Table of Contents

1. Variable & Data Types

Variable

What is Variable?

A variable name is the name assigned to the memory location where the user stores the data and that data can be fetched when required with the help of the variable by calling its variable name.

There are various types of variables which are used to store the data. The type which will be used to store data depends upon the type of data to be stored.

Syntax for variable declaration:

type_of_variable name_of_variable;

Example: var name;

Syntax for multiple variables declaration:

type_of_variable name_of_variable1,name_of_variable2, … ,name_of_variablen;

Example: var a,b,c,d;

Rules for variable name:

- Variable names can’t be keywords.
- Variable names contain only numbers and alphabets.
- Variable names can’t start with numbers.
- Variable names can’t contain any white space and special characters, except the underscore(_) and dollar($) sign.

Some Valid Variable name example:

```
int age;
String name;
double salary;
bool isStudent;
```

```
final int MAX_LENGTH = 100;
var count;
dynamic data;
var _privateVar;
final PI = 3.14159;
final bool isDartAwesome = true;
var $price;
```

Some Invalid Variable name example:

```
int 123abc; // Cannot start with a digit
String my-name; // Hyphen (-) is not allowed in variable names
bool is?valid; // Special characters like question marks (?) are not allowed
double 3.14; // Cannot start with a digit
final class; // 'class' is a reserved keyword and cannot be used as a variable name
```

Syntax for variable initialization:

```
type_of_variable name_of_variable = value;
```

Example: int a = 10;

Keywords

Keyword: var

The **var** keyword is used to declare a variable. The Dart compiler automatically knows the type of data based on the assigned to the variable because Dart is an infer type language. The syntax is given below.

Syntax: var variable_name;

Example:

```
var age = 30; // Dart infers the type as int
var name = "John"; // Dart infers the type as String
var isStudent = true; // Dart infers the type as bool
```

Keyword: dynamic

When we declare a variable with the **dynamic** type, the Dart compiler will not perform any type checks on that variable. This means you can assign values of different types to the same variable without any compile-time errors. However, if the actual type of the value doesn't match the expected type at runtime, you might encounter runtime errors.

Example:

```
dynamic variable = 42; // variable is of type int
print(variable); // Output: 42
variable = "Hello, world!"; // Now variable is of type String
print(variable); // Output: Hello, world!
variable = true; // Now variable is of type bool
print(variable); // Output: true
```

Keyword: const

The **const** keyword is used to declare constants. Constants are values that are known at compile-time and cannot be changed during runtime. When you declare a variable as a constant using the const keyword, its value must be determined at compile-time and cannot be modified later in the code.

Example:

```
const int age = 30;
const double pi = 3.14159;
const String name = "John";
const bool isStudent = true;
```

Keyword: final

The **final** keyword is used to declare variables whose value can be assigned only once. Once a variable is assigned a value using the final keyword, its value cannot be changed for the rest of the program's execution. This makes the variable effectively read-only.

Example:

```
final int age = 30;
final double pi = 3.14159;
final String name = "John";
final bool isStudent = true;
```

In summary, const is used for values that are known at compile-time, and final is used for values that are determined at runtime during program execution. Both const and final variables are immutable once their values are assigned, but const is more restrictive because its value must be known at compile-time and cannot change across different program executions. On the other hand, final allows values to be determined at runtime, but once assigned, it remains constant during the program's execution.

Data Types

Like other languages (C, C++, Java), whenever a variable is created, each variable has an associated data type. In Dart language, there is the type of values that can be represented and manipulated in a programming language. The data type classification is as given below:

Data Type	Keyword	Description
Number	int, double, num, BigInt	Numbers in Dart are used to represent numeric literals.
Strings	String	Strings represent a sequence of characters.
Booleans	bool	It represents Boolean values true and false.
List	List	It is an ordered group of objects.
Map	Map	It represents a set of values as key-value pairs.

1. Number: The number in Dart Programming is the data type that is used to hold the numeric value. Dart numbers can be classified as:

- The int data type is used to represent whole numbers.

```
int score = 100;
int temperature = -5;
```

- The double data type is used to represent 64-bit floating-point numbers.

```
double pi = 3.14159;
double temperature = 25.5;
```

- The num type is an inherited data type of the int and double types.

```
num value1 = 42;
num value2 = 3.14;
```

2. String: It used to represent a sequence of characters. It is a sequence of UTF-16 code units. The keyword string is used to represent string literals. String values are embedded in either single or double-quotes.

Example:

```
String firstName = 'John';
String lastName = "Doe";
String message = "Hello, World!";
String url = 'https://www.example.com';
String multilineText = '''
This is a multiline string.
It can span multiple lines.
''';
```

Dart provides many built-in methods and operators to work with strings. Here are a few examples:

```
String name = 'John Doe';
print(name.length); // Output: 8 (number of characters in the string)
print(name.toUpperCase()); // Output: JOHN DOE (converts the string to uppercase)
print(name.substring(0, 4)); // Output: John (extracts a substring from the original string)
```

```
print(name.contains('Doe')); // Output: true (checks if the string contains a specific substring)
print(name.split(' ')); // Output: [John, Doe] (splits the string into a list based on the delimiter)
```

Dart also allows string interpolation using '${}' within double-quoted strings. This allows you to embed expressions or variables directly into the string:

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

String fullName = '$firstName $lastName';
print(fullName); // Output: John Doe
```

3. Boolean: It represents Boolean values true and false. The keyword bool is used to represent a Boolean literal in DART. Booleans are often used in programming to control the flow of logic, make decisions, and implement conditions.

Example:

```
bool isStudent = true;
bool isWorking = false;
bool hasPermission = true;
```

4. List: List data type is similar to arrays in other programming languages. A list is used to represent a collection of objects. It is an ordered group of objects.

In Dart, a List is an ordered collection of elements, and it is one of the most commonly used data structures. Lists can store elements of any data type, such as integers, strings, booleans, or even objects. Lists can be of fixed length (array) or dynamic length.

Here are some examples of how to declare and use lists in Dart:

Creating a List:

You can create a list using the square brackets [], and you can initialize it with elements separated by commas.

```
List<int> numbers = [1, 2, 3, 4, 5];
List<String> names = ['Alice', 'Bob', 'Charlie'];
List<dynamic> mixedList = [1, 'Hello', true, 3.14];
```

Accessing Elements:

You can access elements in a list using their index. Indexing starts at 0 for the first element, 1 for the second element, and so on.

```
List<int> numbers = [1, 2, 3, 4, 5];
int firstNumber = numbers[0]; // 1
int thirdNumber = numbers[2]; // 3
```

Modifying Elements:

You can modify elements in a list by assigning new values to their respective indices.

```
List<int> numbers = [1, 2, 3, 4, 5];
numbers[2] = 10;
print(numbers); // Output: [1, 2, 10, 4, 5]
```

List Operations:

Dart provides various methods to perform operations on lists, such as adding elements, removing elements, finding elements, sorting, and more.

```
List<int> numbers = [1, 2, 3];
numbers.add(4); // Adding an element to the end of the list
numbers.remove(2); // Removing a specific element from the list
numbers.indexOf(3); // Finding the index of an element
numbers.sort(); // Sorting the list in ascending order
print(numbers); // Output: [1, 3, 4]
```

List Length:

You can get the number of elements in a list using the length property.

```
List<String> fruits = ['apple', 'banana', 'orange'];
```

```
int numFruits = fruits.length; // 3
```

Fixed-Length List:

You can create a fixed-length list (array) using the List.filled constructor or the List.generate constructor.

```
List<int> fixedList = List.filled(5, 0); // Creates a list of length 5 with all elements initialized to 0.

List<int> generatedList = List.generate(3, (index) => index * 2); // Creates a list of length 3 with elements [0, 2, 4].
```

Lists are versatile and essential data structures in Dart, allowing you to work with collections of data in a flexible and efficient manner.

5. Map: The Map object is a key and value pair. Keys and values on a map may be of any type. It is a dynamic collection.

Here are some examples of how to declare and use maps in Dart:

Creating a Map:

You can create a map using curly braces {} and specifying the key-value pairs separated by colons (:).

```
Map<String, int> ages = {
  'John': 30,
  'Jane': 25,
  'Bob': 28,
};
Map<int, String> cities = {
  1: 'New York',
  2: 'London',
  3: 'Tokyo',
};
```

Accessing Values:

You can access values in a map using their associated keys.

```
Map<String, int> ages = {
  'John': 30,
  'Jane': 25,
  'Bob': 28,
};
int johnsAge = ages['John']; // 30
```

Modifying Values:

You can modify the values of a map using their associated keys.

```
Map<String, int> ages = {
  'John': 30,
  'Jane': 25,
  'Bob': 28,
};
ages['Bob'] = 29;
print(ages); // Output: {John: 30, Jane: 25, Bob: 29}
```

Map Operations:

Dart provides various methods to perform operations on maps, such as adding new key-value pairs, removing key-value pairs, checking if a key exists, getting all keys or values, and more.

```
Map<String, int> ages = {
  'John': 30,
  'Jane': 25,
  'Bob': 28,
};


ages['Alice'] = 24; // Adding a new key-value pair
ages.remove('Jane'); // Removing a key-value pair
bool hasJohn = ages.containsKey('John'); // Checking if a key exists
List<String> names = ages.keys.toList(); // Getting all keys as a list
List<int> ageList = ages.values.toList(); // Getting all values as a list
```

```
print(names); // Output: [John, Bob, Alice]
print(ageList); // Output: [30, 28, 24]
```

Map Length:

You can get the number of key-value pairs in a map using the length property.

```
Map<String, int> ages = {
  'John': 30,
  'Jane': 25,
  'Bob': 28,
};

int numPeople = ages.length; // 3
```

2. Null statements and Null Aware

Null Statements

In Dart, a "null statement" refers to a statement that performs no operation or action. It is essentially a placeholder that represents the absence of any meaningful code or operation. The null statement is denoted by a semicolon (;) without any preceding code.

Example:

```
int x = 10;
if (x < 5); // Null statement, does nothing
print('x is less than 5'); // This statement is outside the if block and will be executed unconditionally
```

Null statements are sometimes used in code for various reasons, such as when you want to add a placeholder for future code development, or as a temporary workaround during debugging or testing.

However, it's essential to use null statements judiciously and with proper comments, as their presence without any clear purpose can make the code less readable and lead to confusion for other developers who might review or maintain the code later. It's generally better to write meaningful code or remove any unnecessary null statements to keep the codebase clean and maintainable.

Null Aware Operators

"null aware" refers to a set of operators and expressions that allow you to handle null values in a more concise and safe manner. These operators help you avoid null reference errors and make your code more robust when dealing with variables that might be null.

1. Default Operator: ??

- We use ?? when you want to evaluate and return an expression if another expression resolves to null.
- It is also called the if-null operator and coalescing operator.
- The null-aware operator is ??, which returns the expression on its left unless that expression’s value is null. In which case it’s null it returns the expression on its right:

Example:

```
int x; // x is currently null
```

```
int y = x ?? 10; // If x is null, y will be assigned 10
print(y); // Output: 10

x = 5;
y = x ?? 10; // x is 5, so y will be assigned 5
print(y); // Output: 5
```

Null-aware assignment (??=):

The null-aware assignment operator allows you to assign a value to a variable only if the variable is currently null. If the variable already has a non-null value, the assignment does not take place.

Example:

```
int x; // x is currently null

x ??= 5; // If x is null, assign 5 to it
print(x); // Output: 5

x ??= 10; // x is already 5, so the assignment doesn't happen
print(x); // Output: 5
```

Conditional access (?.):

The conditional access operator allows you to access properties or call methods on an object that might be null without causing a null reference error. If the object is null, the expression evaluates to null instead of throwing an exception.

Example:

```
class Person {
  String? name;
  String getName() => name ?? 'Unknown';
}

Person? person; // person is currently null

String? name = person?.name; // Safe access to the 'name' property, name will be null if 'person' is null
String personName = person?.getName() ?? 'Anonymous'; // Safe access to the 'getName()' metho
```

3. Operators

An "operator" refers to a symbol or a set of symbols that represent an operation or an action to be performed on one or more operands. Operators are fundamental building blocks of any programming language and are used to perform various tasks, such as arithmetic calculations, logical comparisons, assignment of values, and more.

In most programming languages, including Dart, operators can be classified into several categories based on the type of operation they perform:

Arithmetic Operators: Perform basic arithmetic calculations on numeric operands.

```
int a = 5;
int b = 3;

int sum = a + b; // Addition
int difference = a - b; // Subtraction
int product = a * b; // Multiplication
double quotient = a / b; // Division (results in a double)
int remainder = a % b; // Modulo (remainder of division)
```

Comparison Operators: Compare two operands and return a boolean value (true or false) based on the comparison.

```
int a = 5;
int b = 3;

bool isEqual = a == b; // Equal to
bool isNotEqual = a != b; // Not equal to
bool isGreater = a > b; // Greater than
bool isLess = a < b; // Less than
bool isGreaterOrEqual = a >= b; // Greater than or equal to
bool isLessOrEqual = a <= b; // Less than or equal to
```

Logical Operators: Perform logical operations on boolean operands and return boolean results.

```
bool isRainy = true;
```

```
bool isCold = false;

bool isWetAndCold = isRainy && isCold; // Logical AND
bool isWetOrCold = isRainy || isCold; // Logical OR
bool isNotRainy = !isRainy; // Logical NOT (negation)
```

Assignment Operators: Assign a value to a variable or modify its value.

```
int x = 10;
x = 5; // Simple assignment
x += 3; // Compound assignment (equivalent to x = x + 3)
x -= 2; // Compound assignment (equivalent to x = x - 2)
x *= 2; // Compound assignment (equivalent to x = x * 2)
x /= 4; // Compound assignment (equivalent to x = x / 4)
```

Bitwise Operators

"Bitwise Operators" are used to perform bitwise operations on integer values at the bit level. These operators manipulate individual bits in integers, treating them as binary numbers. Bitwise operators are useful in scenarios where you need to work with individual bits of integer values directly.

```
int a = 10; // Binary: 00001010
int b = 5;  // Binary: 00000101

// Bitwise AND (a & b)
int resultAnd = a & b; // Binary: 00000000 (0 in decimal)
print("Bitwise AND: $resultAnd"); // Output: 0

// Bitwise OR (a | b)
int resultOr = a | b; // Binary: 00001111 (15 in decimal)
print("Bitwise OR: $resultOr"); // Output: 15

// Bitwise XOR (a ^ b)
int resultXor = a ^ b; // Binary: 00001111 (15 in decimal)
print("Bitwise XOR: $resultXor"); // Output: 15

// Bitwise NOT (~a)
int resultNotA = ~a; // Binary: 11110101 (4294967285 in decimal)
print("Bitwise NOT of a: $resultNotA"); // Output: 4294967285

// Bitwise NOT (~b)
int resultNotB = ~b; // Binary: 11111010 (4294967290 in decimal)
print("Bitwise NOT of b: $resultNotB"); // Output: 4294967290
```

```
// Left Shift (a << 2)
int resultLeftShift = a << 2; // Binary: 00101000 (40 in decimal)
print("Left Shift of a: $resultLeftShift"); // Output: 40

// Right Shift (b >> 1)
int resultRightShift = b >> 1; // Binary: 00000010 (2 in decimal)
print("Right Shift of b: $resultRightShift"); // Output: 2
```

Spread operators

spread operators are used to "spread" the elements of a collection (such as a list or a map) into another collection or to function arguments. There are two types of spread operators: the spread operator ... and the null-aware spread operator ...?.

Spread Operator (...):

The spread operator ... is used to "spread" the elements of an iterable (e.g., a list or a set) into another collection or as function arguments. It takes the individual elements of the source iterable and adds them to the target collection.

Using the spread operator with a list:

```
List<int> list1 = [1, 2, 3];
List<int> list2 = [4, 5, 6];
List<int> combinedList = [...list1, ...list2]; // Combines the elements of list1 and list2 into a new list.
print(combinedList); // Output: [1, 2, 3, 4, 5, 6]
```

Using the spread operator with function arguments:

```
void printValues(int a, int b, int c) {
  print('a: $a, b: $b, c: $c');
}

List<int> values = [1, 2, 3];

printValues(...values); // Spreads the elements of the 'values' list as function arguments.
// Output: a: 1, b: 2, c: 3
```

Null-aware Spread Operator (...?):

The null-aware spread operator ...? is used to "spread" the elements of an iterable that might be null. If the iterable is null, the spread operation results in an empty collection (an empty list or an empty set).

Using the null-aware spread operator with a nullable list:

```
List<int>? nullableList = [1, 2, 3];
List<int> nonNullableList = [4, 5, 6];

List<int> combinedList = [...?nullableList, ...nonNullableList];
// If 'nullableList' is null, it will be spread as an empty list, and 'combinedList' will only contain the elements of 'nonNullableList'.

print(combinedList); // Output: [1, 2, 3, 4, 5, 6]
```

The spread operators offer a concise way to combine or spread elements of collections and function arguments. They are particularly useful when you need to create new collections based on existing ones or when you want to pass multiple values to a function as arguments.

Ternary operator

The ternary operator, also known as the conditional operator, is a concise way to write a simple if-else statement in a single line. It evaluates a Boolean expression and returns one of two values based on the result of the evaluation.

The syntax of the ternary operator is:

```
condition ? expression1 : expression2;
```

Here's how the ternary operator works:

- condition: A Boolean expression that is evaluated. If the condition is true, the value of expression1 is returned; otherwise, the value of expression2 is returned.

Example using the ternary operator:

```
int a = 10;
int b = 5;

// If 'a' is greater than 'b', assign 'a' to 'maxValue', else assign 'b'.
int maxValue = a > b ? a : b;

print(maxValue); // Output: 10
```

The ternary operator is useful when you need to make a simple decision based on a condition and assign a value accordingly. It can be used as a concise alternative to the if-else statement in cases where you have a single expression for each branch of the condition. However, for more complex conditions or multiple expressions in each branch, the if-else statement is usually preferred for better readability and maintainability.

4. String Class

The String data type represents a sequence of characters. A Dart string is a sequence of UTF 16 code units.

A string can be either single or multiline. Single line strings are written using matching single or double quotes, and multiline strings are written using triple quotes. The following are all valid Dart strings:

```
'Single quotes';
"Double quotes";
'Double quotes in "single" quotes';
"Single quotes in 'double' quotes";

'''A
multiline
string''';

"""
Another
multiline
string""";
```

Strings are immutable. Although you cannot change a string, you can perform an operation on a string which creates a new string:

```
const string = 'Dart is fun';
print(string.substring(0, 4)); // 'Dart'

//You can use the plus (+) operator to concatenate strings:
const string = 'Dart ' + 'is ' + 'fun!';
print(string); // 'Dart is fun!'
```

Adjacent string literals are concatenated automatically:

```
const string = 'Dart ' 'is ' 'fun!';
print(string); // 'Dart is fun!'
```

You can use ${} to interpolate the value of Dart expressions within strings. The curly braces can be omitted when evaluating identifiers:

```
const string = 'dartlang';
print('$string has ${string.length} letters'); // dartlang has 8 letters
```

A string is represented by a sequence of Unicode UTF-16 code units accessible through the codeUnitAt or the codeUnits members:

```
const string = 'Dart';
final firstCodeUnit = string.codeUnitAt(0);
print(firstCodeUnit); // 68, aka U+0044, the code point for 'D'.
final allCodeUnits = string.codeUnits;
print(allCodeUnits); // [68, 97, 114, 116]
```

A string representation of the individual code units is accessible through the index operator:

```
const string = 'Dart';
final charAtIndex = string[0];
print(charAtIndex); // 'D'
```

The characters of a string are encoded in UTF-16. Decoding UTF-16, which combines surrogate pairs, yields Unicode code points. Following a similar terminology to Go, Dart uses the name 'rune' for an integer representing a Unicode code point. Use the runes property to get the runes of a string:

```
const string = 'Dart';
final runes = string.runes.toList();
print(runes); // [68, 97, 114, 116]
```

Methods to Manipulate Strings

In Dart, the String class provides several methods to manipulate strings. These methods allow you to perform various operations on strings, such as finding substrings, converting cases, replacing parts of the string, splitting and joining strings, and more. Here are some commonly used methods to manipulate strings:

Length:

length: Returns the length of the string.

```
String text = "Hello, Dart!";
```

```
int length = text.length; // Output: 13
```

Substrings:

substring(startIndex, [endIndex]): Returns a new string that is a substring of the original string, starting from the startIndex and optionally ending at the endIndex.

```
String text = "Hello, Dart!";
String substring1 = text.substring(0, 5); // Output: "Hello"
String substring2 = text.substring(7); // Output: "Dart!"
```

Case Conversion:

toUpperCase(): Converts all characters in the string to uppercase.

toLowerCase(): Converts all characters in the string to lowercase.

```
String text = "Hello, Dart!";
String uppercaseText = text.toUpperCase(); // Output: "HELLO, DART!"
String lowercaseText = text.toLowerCase(); // Output: "hello, dart!"
```

Searching:

contains(pattern): Checks if the string contains the specified pattern.

startsWith(pattern): Checks if the string starts with the specified pattern.

endsWith(pattern): Checks if the string ends with the specified pattern.

indexOf(pattern, [startIndex]): Returns the index of the first occurrence of the pattern, starting from the optional startIndex.

lastIndexOf(pattern, [startIndex]): Returns the index of the last occurrence of the pattern, starting from the optional startIndex.

```
String text = "Hello, Dart!";
bool containsDart = text.contains("Dart"); // Output: true
bool startsWithHello = text.startsWith("Hello"); // Output: true
bool endsWithExclamation = text.endsWith("!"); // Output: true
int indexOfD = text.indexOf("D"); // Output: 7
int lastIndexOfL = text.lastIndexOf("l"); // Output: 3
```

Replacing:

replaceAll(from, to): Replaces all occurrences of the from substring with the to substring.

```
String text = "Hello, Dart!";
String replacedText = text.replaceAll("Dart", "World"); // Output: "Hello, World!"
```

Splitting and Joining:

split(separator): Splits the string into a list of substrings based on the separator.

join(separator): Joins a list of strings into a single string, using the separator.

```
String text = "apple,banana,orange";
List<String> fruits = text.split(","); // Output: ["apple", "banana", "orange"]
String joinedFruits = fruits.join(" and "); // Output: "apple and banana and orange"
```

Trimming:

trim(): Removes leading and trailing whitespace characters from the string.

trimLeft(): Removes leading whitespace characters from the string.

trimRight(): Removes trailing whitespace characters from the string.

```
String text = "   Hello, Dart!   ";
String trimmedText = text.trim(); // Output: "Hello, Dart!"
```

5. Conditionals

1. **if statement:**

The if statement is used to execute a block of code if a specified condition is true.

Syntax:

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

Example:

```
int age = 25;
if (age >= 18) {
  print("You are an adult.");
}
```

2. **if-else statement:**

The if-else statement is used to execute one block of code if the condition is true and another block of code if the condition is false.

Syntax:

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

Example:

```
int age = 15;

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

```
} else {
  print("You are a minor.");
}
```

3. **if-else if-else statement:**

The if-else if-else statement is used to check multiple conditions and execute different blocks of code based on the first condition that is true.

Syntax:

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

Example:

```
int score = 85;

if (score >= 90) {
  print("A grade.");
} else if (score >= 80) {
  print("B grade.");
} else if (score >= 70) {
  print("C grade.");
} else {
  print("Fail.");
}
```

4. **Nested if-else condition**

Nested if-else statements to create multiple levels of conditions. A nested if-else statement means that you have an if-else statement inside another if or else block. This allows you to

test for more complex conditions and execute different blocks of code based on various combinations of conditions.

Syntax:

```
if (condition1) {
  // Code to be executed if condition1 is true
  if (condition2) {
    // Code to be executed if both condition1 and condition2 are true
  } else {
    // Code to be executed if condition1 is true, but condition2 is false
  }
} else {
  // Code to be executed if condition1 is false
}
```

Example:

```
int score = 85;
bool isQualified = true;

if (score >= 90) {
  print("A grade.");
  if (isQualified) {
    print("You are qualified for a scholarship.");
  } else {
    print("You are not qualified for a scholarship.");
  }
} else if (score >= 80) {
  print("B grade.");
} else {
  print("Fail.");
}
```

6. Loops

Loops are used to repeat a block of code multiple times until a certain condition is met. Dart provides several types of loops to cater to different scenarios:

- **for loop:**

The for loop is used to execute a block of code for a specified number of times. It consists of an initialization, a condition, and an update expression.

Syntax:

```
for (initialization; condition; update) {
  // Code to be executed repeatedly until the condition is false
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
  print('Iteration $i');
}
```

- **while loop:**

The while loop is used to execute a block of code as long as a specified condition is true. It checks the condition before entering the loop.

Syntax:

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

Example:

```
int count = 1;
while (count <= 5) {
```

```
  print('Iteration $count');
  count++;
}
```

- **do-while loop:**

The do-while loop is similar to the while loop, but it checks the condition after executing the block of code. This ensures that the code inside the loop is executed at least once.

Syntax:

```
do {
  // Code to be executed repeatedly until the condition is false
} while (condition);
```

Example:

```
int count = 1;
do {
  print('Iteration $count');
  count++;
} while (count <= 5);
```

- **for-in loop:**

The for-in loop is used to iterate over elements in an iterable object, such as a list or a set.

Syntax:

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

Example:

```
List<int> numbers = [1, 2, 3, 4, 5];
for (var number in numbers) {
  print('Number: $number');
}
```

- **forEach loop:**

The forEach loop is used to iterate over elements in an iterable object, such as a list or a set. It is a higher-order function that takes a callback function as an argument.

Syntax:

```
iterable.forEach((element) {
  // Code to be executed for each element in the iterable
});
```

Example:

```
List<int> numbers = [1, 2, 3, 4, 5];
numbers.forEach((number) {
  print('Number: $number');
});
```

Loops are essential for automating repetitive tasks and processing collections of data in your Dart programs. They offer a powerful way to efficiently handle a large number of iterations and perform tasks accordingly.

7. Functions

Functions are blocks of code that can be defined once and executed multiple times with different arguments. They allow you to encapsulate reusable pieces of code, making your program more organized and easier to maintain. Dart supports both named functions and anonymous functions (also known as lambda or anonymous closures). Here's how you can define and use functions in Dart:

- **Named Function:**

A named function is a regular function with a name that can be invoked using its identifier.

Syntax:

```
returnType functionName(parameter1, parameter2, ...) {
  // Function body
  // Code to be executed when the function is called
  return returnValue;
}
```

Example:

```
// Named function
int addNumbers(int a, int b) {
  return a + b;
}

void main() {
  int result = addNumbers(5, 3);
  print('Result: $result'); // Output: Result: 8
}
```

- **Anonymous Function (Lambda or Closure):**

An anonymous function is a function without a name. It can be assigned to a variable or passed as an argument to another function.

Syntax:

```
(parameter1, parameter2, ...) {
  // Function body
  // Code to be executed when the function is called
  return returnValue;
}
```

Example:

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

void main() {
  int result = multiply(5, 3);
  print('Result: $result'); // Output: Result: 15
}
```

- **Arrow Function (Short Syntax):**

For single-expression functions, you can use the arrow function syntax, which provides a shorter and more concise way to define functions.

Syntax:

```
returnType functionName(parameter1, parameter2, ...) => expression;
```

Example:

```
// Arrow function
int square(int x) => x * x;
```

```
void main() {
  int result = square(5);
  print('Result: $result'); // Output: Result: 25
}
```

Functions can have parameters, which are variables used to pass values into the function when it's called. They can also have a return type to specify the type of value the function will return. If a function does not explicitly return a value, it implicitly returns null.

Dart functions are first-class objects, which means you can pass them as arguments to other functions, return them from functions, and assign them to variables. This feature makes Dart a versatile and powerful language for creating flexible and modular code.

Functions with Optional Parameter:

you can define optional parameters using both positional and named syntax. Additionally, you can assign default values to optional parameters, making them optional with default values. Let's explore each type of optional parameter in Dart:

- **Optional Positional Parameter:**

Optional positional parameters are defined using square brackets [] in the function signature. They are identified by their positions when calling the function. You can provide values for these parameters in the order they are defined, and they are considered optional since you can omit them when calling the function.

Syntax:

```
returnType functionName(parameter1, [parameter2, ...]) {
  // Function body
}
```

Example:

```
void printInfo(String name, [int age = 18, String? occupation]) {
  print('Name: $name, Age: $age, Occupation: ${occupation ?? 'Unknown'}');
}
```

```
void main() {
  printInfo('Alice'); // Output: Name: Alice, Age: 18, Occupation: Unknown
  printInfo('Bob', 30); // Output: Name: Bob, Age: 30, Occupation: Unknown
   printInfo('Charlie', 25, 'Engineer'); // Output: Name: Charlie, Age: 25, Occupation: Engineer
}
```

- **Optional Named Parameter:**

Optional named parameters are defined using curly braces {} in the function signature. They are identified by their names when calling the function. Named parameters allow you to specify the parameter name along with the corresponding value, making function calls more explicit and allowing you to provide values in any order.

Syntax:

```
returnType functionName({parameter1, parameter2, ...}) {
  // Function body
}
```

Example:

```
void greet({String? name, int age = 18, String? occupation}) {
  print('Hello, ${name ?? 'Guest'}! You are $age years old. Occupation: ${occupation ?? 'Unknown'}');
}


void main() {
  greet(); // Output: Hello, Guest! You are 18 years old. Occupation: Unknown
   greet(name: 'Alice'); // Output: Hello, Alice! You are 18 years old. Occupation: Unknown
  greet(name: 'Bob', age: 25); // Output: Hello, Bob! You are 25 years old. Occupation: Unknown
    greet(occupation: 'Engineer'); // Output: Hello, Guest! You are 18 years old. Occupation: Engineer
}
```

- **Optional Parameter with Default Values:**

Both optional positional and named parameters can have default values assigned to them. If a value is not provided for an optional parameter during the function call, the default value will be used instead.

Syntax for Optional Positional Parameter:

```
returnType functionName([parameter1 = defaultValue, parameter2 = defaultValue, ...]) {
  // Function body
}
```

Syntax for Optional Named Parameter:

```
returnType functionName({parameter1 = defaultValue, parameter2 = defaultValue, ...}) {
  // Function body
}
```

Example:

```
void printInfo(String name, [int age = 18, String occupation = 'Unknown']) {
  print('Name: $name, Age: $age, Occupation: $occupation');
}

void main() {
  printInfo('Alice'); // Output: Name: Alice, Age: 18, Occupation: Unknown
  printInfo('Bob', 30); // Output: Name: Bob, Age: 30, Occupation: Unknown
   printInfo('Charlie', 25, 'Engineer'); // Output: Name: Charlie, Age: 25, Occupation: Engineer
}
```

Optional parameters with default values are a powerful feature in Dart that provides flexibility and convenience when defining functions. They allow you to create functions that can handle various argument scenarios with meaningful default values and enable you to use functions more intuitively in your code.

8. Anonymous Function and Closures

Anonymous Function

An anonymous function, also known as a lambda function or a closure, is a function that doesn't have a name. It's defined using a concise syntax and can be used to encapsulate a block of code for later execution. Anonymous functions are often used as arguments to higher-order functions, like forEach, map, and filter, or to define inline behavior for callbacks.

Syntax:

```
(parameter1, parameter2, ...) {
  // Function body
  // Code to be executed
}
```

Example:

```
void main() {
  // Using an anonymous function as an argument to forEach
  List<int> numbers = [1, 2, 3, 4, 5];
  numbers.forEach((number) {
    print('Number: $number');
  });

  // Using an anonymous function to filter even numbers
  List<int> evenNumbers = numbers.where((number) => number % 2 == 0).toList();
  print('Even numbers: $evenNumbers');

  // Using an anonymous function for a custom sort
  List<String> names = ['Alice', 'Bob', 'Charlie', 'David'];
  names.sort((a, b) => a.compareTo(b));
  print('Sorted names: $names');
}
```

Closures

A closure is a special type of function that can capture and remember the variables and parameters of its containing scope, even after the scope has exited. Closures are created when a function references variables from its surrounding context, and they allow functions to retain access to those variables even if they are no longer in scope. Closures are a powerful feature in Dart and are often used for callbacks, asynchronous programming, and encapsulating behavior.

Example:

```
Function makeMultiplier(int factor) {
  return (int number) => number * factor;
}

void main() {
  var doubler = makeMultiplier(2);
  var tripler = makeMultiplier(3);

  print(doubler(5)); // Output: 10
  print(tripler(5)); // Output: 15
}
```

In this example:

- We define a function makeMultiplier that takes a factor parameter and returns a closure (anonymous function). The closure takes a number parameter and multiplies it by the given factor.
- We create two closures, doubler and tripler, by calling makeMultiplier(2) and makeMultiplier(3), respectively. These closures "capture" the factor values (2 and 3) from their surrounding context.
- When we call doubler(5) and tripler(5), the closures execute and use the captured factor values to perform the multiplication.

The concept of closures is fundamental in Dart's functional programming paradigm. Closures allow you to create functions that "remember" the values of variables from their enclosing scope, making it possible to pass behavior along with data. This is particularly useful for scenarios like callbacks, where you want to specify behavior that uses values from the current context.