocks based on the first condition that evaluates to true. It provides an alternative code block for each condition, allowing you to handle multiple cases or scenarios.

Dart if else if Statement Flow Diagram



Here's the syntax for the if-else if statement:

```
if (condition1) {
  // Code to be executed if condition1 is true
} else if (condition2) {
  // Code to be executed if condition1 is false and condition2 is true
} else if (condition3) {
  // Code to be executed if condition1 and condition2 are false and condition3 is true
}
// ...
else {
  // Code to be executed if none of the conditions are true
}
```

condition1, **condition2**, **condition3**, and so on: Boolean expressions that determine whether the corresponding code block should be executed. They can be any expressions that evaluate to either true or false.

Code to be executed if the condition is true: The block of code to execute if the corresponding condition is true. It can contain one or more statements.

else if (optional): Allows you to add additional conditions to check if the previous conditions are false.

else (optional): Specifies a default code block to execute if none of the conditions evaluate to true.

```
int score = 85;

if (score >= 90) {
  print("Excellent");
} else if (score >= 80) {
  print("Good");
} else if (score >= 70) {
  print("Average");
} else {
  print("Below Average");
}
```

In this example, the value of the **score** is 85. The first condition **score >= 90** evaluates to false. The second condition **score >= 80** evaluates to true, so the code blocks within the corresponding **else if** statement is executed, and it prints "Good" to the console.

The **if-else if** statement allows you to handle multiple cases or scenarios based on different conditions. It provides a way to test multiple conditions sequentially and execute the code block associated with the first condition that evaluates to true.

Dart Nested If else

You can nest **if-else** statements, which means you can include one **if-else** statement within another **if** or **else** block. This allows you to create more complex conditional logic and handle nested conditions.

Here's an example of nested if-else statements:

```
int x = 10;
int y = 5;

if (x > 5) {
  if (y > 3) {
    print("Both x and y are greater than their respective thresholds");
  } else {
    print("x is greater than 5, but y is not greater than 3");
  }
} else {
  print("x is not greater than 5");
}
```

In this example, we have nested if-else statements. The outer if statement checks if x is greater than 5. If it is, the code inside the outer if block is executed. Within that block, we have another if-else statement that checks if y is greater than 3. If it is, the code inside the inner if block is executed. Otherwise, the code inside the inner else block is executed.

Based on the values of x and y in this example, the output will be "Both x and y are greater than their respective thresholds" because both conditions are true.

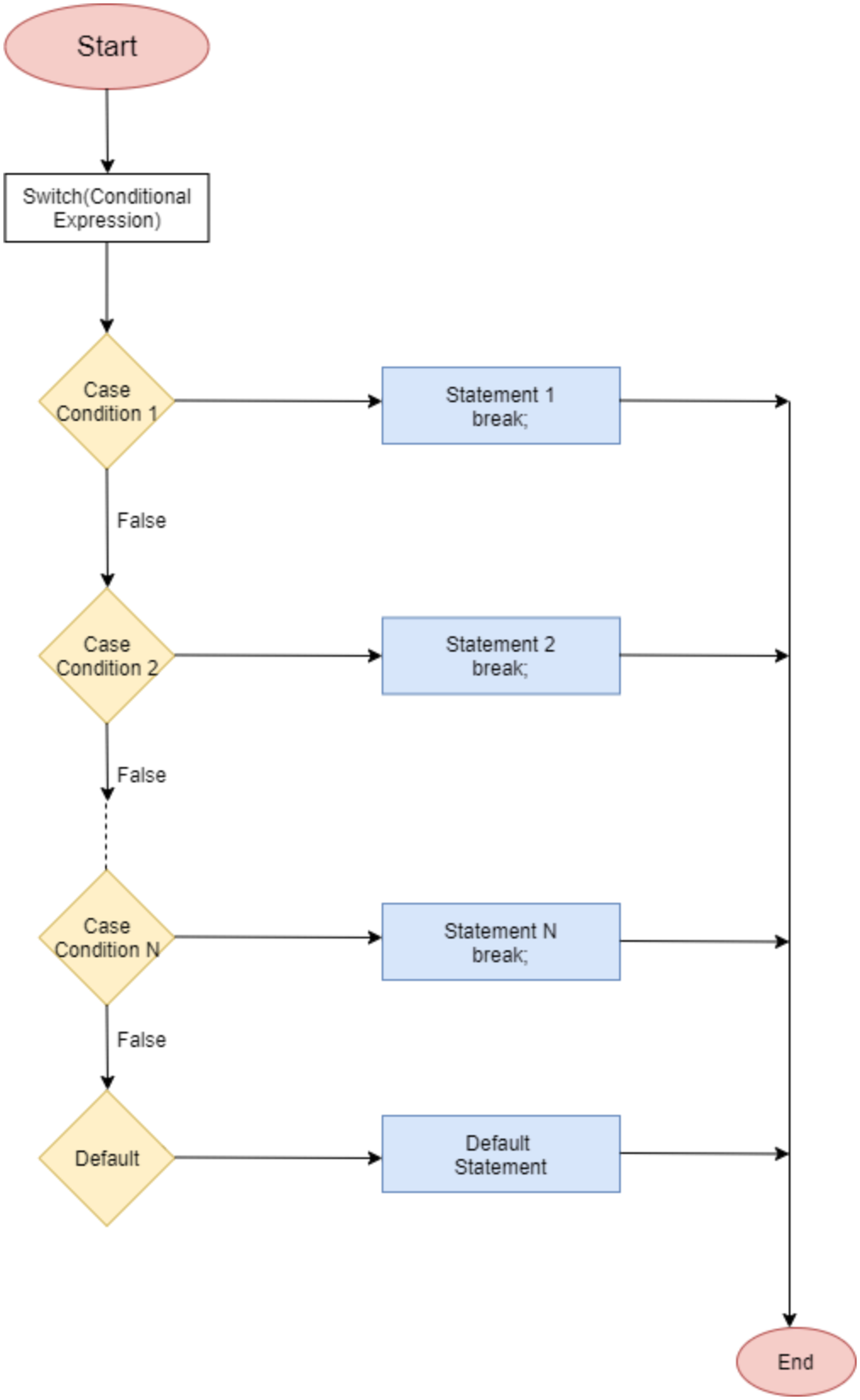
Nested if-else statements are useful when you need to handle more intricate conditions that involve multiple levels of decision-making. They allow you to build complex conditional structures and execute different code blocks based on nested conditions. However, it's important to maintain clarity and readability by properly indenting and organizing your nested statements.

Dart Switch Case Statement

Dart Switch case statement is used to avoid the long chain of the if-else statement. It is the simplified form of a nested if-else statement. The value of the variable compares with the multiple cases, and if a match is found, then it executes a block of statements associated with that particular case.

The assigned value is compared with each case until the match is found. Once the match is found, it identifies the block of code to be executed.

Dart Switch Case Statement Flowchart


Start
Switch(Conditional Expression)
Case Condition 1
Statement 1
break;
False
Case Condition 2
Statement 2
break;
False
Case Condition N
Statement N
break;
False
Default
Default Statement
End

Here's the basic syntax of the switch statement in Dart:

```
switch (expression) {
  case value1:
    // Code to be executed if expression matches value1
    break;
  case value2:
    // Code to be executed if expression matches value2
    break;
  // Additional case statements
  default:
    // Code to be executed if none of the above cases match
}
```

Here, the expression can be integer expression or character expression. The value 1, 2, n represents the case labels and they are used to identify each case particularly. Each label must be ended with the colon(:).

The labels must be unique because the same name label will create the problem while running the program.

A block is associated with the case label. Block is nothing but a group of multiple statements for a particular case.

Once the switch expression is evaluated, the expression value is compared with all cases which we have defined inside the switch case. Suppose the value of the expression is 2, then compared with each case until it found the label 2 in the program.

The **break statement** is essential to use at the end of each case. If we do not put the break statement, then even if a specific case is found, it will execute all the cases until the program end is reached. The **break** keyword is used to declare the break statement.

Sometimes the value of the expression is not matched with any of the cases; then the default case will be executed. It is optional to write in the program.

Now let's look at an example to better understand how the switch statement works:

```dart
String fruit = 'apple';

switch (fruit) {
case 'apple':
  print('It is an apple.');
  break;
case 'banana':
  print('It is a banana.');
  break;
case 'orange':
  print('It is an orange.');
  break;
default:
  print('It is an unknown fruit.');
}
```

In this example, the **switch** statement evaluates the value of the **fruit** variable. If it matches any of the cases, the corresponding code block is executed. In this case, since **fruit** is set to **'apple'**,
the output will be: It is an apple.

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\switch.dart"
It is an apple.
```

If the value doesn't match any of the cases, the code block associated with the **default** keyword will be executed. In this example, if the value of fruit was something other than **'apple'**, **'banana'**, or **'orange'**,

the output would be: It is an unknown fruit.

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\switch.dart"
It is an unknown fruit.
```

Let's understand the concept with other example.

```
// declaring a interger variable
int Roll_num =  90014;

// Evalaute the test-expression to find the match
  switch (Roll_num) {
  case 90009:
    print("My name is Arslan");
    break;
  case 90010:
    print("My name is Ibraheem");
    break;
  case 090011:
    print("My name is Adnan");
    break;

// default block
  default:
    print("Roll number is not found");
}
```

Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\switch.dart"
Roll number is not found
```

In the above program, we have initialized the variable Roll_num with the value of 90014. The switch test-expression checked all cases which are declared inside the switch statement. The test-expression did not find the match in cases; then it printed the default case statement.

Benefit of Switch case

As we discussed above, the switch case is a simplified form of if nested if-else statement. The problem with the nested if-else, it creates complexity in the program when the multiple paths increase. The switch case reduces the complexity of the program. It enhances the readability of the program.

Dart Loops

What is Loop?

A loop is a fundamental control structure in programming that allows a block of code to be executed repeatedly. It helps automate repetitive tasks and perform operations on a set of data or iterate over a collection of elements.

Loops typically consist of three main components:

1. **Initialization**: It initializes the loop control variable(s) or sets up any required variables before entering the loop.

2. **Condition**: It defines the condition that is checked before each iteration. If the condition evaluates to true, the loop body is executed. If the condition evaluates to false, the loop terminates.

3. **Iteration**: It specifies the actions to be performed within the loop body during each iteration. It typically includes statements that update the loop control variable(s) or perform other necessary operations.

There are different types of loops available in programming languages, including:

1. **For Loop**: It is used when you know the number of iterations in advance and provides a compact way to express looping with an initialization, condition, and iteration expression.

2. **While Loop**: It continues executing the loop body as long as a given condition remains true. The condition is evaluated before each iteration.

3. **Do-While Loop**: It is similar to the while loop but guarantees that the loop body is executed at least once before checking the condition.

Loops provide a powerful way to handle repetitive tasks and iterate over data structures, such as arrays or lists. They are essential for implementing algorithms, processing collections, and performing tasks that require repetitive operations.

It's important to ensure that the loop termination condition is properly defined to avoid infinite loops, which can lead to programs getting stuck or consuming excessive resources.

Dart Loops

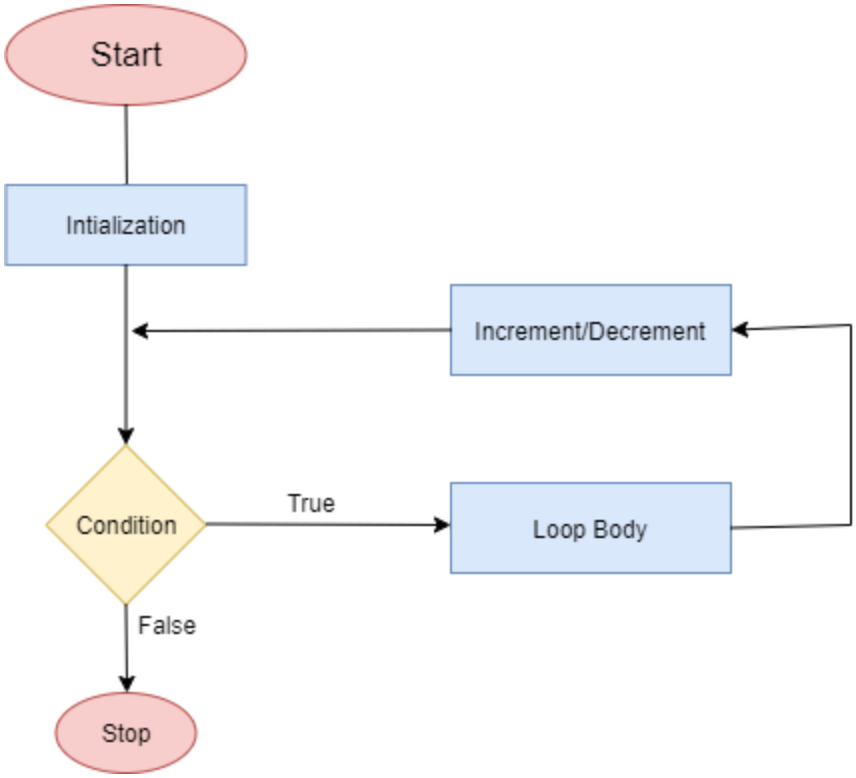
Dart Loop is used to run a block of code repetitively for a given number of times or until it matches the specified condition. Loops are essential tools for any programming language. It is used to iterate the Dart iterable such as list, map, etc. and perform operations for multiple times. A loop can have two parts - a body of the loop and control statements. The main objective of the loop is to run the code multiple times. Dart supports the following type of loops.

- Dart for loop
- Dart for...in loop
- Dart while loop
- Dart do-while loop

Dart for loop

Dart for loop is used when we are familiar with the number of execution of a block of code. It is similar to the C, C++, and Java for loop. It takes an initial variable to start the loop execution. It executes a block of code until it matches the specified condition. When the loop is executed, the value of the iterator is updated each iteration, and then the test-expression is evaluated. This process will continue until the given test-expression is true. Once the test-expression is false, the for loop is terminated.

Dart for Loop Flowchart



Here's how the for loop is structured in Dart:

```
for (initialization; condition; increment/decrement) {
  // Code to be executed
}
```

Let's break down the different parts of the for loop:

1. **Initialization**: It's where you initialize a loop control variable to a starting value before the loop begins. This variable is often used to keep track of the loop's progress.

2. **Condition**: It defines the condition that is checked before each iteration of the loop. As long as the condition evaluates to true, the loop will continue executing. If the condition becomes false, the loop will terminate.

3. **Increment/Decrement**: It specifies how the loop control variable is updated after each iteration. It's used to control the progress of the loop and eventually make the condition evaluate to false to end the loop.

Now, let's see an example to make it clearer:

```dart
// Dart forloop
for (int i = 1; i <= 5; i++) {
  print(i);
}
```

In this example, we use a for loop to print the numbers from 1 to 5. Here's how the loop works:

1. **Initialization**: int i = 1 sets the initial value of i to 1.

2. **Condition**: **i <= 5** checks if **i** is less than or equal to 5. As long as this condition is true, the loop will continue executing.

3. **Code Execution**: The code block inside the loop, in this case, **print(i)**, is executed for each iteration of the loop. In our example, it prints the value of **i**.

4. **Increment**: **i++** increments the value of **i** by **1** after each iteration.

5. **Condition Check**: After executing the code block and updating it, the loop checks the condition again. If **i** is still less than or equal to 5, the loop continues executing. Otherwise, the loop terminates.

Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\switch.dart"
1
2
3
4
5
```

As you can see, the **for** loop executes the code block five times, printing the numbers from 1 to 5.

That's the basic idea behind the Dart **for** loop. You can modify the initialization, condition, and increment/decrement expressions to fit your specific needs and perform different operations within the loop.

We can skip the initial value from the for loop.

Consider the following example.

```dart
var i = 1;

//for loop iteration skipping the initial value from for loop
for(; i <= 5;i++)
{
  print(i);
}
```

It will give the same output as previous code.

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\switch.dart"
1
2
3
4
5
```

Also, we can skip the condition, increment, or decrement by using a semicolon

Nested ForLoop

The nested for loop means, "the for loop inside another for loop". A for inside another loop is called an inner loop and outside loop is called the outer loop. In each iteration of the outer loop, the inner loop will iterate for the entire its cycle.

It allows you to iterate over multiple dimensions or levels of data. In simple terms, a nested for loop is a loop inside another loop.

The outer loop controls the overall iteration, while the inner loop executes repeatedly for each iteration of the outer loop. This creates a hierarchical structure where the inner loop completes its entire iteration for each iteration of the outer loop.
Here's an example to illustrate a nested for loop:

```
for (int i = 1; i <= 3; i++) {
  for (int j = 1; j <= 3; j++) {
    print('Outer loop: $i, Inner loop: $j');
  }
}
```

In this example, we have an outer for loop and an inner for loop. The outer loop iterates from 1 to 3, and for each iteration of the outer loop, the inner loop iterates from 1 to 3 as well. For every combination of the outer and inner loop values, it prints a message indicating the current iteration of each loop.

Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\switch.dart"
Outer loop: 1, Inner loop: 1
Outer loop: 1, Inner loop: 2
Outer loop: 1, Inner loop: 3
Outer loop: 2, Inner loop: 1
Outer loop: 2, Inner loop: 2
Outer loop: 2, Inner loop: 3
Outer loop: 3, Inner loop: 1
Outer loop: 3, Inner loop: 2
Outer loop: 3, Inner loop: 3
```

As you can see, the inner loop completes its entire iteration for each iteration of the outer loop. This allows you to perform more complex operations that involve combinations of different values of multidimensional data structures.

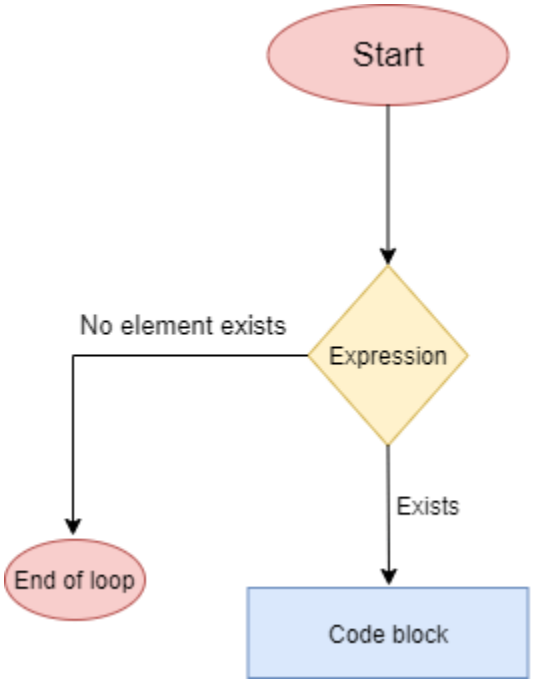
Nested for loops can be extended to more levels if needed. Just remember to manage the loop control variables appropriately for each level, and be mindful of the execution order and termination conditions to avoid infinite loops.

Nested for loops are commonly used in tasks such as working with multi-dimensional arrays, generating combinations or permutations, and traversing complex data structures.

Dart for..in Loop

The for..in loop in Dart provides a convenient way to iterate over the elements of an iterable object, such as lists, sets, or maps. It allows you to access each element directly without worrying about managing index values or explicit loop counters.

Dart For In Loop Flow Diagram



Here's the syntax of the for..in loop in Dart:

```
for (var element in iterable) {
  // Code to be executed
}
```

Let's break down the components of the **for..in** loop:

1. **element**: It represents a variable that holds the value of each element in the iterable. You can choose any valid variable name to represent the current element.

2. **iterable**: It refers to an object that provides a sequence of values. It can be a list, set, map, or any other iterable object.

The **for..in** loop automatically iterates over the elements of the iterable object, assigning each element to the **element** variable. The loop continues until all elements of the iterable have been processed.

Here's an example that demonstrates the usage of the **for..in** loop:

```dart
Run | Debug
void main() {
  List<String> fruits = ['apple', 'banana', 'orange'];

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

In this example, we have a list of fruits, and we use a **for..in** loop to iterate over each fruit in the list. The loop assigns each **fruit** to the fruit variable, and we simply print it.

Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\loops.dart"
apple
banana
orange
```

As you can see, the **for..in** loop iterates over each element in the **fruits** list and prints them.

The **for..in** loop is particularly useful when you want to process all elements of an iterable without worrying about the specific indices or managing a loop counter. It provides a cleaner and more concise syntax for iterating over collections.

Note that the **for..in** loop works only on iterable objects and not on traditional arrays or objects that do not implement the **Iterable** interface.

Let's Take Another Example:

```
Run | Debug
void main() {

  var list1 = [10,20,30,40,50];
   // create an integer variable
   int sum = 0;
   print("Dart for..in loop Example");

   for(var i in list1) {
    // Each element of iterator and added to sum variable.
    sum = i+ sum;
  }
  print("The sum is : ${sum}");

}
```

Explanation:

In the above example, we have declared a variable **sum** with value 0. We have a for..in loop with the iterator list, and each element of the list added to the sum variable after each iteration.

In the first iteration, the value of the sum is equal to 10. In the next iteration, the value of sum became 30 after adding 20 to it. After completing the iteration, it returned the sum of all elements of the list.

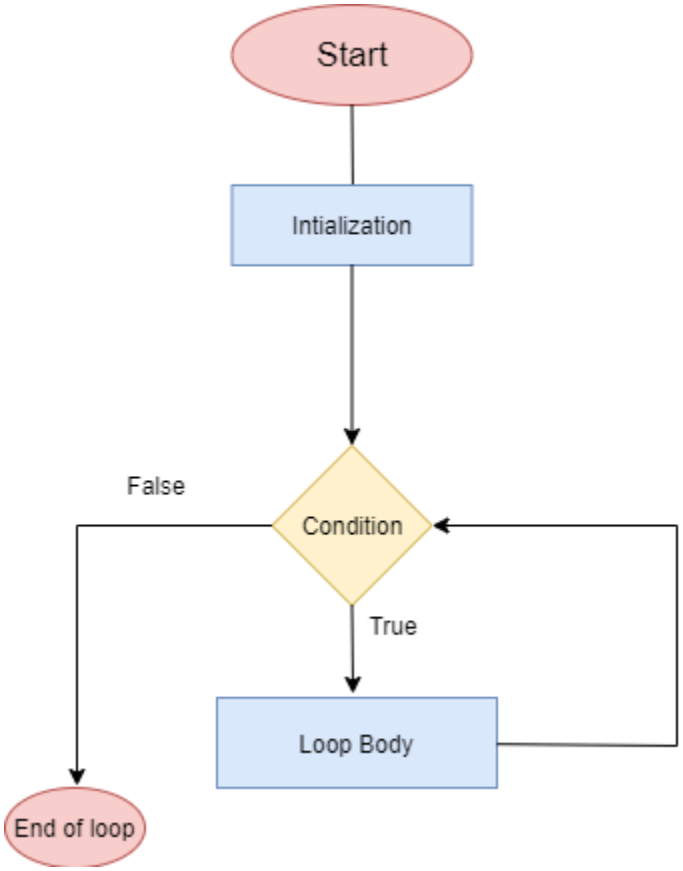
Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\loops.dart"
Dart for..in loop Example
The sum is : 150
```

Dart While Loop

The while loop is used when the number of execution of a block of code is not known. It will execute as long as the condition is true. It initially checks the given condition then executes the statements that are inside the while loop. The while loop is mostly used to create an infinite loop.

Dart While Loop Flowchart



Here's the basic syntax of the while loop in Dart:

```
while (condition) {
  // Code to be executed
}
```

Let's break down the different parts of the while loop:

1. **Condition**: It specifies the condition that is evaluated before each iteration of the loop. As long as the condition remains **true**, the loop body is executed. If the condition becomes **false**, the loop terminates, and the program continues with the next statement after the loop.

2. **Code Execution**: The code block inside the loop is executed only if the condition is initially **true** and remains **true** before each iteration. It can contain any valid Dart statements or a combination of statements.

Here's an example that demonstrates the usage of the **while** loop:

```
Run | Debug
void main() {
  int count = 1;

  while (count <= 5) {
    print(count);
    count++;
  }
}
```

In this example, we have a variable **count** initialized to 1. The **while** loop checks the condition **count <= 5** before each iteration. If the condition is **true**, the loop body is executed. Inside the loop, we print the value of count and increment it by 1 (**count++**). The loop continues as long as **count** is less than or equal to 5.

Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\loops.dart"
1
2
3
4
5
```

As you can see, the **while** loop executes the code block repeatedly until the condition **count <= 5** becomes **false**. It prints the numbers from 1 to 5.

It's important to ensure that the condition in a **while** loop eventually becomes **false** to avoid an infinite loop. If the condition never evaluates to **false**, the loop will continue indefinitely, and the program will not proceed beyond the loop.

The **while** loop provides flexibility in controlling the flow of execution based on a specific condition. It's commonly used when you want to repeatedly execute code until a certain condition is no longer satisfied.

Logical Operator while loop

In Dart, logical operators can be used within a while loop to control the loop's behavior based on multiple conditions. The logical operators allow you to combine or modify conditions to determine whether the loop should continue or terminate.

The three main logical operators in Dart are:

1. **&& (AND)**: Returns true if both conditions on either side of the operator are true.

2. **|| (OR)**: Returns true if at least one of the conditions on either side of the operator is true.

3. **! (NOT)**: Negates the condition, returning true if the condition is initially false, and vice versa.

Let's see an example that demonstrates the usage of logical operators within a while loop:

```
Run | Debug
void main() {
  int number = 1;

  while (number <= 10 && number % 2 == 0) {
    print(number);
    number++;
  }
}
```

In this example, the while loop has two conditions combined using the && (AND) operator:

1. **number <= 10**: The loop will continue as long as the number is less than or equal to 10.

2. **number % 2 == 0**: The loop will continue if the number is an even number (divisible by 2).

The loop body will only execute if both conditions are true. If either condition evaluates to false, the loop will terminate.

Output:

(no output, as the conditions are not initially satisfied)

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\loops.dart"
```

Since the initial value of **number** is 1, which is not an even number, the loop conditions are not met, and the loop body does not execute.

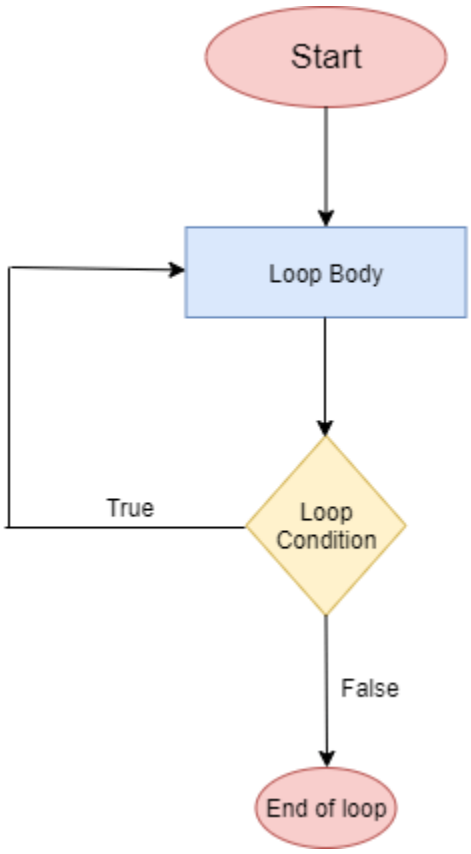
By modifying the conditions or using different logical operators (**||, !**), you can control the flow of execution within the **while** loop based on specific conditions or combinations of conditions.

Remember to ensure that the loop conditions eventually become **false** to prevent an infinite loop that can lead to the program getting stuck.

Dart do while loop

In Dart, the do-while loop is a control structure that executes a block of code once before checking the loop condition. It guarantees that the loop body is executed at least once, even if the condition is initially false. After executing the code block, the loop condition is evaluated, and if it's true, the loop continues executing. If the condition is false, the loop terminates.

Dart do-while loop Flowchart



Here's the basic syntax of the do-while loop in Dart:

```
void main(){
  do {
    // Code to be executed
  } while (condition);
}
```

Let's break down the different parts of the do-while loop:

1. **Code Execution**: The code block inside the loop is executed once before checking the condition. It can contain any valid Dart statements or a combination of statements.

2. **Condition**: It specifies the condition that is checked after executing the code block. If the condition is **true**, the loop continues executing. If the condition is **false**, the loop terminates, and the program continues with the next statement after the loop.

Here's an example that demonstrates the usage of the **do-while** loop:

```
Run | Debug
void main() {
  int count = 1;

  do {
    print(count);
    count++;
  } while (count <= 5);
}
```

In this example, we have a variable **count** initialized to 1. The **do-while** loop executes the code block inside the loop, which prints the value of **count** and increments it by 1 (**count++**). After each iteration, the loop checks the condition **count <= 5.** If the condition is **true**, the loop continues executing. If the condition becomes **false**, the loop terminates.

Output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\loops.dart"
1
2
3
4
5
```

As you can see, the **do-while** loop executes the code block once, printing the number 1, and then continues executing as long as the condition **count <= 5** remains **true**. It prints the numbers from 1 to 5.

The **do-while** loop is particularly useful when you want to execute a block of code at least once and then continue looping based on a specific condition. It ensures that the code block is executed before checking the loop condition.

Just like with other loop structures, ensure that the loop condition eventually becomes **false** to prevent an infinite loop.

Dart forEach Loop

The forEach loop is a convenient way to iterate over elements in a collection, such as a list or a set, and perform an action on each element. It allows you to execute a specified function for each item in the collection.

Here's a simple example to illustrate how to use the forEach loop in Dart:

```
Run | Debug
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];

  numbers.forEach((number) {
    print(number);
  });
}
```

In the example above:

We have a list of integers called numbers containing [1, 2, 3, 4, 5].
We use the forEach method on the numbers list to iterate over each element.
Inside the forEach method, we provide an anonymous function that takes a single parameter number.
The anonymous function is executed for each element in the numbers list.
In this case, we simply print out each number using the print statement.

When you run this code, it will output:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\loops.dart"
1
2
3
4
5
```

The forEach loop is a concise way to iterate over a collection without explicitly managing the index or using a traditional for loop. It helps make your code more readable and expressive.

Dart Boolean

Dart Boolean data type is used to check whether a given statement is true or false. The true and false are the two values of the Boolean type, which are both compile-time constants. In Dart, The numeric value 1 or 0 cannot be used to specify the true or false. The bool keyword is used to represent the Boolean value. The syntax of declaring the Boolean variable is given below.

Syntax -
bool var_name = true;
OR
bool var_name = false;

```dart
void main() {
  bool check;
  check = 20>10;
  print("The statement is = ${check}");
}
```

Output: The Statement is = True

Break and Continue Statements in Dart

In Dart, the break and continue statements are used to control the flow of loops, such as for loops, while loops, or do-while loops. Here's an explanation of how break and continue work:

break statement:

When a break is encountered inside a loop, it immediately terminates the loop and resumes execution at the next statement after the loop.

It allows you to prematurely exit a loop based on a certain condition or when a specific situation occurs.

Here's an example of using break in a for loop:

```dart
Run | Debug
void main() {
  for (int i = 1; i <= 10; i++) {
    if (i == 5) {
      break; // exit the loop when i equals 5
    }
    print(i);
  }
}
```

The output of this code will be:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart
Master\lib\break_continue.dart"
1
2
3
4
```

continue statement:

When continue is encountered inside a loop, it immediately skips the remaining code in the current iteration and moves to the next iteration of the loop.

It allows you to skip specific iterations or certain code blocks within a loop.

Here's an example of using continue in a for loop:

```dart
Run | Debug
void main() {
  for (int i = 1; i <= 5; i++) {
    if (i == 3) {
      continue; // skip iteration when i equals 3
    }
    print(i);
  }
}
```

The output of this code will be:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart
Master\lib\break_continue.dart"
1
2
4
5
```

Both break and continue statements are useful in controlling the flow of loops based on specific conditions. break allows you to exit the loop entirely, while continue allows you to skip the remaining code within the current iteration and move to the next iteration.

Functions in Dart

Before learning about dart functions let's understand what function is? & why we use functions in any programming language & what are the important characteristics of functions.

What function is?

A function is a self-contained block of code that performs a specific task or a set of instructions. It is a fundamental concept in programming that allows you to encapsulate a sequence of statements and give it a name. Functions can be invoked (called) from different parts of a program, enabling code reuse and modularity.

Functions have the following characteristics:

1. **Name**: A function is identified by a unique name, which is used to call the function when needed.

2. **Parameters**: Functions can accept zero or more parameters (also known as arguments) that allow you to pass values into the function. Parameters are variables that hold the values passed to the function.

3. **Return Value**: A function may return a value after executing its code. The return value represents the result of the function's computation. Not all functions have a return value; some functions may simply perform actions without producing a result.

4. **Code Block**: The code block within a function contains a series of statements that define the behavior of the function. The code block is enclosed in curly braces **{}**.

User defined function and built-in functions

In Dart, you can define your own functions, known as user-defined functions, as well as use built-in functions provided by the Dart language or external libraries. Let's explore both types:

User-Defined Functions:

User-defined functions are functions created by developers to perform specific tasks based on their requirements. You define these functions in your own codebase. Here's an example of a user-defined function in Dart:

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

In this example, greet is a user-defined function that takes a String parameter named name and prints a greeting message using the provided name.

Built-in Functions:

Built-in functions are pre-defined functions that are provided by the Dart language or libraries. They are readily available for you to use without requiring any additional setup or configuration. Built-in functions provide common functionalities and utility operations. Here are a few examples of built-in functions in Dart:

- print(): Prints the specified message to the console.
- sqrt(): Computes the square root of a given number.
- toUpperCase(): Converts a string to uppercase.
- sort(): Sorts a list in ascending order.
- toString(): Converts an object to its string representation.

Built-in functions are part of the Dart language or external libraries. They are typically documented, and you can readily use them by calling their names and providing the required arguments.

It's important to note that Dart also supports higher-order functions and closures. Higher-order functions are functions that can accept other functions as parameters or return functions as results. Closures are functions that can access and modify variables from their surrounding scope.

When programming in Dart, you have the flexibility to create your own functions to address specific needs while utilizing the built-in functions available in the language or libraries to leverage existing functionality and optimize development efforts.

Now we can briefly discuss user-defined functions first then will talk about built-in Functions.

Dart Function

Dart function is a set of codes that together perform a specific task. It is used to break the large code into smaller modules and reuse it when needed. Functions make the program more readable and easy to debug. It improves the modular approach and enhances the code reusability.

Suppose, we write a simple calculator program where we need to perform operations number of times when the user enters the values. We can create different functions for each calculator operator. By using these functions, we don't need to write code for adding, subtracting, multiplying, and dividing again and again. We can use the functions multiple times by calling.

The function provides the flexibility to run a code several times with different values. A function can be called anytime as its parameter and returns some value to where it is called.

The few benefits of the Dart function are given below.

- It increases the module approach to solve the problems.
- It enhances the re-usability of the program.
- We can do the coupling of the programs.
- It optimizes the code.
- It makes debugging easier.
- It makes development easy and creates less complexity.

Let's understand the basic concept of functions in Dart:

Defining a Function

A function can be defined by providing the name of the function with the appropriate parameter and return type. A function contains a set of statements which are called function body.

To define a function in Dart, you need to follow a specific syntax. Here's an example of how to define a function in Dart:

The syntax is given below.

```
returnType functionName(parameter1, parameter2, ...) {
  // Function body
  // Code to be executed
  // Optional return statement
}
```

Let's break down the components of a function definition:

1. **returnType**: Specifies the type of value that the function returns. It can be any valid Dart data type or void if the function doesn't return a value. For example, int, double, String, bool, or a custom-defined class.

2. **functionName**: The name you give to the function. Choose a meaningful and descriptive name that reflects the purpose of the function. The function name follows the Dart naming conventions, starting with a lowercase letter and using camel case (e.g., calculateSum, printMessage).

3. **parameters**: Optional input values that you can pass to the function. Parameters are enclosed in parentheses () and separated by commas. Each parameter consists of a type followed by a name. Parameters allow you to pass values into the function for it to work with. For example, (int a, double b) or (String name, int age).

4. **Function Body**: The code block within the function is enclosed in curly braces {}. It contains the statements and instructions that define the behavior of the function. This is where you write the code that will be executed when the function is called.

5. **Return Statement (optional)**: If the function has a return type other than void, you can use the return statement to specify the value to be returned

from the function. The return statement ends the function's execution and sends the result back to the caller.

Here's a simple example of a function in Dart:

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

In this example:

- The function is named greet.
- It takes a single parameter named type String.
- It uses the print statement to output a greeting message using the provided name.

Calling a Function

You can call (invoke) the greet function like this:

```dart
Run | Debug
void main(List<String> args) {

  greet('Arslan');

}
```

When the function is invoked with the argument 'Arslan',

it will print:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\functions.dart"
Hello, Arslan!
```

Functions help in organizing code, promoting reusability, and making programs easier to understand and maintain. By breaking down tasks into smaller functions, you can write cleaner and more modular code, reducing redundancy and improving code readability.

Passing Arguments to Function

When a function is called, it may have some information as per the function prototype is known as a parameter (argument). The number of parameters passed and data type while the function call must be matched with the number of parameters during function declaration. Otherwise, it will throw an error. Parameter passing is also optional, which means it is not compulsory to pass during function declaration. The parameter can be of two types.

1. **Actual Parameter** - A parameter which is passed during a function definition is called the actual parameter.

2. **Formal Parameter** - A parameter which is passed during a function call is called the formal parameter.

Here's an example of a function in Dart that calculates the sum of two integers:

```
int calculateSum(int a, int b) {
  int sum = a + b;
  return sum;
}
```

In this example, the function is named calculateSum, and it takes two parameters of type int named a and b. The function body calculates the sum of a and b and stores it in the sum variable. Finally, the return statement sends the value of sum back to the caller.

Return a Value from Function

A function always returns some value as a result to the point where it is called. The return keyword is used to return a value. The return statement is optional. A function can have only one return statement.

You can then call this function in your Dart program like this:

```
Run | Debug
void main(List<String> args) {
  int result = calculateSum(3, 4);
  print(result); // Output: 7
}
```

In this example, the calculateSum function is invoked with the arguments 3 and 4, and the returned value is stored in the result variable. The print statement displays the value of result, which is 7 in this case.

we declared a function named **calculateSum()** and passed two integer variables as actual parameters. In the function body,

In order to add two values, we called a function with the same name, passed formal parameters 30 and 20. The **calculateSum()** returned a value which we stored in the variable **result** and printed the **result** on the console.

Dart Function with No Parameter and Return Value

As we discussed earlier, the parameters are optional to pass while defining a function. We can create a function without parameter return value.

The syntax is given below.

```
Run | Debug
void main(){

  // Creating a function without argument
  String greetings(){
    return "Welcome to JavaTpoint";
  }

  // Calling function inside print statement
  print(greetings());

}
```

In the above example, we created a function named greetings() without argument and returned the string value to the calling function. Then, we called the greeting() function inside the print statement and printed the result to the console.

Dart Function with No Parameter and without a Return Value

We can declare a function without parameters and no return value. The syntax is given below.

```
Run | Debug
void main() {
  print("The example of Dart Function");

  // function callling
  greetings();
}


// Creating a function without argument
void greetings()  {
  print("Welcome to JavaTpoint");
}
```

In the above example, we created a function called greeting() outside the main() function and wrote the print statement. Inside the main() function, we called the defined function and printed the output to the console.

Positional Parameter

In Dart, positional parameters are a way to pass arguments to a function based on their position or order. Positional parameters are defined without specifying their names and are accessed in the function based on their position in the argument list.

Here's an example that demonstrates the usage of positional parameters:

```dart
void printOrderDetails(String productName, int quantity, double price) {
  print('Product: $productName');
  print('Quantity: $quantity');
  print('Price: \$${price.toStringAsFixed(2)}');
}

Run | Debug
void main() {
  printOrderDetails('Book', 2, 19.99);
}
```

In this example, the **printOrderDetails** function takes three positional parameters: **productName**, **quantity**, and **price**. When calling the function in the **main** function, the arguments are provided in the same order as the parameters.

The output of the above code will be:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\functions.dart"
Product: Book
Quantity: 2
Price: $19.99
```

Positional parameters allow you to pass arguments to a function without explicitly specifying their names. However, it's important to ensure that the order of the arguments matches the order of the parameters defined in the function.

Optional Positional Parameter

In Dart, you can define optional positional parameters in a function, which allows you to pass arguments to the function without requiring them to be provided in a specific order. Optional positional parameters are enclosed within square brackets [] in the function declaration.

Here's an example that demonstrates the usage of optional positional parameters:

```dart
void printOrderDetails(String productName, [int? quantity, double? price]) {
  print('Product: $productName');
  if (quantity != null) {
    print('Quantity: $quantity');
  }
  if (price != null) {
    print('Price: \$${price.toStringAsFixed(2)}');
  }
}

Run | Debug
void main() {
  printOrderDetails('Book');
  printOrderDetails('Pen', 5);
  printOrderDetails('Notebook', 3, 7.99);
}
```

In this example, the printOrderDetails function has one required positional parameter productName, followed by two optional positional parameters quantity and price. The optional parameters are enclosed within square brackets [].

When calling the printOrderDetails function, you can omit the optional parameters or provide them in any order you prefer. If an optional parameter is not provided, its value will be null.

By using int? and double? as the parameter types, you indicate that the parameters can accept null values. Now, the code should execute without errors, and the output will be as expected:

The output of the above code will be:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart Master\lib\functions.dart"
Product: Book
Product: Pen
Quantity: 5
Product: Notebook
Quantity: 3
Price: $7.99
```

As you can see, the function can be called with different combinations of arguments, and the output adjusts accordingly.

Optional positional parameters provide flexibility in function invocation by allowing you to omit certain arguments or provide them in any order, making your code more versatile and accommodating different use cases.

Default Value Parameter

In Dart, you can define default values for function parameters, which are used when the caller does not provide a value for those parameters. Default values are specified using the = sign in the function declaration.

Here's an example that demonstrates the usage of default value parameters:

```dart
// Default Value Parameter
void printOrderDetails(String productName, {int quantity = 1, double price = 0.0}) {
  print('Product: $productName');
  print('Quantity: $quantity');
  print('Price: \$${price.toStringAsFixed(2)}');
}

Run | Debug
void main() {
  printOrderDetails('Book');
  printOrderDetails('Pen', quantity: 5);
  printOrderDetails('Notebook', quantity: 3, price: 7.99);
}
```

In this example, the printOrderDetails function has a required named parameter productName and two optional named parameters quantity and price. The default values for quantity and price are specified as 1 and 0.0, respectively.

When calling the printOrderDetails function, you can omit the optional parameters or provide them using the named parameter syntax (parameterName: value). If you omit an optional parameter, its default value will be used.

The output of the above code will be:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart
Product: Book
Quantity: 1
Price: $0.00
Product: Pen
Quantity: 5
Price: $0.00
Product: Notebook
Quantity: 3
Price: $7.99
```

As you can see, the default values are used when the caller does not provide specific values for the optional parameters.

Default value parameters provide a convenient way to define fallback values for optional parameters in functions. They allow you to provide sensible defaults that are used when the caller doesn't explicitly specify values, making your code more flexible and robust.

Named Parameter

In Dart, named parameters allow you to pass arguments to a function by specifying the parameter names along with their corresponding values. Named parameters are enclosed within curly braces {} in the function declaration.

Here's an example that demonstrates the usage of named parameters:

```dart
void printOrderDetails({String? productName, int? quantity, double? price}) {
  print('Product: $productName');
  print('Quantity: $quantity');
  print('Price: \$${price?.toStringAsFixed(2)}');
}

Run | Debug
void main() {
  printOrderDetails(productName: 'Book', quantity: 2, price: 19.99);
}
```

In this example, the printOrderDetails function has named parameters: productName, quantity, and price. Each parameter is preceded by its name and followed by its type.

When calling the printOrderDetails function, you provide the arguments using the named parameter syntax (parameterName: value). This allows you to pass arguments in any order, omit optional parameters, and make the code more readable.

The output of the above code will be:

```
[Running] dart "c:\Users\Muhammad Arslan\Desktop\Dart
Product: Book
Quantity: 2
Price: $19.99
```

As you can see, the output remains the same even if the order of the named parameters is changed.

Named parameters provide flexibility in function invocation by allowing you to explicitly specify which parameter each argument corresponds to. They enhance

code readability and make it easier to understand the purpose of the passed values, especially when a function has many parameters or optional arguments.

Additionally, named parameters can have default values assigned to them. This means you can define a fallback value that will be used if the caller doesn't provide a value for that parameter. To assign default values to named parameters, you use the = sign in the function declaration. For example:

```dart
void printOrderDetails({String? productName, int quantity = 1, double price = 0.0}) {
  // Function body
}
```

With default values assigned, you can omit those parameters when calling the function, and the default values will be used.

Anonymous Functions (Lambda Functions)

In Dart, anonymous functions, also known as lambda functions or function literals, allow you to define functions without explicitly giving them a name. They are commonly used for shorter, one-time functions or as arguments to other functions. Anonymous functions can be defined using the **() {}** syntax.

Here's an example that demonstrates the usage of anonymous functions:

```dart
Run | Debug
void main() {

  // Example 1: Anonymous function assigned to a variable
  var multiply = (int a, int b) {
    return a * b;
  };

  print(multiply(3, 5));  // Output: 15

}
```

In Example 1, an anonymous function is assigned to the variable multiply. The function takes two parameters a and b, and it returns their product. The multiply variable can then be invoked as a function with the arguments (3, 5).

```
Run | Debug
void main() {

  // Example 2: Anonymous function as an argument to another function
  void performOperation(int x, int y, Function operation) {
    int result = operation(x, y);
    print('Result: $result');
  }

  performOperation(10, 5, (a, b) => a + b);  // Output: Result: 15

}
```

In Example 2, the performOperation function accepts three parameters: x, y, and operation, where operation is a function. Inside performOperation, the operation function is invoked with the arguments x and y, and the result is printed.

Anonymous functions are useful when you need to define a small function on the fly or pass a function as an argument without the need for a named function declaration. They provide flexibility and allow for concise code when working with functions in Dart.

Arrow Function

In Dart, an arrow function, also known as a fat arrow function or a lambda function, is a concise syntax for defining anonymous functions. Arrow functions provide a more compact and readable way to express short, single-expression functions. They are defined using the => operator.

Here's an example that demonstrates the usage of arrow functions:

```dart
Run | Debug
void main() {
  // Example 1: Arrow function assigned to a variable
  var multiply = (int a, int b) => a * b;
  print(multiply(3, 5));  // Output: 15

  // Example 2: Arrow function as an argument to another function
  void performOperation(int x, int y, Function operation) {
    int result = operation(x, y);
    print('Result: $result');
  }

  performOperation(10, 5, (a, b) => a + b);  // Output: Result: 15
}
```

In Example 1, an arrow function is assigned to the variable multiply. The function takes two parameters a and b, and it returns their product using the expression a * b. The multiply variable can be invoked as a function with the arguments (3, 5).

In Example 2, the performOperation function accepts three parameters: x, y, and operation, where operation is a function. Inside performOperation, the operation function is invoked with the arguments x and y, and the result is printed.

Arrow functions are particularly useful when the function body consists of a single expression. They eliminate the need for curly braces {} and the return keyword. The expression on the right side of the => is automatically returned.

Arrow functions provide a more concise and expressive way to define anonymous functions, making the code more readable and reducing the boilerplate code.

Lexical Scope

```dart
Run | Debug
void main(List<String> args) {

  var personName = "Arslan";

  void outerFunction() {

    var personName = "Ibraheem";
    print(personName);

    void innerFunction(){

      var personName = "Adnan";
      print(personName);
    }
    innerFunction();

  }

  outerFunction();

  print(personName);

}
```

In this example, we have three nested functions: main, outerFunction, and innerFunction. Each function defines its own lexical scope.

- The main function is the entry point of the Dart program. It declares a variable personName and assigns it the value "Arslan". This variable is in the scope of the main function.

- Inside the main function, there is an outerFunction. It also declares a variable personName and assigns it the value "Ibraheem". This variable is in the scope of the outerFunction.

- The outerFunction prints the value of its personName variable, which is "Ibraheem". It then declares an innerFunction.

- Inside the innerFunction, there is another variable personName assigned the value "Adnan". This variable is in the scope of the innerFunction.

- The innerFunction prints the value of its personName variable, which is "Adnan".

- After defining the innerFunction, it is called within the outerFunction. This results in the output "Adnan".

- After the execution of the innerFunction, the outerFunction continues executing and finishes. It then prints the value of its personName variable, which is "Ibraheem".

- Finally, outside the outerFunction, the value of the personName variable from the main function is printed, which is "Arslan".

The code demonstrates the concept of variable shadowing, where variables with the same name are declared in different scopes. The variable with the narrowest scope takes precedence over variables with the same name in outer scopes. As a result, when accessing personName within the functions, the values from the respective scopes are printed.

The example demonstrates how lexical scoping allows variables to be accessed based on their defined scope hierarchy, providing encapsulation and control over variable visibility within nested scopes.

What is Recursion

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem by breaking it down into smaller subproblems. In other words, a recursive function is a function that solves a problem by reducing it to smaller instances of the same problem.

Here are the key characteristics of recursion:

1. **Base Case**: Recursive functions have a base case, which is the condition that determines when the function should stop calling itself. It acts as the termination condition for the recursive process and prevents infinite recursion.

2. **Recursive Case**: Recursive functions also have a recursive case, which is the part of the function where it calls itself to solve a smaller instance of the problem. By solving the smaller instance, the function progresses towards reaching the base case.

3. **Stack Frames**: Each time a recursive function calls itself, a new instance of the function is added to the call stack. The call stack keeps track of the sequence of function calls and their respective variables. As the recursive function reaches the base case, the stack starts unwinding, and the function calls are resolved one by one.

Recursion can be a powerful and elegant solution for solving problems that exhibit self-similarity or can be divided into smaller subproblems. It allows you to express complex problems concisely and intuitively by leveraging the concept of "divide and conquer."

Dart Recursive Function

Recursive functions are quite similar to the other functions, but the difference is in calling itself recursively. A recursive function repeats multiple times until it returns the final output. It allows programmers to solve complex problems with minimal code.

Here's a simple example of a recursive function that calculates the factorial of a number:

```dart
// Recursion
int factorial(int n) {
  if (n == 0) {
    return 1;  // Base case: factorial of 0 is 1
  } else {
    return n * factorial(n - 1);  // Recursive case: n! = n * (n-1)!
  }
}

Run | Debug
void main() {
  int result = factorial(5);
  print(result);  // Output: 120
}
```

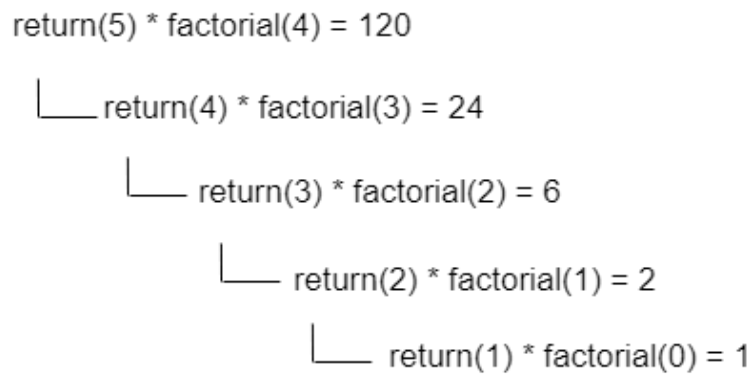
In this example, the **factorial** function calculates the factorial of a number **n**. It calls itself recursively, reducing the problem to a smaller instance by computing the factorial of (**n - 1**). The function reaches the base case when **n** becomes 0, and it returns 1 to start unwinding the stack.

Recursion can be a powerful tool in solving problems, but it requires careful consideration of the base case and ensuring that the recursive calls eventually reach the base case. It's important to handle recursion correctly to avoid infinite loops and stack overflow errors.

How does recursion work?

Let's understand the concept of the recursion of the example of a factorial of a given number. In the following example, we will evaluate the factorial of n numbers. It is the series of the multiplication as follows.

Factorial of n (n!) = n*(n-1)*(n-2).....1



Characteristics of Recursive Function

The characteristics of the recursive function are given below.

- A recursive function is a unique form of a function where the function calls itself.
- A valid base case is required to terminate the recursive function.
- It is slower than the iteration because of stack overheads.

Let's have a look at recursion syntax:

```
void recurse() {
  //statement(s)
 recurse();
//statement(s);
}
void main(){
   //statement(s)
  recurse();
 //statement(s)
}
```

Let's understand the following example.

Example - 1

```dart
int factorial(int num){

  //base case of recursion.
  if(num<=1) { // base case
        return 1;
}
  else{
        return num*factorial(num-1);    //function call itself.
 }
}
Run | Debug
void main() {
  var num = 5;
  // Storing function call result in fact variable.
  var fact = factorial(num);
  print("Factorial Of 5 is: ${fact}");
}
```

In the above example, the factorial() is a recursive function as it calls itself. When we call the factorial() function by passing the integer value 5, it will recursively call itself by decreasing the number.

The factorial() function will be called every time until it matches the base condition, or it is equal to one. It multiplied the number with the factorial of the number. Consider the following explanation of the recursive call.

1. factorial(5) # 1st call with 5
2. 5 * factorial(4) # 2nd call with 4
3. 5 * 4 * factorial(3) # 3rd call with 3
4. 5 * 4 * 3 * factorial(2) # 4th call with 2
5. 5 * 4 * 3 * 2 * 1 # **return** from 2nd call
6. 120 # **return** from 1st call

The recursion is ended when the number is reduced to 1, and it is the base condition of recursion.

A recursion function must have a base condition to avoid an infinite call.

Disadvantage of Recursion

- The recursive calls consume a lot of memory; that's why these are inefficient.
- The recursive functions are difficult to debug.
- Sometimes, It is hard to follow the logic behind the recursion.

Object oriented Programming

What is OOP?

Object-oriented programming is a way of organizing code and solving problems by thinking about objects and their interactions. In the real world, objects are things you can see and interact with. For example, think about a car, a dog, or a smartphone. Each of these objects has certain characteristics (properties) and can perform certain actions (behaviors).

Let's take the example of a car to understand OOP:

1. **Properties**: Properties describe the characteristics or attributes of an object. In the case of a car, properties can include its color, brand, model, and speed.

   For example, if we have a car object named "myCar," its properties could be:

   Color: Red
   Brand: Toyota
   Model: Camry
   Speed: 60 km/h

2. **Behaviors**: Behaviors represent the actions an object can perform. In the case of a car, behaviors can include starting the engine, accelerating, braking, and changing gears.

   For example, the behaviors of the "myCar" object could be:

   Start the engine
   Accelerate
   Brake
   Change gears

In OOP, we create classes that act as blueprints for objects. A class defines what properties an object will have and what actions it can perform. Using the car example, we can create a class called "Car" with properties like color, brand, model, and speed, and methods (functions) like startEngine(), accelerate(), brake(), and changeGears().

Once we have defined the class, we can create multiple car objects (instances) based on that class. Each object will have its own set of property values, but they will share the same behaviors defined by the class.

For instance, we can create two car objects named "myCar" and "friendCar" from the "Car" class. "myCar" can be a red Toyota Camry, while "friendCar" can be a blue Honda Accord. Both cars can start the engine, accelerate, brake, and change gears.

OOP allows us to model and organize code in a way that mirrors real-world objects and their relationships. It promotes reusability, modularity, and makes code easier to understand and maintain.

Classes and Object

Let's explore the concepts of classes and objects, including class definition, object creation, instance variables, and methods, using a meaningful example.

Example 1 :

Imagine we're building a virtual pet simulation game. In this game, we can create and interact with different types of pets, each having its own characteristics and behaviors.

Class Definition

We start by defining a class called "Pet" that will serve as the blueprint for creating pet objects. The "Pet" class will have properties and methods that all pets share.

```dart
class Pet {
  String ? name;
  int ? age;
  String ? breed;

  void eat() {
    print('$name is eating.');
  }

  void sleep() {
    print('$name is sleeping.');
  }
}
```

In the above example, the "Pet" class has properties like name, age, and breed. It also has methods like **eat()** and **sleep()** that represent common behaviors of pets.

Object Creation

Now, let's create specific pet objects from the "Pet" class. For instance, we can create a dog named "Buddy" and a cat named "Whiskers" as follows:

```dart
Run | Debug
void main() {
  var buddy = Pet();
  buddy.name = 'Buddy';
  buddy.age = 3;
  buddy.breed = 'Golden Retriever';

  var whiskers = Pet();
  whiskers.name = 'Whiskers';
  whiskers.age = 2;
  whiskers.breed = 'Siamese';

  print('${buddy.name} is a ${buddy.breed} and is ${buddy.age} years old.');
  buddy.eat(); // Buddy is eating.

  print('${whiskers.name} is a ${whiskers.breed} and is ${whiskers.age} years old.');
  whiskers.sleep(); // Whiskers is sleeping.
}
```

In the above code, we create two pet objects: buddy and whiskers. We set the properties of each object to define their specific characteristics.

Let's Take another example:

Consider a scenario where we are building a banking application. We want to represent bank accounts as objects, each having properties like an account number, account holder name, and balance. We'll also define methods to deposit and withdraw funds from these accounts.

Class Definition:

We start by defining a class called "BankAccount" that represents a bank account. The class will have properties and methods associated with bank accounts.

```
class BankAccount {
  String  accountNumber  = "";
  String  accountHolderName = "";
  double  balance = 0.0;

  void deposit(double amount) {
    balance += amount;
    print('Amount $amount deposited successfully.');
  }

  void withdraw(double amount) {
    if (balance >= amount) {
      balance -= amount;
      print('Amount $amount withdrawn successfully.');
    } else {
      print('Insufficient balance.');
    }
  }
}
```

In the above example, the "BankAccount" class has properties like accountNumber, accountHolderName, and balance. It also has methods like deposit() and withdraw() to interact with the account.

Object Creation:

Now, let's create specific bank account objects from the "BankAccount" class. For example, we can create two bank account objects for different account holders.

```dart
Run | Debug
void main() {
  var account1 = BankAccount();
  account1.accountNumber = '123456';
  account1.accountHolderName = 'Arslan Yousaf';
  account1.balance = 5000;

  var account2 = BankAccount();
  account2.accountNumber = '789012';
  account2.accountHolderName = 'Kamran Khan';
  account2.balance = 10000;

  print('${account1.accountHolderName} has a balance of ${account1.balance}.');
  account1.deposit(2000); // Amount 2000 deposited successfully.

  print('${account2.accountHolderName} has a balance of ${account2.balance}.');
  account2.withdraw(5000); // Amount 5000 withdrawn successfully.
}
```

In the above code, we create two bank account objects: account1 and account2. We set the properties of each object to define the account number, account holder name, and initial balance.

1. Instance Variables:

   Instance variables are the properties of an object that hold its state or data. In our example, accountNumber, accountHolderName, and balance are instance variables of the "BankAccount" class.

   Instance variables are specific to each object. When we create a bank account object, it will have its own set of instance variable values.

2. Methods:

   Methods define the actions or behaviors that an object can perform. In our "BankAccount" class, we defined two methods: deposit() and withdraw().

   The deposit() method takes an amount as a parameter and adds that amount to the account's balance. The withdraw() method also takes an amount as a parameter but checks if the account has sufficient balance before subtracting the amount.

   When we call these methods on the bank account objects, they perform the corresponding actions on that specific account.

The combination of class definitions, object creation, instance variables, and methods allows us to create and interact with objects that represent bank accounts in our banking application.

Constructor

Let's explore constructors in Dart, including the default constructor, parameterized constructor, named constructors, and constructor chaining, with a meaningful example.

Consider a scenario where we are building a messaging application. We want to represent a message as an object, each having properties like the sender, recipient, and message content. We'll define different constructors to create message objects in various ways.

Default Constructor:

A default constructor is automatically provided by Dart if we don't explicitly define any constructors. It initializes the object with default values or null for each property.

```dart
class Message {
  String ? sender;
  String ? recipient;
  String ? content;

  Message() {
    print('Default constructor called.');
  }
}

Run | Debug
void main() {
  var message = Message(); // Default constructor
  print(message);
}
```

In the above example, the "Message" class has three properties: sender, recipient, and content. We haven't defined a constructor explicitly, so Dart provides a default constructor.

Parameterized Constructor:

A parameterized constructor allows us to define custom parameters during object creation and use those values to initialize the object's properties.

```dart
class Message {
  String sender;
  String recipient;
  String content;

  Message(this.sender, this.recipient, this.content) {
    print('Parameterized constructor called.');
  }
}

Run | Debug
void main() {
  var message = Message('Ali', 'Akmal', 'Hello!'); // Parameterized constructor
  print(message);
}
```

In this example, we define a class Message with a parameterized constructor. The constructor takes three parameters (sender, recipient, and content) and initializes the corresponding properties using this keyword. When we create an object using Message(Ali, 'Akmal, 'Hello!'), the parameterized constructor is called and the associated print statement is executed.

Named Constructors:

Dart allows us to define named constructors, which have a name different from the class name. Named constructors provide an alternative way to create objects with specific initialization logic.

```dart
class Message {
  String ? sender;
  String ? recipient;
  String ? content;

  Message.fromSender(this.sender) {
    print('Named constructor called: fromSender.');
  }

  Message.fromRecipient(this.recipient) {
    print('Named constructor called: fromRecipient.');
  }
}

Run | Debug
void main() {
  var message1 = Message.fromSender('Arslan'); // Named constructor: fromSender
  var message2 = Message.fromRecipient('Ali'); // Named constructor: fromRecipient

  print(message1);
  print(message2);
}
```

In this example, we define a class Message with two named constructors: fromSender and fromRecipient. Each named constructor takes a single parameter and initializes the corresponding property. When we create objects using the named constructors (Message.fromSender(‘Arslan’) and Message.fromRecipient('Ali')), the respective named constructors are called, and the associated print statements are executed.

Constructor Chaining:

Dart allows constructors to call other constructors within the same class using this keyword. This is known as constructor chaining and helps avoid code duplication.

```dart
class Message {
  String sender;
  String recipient;
  String content;

  Message(this.sender, this.recipient, this.content) {
    print('Parameterized constructor called.');
  }

  Message.empty() : this('', '', '');
}

Run | Debug
void main() {
  var message = Message.empty(); // Constructor chaining
  print(message);
}
```

In this example, we define a class Message with a parameterized constructor and a named constructor empty. The named constructor empty uses constructor chaining by calling the parameterized constructor with empty values. When we create an object using Message.empty(), the named constructor empty is called, which then calls the parameterized constructor. The associated print statements for both the named and parameterized constructors are executed.

Inheritance

Inheritance is a concept in object-oriented programming where one class can inherit or acquire properties and behaviors from another class. The class that is inherited from is called the "superclass" or "base class," and the class that inherits from it is called the "subclass" or "derived class."

Think of inheritance as a parent-child relationship, where the parent (superclass) passes down its traits to the child (subclass). The subclass can access and use the methods and properties of the superclass without having to redefine them.

Example:

Let's consider a real-world example of different types of vehicles. We have a general class called "Vehicle," which defines basic properties and behaviors common to all vehicles. Then, we have specific vehicle classes like "Car," "Bike," and "Truck" that inherit from the "Vehicle" class and add their unique features.

```dart
// Superclass (Base class)
class Vehicle {
  String brand;
  int year;

  Vehicle(this.brand, this.year);

  void start() {
    print('Vehicle engine started.');
  }

  void stop() {
    print('Vehicle engine stopped.');
  }
}
```

```dart
// Subclass (Derived class)
class Car extends Vehicle {
  int numberOfDoors;

  Car(String brand, int year, this.numberOfDoors) : super(brand, year);

  void honk() {
    print('Car honks: Beep Beep!');
  }
}

// Subclass (Derived class)
class Bike extends Vehicle {
  bool isElectric;

  Bike(String brand, int year, this.isElectric) : super(brand, year);

  void ringBell() {
    print('Bike rings the bell: Ding Ding!');
  }
}
```

```dart
Run | Debug
void main() {
  // Creating objects using subclass constructors
  var car = Car('Toyota', 2022, 4);
  var bike = Bike('Kawasaki', 2021, true);

  // Accessing properties and methods from the superclass and subclass
  print('Car - Brand: ${car.brand}, Year: ${car.year}, Doors: ${car.numberOfDoors}');
  car.start(); // Vehicle engine started.
  car.honk(); // Car honks: Beep Beep!
  car.stop(); // Vehicle engine stopped.

  print('\nBike - Brand: ${bike.brand}, Year: ${bike.year}, Electric: ${bike.isElectric}');
  bike.start(); // Vehicle engine started.
  bike.ringBell(); // Bike rings the bell: Ding Ding!
  bike.stop(); // Vehicle engine stopped.
}
```

In this example, the "Vehicle" class is the superclass that defines common properties and methods for all vehicles, like "brand" and "year" of the vehicle, as well as the "start()" and "stop()" methods.

The "Car" class is a subclass that extends the "Vehicle" class, inheriting its properties and methods. Additionally, it adds its own property "numberOfDoors" and a unique method "honk()" to represent a car's specific features.

Similarly, the "Bike" class is another subclass that extends the "Vehicle" class. It inherits the "brand" and "year" properties along with the "start()" and "stop()" methods. Additionally, it introduces its own property "isElectric" and a specific method "ringBell()" to represent bike-related attributes.

With inheritance, we can create a hierarchy of classes, making it easier to manage and organize our code while promoting code reuse. The subclasses benefit from the common functionality provided by the superclass, and they can extend or modify it to cater to their specific needs.

Superclass (Base Class):

In object-oriented programming, a superclass, also known as a base class or parent class, is a class that serves as a blueprint for other classes. It defines common properties and behaviors that can be shared among multiple classes.

Think of a superclass as a general category or a template that provides a set of attributes and methods that can be inherited by more specific classes. It encapsulates common characteristics and functionalities that subclasses can reuse.

Example:

Let's consider a simple example of a "Shape" superclass that represents basic shapes like circles, squares, and triangles. The "Shape" class defines common properties like "color" and "area" and a method "calculateArea()" to calculate the area of any shape.

```
// Superclass (Base class)
class Shape {
  String color;

  Shape(this.color);

  double calculateArea() {
    return 0.0; // Default area (to be overridden by subclasses)
  }
}
```

Subclass (Derived Class):

A subclass, also known as a derived class or child class, is a class that inherits properties and behaviors from a superclass. It extends or specializes the superclass, adding its specific attributes and methods.

Think of a subclass as a more specific version of the superclass. It inherits all the characteristics of the superclass and can have additional features unique to its type.

Example:

Continuing with the "Shape" superclass, let's create specific subclasses like "Circle" and "Square" that inherit from the "Shape" class. Each subclass adds its specific properties like "radius" for circles and "sideLength" for squares and overrides the "calculateArea()" method with the appropriate formula for their shape.

```
// Subclass (Derived class) - Circle
class Circle extends Shape {
  double radius;

  Circle(String color, this.radius) : super(color);

  @override
  double calculateArea() {
    return 3.14 * radius * radius;
  }
}

// Subclass (Derived class) - Square
class Square extends Shape {
  double sideLength;

  Square(String color, this.sideLength) : super(color);

  @override
  double calculateArea() {
    return sideLength * sideLength;
  }
}
```

```
Run | Debug
void main() {
  // Create objects of the subclasses and test their functionality
  var circle = Circle('Red', 5.0);
  var square = Square('Blue', 4.0);

  print('Circle - Color: ${circle.color}, Area: ${circle.calculateArea()}');
  print('Square - Color: ${square.color}, Area: ${square.calculateArea()}');
}
```

In this example, "Circle" and "Square" are subclasses of the "Shape" superclass. They inherit the "color" property and the "calculateArea()" method from the "Shape" class. Additionally, each subclass defines its own properties ("radius" for circles and "sideLength" for squares) and overrides the "calculateArea()" method to provide the area calculation specific to their shape.

In summary, a superclass is a general class that defines common properties and behaviors, while a subclass is a more specific class that inherits and extends the functionality of the superclass, adding its specific characteristics. This inheritance mechanism helps create a hierarchy of classes and promotes code reusability in object-oriented programming.

Single Inheritance:

Single inheritance is a concept in object-oriented programming where a class can inherit properties and behaviors from only one single superclass. It means that a subclass can have only one direct parent class, which restricts it from inheriting from multiple classes at the same time.

Think of single inheritance as a family tree, where a child can have only one biological parent (the immediate parent). Similarly, in single inheritance, a subclass can extend only one superclass.

Example :

In this example, we'll create a hierarchy of vehicles. We'll start with a superclass called "Vehicle," which represents the common properties and methods shared by all vehicles. Then, we'll have specific vehicle classes like "Car" and "Bike" that inherit from the "Vehicle" class.

```dart
// Superclass (Base class)
class Vehicle {
  String brand;
  int year;

  Vehicle(this.brand, this.year);

  void start() {
    print('Vehicle engine started.');
  }

  void stop() {
    print('Vehicle engine stopped.');
  }
}
```

SubClasses

```dart
// Subclass (Derived class)
class Car extends Vehicle {
  int numberOfDoors;

  Car(String brand, int year, this.numberOfDoors) : super(brand, year);

  void honk() {
    print('Car honks: Beep Beep!');
  }
}

// Subclass (Derived class)
class Bike extends Vehicle {
  bool isElectric;

  Bike(String brand, int year, this.isElectric) : super(brand, year);

  void ringBell() {
    print('Bike rings the bell: Ding Ding!');
  }
}
```

In this example:

1. The "Vehicle" class is the superclass, representing common features of all vehicles, such as the "brand" and "year" of the vehicle. It also has methods "start()" and "stop()" that represent the engine starting and stopping.

2. The "Car" class is a subclass that extends the "Vehicle" class. It inherits the "brand" and "year" properties along with the "start()" and "stop()" methods. Additionally, it introduces its specific property "numberOfDoors" and a method "honk()" to represent car-specific features.

3. The "Bike" class is another subclass that extends the "Vehicle" class. It inherits the "brand" and "year" properties along with the "start()" and "stop()" methods. Additionally, it introduces its specific property "isElectric" and a method "ringBell()" to represent bike-specific features.

Now, we can use this hierarchy to create instances of cars and bikes and work with their properties and methods:

```
Run | Debug
void main() {
  var car = Car('Toyota', 2022, 4);
  var bike = Bike('Schwinn', 2021, true);

  print('Car - Brand: ${car.brand}, Year: ${car.year}, Doors: ${car.numberOfDoors}');
  car.start();
  car.honk();
  car.stop();

  print('\nBike - Brand: ${bike.brand}, Year: ${bike.year}, Electric: ${bike.isElectric}');
  bike.start();
  bike.ringBell();
  bike.stop();
}
```

This example demonstrates how single inheritance allows us to create a hierarchy of classes, where subclasses inherit and extend functionality from the superclass. It enables code reuse and promotes a well-organized design in object-oriented programming.

Method Overriding

Method overriding is a concept in object-oriented programming where a subclass provides a specific implementation for a method that is already defined in its superclass. When a subclass overrides a method, it replaces the implementation inherited from the superclass with its own implementation, tailored to the subclass's specific behavior.

Key points about method overriding:

1. Inheritance: Method overriding is only applicable in the context of inheritance. It occurs when a subclass extends a superclass, and both classes have a method with the same name and signature.

2. Same method signature: The method in the subclass must have the same name, return type, and parameters (method signature) as the method in the superclass to be considered an override.

3. @override annotation: In programming languages like Dart, you can use the @override annotation to explicitly indicate that a method is intended to override a method in the superclass. This annotation helps prevent accidental mistakes when overriding methods.

4. Polymorphism: Method overriding is a crucial feature for achieving polymorphism. It allows you to treat objects of different subclasses as objects of the common superclass, and their overridden methods will be called based on the actual type of the object at runtime.

Why use method overriding?

Method overriding is essential for building a hierarchy of classes and creating specialized behavior for subclasses while maintaining a consistent interface through the superclass. It promotes code reuse, enables polymorphism, and allows for more flexible and extensible designs in object-oriented programming.

```
// Superclass (Base class)
class Shape {
  void draw() {
    print('Drawing a shape.');
  }
}

// Subclass (Derived class) - Circle
class Circle extends Shape {
  @override
  void draw() {
    print('Drawing a circle.');
  }
}
```

```
// Subclass (Derived class) - Square
class Square extends Shape {
  @override
  void draw() {
    print('Drawing a square.');
  }
}

// Subclass (Derived class) - Triangle
class Triangle extends Shape {
  @override
  void draw() {
    print('Drawing a triangle.');
  }
}
```

In this example, the "Shape" class is a superclass with a method "draw()." The subclasses "Circle," "Square," and "Triangle" override the "draw()" method to provide their specific implementation for drawing their respective shapes.

Let's Take Another example --------> Employee Hierarchy:

```
// Superclass (Base class)
class Employee {
  String name;
  int experience;

  Employee(this.name, this.experience);

  void showDetails() {
    print('Name: $name, Experience: $experience years');
  }
}
```

```
// Subclass (Derived class) - Manager
class Manager extends Employee {
  int numberOfTeams;

  Manager(String name, int experience, this.numberOfTeams) : super(name, experience);

  @override
  void showDetails() {
    print('Manager: $name, Experience: $experience years, Teams: $numberOfTeams');
  }
}

// Subclass (Derived class) - Engineer
class Engineer extends Employee {
  String specialization;

  Engineer(String name, int experience, this.specialization) : super(name, experience);

  @override
  void showDetails() {
    print('Engineer: $name, Experience: $experience years, Specialization: $specialization');
  }
}
```

In this example, the "Employee" class is a superclass with instance variables "name" and "experience" and a method "showDetails()." The subclasses "Manager" and "Engineer" override the "showDetails()" method to display additional details specific to their roles.

These examples demonstrate how method overriding allows subclasses to customize the behavior inherited from the superclass, allowing for polymorphism and more flexible and specialized implementations based on the actual type of the object at runtime.

Abstract classes

Abstract classes are classes in object-oriented programming that cannot be instantiated directly. They are meant to serve as a blueprint for other classes and define a set of common methods and properties that subclasses should implement. Abstract classes provide a way to enforce certain behaviors and structure in subclasses while allowing each subclass to have its own unique implementation.

Key points about abstract classes:

1. Abstract classes cannot be instantiated: You cannot create objects directly from an abstract class. Instead, you use abstract classes as a foundation to create concrete (non-abstract) subclasses.

2. Abstract methods: Abstract classes may contain abstract methods, which are method declarations without implementations. Subclasses of an abstract class must implement all the abstract methods defined in the superclass.

3. Abstract properties: Abstract classes may contain abstract properties, which are properties without initial values. Subclasses must provide values for these abstract properties.

Example: Shape Hierarchy

Let's consider a classic example of an abstract class representing shapes. We'll define an abstract class called "**Shape**" with an abstract method "**calculateArea()**". Different shape subclasses (e.g., Circle, Square) will inherit from the "Shape" class and provide their specific implementations for calculating the area.

```
// Abstract class Shape
abstract class Shape {
  // Abstract method: Subclasses must override this method
  double calculateArea();
}
```

```
// Subclass - Circle
class Circle extends Shape {
  double radius;

  Circle(this.radius);

  @override
  double calculateArea() {
    return 3.14 * radius * radius;
  }
}
```

```
// Subclass - Square
class Square extends Shape {
  double sideLength;

  Square(this.sideLength);

  @override
  double calculateArea() {
    return sideLength * sideLength;
  }
}
```

In this example, the "**Shape**" class is an abstract class that defines the abstract method "**calculateArea()**." It is meant to be a common blueprint for different shapes without providing a concrete implementation for calculating the area.

The "**Circle**" and "**Square**" classes are subclasses of "**Shape**" and override the "**calculateArea()**" method with their specific area calculation logic.

```dart
Run | Debug
void main() {
  Circle circle = Circle(5.0);
  Square square = Square(4.0);

  print('Area of the circle: ${circle.calculateArea()}'); // Output: Area of the circle: 78.5
  print('Area of the square: ${square.calculateArea()}'); // Output: Area of the square: 16.0
}
```

In this example, we create objects of "**Circle**" and "**Square**" and call the "**calculateArea()**" method on each object. The overridden method in each subclass provides the specific area calculation for the corresponding shape.

By using an abstract class in this way, we ensure that all subclasses adhere to the common structure defined by the "**Shape**" superclass, providing consistent behavior while accommodating the unique characteristics of each shape.

Interfaces (through mixins)

In Dart, interfaces are represented using mixins. A mixin is a way to reuse a class's code in multiple class hierarchies. It allows a class to inherit from multiple sources of behavior by mixing in the functionality of one or more classes.

Key points about interfaces through mixins:

1. Reusability: Mixins promo code reusability by allowing classes to incorporate behavior from multiple sources (mixins) without the need for multiple inheritance.

2. No direct instantiation: Mixins cannot be directly instantiated, and they are intended to be used in conjunction with other classes.

3. Use of 'with' keyword: To apply a mixin to a class, you use the 'with' keyword followed by the mixin class name in the class declaration.

Example: Flying Mixin

Let's create a mixin called "Flying" that defines a method "fly()". We'll then create two classes, "Bird" and "Airplane," and use the "Flying" mixin in both classes to provide the "fly()" behavior.

```
// Mixin - Flying
mixin Flying {
  void fly() {
    print('Flying high!');
  }
}
```

```
// Class - Bird
class Bird with Flying {
  String name;

  Bird(this.name);
}

// Class - Airplane
class Airplane with Flying {
  String model;

  Airplane(this.model);
}
```

In this example, the "Flying" mixin provides the behavior of flying, represented by the "fly()" method.

The "Bird" class uses the "Flying" mixin by declaring "with Flying" in the class definition. This allows instances of "Bird" to access the "fly()" method provided by the "Flying" mixin.

Similarly, the "Airplane" class also uses the "Flying" mixin, which enables instances of "Airplane" to access the "fly()" method.

```dart
Run | Debug
void main() {
  var bird = Bird('Eagle');
  var airplane = Airplane('Boeing 747');

  bird.fly(); // Output: Flying high! (method from the Flying mixin)
  airplane.fly(); // Output: Flying high! (method from the Flying mixin)
}
```

In this example, both the "Bird" and "Airplane" classes have access to the "fly()" method through the "Flying" mixin. The "Bird" class is not related to the "Airplane" class, yet they can both use the same functionality through the mixin, promoting code reuse and maintainability.

Mixins enable a powerful way to add functionality to classes without the constraints of multiple inheritance. They provide a flexible and efficient mechanism to achieve code reusability and enhance the structure of class hierarchies in Dart.

Encapsulation

Encapsulation is one of the four fundamental principles of object-oriented programming (OOP), alongside inheritance, polymorphism, and abstraction. It refers to the bundling of data (variables) and methods (functions) that operate on that data within a single unit called a class. Encapsulation allows us to control the access to the internal state of an object and protects it from unauthorized access or modification from outside the class.

Key points about encapsulation:

1. Data hiding: Encapsulation hides the internal details and data of an object, so they are not directly accessible from outside the class. Access to the data is provided only through specific methods, known as getters and setters.

2. Access modifiers: In many programming languages, access modifiers (e.g., private, public, protected) are used to control the visibility of class members

(variables and methods). Private members are only accessible within the class, while public members can be accessed from outside the class.

3. Information hiding: Encapsulation allows for information hiding, ensuring that the internal representation of the object is not visible to the outside world. This promotes better code organization and reduces dependencies between different parts of the code.

Example: Bank Account

Let's illustrate encapsulation using an example of a BankAccount class:

```
class BankAccount {
  // Private instance variables (encapsulated data)
  String  _accountNumber = "";
  String _accountHolderName = "";
  double _balance = 0.0;

  // Public getter methods (accessors) to access private data
  String get accountNumber => _accountNumber;
  String get accountHolderName => _accountHolderName;
  double get balance => _balance;

  // Public setter method (mutator) to update private data
  void deposit(double amount) {
    if (amount > 0) {
      _balance += amount;
    }
  }

  void withdraw(double amount) {
    if (amount > 0 && amount <= _balance) {
      _balance -= amount;
    }
  }
}
```

In this example:

1. The BankAccount class encapsulates the data related to a bank account, including _accountNumber, _accountHolderName, and _balance.

2. These instance variables are marked as private by prefixing them with an underscore _, which means they can only be accessed within the same class.

3. Public getter methods like accountNumber, accountHolderName, and balance allow controlled access to the private data from outside the class.

4. The deposit() and withdraw() methods are public and provide controlled ways to update the account balance.

```
Run | Debug
void main() {
  var account = BankAccount();
  account.deposit(1000);
  account.withdraw(500);

  print('Account Number: ${account.accountNumber}');
  print('Account Holder: ${account.accountHolderName}');
  print('Account Balance: ${account.balance}');
}
```

In this example, the internal details of the bank account (e.g., account number, account holder name) are encapsulated and not directly accessible from outside the BankAccount class. Instead, we use the public getter methods to access this information, ensuring proper encapsulation and data hiding.

Encapsulation helps in building more secure and maintainable code by hiding the implementation details and exposing only essential functionality through well-defined interfaces. It allows objects to protect their state and behavior, promoting a cleaner design and reducing the risk of unintended interference from external code.

Another Example Encapsulation

Encapsulation is a principle of object-oriented programming that aims to hide the internal implementation details of a class and expose only the essential features and behavior through well-defined interfaces. It helps in creating more maintainable and robust code by limiting direct access to the internal state of an object.

Dart provides three levels of access control for class members:

1. Public Members:

Public members are accessible from anywhere in the code, both within and outside the class. In Dart, all class members are public by default, meaning that if you do not explicitly specify an access modifier, the member will be public. You can access public members using the dot notation.

```dart
class Car {
  String ? brand; // Public member

  void start() {
    print('$brand is starting.');
  }
}

Run | Debug
void main() {
  Car myCar = Car();
  myCar.brand = 'Toyota'; // Accessing the public member 'brand'
  myCar.start(); // Accessing the public method 'start'
}
```

2. Private Members:

Private members are only accessible within the class they are declared in. To make a member private in Dart, use an underscore (_) before its name. This convention indicates that the member should not be accessed outside the class. Attempting to access a private member from outside the class will result in a compilation error.

```dart
class Car {
  String ? _registrationNumber; // Private member

  void _privateMethod() {
    print('This is a private method.');
  }

  void start() {
    print('Starting car with registration number $_registrationNumber.');
    _privateMethod();
  }
}

Run | Debug
void main() {
  Car myCar = Car();
  myCar._registrationNumber = '12345'; // This will not work - private member access not allowed
  myCar._privateMethod(); // This will not work - private method access not allowed
}
```

3. Protected Members:

Dart does not have a built-in access modifier for protected members like some other languages do. In other languages, protected members can be accessed within the class they are defined in and by subclasses. However, in Dart, there's no explicit keyword for protected access control.

As a convention, developers often use an underscore (_) prefix for members that are intended to be protected. This suggests that the member should only be accessed within the class and its subclasses. Developers should follow this convention and exercise discipline when accessing such "protected" members to maintain encapsulation.

```dart
class Vehicle {
  String ? _type; // Protected member

  void _protectedMethod() {
    print('This is a protected method.');
  }

  void start() {
    print('Starting the $_type.');
    _protectedMethod();
  }
}

class Car extends Vehicle {
  Car() {
    _type = 'car'; // Accessing the "protected" member from the subclass
    _protectedMethod(); // Accessing the "protected" method from the subclass
  }
}
```

Remember that Dart's access control is mainly a convention. Even though Dart allows you to access private members from outside the class if they are in the same library, it's best to respect encapsulation and avoid accessing private members from outside the class to maintain code integrity and minimize potential issues in the future.

Getters and setters

Getters and setters, also known as property accessors, are special methods used to read and write the values of class fields (also called properties or instance variables). They allow you to control access to the fields, enforce validation rules, and execute custom logic when getting or setting the values.

1. Getters:

A getter is a method that retrieves the value of a class field. It allows you to access the value of a private field without directly exposing the field itself. Getters are defined using the get keyword followed by the property name (without parentheses).

```dart
class Circle {
  double _radius = 0.0; // Non-nullable private field

  double get area {
    return 3.14 * _radius * _radius; // Getter logic
  }
}

Run | Debug
void main() {
  Circle circle = Circle();
  circle._radius = 5;
  print('Area of the circle: ${circle.area}'); // Accessing the getter
}
```

> If you expect `_radius` to be nullable and want to handle the null case explicitly, you can use a null-aware operator (`??`) to provide a default value when `_radius` is null.

```dart
class Circle {
  double? _radius; // Nullable private field

  double get area {
    return 3.14 * (_radius ?? 0.0) * (_radius ?? 0.0); // Getter logic with null-aware operator
  }
}

Run | Debug
void main() {
  Circle circle = Circle();
  circle._radius = 5.0;
  print('Area of the circle: ${circle.area}'); // Accessing the getter
}
```

2. Setters:

A setter is a method that allows you to assign a value to a class field. It enables you to control the assignment process and apply validation checks before setting the value. Setters are defined using the set keyword followed by the property name (without parentheses) and a parameter to hold the new value.

```dart
class Temperature {
  double? _celsius;

  double get celsius {
    return _celsius ?? 0.0; // Return _celsius or a default value (0.0) if it's null
  }

  set celsius(double value) {
    if (value >= -273.15) {
      _celsius = value; // Setter logic with validation check
    } else {
      throw ArgumentError('Temperature cannot be below -273.15°C.');
    }
  }
}

Run | Debug
void main() {
  Temperature temp = Temperature();
  temp.celsius = 25; // Assigning a value using the setter
  print('Current temperature: ${temp.celsius}°C'); // Accessing the getter
}
```

3. Property Accessors:

In many cases, you may want to define both a getter and a setter for a class field to encapsulate its access and manipulation. Dart provides an easy way to create both together using property accessors.

```dart
class BankAccount {
  double _balance = 0;

  double get balance {
    return _balance; // Getter logic
  }

  set balance(double value) {
    if (value >= 0) {
      _balance = value; // Setter logic with validation check
    } else {
      throw ArgumentError('Balance cannot be negative.');
    }
  }
}

Run | Debug
void main() {
  BankAccount account = BankAccount();
  account.balance = 1000; // Using the setter to set the balance
  print('Account balance: \$${account.balance}'); // Using the getter to get the balance
}
```

In this example, we define both a getter and a setter for the balance property of a BankAccount class. The setter allows us to set the balance, while the getter allows us to retrieve the current balance. We also apply a validation check in the setter to ensure that the balance is not negative.

Property accessors provide a clean and controlled way to manage the access and modification of class fields, enhancing the encapsulation and maintainability of your code.