Dart Syllabus

Basics

1. Introduction to Dart Programming Language
2. SDK Installation
3. Comments
4. Variables
5. Operators
6. Standard Input Output

Data Types

1. Dart Data Types
2. Numbers
3. Strings
4. List
5. Sets
6. Map
7. Queues in Dart
8. Dart Enums

Control Flow

1. If-else Statement
2. Switch case Statements
3. Loops
4. Loops Control Statements
5. Labels in Dart

Key Functions

1. Function
2. Types of Functions
3. Anonymous Function
4. Main Function
5. Common Collection Methods
6. Exit Function
7. Getter and Setter Methods in Dart

Object – Oriented Programming

1. Classes and Objects
2. Constructors
3. Super Constructor
4. This Keyword
5. Static Keyword
6. Super Keyword
7. Constant and Final Keyword
8. Inheritance
9. Methods
10. Method Overloading
11. Getters and Setters
12. Abstract Classes
13. Builder Class
14. Concepts of Callable Classes in Dart
15. Dart Interfaces
16. Dart extends Vs with Vs implements

Dart Utilities

1. Dart Date and Time
2. Using await async in Dart
3. Data Enumeration in Dart
4. Dart Type System
5. Generator in Dart

Dart Programs

1. How to combine Lists in Dart?
2. Finding minimum and maximum value in List?
3. Dart – Splitting of Strings
4. How to append or concatenate strings in Dart?
5. How to find the length of a string in Dart?
6. Dart sort a List
7. How to convert a lower-case string to an upper-case string?
8. How to convert all characters of a string to lowercase?
9. How to replace a substring of string in Dart?
10. How to check string is Empty in Dart?

Advanced Concepts

1. Exception Handling in Dart
2. Assert Statements in Dart
3. Fallthrough Condition in Dart
4. Concept of Isolates in Dart
5. Dart Typedef
6. Dart URIs
7. Dart Collections
8. Dart Packages
9. Dart Generators
10. Dart Callable Classes
11. Dart Isolates
12. Dart Async
13. Dart String codeUnits Property
14. Dart HTML DOM

Dart

Dart is an open-source, general-purpose, object-oriented programming language with C – style syntax developed by Google in 2011.

The purpose of Dart programming is to create a frontend user interfaces for the web and mobile apps. It meant for both server side as well as the user side.

It supports most of the common concepts of programming like classes, interface, functions unlike other programming languages. Dart language does not support arrays directly. It supports collection, which is used to replicate the data structure such as array, generics and optional typing.

Dart is extensively use to create single-page websites and web-applications. Best example of dart application is **Gmail**.

Why Dart

- Dart is really is flexible.
- Dart embraced the open-source ecosystem.
- Dart is pretty easy to learn.
- Dart has great tooling support.
- Data is a robust language.
- Dart ensures productivity.
- Dart is backed by a tech giant, so it must have some great scope.
- Dart solved a lot of problems efficiently.

Dart: Comments

In all programming language comments play an important role for a better understanding of the code in the future or by any other programmer. Comments are a set of statements that are not meant to be executed by the compiler. They provide proper documentation of the code.

Types of Dart Comments:

1. Dart Single line comment
2. Dart Multiline comment
3. Dart Documentation comment

Dart Single line comment

Dart single line comment is used to comment a line until break occurs. It is done using a double forward-slash (//).

```
// This is a single line comment.
```

Dart Multiline Comment

Dart multiline comment is used to comment out a whole section of code. It uses '/*' and '*/' to start and end a multiline comment respectively.

```
/*
This the first Dart Program.
It contains a Main method,
Which is the entry point for the program.
*/
```

Dart Documentation Comment

Dart documentation comments are a special type of comment used to provide references to packages, software, or projects. Dart supports two types of documentation comments "///" (C# style) and "/**.... */" (JavaDoc Style). It is preferred to use "///" for doc comments as many times * used to mark items in a bulleted list which makes it difficult to read the comments. Doc comments are recommended for writing public APIs.

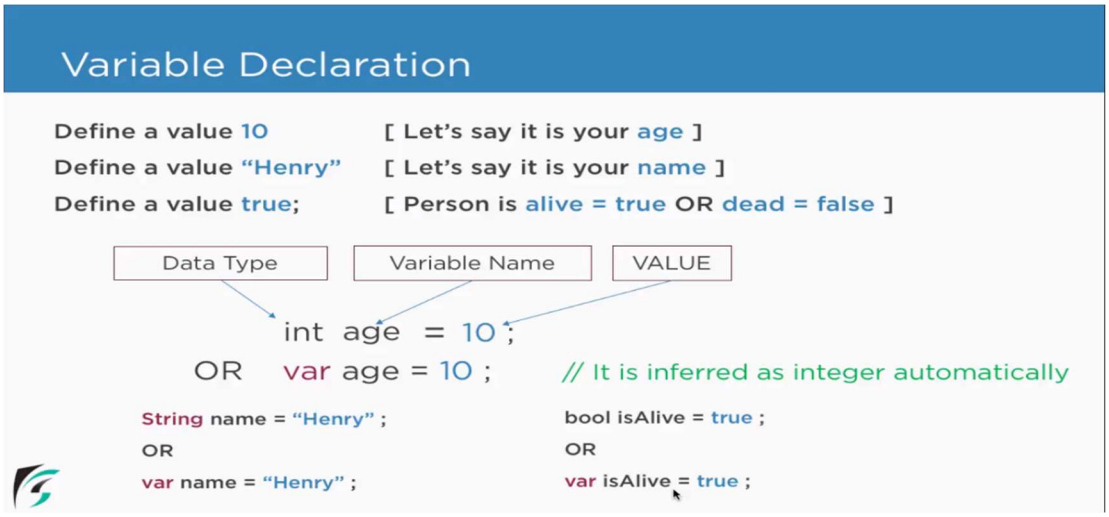
```
/// This is
/// a documentation
/// comment
```

Dart Variables

A variable name is the name assign to the memory location where the user stores the data and the 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.

Variable Declaration



Conditions to write variable name or identifiers:

1. Variable name or identifiers can't be the **Keyword**.
2. Variable name or identifiers can contain alphabets and numbers.
3. Variable name or identifiers can't contain spaces and special characters, except the **underscore** (_) and the **dollar** ($) sign.
4. Variable name or identifiers can't begin with number.

Built-in Data Types

Dart has special support for these data types,

- Numbers
  - int
  - double
- Strings
- Booleans (true or false)
- Lists (also known as Arrays)
- Maps
- Runes (for expressing Unicode characters in a String)
- Symbols

NOTE:

- All data types in Dart are **Objects**. Therefore, their initial values are by default **null**.
- Dart supports **type-checking**, it means that it checks whether the data type and the data that variable holds are specific to that data or not.

Dynamic type variable in Dart

This is a special variable initialized with keyword **dynamic**. The variable declared with this data type can store implicitly any value during running the program. It is quite similar to **var** datatype in Dart, but the difference between them is the moment you assign the data to variable with var keyword it is replaced with the appropriate data type.

Example:

```
void main() {
  // Assigning value to geek variable
  dynamic geek = "Geeks For Geeks";

  // Printing variable geek
  print(geek);

  // Reassigning the data to variable and printing it
  geek = 3.14157;
  print(geek);
}
```

```
Output:
Geeks For Geeks
3.14157
```

NOTE: If we use **var** instead of **dynamic** in the above code then it will show error.

Final and Const Keyword in Dart

These keywords are used to define constant variable in Dart i.e., once a variable is defined using these keywords then its value can't be changed in the entire code. These keywords can be used with or without data type name.

If you never want to change a value then use final and const keywords:

```
final name = "Peter";
const PI = 3.14;
```

Difference between final and const

- Final variable can only be set once and it is initialized when accessed.
- Const variable is implicitly final but it is a compile-time constant i.e., it is initialized during compilation.

Instance Variable can be **final** but cannot be **const**. If you want Constant at class level then make it **static** const.

Operators in Dart

Arithmetic Operators

These classes of operators contain those operators which are used to perform arithmetic operation on the operands. They are binary operators i.e.; they act on two operands.

Operator Symbol	Operator Name	Operator Description
+	Addition	Use to add two operands
-	Subtraction	Use to subtract two operands
-expr	Unary Minus	It is used to reverse the sign of the expression
*	Multiply	Use to multiply two operands
/	Division	Use to divide two operands
~/	Division	Use to divide two operands but give output in integer
%	Modulus	Use to give remainder of two operands

Relational Operators: These classes of operators contain those operators which are used to perform relational operation on the operands.

Operator Symbol	Operator Name	Operator Description
>	Greater than	Check which operand is bigger and give the result as Boolean expression
<	Less than	Check which operand is smaller and give result as Boolean expression
>=	Greater than or equal to	Check which operand is greater or equal to each other and give result as Boolean expression
<=	Less than equal to	Check which operand is less than or equal to each other and give result as Boolean expression
==	Equal to	Check whether the operand is equal to each other or not and give result as Boolean expression
!=	Not equal to	Check whether the operand is not equal to each other or not and give result as Boolean expression

Bitwise Operators: These classes of operators contain those operators which are used to perform bitwise operation on the operands.

Operator Symbol	Operator Name	Operator Description
&	Bitwise AND	Performs bitwise and operation on two operands
|	Bitwise OR	Performs bitwise or operations on two operands
^	Bitwise XOR	Performs bitwise XOR operations on two operands
~	Bitwise NOT	Performs bitwise NOT operation on two operands
<<	Left Shift	Shifts a in binary representation to b bits to left and inserting 0 from right
>>	Right Shift	Shifts a in binary representation to b bits to left and inserting 0 from left

Cascade Notation Operators

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

Operator Symbol	Operator Name	Operator Description
..	Cascading Method	It is used to perform multiple methods on the same object.

Type Test Operators

These classes of operators contain those operators which are used to perform comparison on the operands.

Operator Symbol	Operator Name	Operator Description
is	is	Gives Boolean value true as output if the object has specific type
is!	is not	Gives Boolean value false as output if the object has specific type

Assignment Operators

These classes of operators contain those operators which are used to assign value to the operands.

Operator Symbol	Operator Name	Operator Description
=	Equal to	Use to assign values to the expression or variable
??=	Assignment operator	Assign the value only if it is null

Logical Operators

These classes of operators contain those operators which are used to logically combine two or more conditions of the operands.

Operator Symbol	Operator Name	Operator Description
&&	And operator	Use to add two conditions and if both are true than it will return true
||	Or operator	Use to add two conditions and if even one of them is true than it will return true
!	Not operator	It is use to reverse the result

Conditional Operators

These classes of operators contain those operators which are used to perform comparison on the operands.

Operator Symbol	Operator Name	Operator Description
Condition? Expression1: expression2	Conditional operator	It is a simple version of if-else statement. If the condition is true than expression1 is executed else expression2 is executed.
Expression1 ?? expression2	Conditional operator	If expression1 is non-null returns its value else returns expression2 value.

Dart: Standard Input Output

Standard Input

In Dart programming language, you can take standard input from the user through the console via the use of **.redLineSync ()** function. To take input from the console you need to import a library, named **dart:io** from libraries of Dart.

Stdin Class

This class allows the user to read data from standard input in both synchronous and asynchronous ways. The method **readLineSync ()** is one of the methods used to take input from the user.

Taking a string input from user:

```
// importing dart:io file
import 'dart:io';

void main()
{
   print("Enter your name?");
   // Reading name of the Geek
   String? name = stdin.readLineSync();

   // Printing the name
   print("Hello, $name! \nWelcome to GeeksforGeeks!!");
}
```

```
Input: Geek
```

```
Output:
Enter your name?
Hello, Geek!
Welcome to GeeksforGeeks!!
```

Standard Output in Dart

In Dart, there are two ways to display output in the console:

1. Using print statement
2. Using stdout.write () statement

NOTE: The **print ()** statement brings the cursor to next line while **stdout.write ()** don't bring the cursor to the next line, it remains in the same line.

Dart: Data Types

Like other languages, 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.

Data Type	Keyword	Description
Number	Int, Double, num	Numbers in Dart are used to represent numeric literals
Strings	String	Stings represent a sequence of characters
Booleans	Bool	It represents Boolean values true and false
Lists	List	It is an ordered group of objects
Maps	Map	It represents a set of values as key-value pairs

Number

The number in Dart Programming language 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.
- The **double** data type is used to represent 64-bit floating point numbers.
- The **num** type is an inherited data type of the int and double types.

Parsing in Dart

The parse () function is used parsing a string containing numeric literal and convert to the number.

Properties

hashcode	This property is used to get hash code of the given number.
isFinite	If the given number is finite, then this property will return true.
isInfinite	If the number is infinite, then this property will return true.
isNan	If the number is non-negative then this property will return true.
isNegative	If the number is negative then this property will return true.
sign	This property is used to get -1, 0 or 1 depending upon the sign of the given number.
isEven	If the given number is an even then this property will return true.
isOdd	If the given number is odd then this property will return true.

Methods

abs ()	This method gives the absolute value of the given number.
ceil ()	This method gives the ceiling value of the given number.
floor ()	This method gives the floor value of the given number.
compareTo ()	This method compares the value of the given number.
remainder ()	This method gives the truncated remainder after dividing the two numbers.
round ()	This method returns the round value of the number.
toDouble ()	This method gives the double equivalent representation of the number.
toInt ()	This method returns the integer equivalent representation of the number.
toString ()	This method returns the String equivalent representation of the number.
truncate ()	This method returns the integer after discarding fraction digits.

String

It is 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.

The string starts with the data type **var**:

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

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

You can put the value of an expression inside a string by using **$(expression)**. It will help the strings to concatenate very easily. If the expression is an identifier, you can skip the **{}**.

```
 void main () {
var string = 'I do coding';
var string1 = '$string on Geeks for Geeks';
print (string1);
}
```

```
I do coding on Geeks for Geeks.
```

Dart also allows us to concatenate the string by + operator as well as can just separate the two strings by Quotes. The concatenation also works over line breaks which is itself a very useful feature.

```
var string = 'Geeks''for''Geeks';
var str = 'Coding is ';
var str1 = 'Fun';
print (string);
print (str + str1);
```

```
GeeksforGeeks
Coding is Fun
```

We can also check whether two strings are equal by == operator. It compares every element of the first string with every element of the second string.

```
void main(){
var str = 'Geeks';
var str1 = 'Geeks';
if (str == str1){
print (True);
}
}
```

```
True
```

Raw String are useful when you want to define a String that has a lot of special characters. We can create a raw string by prefixing it with **r**.

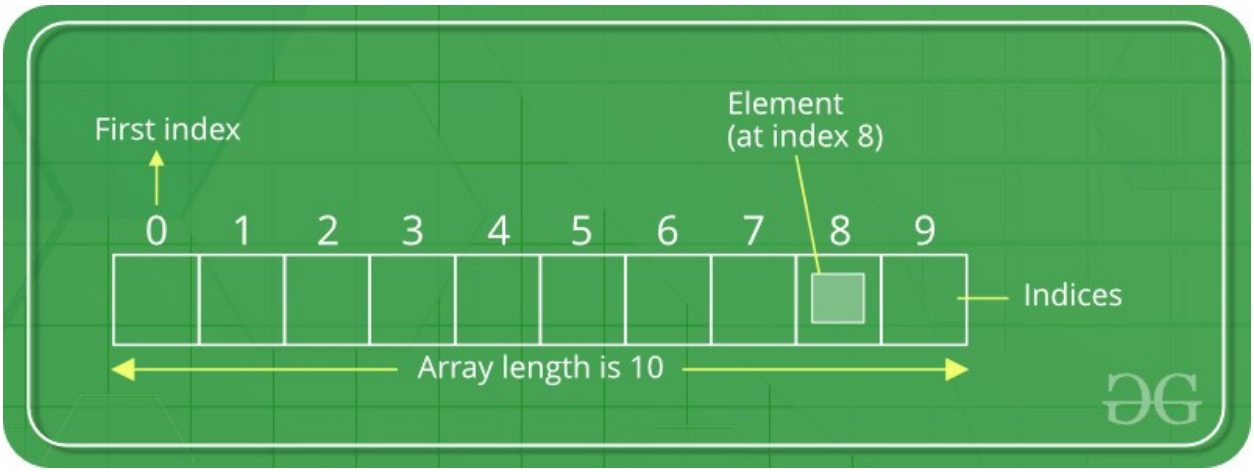
```
void main() {

  var gfg = r'This is a raw string';
  print(gfg);
}
```

```
This is a raw string
```

Dart Programming: List

In Dart Programming, List datatype is similar to arrays in other programming languages. List is used to represent a collection of objects. It is an **ordered** (ordered collection: elements are ordered in sequence) group of objects. The **core** libraries in Dart are responsible for the existence of List class, its creation and manipulation.

Logical Representation of List:



Index of the element represents the position of the specific data and when the list item of that index is called the element is displayed. Generally, the list item is called from its index.

Types of Lists

There are broadly two types of lists on the basis of its length:

1. Fixed Length List: Length once defined cannot be changed
2. Growable List: Length is dynamic

Fixed Length List

Here, the size of the list is declared initially and can't be changed during runtime.

Syntax:

```
Declaring a list
var list_name = new List (initial_size);

Initializing a list
list_name [index] = value;
```

NOTE: Collections are groups of objects that represent a particular element. The **dart::collection** library is used to implement the collection in dart. There are a variety of collections available in dart.

Examples:

```
void main()
{
   var gfg = new List(3);
   gfg[0] = 'Geeks';
   gfg[1] = 'For';
   gfg[2] = 'Geeks';

   // Printing all the values in List
   print(gfg);

   // Printing value at specific position
   print(gfg[2]);
}
```

```
[Geeks, For, Geeks]
Geeks
```

Growable List

This type of list is declared without declaring size of the list. Its length can be changed during runtime.

Syntax:

```
Declaring a List
var list_name = [val1,val2,val3]
--- creates a list containing the specified values
OR
var list_name = new List()
--- creates a list of size zero
Initializing a List
list_name[index] = value;
```

Adding value to the growable list:

```
void main()
{
   var gfg = [ 'Geeks', 'For' ];
   // Printing all the values in List
   print(gfg);
   // Adding new value in List and printing it
   gfg.add('Geeks'); // list_name.add(value);
   print(gfg);
}
```

```
[Geeks, For]
[Geeks, For, Geeks]
```

Adding multiple value to growable list:

```
void main()
{
   var gfg = [ 'Geeks' ];

   // Printing all the values in List
   print(gfg);

   // Adding multiple values in List and printing it

   // list_name.addAll([val 1, val 2, ...]);
   gfg.addAll([ 'For', 'Geeks' ]);
   print(gfg);
}
```

```
[Geeks]
[Geeks, For, Geeks]
```

Adding a value to growable list at specific index:

```
void main()
{
   var gfg = [ 'Geeks', 'Geeks' ];

   // Printing all the values in List
   print(gfg);

   // Adding new value in List at specific index and printing it

   // list_name.insert(index, value);
   gfg.insert(1, 'For');
   print(gfg);
}
```

```
[Geeks, Geeks]
[Geeks, For, Geeks]
```

Adding multiple value to growable list at specific indexes:

```
void main()
{
   var gfg = [ 'Geeks' ];

   // Printing all the values in List
   print(gfg);

   // Adding new value in List at specific index and printing it

   // list_name.insertAll(index, list_of_values);
   gfg.insertAll(1, [ 'For', 'Geeks' ]);
   print(gfg);

   // Element at index 1 in list
   print(gfg[1]);
}
```

```
[Geeks]
[Geeks, For, Geeks]
For
```

Dart: Sets

Sets in Dart is a special case in List where all the inputs are **unique** i.e., it doesn't contain any repeated input. It can also be interpreted as an **unordered** array with unique inputs. The set comes in play when we want to store unique values in a single variable without considering the order of the inputs. The sets are declared by the use of a **set** keyword.

```
var variable_name = <variable_type>{};

or,

Set <variable_type> variable_name = {};
```

NOTE:

1. Set is a collection of unique items and it does not contain duplicate elements.
2. You cannot get elements by Index, since the items are unordered.
3. **HashSet**: This is an implementation of unordered set. It is based on hash-table based Set implementation.

Example 1: Declaring set in two different ways.

```
// Dart program to show the Sets concept
void main()
{
  // Declaring set in First Way
  var gfg1 = <String>{'GeeksForGeeks'};

  // Printing First Set
  print("Output of first set: $gfg1");

  // Declaring set in Second Way
  Set<String> gfg2 = {'GeeksForGeeks'};

  // Printing Second Set
  print("Output of second set: $gfg2");
}
```

```
Output of first set: {GeeksForGeeks}
Output of second set: {GeeksForGeeks}
```

Example 2: Declaring repeated value in a set and a list and then comparing it.

```
// Dart Program to declare repeated value
// in a set and a list and then
// comparing it

void main()
{
  // Declaring list with repeated value
  var gfg = ['Geeks','For','Geeks'];

  // Printing List
  print("Output of the list is: $gfg");

  // Declaring set with repeated value
  var gfg1 = <String>{'Geeks','For','Geeks'};

  // Printing Set
  print("Output of the set is: $gfg1");
}
```

```
Output of the list is: [Geeks, For, Geeks]
Output of the set is: {Geeks, For}
```

NOTE: You can see that repeated value was simply ignored in the case of the set.

Adding Element in Set: To add element in the set we make use of **".add ()"** function or **".addAll ()"** function. But you must note that if you try to add a duplicate value using these functions than too they will get ignored in a set.

```
// To add single value
variable_name.add(value);

// To add multiple value
variable_name.addAll(value1, value2, value3, ...valueN);
```

Some other functions involving Sets are follows:

Serial No.	Syntax	Description
1	Variable_name.elementAt (index);	It returns the element at the corresponding index. The type of output depends on the type of set defined.
2	Variable_name.length;	It returns the length of the set. The output is of integer type.
3	Variable_name.contains (element_name);	It returns Boolean true if the element_name is present in the set else return false.
4	Variable_name.remove (element_name);	It deletes the element whose name is present inside it.
5	Variable_name.forEach (...);	It runs the command defined inside forEach () function for all the elements inside the set.
6	Variable_name.clear ();	It deletes all the element inside the set.

Converting Set to List in Dart: Dart also provides us with the method to convert the set into the list. To do, so we make use of **toList ()** method in Dart.

```
List<type> list_variable_name = set_variable_name.toList();
```

NOTE: It is useful in the way as the list we will get will contain unique values and no repeated values.

Converting Set to Map in Dart: Dart also provides us with the method to convert the set into the map.

```
// Declaring set 1 with value
 var gfg = <String>{"GeeksForGeeks","Geek1","Geek2","Geek3"};

 var geeksforgeeks = gfg.map((value) {
   return 'mapped $value';
 });
```

Dart Programming: Map

In Dart Programming, Maps are dictionary-like data types that exist in **key-value** form (known as lock-key). There is no restriction on the type of data that goes in a map data type. Maps are very flexible and can mutate their size based on the requirements. However, it is important to note that all locks (keys) need to be **unique** inside a map data type.

NOTE:

1. It is **unordered** collection of **key-value** pair.
2. Key-value can be of any object type. Each key in a map should be **unique** but the value can be **repeated**.
3. Map commonly called as **hash** or **dictionary**.
4. Size of map is **not fixed**; it can increase or decrease as per the number of elements.
5. **HashMap**: Implementation of Map and based on hash-table.

We can declare Map in two ways:

1. Using Map Literals
2. Using Map Constructors

Map Literals

Map can be declared using map literals as shown below:

```
Syntax:
// Creating the Map using Map Literals
var map_name = { key1 : value1, key2 : value2, ..., key n : value n }
```

Example 1: Creating Map using Map Literals

```
void main() {
  // Creating Map using is literals
  var gfg = {'position1' : 'Geek', 'position2' : 'for', 'position3' : 'Geeks'};
  // Printing Its content
  print(gfg);
  // Printing Specific Content
  // Key is defined
  print(gfg['position1']);
  // Key is not defined
  print(gfg[0]);
}
```

```
{position1: Geek, position2: for, position3: Geeks}
Geek
null
```

Example 2: You have noticed that different strings get concatenated to one.

```
void main() {
  // Creating Map using is literals
  var gfg = {'position1' : 'Geek' 'for' 'Geeks'};

  // Printing Its content
  print(gfg);

  // Printing Specific Content
  // Key is defined
  print(gfg['position1']);
}
```

```
{position1: GeekforGeeks}
GeekforGeeks
```

Example 3: Inserting a new value into map

```
void main() {
  // Creating Map
  var gfg = {'position1' : 'Geeks' 'for'  'Geeks'};

  // Printing Its content before insetion
  print(gfg);

  // Inserting a new value in Map
  gfg ['position0'] = 'Welcome to ';

  // Printing Its content after insertion
  print(gfg);

  // Printing Specific Content
  // Keys is defined
  print(gfg['position0'] + gfg['position1']);
}
```

```
{position1: GeeksforGeeks}
{position1: GeeksforGeeks, position0: Welcome to }
Welcome to GeeksforGeeks
```

Map Constructors

```
Syntax:
// Creating the Map using Map Constructor
var map_name = new Map();
// Assigning value and key inside Map
map_name [ key ] = value;
```

Example 1: Creating Map using Map Constructors

```
void main() {
  // Creating Map using Constructors
  var gfg = new Map();

  // Inserting values into Map
  gfg [0] = 'Geeks';
  gfg [1] = 'for';
  gfg [2] = 'Geeks';

  // Printing Its content
  print(gfg);

  // Printing Specific Content
  // Key is defined
  print(gfg[0]);
}
```

```
{0: Geeks, 1: for, 2: Geeks}
Geeks
```

Example 2: Assigning same key to different element

```
void main() {
  // Creating Map using Constructors
  var gfg = new Map();

  // Inserting values into Map
  gfg [0] = 'Geeks';
  gfg [0] = 'for';
  gfg [0] = 'Geeks';

  // Printing Its content
  print(gfg);

  // Printing Specific Content
  // Key is defined
```

```
  print(gfg[0]);
}
```

```
{0: Geeks}
Geeks
```

You have noticed that the other two values were simply ignored.

Queues in Dart

Dart also provides the user to manipulate a collection of data in the form of a queue. A queue is a **FIFO** (First in First Out) data structure where the element that is added first will be deleted first.

It takes the data from one end and removes it from the other end. Queues are useful when you want to build a first-in, first-out collection of data. It is the special case of list implementation of data in Dart.

Creating a Queue in Dart:

Using Constructor:

```
Queue variable_name = new Queue();
```

Through Existing List:

```
// With type notation(E)
Queue<E> variable_name = new Queue<E>.from(list_name);

// Without type notation
var variable_name = new Queue.from(list_name);
```

It must be noted that to use a queue in Dart program you have to import "**dart: collection**" module. If you don't do so then you will see some error.

Example 1: Creating a queue through a constructor and then inserting the elements in it.

```
import 'dart:collection';

void main()
{
  // Creating a Queue
  Queue<String> geek = new Queue<String>();

  // Printing default
  // value of queue
  print(geek);

  // Adding elements in a Queue
  geek.add("Geeks");
  geek.add("For");
  geek.add("Geeks");

  // Printing the
  // inserted elements
  print(geek);
}
```

```
{}
{Geeks, For, Geeks}
```

Example 2: Creating a queue through list

```
import 'dart:collection';

void main()
{
  // Creating a List
  List<String> geek_list = ["Geeks","For","Geeks"];

  // Creating a Queue through a List
  Queue<String> geek_queue = new Queue<String>.from(geek_list);

  // Printing the elements
  // in the queue
  print(geek_queue);
}
```

```
{Geeks, For, Geeks}
```

Functions of Queue in Dart

Dart also provides functions to manipulate queue created in the Dart program. Some important functions are listed in the table below.

Serial No.	Function Syntax	Description
1	queue_name.add (element)	Adds the element inside the queue from the front.
2	queue_name.addAll (collection_name)	Adds all the element present in the collection_name (generally List).
3	queue_name,addFirst (element)	Adds the element from front inside the queue.
4	queue_name,addLast (element)	Adds the element from back in the queue.
5	queue_name.clear ()	Removes all the elements from the queue.
6	queue_name.first ()	Returns the first element from the queue.
7	queue_name.forEach (f(element))	Returns all the element present in the queue.
8	queue_name.isEmpty ()	Returns Boolean true if the queue is empty else return false.
9	queue_name.length	Returns the length of the queue.
10	queue_name.removeFirst ()	Removes the first element from the queue.
11	queue_name.removeLast ()	Removes the last element from the queue.

Dart: Data Enumeration

Enumerated types (also known as **enumerations** or **enums**) are primarily used to define named constant values. The **enum** keyword is used to define an enumeration type in Dart. The use of enumeration is to store infinite data members under the same type definition.

```
Syntax:
enum variable_name{
  // Insert the data members as shown
  member1, member2, member3, ...., memberN
}
```

Let's analyze the above syntax:

1. The **enum** is the keyword used to initialize enumerated data type.
2. The **variable_name** as the name suggests is used for naming the enumerated class.
3. The data members inside the enumerated class must be separated by the commas.
4. Each data member is assigned an integer greater than the previous one, starting with 0 (by default).
5. Make sure not to use semi-colon or comma at the end of the last data member.

Example 1: Printing all the elements from the enum data class.

```
// dart program to print all the
// elements from the enum data class

// Creating enum with name Gfg
enum Gfg {

  // Inserting data
  Welcome, to, GeeksForGeeks
}

void main() {

  // Printing the value
  // present in the Gfg
  for (Gfg geek in Gfg.values) {
    print(geek);
  }
}
```

```
Gfg.Welcome
Gfg.to
Gfg.GeeksForGeeks
```

NOTE: Notice in the above example the strings are not enclosed with quotes, so that it can be used to print different results by comparing them with the values inside the enum.

Example 2: Using switch-case to print result

```
enum Gfg {
  Welcome, to, GeeksForGeeks
}

void main() {

  // Assign a value from
  // enum to a variable geek
  var geek = Gfg.GeeksForGeeks;

  // Switch-case
  switch(geek) {
    case Gfg.Welcome: print("This is not the correct case.");
    break;
    case Gfg.to: print("This is not the correct case.");
    break;
    case Gfg.GeeksForGeeks: print("This is the correct case.");
    break;
  }
}
```

```
This is the correct case.
```

NOTE: The enumerated class does not hold all types of data, rather it stores only string values without the quotes over them.

Limitation of Enumerated Datatype:

1. It cannot be subclassed or mixed in.
2. It is not possible to explicitly instantiate an enum.

Dart: Control Flow

If Else Statement (if, if-else, Nested if, if-else-if)

If the condition is true **then do something** else **do something else**.

```
Syntax:
if (condition){
// Execute the code if condition is true
} else {
// Execute code here if condition is false
}
```

If Else if Ladder Statement

Keep on checking the condition one by one.

```
if (condition1){
// Execute code here
} else if (condition2){
// Execute code here
} else if (condition3){
// Execute code here
} else {
// Finally execute code here if nothing works out
}
```

Switch Case Statement

It is similar to If Else If Ladder statements. The variable used in switch case can be: **Integer** [int] and **String** [String].

```
switch ( expression ) {
  case value1: {
    // Body of value1
  } break;
  case value2: {
    //Body of value2
  } break;
  .
  .
  default: {
    //Body of default case
  } break;
}
```

Dart: Loops

Looping statement in Dart or any other programming language is used to repeat certain set of commands until certain conditions are not completed.

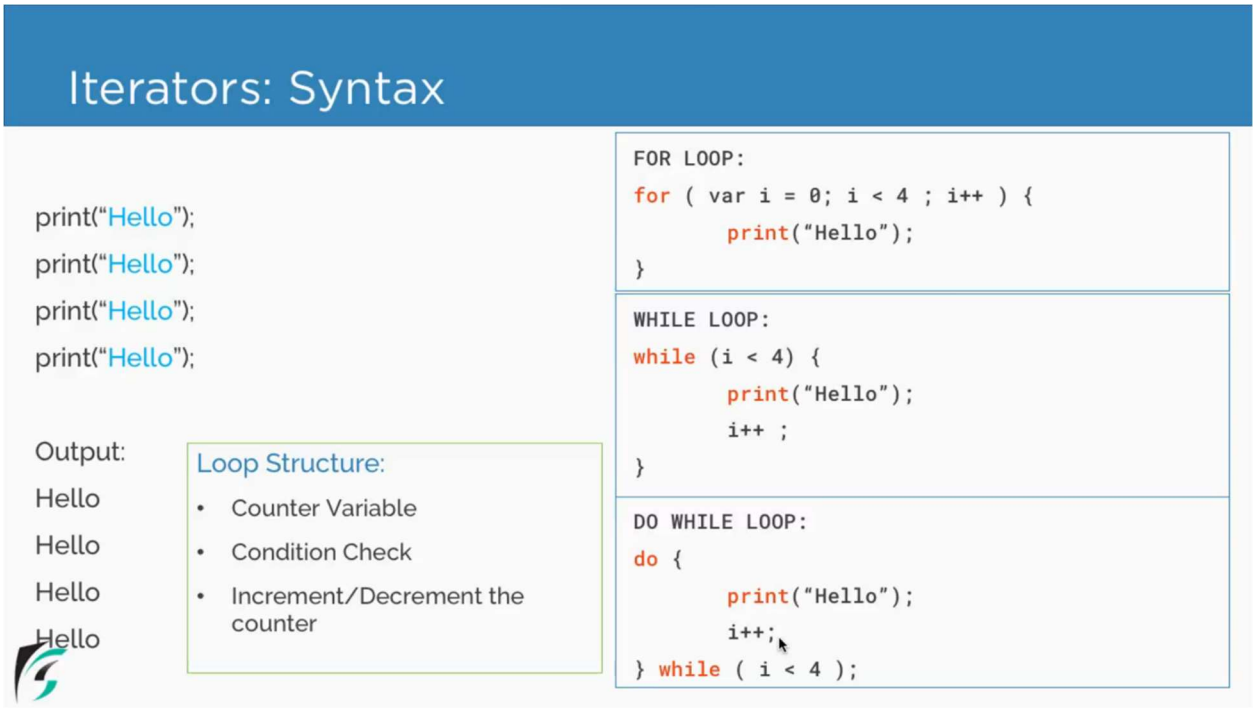
Loops are two types:

1. **Definite** Loops: For Loop
2. **Indefinite** Loops: While Loop, Do-While Loop

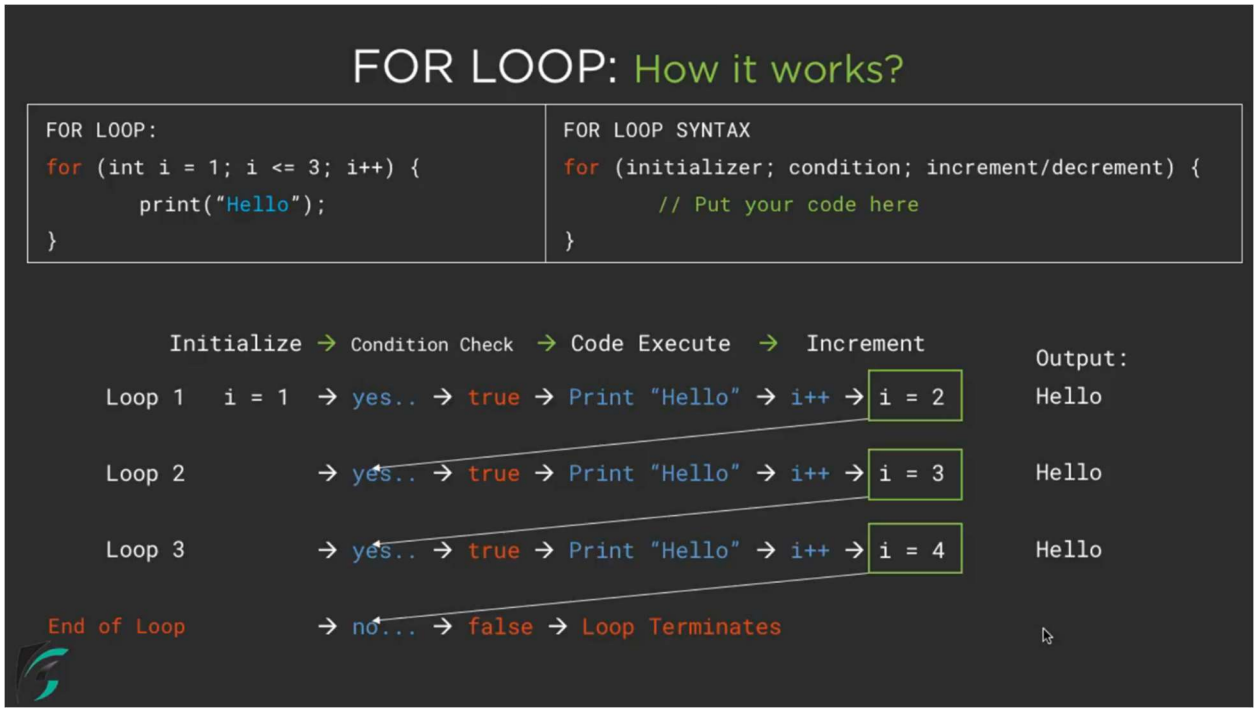
Difference between For, While and Do-While loops:

When to use:

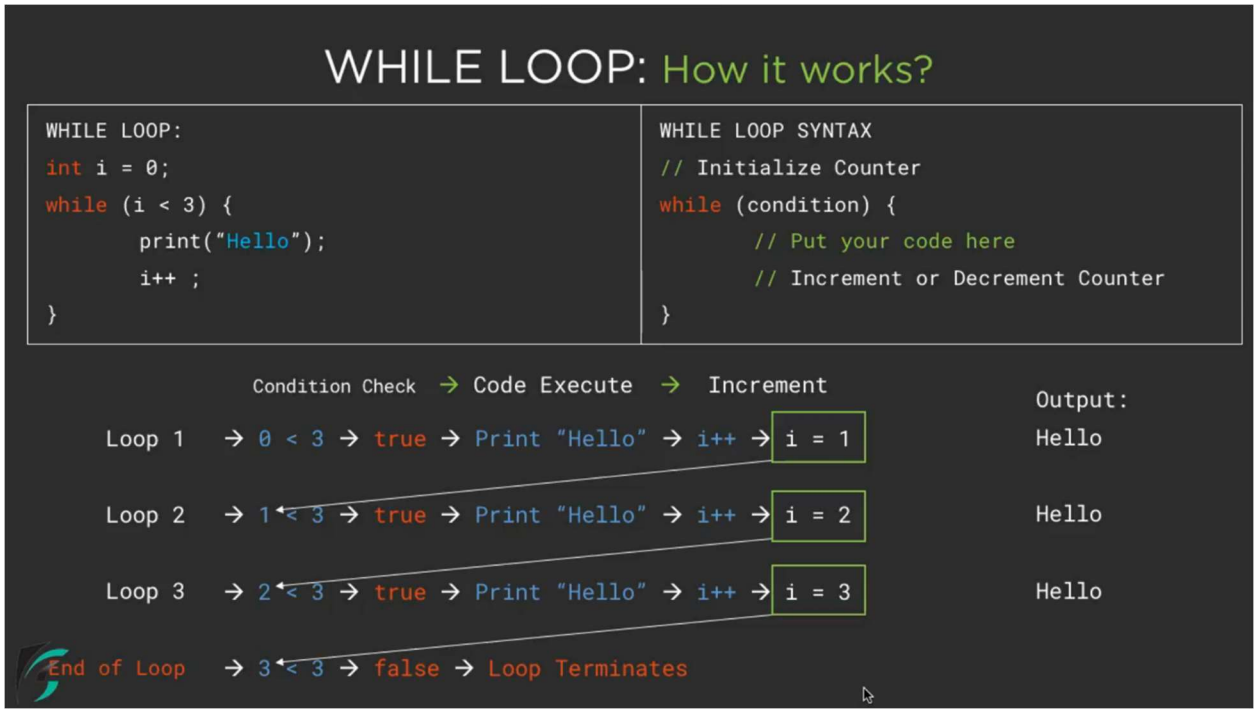
- If the number of iterations is fixed, it is recommended to use **for** loop.
- If the number of iterations is not fixed, it is recommended to use **while** loop.
- If the number of iterations is not fixed and you must have to execute the loop at least once, it is recommended to use the **do-while** loop.



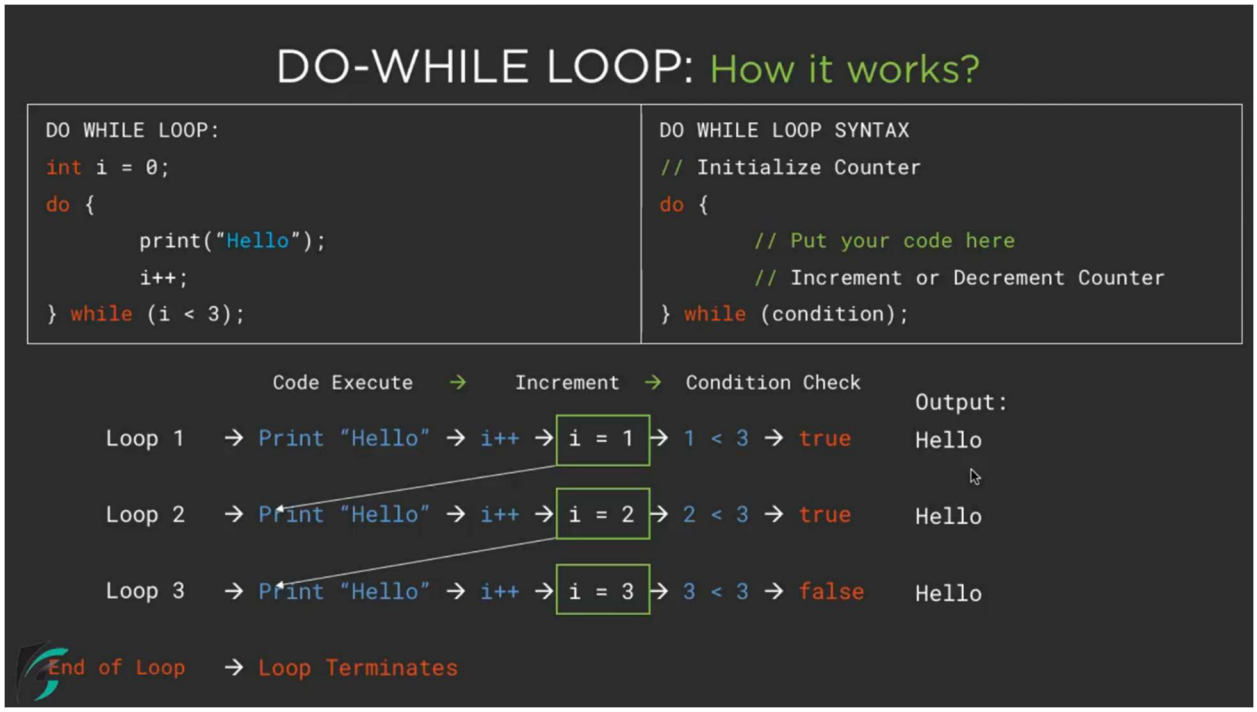
For Loop



While Loop



Do-While Loop



Dart: Loop Control Statements (Break and Continue)

Dart supports two types of loop control statements:

1. Break Statement
2. Continue Statement

Break Statement

This statement is used to break the flow of control of the loop i.e., if it is used within a loop then it will terminate the loop whenever encountered. It will bring the flow of control out of the nearest loop.

Example: Using Break inside for loop

```
void main()
{
  for (int i = 1; i <= 10; ++i) {
    if (i == 2)
      break;

    print("Geek, you are inside loop $i");
  }

  print("Geek, you are out of loop");
}
```

```
Geek, you are inside loop 1
Geek, you are out of loop
```

Break keyword is only applicable for its nearest **parent** loop means the inner loop. It has no impact on the outer loop.

Continue Statement

While the **Break** is used to end the flow of control, **continue** on the other hand is used to continue the flow of control. When a continue statement is encountered in a loop it doesn't terminate the loop but rather jump the flow to next iteration.

Example: using continue inside for loop

```
void main()
{
   for (int i = 1; i <= 10; ++i) {

      if (i == 2) {
         print("Geek, you are inside loop $i");
         continue;
      }
   }

   print("Geek, you are out of loop");
}
```

```
Geek, you are inside loop 2
Geek, you are out of loop
```

Labels in Dart

Dart has **lebels** which can be used with continue and break statements and help them to take a bigger leap in the code. It must be noted that line-breaks are not allowed between **'label-name'** and loop control statements.

Example 1: Using lebel with the break statement

```
void main() {

 // Defining the label
 Geek1:for(int i=0; i<3; i++)
 {
  if(i < 2)
  {
    print("You are inside the loop Geek");
```

```
    // breaking with label
    break Geek1;
   }
   print("You are still inside the loop");
  }
}
```

```
You are inside the loop Geek
```

The above code results into only one-time printing of statemen because once the loop is broken it doesn't go back into it.

Example 2: using label with the continue statement

```
void main() {

  // Defining the label
  Geek1:for(int i=0; i<3; i++)
  {
   if(i < 2)
   {
    print("You are inside the loop Geek");

    // Continue with label
    continue Geek1;
   }
   print("You are still inside the loop");
  }
}
```

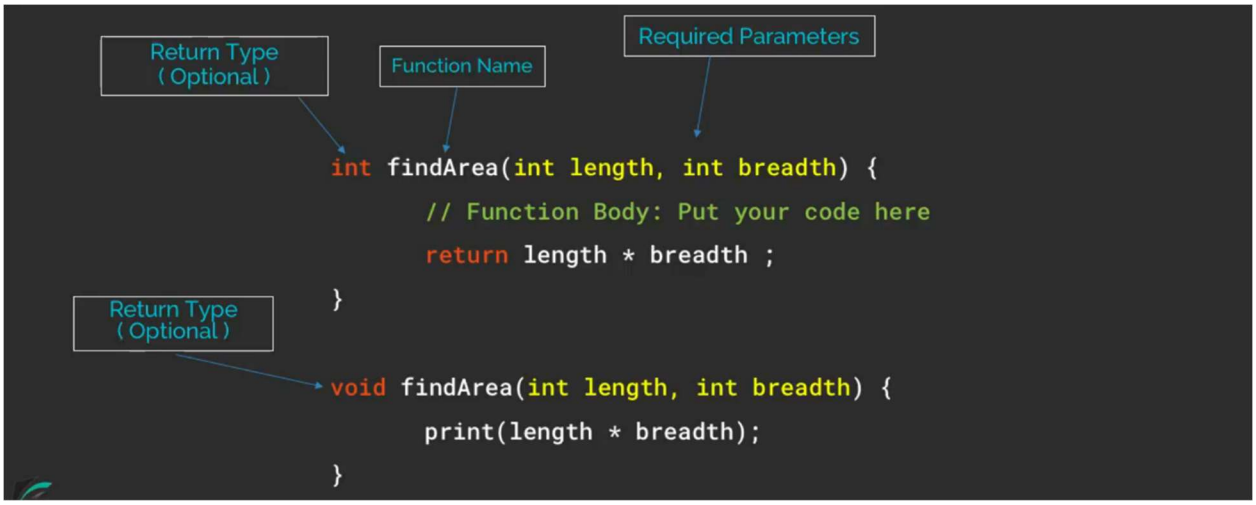
```
You are inside the loop Geek
You are inside the loop Geek
You are still inside the loop
```

The above code results in printing of the statement twice because of the condition it didn't break out of the loop and thus printing it twice.

Dart Programming: Function

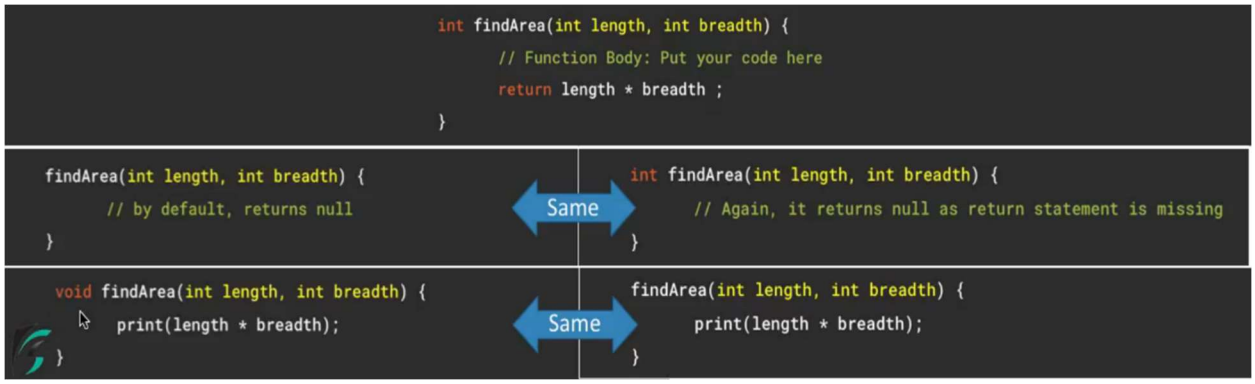
Function is a collection of statements grouped together to perform an operation.

```
return_type function_name ( parameters ) {
  // Body of function
  return value;
}
```



Properties of Functions in Dart

- Functions in Dart are **Objects**. Functions can be assigned to a variable or passed as parameter to other functions.
- All functions in Dart returns a value. If no return value is specified the function returns **null**.
- Specifying the return type is optional but is recommended as per code convention.



Parameters in Function

In Dart, Functions has two types of parameters:

1. Required Parameters
2. Optional parameters (Positional Parameter, Named Parameter, Default Parameter)

Named Parameter

- Named parameter prevent errors if there are large number of parameters.
- When we pass this parameter, it is mandatory to pass it while passing values. It is specified by **curly {}** brackets.

```
int findVolume( int length, { int breadth, int height } ) {
       return length * breadth * height;
}

var result = findVolume(2, breadth:3, height: 10);
print(result);      // 2*3*10 = 60

var result = findVolume(2, height: 10, breadth: 3); // Sequence does not matters
print(result);             // 2*3*10 = 60
```

Default Parameter

You can assign default values to parameters.

```
int findVolume( int length, int breadth, { int height = 10 } ) {
       return length * breadth * height;
}

var result = findVolume(2, 3);
print(result);      // 2*3*10 = 60

var result = findVolume(2, 3, height: 20);          // Overrides the default value
print(result);             // 2*3*20 = 120
```

Positional Parameter

To specify it use **square [ ]** brackets.

```
void gfg1(int g1, [ int g2 ])
{
   // Creating function 1
   print("g1 is $g1");
   print("g2 is $g2");
}

void main()
{
   // Calling the function with optional positional parameter
   print("Calling the function with optional parameter:");
   gfg1(01);
}
```

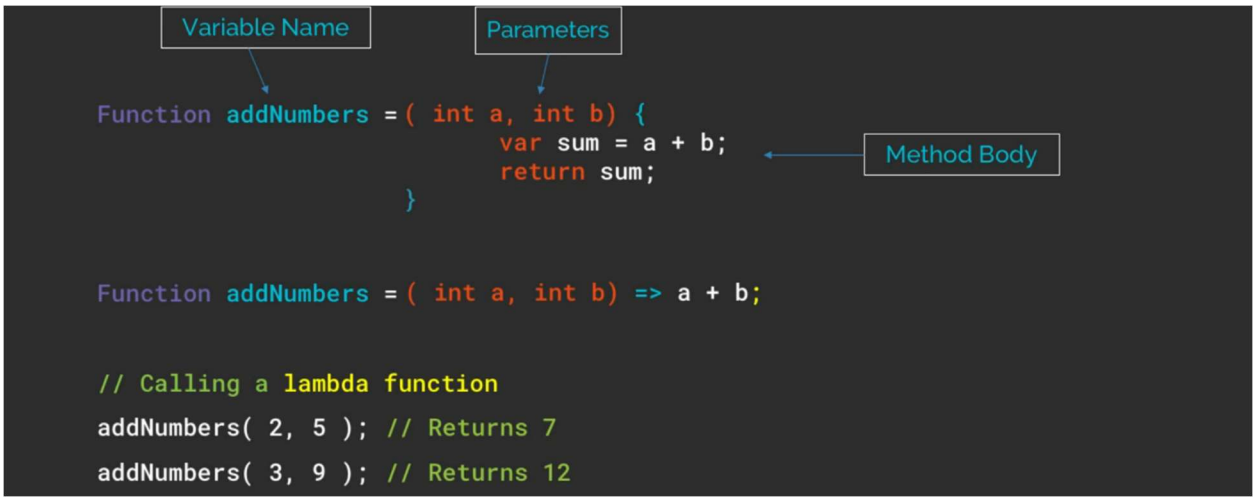
```
Calling the function with optional parameter:
g1 is 1
g2 is null
```

Dart: Lambda Function

They are the short way of representing a function in Dart. They are also called arrow function. But you should note that with lambda function you can return value for only one expression.

Lambda Expression:

- A function without a name.
- Also known as **anonymous** function or **lambda**.
- A function in Dart is an **object**.



Dart: Anonymous Functions

An **anonymous** function in Dart is like a named function but they do not have names associated with it. An **anonymous** function can have zero or more parameters with optional type annotations. An **anonymous** function consists of self-contained blocks of code and that can be passed around in our code as a function as a function parameter.

In Dart most of the functions are named functions we can also create nameless function known as an **anonymous** function, **lambda** or **closure**.

In Dart we can assign an anonymous function to constants or variables, later we can access or retrieve the value of closure based on our requirements:

```
Syntax:
(parameter_list)
{
    statement(s)
}
```

Example:

```
// Dartprogram to illustrate
// Anonymous functions in Dart
void main()
{
  var list = ["Shubham","Nick","Adil","Puthal"];
  print("GeeksforGeeks - Anonymous function in Dart");
  list.forEach((item) {
    print('${list.indexOf(item)} : $item');
  });
}
```

```
Output:
GeeksforGeeks – Anonymous function in Dart
0 : Shubham
1 : Nick
2 : Adil
3 : Puthal
```

This example defines an anonymous function with an untyped parameter, **item**. The function, invoked for each item in the list, prints a string that includes the value at the specified index.

Dart: Main () Function

The main () function is a predefined method in Dart. It is the most important and mandatory part of any Dart Program. Any Dart script requires the main () method for its execution. This method acts as the entry point for any Dart application. It is responsible for executing all library functions, user-defined statements, and user-defined functions.

```
void main()
{
   //main() function body
}
```

The main function can be further structured to variable declaration, function declaration, and executable statements. The main function returns void. Also, optional parameters List<String> may be used as arguments to the function. These arguments may be used in case we need to control our program from outside.

Example 1: The following example is a basic example of how the main function is the entry point of a Dart program.

```
main(){
   print("Main is the entry point!");
}
```

```
Main is the entry point!
```

Example 2: The following example shows how we can pass arguments inside the main () function.

```
main(List<String> arguments){
    //printing the arguments along with length
    print(arguments.length);
    print(arguments);
}
```

```
2
[Argument1, Argument2]
```

Dart: Common Collection Methods

List, Set, and Map share common functionality found in many collections. Some of this common functionality is defined by the Iterable class, which List and Set implement.

1. isEmpty () or isNotEmpty (): use isEmpty or isNotEmpty to check whether a list, set or map has items,

```
void main(){

var coffees = [];
var teas = ['green', 'black', 'chamomile', 'earl grey'];
print(coffees.isEmpty);
print(teas.isNotEmpty);
}
```

```
true
true
```

2. forEach (): To apply a function to each item in a list, set or map, you can use forEach ().

```
void main(){

var teas = ['green', 'black', 'chamomile', 'earl grey'];

var loudTeas = teas.map((tea) => tea.toUpperCase());
loudTeas.forEach(print);
}
```

```
GREEN
BLACK
CHAMOMILE
EARL GREY
```

3. where (): Use Iterable's where () method to get all the items that match a condition. Use Iterable's any () and every () methods to check whether some or all items match a condition.

```
void main(){
var teas = ['green', 'black', 'chamomile', 'earl grey'];
// Chamomile is not caffeinated.
bool isDecaffeinated(String teaName) =>
   teaName == 'chamomile';
// Use where() to find only the items that return true
// from the provided function.
// Use any() to check whether at least one item in the
// collection satisfies a condition.
print(teas.any(isDecaffeinated));
// Use every() to check whether all the items in a
// collection satisfy a condition.
print(!teas.every(isDecaffeinated));
}
```

```
true
true
```

how to Exit a Dart application unconditionally?

The exit () method exits the current program by terminating running Dart VM. This method takes a status code. A non-zero value of status code is generally used to indicate abnormal termination. This is a similar exit in C/C++, Java. This method doesn't wait for any asynchronous operations to terminate.

```
Syntax: exit(exit_code);
```

To use this method, we have to import **'dart:io'** package. The handling of exit code is platform-specific.

The **exit (0),** generally used to indicate successful termination while rest generally indicates unsuccessful termination.

Implementation of the exit () method is as:

```
void exit(int code) {
  ArgumentError.checkNotNull(code, "code");
  if (!_EmbedderConfig._mayExit) {
    throw new UnsupportedError(
        "This embedder disallows calling dart:io's exit()");
  }
  _ProcessUtils._exit(code);
}
```

Dart: Getters and Setters

Getters and **Setters**, also called **accessors** and **mutators**, allows the program to initialize and retrieve the values of class fields respectively.

- Getters or accessors are defined using the get keyword.
- Setters or mutators are defined using the set keyword.

A default getter/setter is associated with every class. However, the default ones can be overridden by explicitly defining a setter/getter. A getter has no parameters and returns a value, and the setter has one parameter and does not return a value.

```
Syntax: Defining a getter

Return_type  get identifier
{
   // statements
}
```

```
Syntax: Defining a setter

set identifier
{
  // statements
}
```

Example 1: The following example shows how you can use getters and setters in Dart class:

```
/ 1. Default Getter and Setter
// 2. Custom Getter and Setter
// 3. Private Instance Variable

void main() {
  var student = Student();
  student.name = "Peter"; // Calling default Setter to set value
  print(student.name); // Calling the Getter to get value

  student.percentage = 438.0; // Calling custom setter to set value
  print(student.percentage); // Calling custom getter to get value
}

class Student {
  String name = "ashu"; // Instance Variable with default Getter and Setter
```

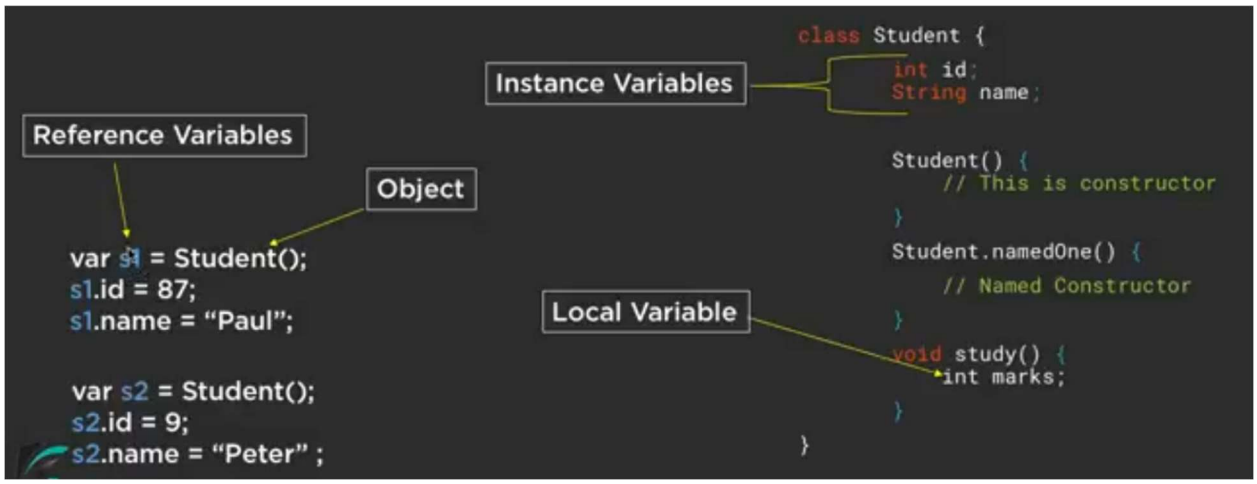
```
  double _percent = 67.8; // Private Instance variable for its own library

// Instance variable with Custom Setter
// Compact code : void set percentage(double marksSecured) => _percent = (marksSecured / 500) * 100;
  void set percentage(double marksSecured) {
    _percent = (marksSecured / 500) * 100;
  }

// Instance Variable with Custom Getter
// Compact code : double get percentage => _percent;
  double get percentage {
    return _percent;
  }
}
```

```
Peter
87.6
```

Dart: Object-Oriented Programming

Dart is an Object-Oriented programming language, so it supports the concept of class, object...etc. In Dart, we can define classes and objects of our own. We use **class** keyword to do so.

A **class** is a blueprint to create objects.



Declaring class in Dart:

```
class class_name {

  // Body of class
}
```

Declaring Objects in Dart:

Objects are the instance of the class and they declared by using **new** keyword followed by the class name, but new keyword is optional:

```
var object_name = new class_name([ arguments ]);
```

after the object is created, there will be the need to access the fields which we will create. We use the **dot (.) operator** for that purpose.

```
// For accessing the property
object_name.property_name;

// For accessing the method
object_name.method_name();
```

Example:

```
void main() {
  // One Object, student1 is reference variable.
  var student1 = Student();
  student1.id = 23;
  student1.name = "Peter";
  print("${student1.id} and ${student1.name}");

  student1.study();
  student1.sleep();

// Two Object, student2 is reference variable.
  print("");
  var student2 = Student();
  student2.id = 22;
  student2.name = "Buttler";
  print("${student2.id} and ${student2.name}");

  student2.study();
  student2.sleep();
}
// Define States (Properties) and behavior of a Student
class Student {
  int id = -1; // Instance or Field Variable, default value is -1.
  String name = "ashu"; // Instance or Field Variable, default value is ashu.

  void study() {
    print("${this.name} is now studying");
  }

  void sleep() {
    print("${this.name} is now sleeping");
  }
}
```

```
23 and Peter
Peter is now studying
Peter is now sleeping

22 and Buttler
Buttler is now studying
Buttler is now sleeping
```

Constructor in Dart

Dart also provides the support of constructors. Constructors are a special method that is used to initialize an object when created in the program. In object-oriented programming when an object is created, it automatically calls the constructor.

All classes have their default constructor which is created by the compiler when class is called, moreover one can also define constructor of its own. But you must note that if you do so then the default constructor will not be created and will be ignored.

Definition

The constructors have the same name as the class name and don't have any return type.

```
class_name( [ parameters ] ){
   // Constructor Body
}
```

In the above syntax:

1. **Class_name**: is the name of the class whose constructor is being created.
2. **Parameters**: are optional features and they can and can't be defined for the constructor. The default constructor has no parameter defined in it.
3. **Constructor body**: is the body of the constructor and is executed when the constructor is called i.e., when an object is created.
4. Constructors don't have any return type.

Example 1: Creating a constructor in Dart

```
// Dart Program to create a constructor

// Creating Class named Gfg
class Gfg{

 // Creating Constructor
 Gfg() {

  // Whenever constructor is called this statement will run
  print('Constructor is being created');
 }

 // Creating Field inside the class
 String geek1;

 // Creating Function inside class
 void geek(){
  print("Welcome to $geek1");
```

```
  }
}

void main() {

  // Creating Instance of class
  Gfg geek = new Gfg();

  // Calling field name geek1 and assigning value to it using object of the class Gfg
  geek.geek1 = 'GeeksforGeeks';

  // Calling function name geek using object of the class Gfg
  geek.geek();
}
```

```
Constructor is being created
Welcome to GeeksforGeeks
```

There are three types of constructors in Dart:

1. Default Constructor
2. Parameterized Constructor
3. Named Constructor

Properties of Constructor:

1. Used to create **objects**
2. You can initialize your **instance** or **field** variables within constructors
3. You **cannot** have both **default** and **parameterized** constructors at the same time
4. You can have as many **Named** constructor as you want to

NOTE:

1. By default, a constructor in a subclass calls the superclass's no-argument constructor.
2. Parent class constructor is always called before child class constructor.
3. If default constructor is missing in parent class, then you must manually call one of the constructors in super class.

Default Constructor

The default constructors are those constructors that don't have any parameters in it. Thus, if a constructor which don't have any parameter, then it will be a type of default constructor.

Example: Creating default constructor in Dart

```
// Dart program to illustrate the Default constructor
// Creating Class named Gfg
class Gfg{
  // Creating Constructor
  Gfg() {
    print('This is the default constructor');
  }
}
void main() {
  // Creating Instance of class
  Gfg geek = new Gfg();
}
```

```
This is the default constructor
```

Parameterized Constructor

In Dart, you can also create a constructor having some parameters. These parameters will decide which constructor will be called and which will be not. Those constructors which accept parameters is known as parameterized constructor.

Example:

```
// Creating parameterized constructor in Dart
// Creating Class named Gfg
class Gfg{
  // Creating Parameterized Constructor
  Gfg(int a) {
    print('This is the parameterized constructor');
  }
}
void main() {
  // Creating Instance of class
  Gfg geek = new Gfg(1);
}
```

```
This is the parameterized constructor
```

NOTE: You can't have two constructors with the same name although they have different parameters. The compiler will display an error.

Named Constructor

As you can't multiple constructors with the same name, this type of constructor is the solution to the problem. They allow the user to make multiple constructors with a different name.

```
class_name.constructor_name ( parameters ){
  // Body of Constructor
}
```

Example:

```
// Creating named constructor in Dart
// Creating Class named Gfg
class Gfg{

  // Creating named and
  // parameterized Constructor
  // with one parameter
  Gfg.constructor1(int a) {
    print('This is the parameterized constructor with only one parameter');
  }
  // Creating named and
  // parameterized Constructor
  // with two parameter
  Gfg.constructor2(int a, int b) {
    print('This is the parameterized constructor with two parameters');
    print('Value of a + b is ${a + b}');
  }
}

void main() {
  // Creating Instance of class
  Gfg geek1 = new Gfg.constructor1(1);
  Gfg geek2 = new Gfg.constructor2(2, 3);
}
```

```
This is the parameterized constructor with only one parameter
This is the parameterized constructor with two parameters
Value of a + b is 5
```

Dart: Super Constructor

In Dart, the subclass can inherit all the variables and methods of the parent class, with the use of **extends** keyword but it can't inherit constructor of the parent class. To do so we make use of **super constructor** in the dart.

There are two ways to call super constructor:

1. Implicitly
2. Explicitly

When calling explicitly we make use of super constructor as:

```
Child_class_constructor() :super() {
...
}
```

Implicit super

In this case, the parent class is called implicitly, when there is object creation of child class. Here we don't make use of the super constructor but when the child class constructor is invoked then it calls default parent class constructor.

Example: Calling parent constructor taking no parameters.

```
// Dart program for calling parent constructor taking no parameter
class SuperGeek {
  // Creating parent constructor
  SuperGeek(){
    print("You are inside Parent constructor!!");
  }
}
class SubGeek extends SuperGeek {

  // Creating child constructor
  SubGeek(){
    print("You are inside Child constructor!!");
  }
}

void main() {
  SubGeek geek = new SubGeek();
}
```

```
You are inside Parent constructor!!
You are inside Child constructor!!
```

Explicit super

If the parent constructor is default, then we call it as followed in implicit super, but if it takes parameters then the superclass is invoked as shown in the syntax mentioned above.

Example: Calling parent constructor taking parameters.

```
class SuperGeek {

  // Creating parent constructor
  SuperGeek(String geek_name){
    print("You are inside Parent constructor!!");
    print("Welcome to $geek_name");
  }
}

class SubGeek extends SuperGeek {

  // Creating child constructor
  // and calling parent class constructor
  SubGeek() : super("Geeks for Geeks"){
    print("You are inside Child constructor!!");
  }
}

void main() {
  SubGeek geek = new SubGeek();
}
```

```
You are inside Parent constructor!!
Welcome to Geeks for Geeks
You are inside Child constructor!!
```

Dart: this keyword

This keyword represents an implicit object pointing to the current class object. It refers to the current instance of the class in a method or constructor. **This** keyword is mainly used to eliminate the ambiguity between class attributes and parameters with the same name.

When the class attributes and the parameter names are the same this keyword is used to avoid ambiguity by prefixing class attributes with this keyword. This keyword can be used to refer to any member of the current object from within an instance method or a constructor.

Used of this keyword:

1. It can be used to refer to the instance variable of the current class.
2. It can be used to make or initiate current class constructor.
3. It can be passed as an argument in the method call.
4. It can be passed as an argument in the constructor call.
5. It can be used to make a current class method.
6. It can be used to return the current class instance.

Example 1: The following example shows the use of this keyword

```
// Dart program to illustrate this keyword
void main()
{
  Student s1 = new Student('S001');
}

class Student
{
  // defining local st_id variable
  var st_id;
  Student(var st_id)
  {
    // using this keyword
    this.st_id = st_id;
    print("GFG - Dart THIS Example");
    print("The Student ID is : ${st_id}");
  }
}
```

```
GFG – Dart THIS Example
The Student ID is : S001
```

Example 2:

```
// Dart program to illustrate
// this keyword
void main() {
  mob m1 = new mobile('M101');
}
class mob {
  String mobile;
  Car(String mobile) {

    // use of this keyword
    this.mobile = mobile;
    print("The mobile is : ${mobile}");
  }
}
```

```
The mobile is : M101
```

Dart: Static Keyword

The **static** keyword is used for **memory management** of global data members. The static keyword can be applied to the fields and methods of a class. The static variables and methods are part of the class instead of a specific instance.

- The **static** keyword is used for a class-level variable and method that is the same for every instance of a class, this means if a data member is static, it can be accessed **without creating** an object.
- The static keyword allows data members to persist values between different instances of a class.
- There is no need to create a class object to access a static variable or call static method: simply put the class name before the static variable or method name to use them.

NOTE:

1. Static variables are also known as **Class Variables**.
2. Static methods are also known as **Class Methods**.
3. Static variables are **lazily** initialized,
    a. i.e., they are not initialized until they are used in program.
    b. So, they consume memory only when they are used.
4. Static method has nothing to do with the class objects or instance.
    a. You cannot use **this** keyword within a static method.
5. Form a static method,
    a. You can **ONLY** access Static method and static variables.
    b. But you **cannot** access normal instance variables and methods of the class.

Static Variables

The static variables belong to the class instead of a specific instance. A static variable is common to all instances of a class: this means only a single copy of the static variable is shared among all the instances of a class. The memory allocation for static variables happens only once in the class area at the time of class loading.

Declaring Static Variables:

Static variables can be declared using static keyword followed by data type then the variable name:

```
Syntax: static [date_type] [variable_name];
```

Accessing Static Variable:

The static variable can be accessed directly from the class name itself rather than creating an instance of it.

```
Syntax: Classname.staticVariable;
```

Static Methods

The static method belongs to a class instead of class instances. A static method is only allowed to access the static variable of class and can invoke only static methods of the class. Usually, utility methods are created as static methods when we want it to be used by other classes without the need of creating an instance.

Declaring Static Method

A static method can be declared using static keyword followed by return type, followed by method name:

```
Syntax:

static return_type method_name()
{
   // Statement(s)
}
```

Calling Static Method

Static methods can be invoked directly from the class name itself rather than creating an instance of it.

```
Syntax: ClassName.staticMethod();
```

Example 1:

```
// Dart Program to show Static methods in Dart
class Employee {
  static var emp_dept;
  var emp_name;
  int emp_salary;

  // Function to show details of the Employee
  showDetails() {
    print("Name of the Employee is: ${emp_name}");
    print("Salary of the Employee is: ${emp_salary}");
    print("Dept. of the Employee is: ${emp_dept}");
  }
}

// Main function
void main() {
  Employee e1 = new Employee();
  Employee e2 = new Employee();
  Employee.emp_dept = "MIS";

  print("GeeksforGeeks Dart static Keyword Example");
  e1.emp_name = 'Rahul';
  e1.emp_salary = 50000;
  e1.showDetails();

  e2.emp_name = 'Tina';
  e2.emp_salary = 55000;
  e2.showDetails();
}
```

```
GeeksforGeeks Dart static Keyword Example
Name of the Employee is: Rahul
Salary of the Employee is: 50000
Dept. of the Employee is: MIS
Name of the Employee is: Tina
Salary of the Employee is: 55000
Dept. of the Employee is: MIS
```

Example 2:

```
// Dart program in dart to
// illustrate static method
class StaticMem {
  static int num;
  static disp() {
    print("#GFG the value of num is ${StaticMem.num}")  ;
  }
}
void main() {
  StaticMem.num = 75;

  // initialize the static variable }
  StaticMem.disp();

  // invoke the static method
}
```

```
#GFG the value of num is 75
```

Dart: Super Keyword

In Dart, **super** keyword is used to refer immediate parent class object. It is used to call properties and methods of the superclass. It does not call the method, whereas when we create an instance of subclass than that of the parent class is created implicitly so super keyword calls that instance.

Advantages of super keyword:

- It can be used to access data members of parent class when both parent and child have member with same name.
- It is used to prevent overriding the parent method.
- It can be used to call parameterized constructor of parent class.

NOTE: While **super** keyword is used to call parent class, **this** keyword is used to call the class itself.

```
Syntax:
// To access parent class variables
super.variable_name;

// To access parent class method
super.method_name();
```

Example 1: Showing the flow of object creation in inheritance

```
class SuperGeek {

   // Creating parent constructor
   SuperGeek()
   {
      print("You are inside Parent constructor!!");
   }
}

class SubGeek extends SuperGeek {

   // Creating child constructor
   SubGeek()
   {
      print("You are inside Child constructor!!");
   }
}

void main()
{
   SubGeek geek = new SubGeek();
}
```

```
You are inside Parent constructor!!
You are inside Child constructor!!
```

Example 2: Accessing parent class variables

```
// Creating Parent class
class SuperGeek {
   String geek = "Geeks for Geeks";
}

// Creating child class
class SubGeek extends SuperGeek {

   // Accessing parent class variable
   void printInfo()
   {
      print(super.geek);
   }
}

void main()
{
```

```
    // Creating child class object
    SubGeek geek = new SubGeek();

    // Calling child class method
    geek.printInfo();
}
```

```
Geeks for Geeks
```

Example 3: Accessing parent class methods

```
class SuperGeek {

    // Creating a method in Parent class
    void printInfo()
    {
        print("Welcome to Gfg!!\nYou are inside parent class.");
    }
}

class SubGeek extends SuperGeek {

    void info()
    {
        print("You are calling method of parent class.");

        // Calling parent class method
        super.printInfo();
    }
}

void main()
{
    SubGeek geek = new SubGeek();
    geek.info();
}
```

```
You are calling method of parent class.
Welcome to Gfg!!
You are inside parent class.
```

Dart: Const and Final Keyword

Dart supports the assignment of constant value to a variable. These are done by the use of the following keyword:

1. Const keyword
2. Final keyword

These keywords are used to keep the value of a variable static throughout the code base, meaning once the variable is defined its state cannot be altered. There are no limitations if these keywords have a defined data type or not.

Final Keyword

The final keyword is used to hardcode the values of the variable and it cannot be altered in future, neither any kind of operations performed on these variables can alter its value (state).

```
// Without datatype
final variable_name;

// With datatype
final data_type  variable_name;
```

Example: using the final keywords in Dart program. If we try to reassign the same variable then it will display error.

```
void main() {

  // Assigning value to geek1
  // variable without datatype
  final geek1 = "Geeks For Geeks";

  // Printing variable geek1
  print(geek1);

  // Assigning value to geek2
  // variable with datatype
  final String geek2 = "Geeks For Geeks Again!!";

  // Printing variable geek2
  print(geek2);
}
```

```
Geeks For Geeks
Geeks For Geeks Again!!
```

Const Keyword

The const keyword in Dart behaves exactly like the final keyword. The only **difference** between final and const is that the const makes the variable constant from **compile-time** only. Using const on an object, makes the object's entire deep state strictly fixed at compile-time and that the object with this state will be considered **frozen** and **completely immutable**.

Example: Using const keywords in a Dart program

```
void main() {

  // Assigning value to geek1
  // variable without datatype
  const geek1 = "Geeks For Geeks";

  // Printing variable geek1
  print(geek1);

  // Assigning value to
  // geek2 variable with datatype
  const String geek2 = "Geeks For Geeks Again!!";

  // Printing variable geek2
  print(geek2);
}
```

```
Geeks For Geeks
Geeks For Geeks Again!!
```

Constant Keyword Properties:

1. You cannot declare const keyword inside the class directly. If you want to, then you can simply use the '**static**' keyword.
2. It is necessary to create them from the data available during the compile time. For instance: setting string "GeeksForGeeks" is fine but setting the current time is not.
3. They are **deeply** and **transitively immutable**.
4. They are canonicalized.

Dart Programming: Inheritance

In Dart, one class can inherit another class i.e., Dart can create a new class from an existing class. We make use of **extend** keyword to do so.

Inheritance is a mechanism in which one object acquires properties of its parent class object.

Super class of any class:

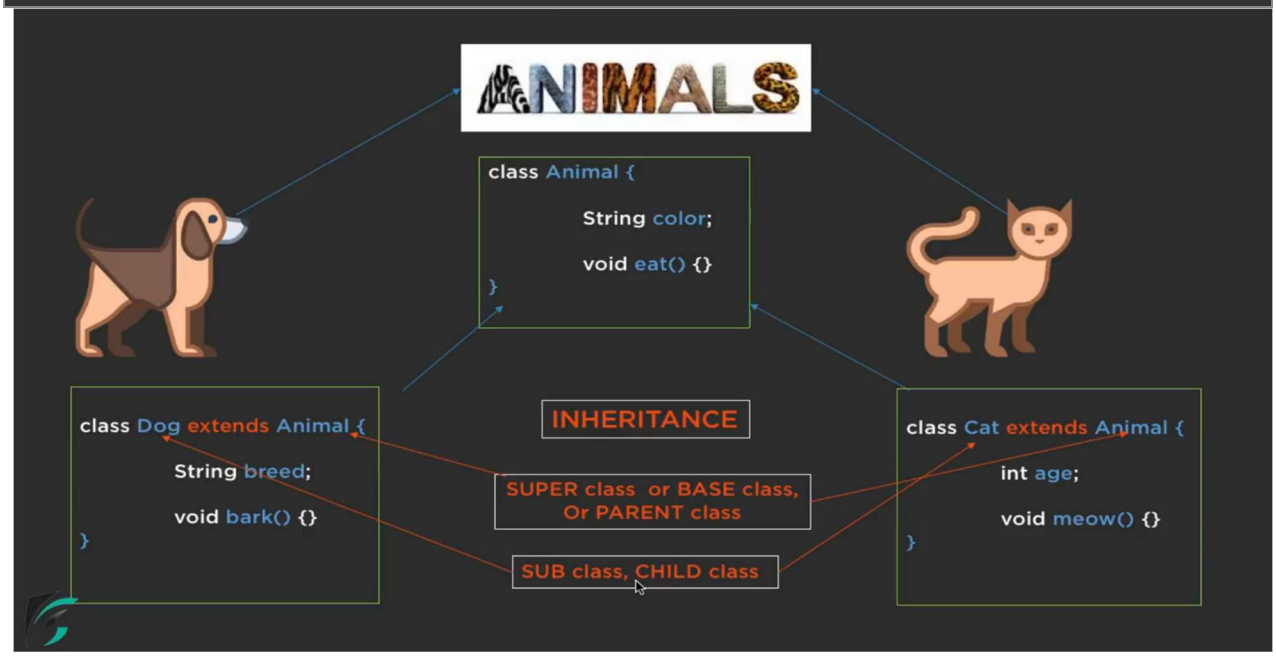
- Provides default implementation of:
  - **toString ()**, returns the String representation of the object
  - **hashCode** Getter, returns the Hash Code of an object
  - **operator = =,** to compare two objects

Advantages:

1. Code reusability
2. Method Overriding
3. Cleaner code: no repetition

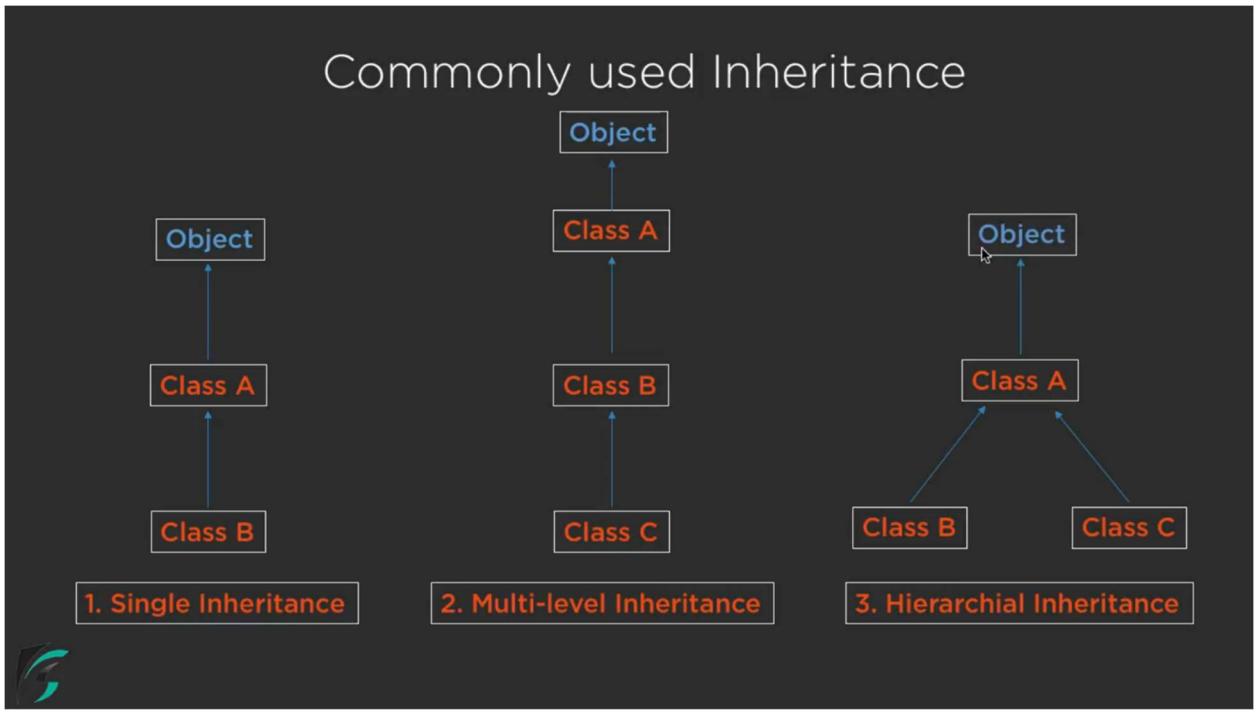
Syntax:

```
class parent_class{
...
}

class child_class extends parent_class{
...
}
```



Types of Inheritance class:

1. **Single Inheritance**: When a class inherits a single parent class then this inheritance occurs.
2. **Multiple Inheritance**: When a class inherits more than one parent class then this inheritance occurs. Dart **doesn't** support this.
3. **Multi-level Inheritance**: When a class inherits another child class then this inheritance occurs.
4. **Hierarchical Inheritance**: More than one classes have the same parent class.



Example 1: Example of Single Inheritance in Dart

```
// Dart program to show the single inheritance
// Creating parent class
class Gfg{

  // Creating a function
  void output(){
    print("Welcome to gfg!!\nYou are inside output function.");
  }
}

// Creating Child class
class GfgChild extends Gfg{
  // We are not defining
  // any thing inside it...
}
void main() {
```

```
  // Creating object of GfgChild class
  var geek = new GfgChild();

  // Calling function
  // inside Gfg(Parent class)
  geek.output();
}
```

```
Welcome to gfg!!
You are inside output function.
```

NOTE:

1. Child class inherit all properties and methods except constructors of the parent class.
2. Unlike Java, Dart also doesn't support multiple inheritance.

Example 2: Multi-level Inheritance

```
// Dart program for multilevel inheritance

// Creating parent class
class Gfg{

  // Creating a function
  void output1(){
    print("Welcome to gfg!!\nYou are inside the output function of Gfg class.");
  }
}

// Creating Child1 class
class GfgChild1 extends Gfg{

  // Creating a function
  void output2(){
    print("Welcome to gfg!!\nYou are inside the output function of GfgChild1 class.");
  }
}

// Creating Child2 class
class GfgChild2 extends GfgChild1{
  // We are not defining
  // any thing inside it...
}

void main() {
  // Creating object
  // of GfgChild class
```

```
  var geek = new GfgChild2();

  // Calling function
  // inside Gfg
  //(Parent class of Parent class)
  geek.output1();

  // Calling function
  // inside GfgChild
  // (Parent class)
  geek.output2();
}
```

```
Welcome to gfg!!
You are inside the output function of Gfg class.
Welcome to gfg!!
You are inside the output function of GfgChild1 class.
```

Example 3: Hierarchical Inheritance

```
// Dart program for Hierarchical inheritance

// Creating parent class
class Gfg{

  // Creating a function
  void output1(){
    print("Welcome to gfg!!\nYou are inside output function of Gfg class.");
  }
}

// Creating Child1 class
class GfgChild1 extends Gfg{
  // We are not defining
  // any thing inside it...
}

// Creating Child2 class
class GfgChild2 extends Gfg{

  // We are not defining
  // any thing inside it...
}

void main() {
  // Creating object
```

```
  // of GfgChild1 class
  var geek1 = new GfgChild1();
  // Calling function
  // inside Gfg(Parent class)
  geek1.output1();
  // Creating object of
  // GfgChild1 class
  var geek2 = new GfgChild2();
  // Calling function
  // inside Gfg(Parent class)
  geek2.output1();
}
```

```
Welcome to gfg!!
You are inside output function of Gfg class.
Welcome to gfg!!
You are inside output function of Gfg class.
```

Dart: Instance and Class Methods

Dart provides us with the ability to create methods of our own. The methods are created to perform certain actions in class. Methods help us to remove the complexity of the program. It must be noted that methods may and may not return any value and also it may or may not take any parameter as input. Methods in a class can be either an object method or a class method.

There are two types of methods in Dart:

1. **Instance Method**
2. **Class Method**

Instance Method in Dart:

Unless the method is declared as static it is classified as an instance method in a class. They are allowed to access instance variables. To call the method of this class you have to first create an object.

Syntax:

```
// Declaring instance method
return_type method_name() {
  // Body of method
}
// Creating object
class_name object_name = new class_name();

// Calling instance method
object_name.method_name();
```

creating instance method in Dart:

```
// Creating Class named Gfg
class Gfg {
   // Declaring instance variable
   int a;
   int b;

   // Creating instance method name
   // sum with two parameters
   void sum(int c, int d)
   {
      // Using this to set the values of the
      // input to instance variable
      this.a = c;
      this.b = d;

      // Printing the result
      print('Sum of numbers is ${a + b}');
   }
}

void main()
{
   // Creating instance of the class Gfg(Creating Object)
   Gfg geek = new Gfg();

   // Calling the method sum with the use of object
   geek.sum(21, 12);
}
```

```
Sum of numbers is 33
```

Class Method in Dart:

All the methods declared with static keyword are termed as class method. They can't access non-static variables and can't invoke non-static methods of the class. It must be noted that unlike instance method class method can directly be called by using class name.

Creating class method in Dart:

```
// Creating Class named Gfg
class Gfg {

   // Creating a class method name
   // sum with two parameters
   static void sum(int c, int d)
   {
      // Printing the result
      print('Sum of numbers is ${c + d}');
   }
}

void main()
{
   // Calling the method sum without the
   // use of object i.e with class name
   Gfg.sum(11, 32);
}
```

```
Sum of numbers is 43
```

Dart: Method Overriding

Method overriding is a mechanism by which the child class redefines a method in its parent class.

Method overriding occurs in dart when a child class tries to override the parent class's method. When a child class extends a parent class, it gets full access to the methods of the parent class and thus it overrides the methods of the parent class. It is achieved by re-defining the same method present in the parent class.

This method is helpful when you have to perform different functions for a different child class, so we can simply re-define the content by overriding it.

Important Points:

1. A method can be overridden only in the child class, not in the parent class itself.
2. Both the methods defined in the child and the parent class should be the exact copy, from name to argument list except the content present inside the method i.e., it can and can't be the same.
3. A method declared final or static inside the parent class can't be overridden by the child class.
4. Constructors of the parent class can't be inherited, so they can't be overridden by the child class.

Example 1: Simple case of method overriding

```
// Dart Program to illustrate the method overriding concept
class SuperGeek {
  // Creating a method
  void show(){
    print("This is class SuperGeek.");
  }
}
class SubGeek extends SuperGeek {

  // Overriding show method
  void show(){
    print("This is class SubGeek child of SuperGeek.");
  }
}
void main() {
  // Creating objects
  //of both the classes
  SuperGeek geek1 = new SuperGeek();
  SubGeek geek2 = new SubGeek();
  // Calling same function
  // from both the classes
  // object to show method overriding
  geek1.show();
  geek2.show();
}
```

```
This is class SuperGeek.
This is class SubGeek child of SuperGeek.
```

Example 2: When there is more than one child class.

```
// Dart Program to illustrate the method overriding concept
class SuperGeek {
  // Creating a method
  void show(){
    print("This is class SuperGeek.");
  }
}

class SubGeek1 extends SuperGeek {
  // Overriding show method
  void show(){
    print("This is class SubGeek1 child of SuperGeek.");
  }
}
class SubGeek2 extends SuperGeek {
  // Overriding show method

  void show(){
    print("This is class SubGeek2 child of SuperGeek.");
  }
}
void main() {
  // Creating objects of both the classes
  SuperGeek geek1 = new SuperGeek();
  SubGeek1 geek2 = new SubGeek1();
  SubGeek2 geek3 = new SubGeek2();

  // Calling same function
  // from both the classes
  // object to show method
  // overriding
  geek1.show();
  geek2.show();
  geek3.show();
}
```

```
This is class SuperGeek.
This is class SubGeek1 child of SuperGeek.
This is class SubGeek2 child of SuperGeek.
```

Dart: Abstract Class

An **Abstract class** in Dart is defined for those classes which contain one or more than one abstract method (methods without implementation) in them. Whereas, to declare abstract class we make use of the **abstract** keyword. So, it must be noted that a class declared abstract may or may not include abstract methods but if it includes an abstract method then it must be an abstract class.

Features of Abstract Class:

A class containing an abstract method must be declared abstract whereas the class declared abstract may or may not have abstract methods i.e., it can have either abstract or concrete methods

A class can be declared abstract by using abstract keyword only.

A class declared as abstract can't be initialized.

An abstract class can be extended, but if you inherit an abstract class then you have to make sure that all the abstract methods in it are provided with implementation.

Generally, abstract classes are used to implement the abstract methods to the extended subclasses.

Syntax:

```
abstract class class_name {

   // Body of the abstract class
}
```

NOTE:

Abstract Method:

- To make a method abstract, use **semicolon** (;) instead of method body.
- Abstract method can only exist with abstract class.
- You need to override abstract methods in sub-class.

Abstract Class:

- Use **abstract** keyword to declare abstract class.
- Abstract class can have **abstract methods**, **normal methods** and **instance variables** as well.
- The abstract class **cannot** be instantiated, you **cannot** create objects.

Overriding abstract method of abstract class:

```
// Understanding Abstract class in Dart

// Creating Abstract Class
abstract class Gfg {
   // Creating Abstract Methods
   void say();
   void write();
}

class Geeksforgeeks extends Gfg{
   @override
   void say()
   {
      print("Yo Geek!!");
   }

   @override
   void write()
   {
      print("Geeks For Geeks");
   }
}

main()
{
   Geeksforgeeks geek = new Geeksforgeeks();
   geek.say();
   geek.write();
}
```

```
Yo Geek!!
Geeks For Geeks
```

Explanation:

First, we declare an abstract class Gfg and create an abstract method geek_info inside it. After that, we extend the Gfg class to the second class and override the methods say () and write (), which result in their respective output.

Note:

It is not mandatory to override the method when there is only one class extending the abstract class, because override is used to change the pre-defined code and as in the above case, nothing is defined inside the method so the above code will work just fine without override.

Overriding abstract method of an abstract class in two different classes:

```
// Understanding Abstract Class in Dart
// Creating Abstract Class
abstract class Gfg {
   // Creating Abstract Method
   void geek_info();
}
// Class Geek1 Inheriting Gfg class
class Geek1 extends Gfg {
   // Overriding method
   @override
   void geek_info()
   {
      print("This is Class Geek1.");
   }
}
// Class Geek2 Inheriting Gfg class
class Geek2 extends Gfg {
   // Overriding method again
   @override
   void geek_info()
   {
      print("This is Class Geek2.");
   }
}
void main()
{
   Geek1 g1 = new Geek1();
   g1.geek_info();
   Geek2 g2 = new Geek2();
   g2.geek_info();
}
```

```
This is Class Geek1.
This is Class Geek2.
```

Explanation:

First, we declare an abstract class Gfg and create an abstract method geek_info inside it. After that, we extend the Gfg class to two other classes and override the method geek_info, which results in two different output strings.

In this code, we have to use override as we are redefining the method in two different class. If we don't use the override in the above example will give result as:

```
This is Class Geek1
This is Class Geek1
```

The same result was printed twice because the method not redefined in the next class.

Dart: Builder Class

Whenever we create a new widget in a flutter there is always a build Widget associated with it and the **BuildContext** parameter is passed by the framework.

```
Widget build ( BuildContext context )
```

Flutter takes care that there need not be any Widget apart from the build that needs the context parameter in their constructors or functions. So, we have to pass the context parameter through the build Widget only otherwise there would be more than one call to the build function.

This is where the Builder class comes into the picture. The main function of the Builder class is to build the child and return it. The Builder class passes a context to the child, it acts as a custom build function.

Builder class constructor

```
Builder({Key key, @required WidgetBuilder builder})
```

The builder argument must not be null.

The different methods available of the class are –

- build(BuildContext context) → Widget
- createElement() → StatelessElement
- debugDescribeChildren() → List<DiagnosticsNode>
- debugFillProperties(DiagnosticPropertiesBuilder properties) → void
- noSuchMethod(Invocation invocation) → dynamic
- toString({DiagnosticLevel minLevel: DiagnosticLevel.info}) → String

We will be using the following example to understand the function of the Builder class. We have actually made a very simple app to demonstrate this. The main screen of the app has a simple Scaffold with an AppBar and the body with a simple button which is made with a GestureDetector. The aim of the button is to display a SnackBar when the person clicks on the button.

The main.dart file is as following:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Builder Demo',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primarySwatch: Colors.green,
      ),
      home: Home(),
    );
  }
}

class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(

      // appbar
      appBar: AppBar(
        title: Text('GeeksforGeeks'),
      ),

      // detect gesture
      body: Center(
        child: GestureDetector(
          onTap: () {
            Scaffold.of(context).showSnackBar(
              new SnackBar(
                content: new Text('GeeksforGeeks'),
              ),
            );
          },

          // box styling
          child: Container(
```

```
          margin: EdgeInsets.all(18),
          height: 40,
          decoration: BoxDecoration(
            color: Colors.blueGrey,
            borderRadius: BorderRadius.circular(8),
          ),
          child: Center(
            child: Text(
              'CLICK ME',
              style:
                  TextStyle(fontWeight: FontWeight.bold, color: Colors.white),
            ),
          ),
        ),
      ),
    ),
  );
 }
}
```

If we run the above program we get the following error –

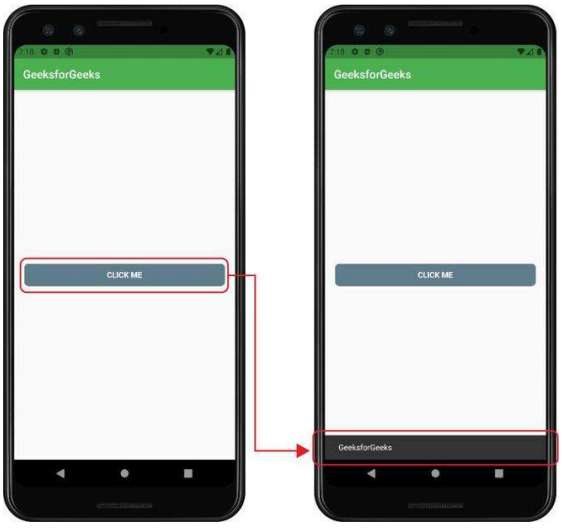
```
======== Exception caught by gesture
=============================================================
Scaffold.of() called with a context that does not contain a Scaffold.
=============================================================================================
==========
```

The error occurs because we are building the Scaffold at the same time, we are calling the SnackBar widget (i.e., the same context is being passed to the Scaffold and to the SnackBar). The context that is being passed does not belong to a Scaffold and for SnackBar to appear there needs to be a Scaffold. So, the app gives the error.

To correct this error, we can wrap the Gesture detector with a builder widget as following. In this case, the context is being passed to the SnackBar widget through the Builder. The SnackBar becomes the child of the Scaffold context being passed. Now if click on the button It gives the desired output as the Scaffold is already present for the SnackBar to appear.

```
body: Center(
      child: Builder(builder: (BuildContext context) {
        return GestureDetector(
          onTap: () {
            Scaffold.of(context).showSnackBar(
              new SnackBar(
                content: new Text('GeeksforGeeks'),
              ),
            );
          },
          child: Container(
            margin: EdgeInsets.all(18),
            height: 40,
            decoration: BoxDecoration(
              color: Colors.blueGrey,
              borderRadius: BorderRadius.circular(8),
            ),
            child: Center(
              child: Text(
                'CLICK ME',
                style: TextStyle(
                    fontWeight: FontWeight.bold, color: Colors.white),
              ),
            ),
          ),
        );
      }),
    ),
```

Now the program works and shows the following output when we tap on the button.



Dart: Callable Classes

Dart allows the user to create a **callable class** which allows the instance of the class to be called **as a function**. To allow an instance of your Dart class to be called like a function, implement the **call () method**.

Syntax:

```
class class_name {
  ... // class content

  return_type call ( parameters ) {
    ... // call function content
  }

}
```

In the above syntax, we can see that to create a callable class we have to define a call method with a return type and parameters within it.

Example 1: Implementing a callable class in Dart

```
// Creating Class GeeksForGeeks
class GeeksForGeeks {

  // Defining call method which create the class callable
  String call(String a, String b, String c) => 'Welcome to $a$b$c!';
}

// Main Function
void main() {
  // Creating instance of class
  var geek_input = GeeksForGeeks();

  // Calling the class through its instance
  var geek_output = geek_input('Geeks', 'For', 'Geeks');

  // Printing the output
  print(geek_output);
}
```

```
Welcome to GeeksForGeeks!
```

It must be noted that Dart doesn't support multiple callable methods i.e., if we try to create more than one callable function for the same class it will display **error**.

Example 2: Implementing multiple callable functions in a class of Dart.

```
// Creating Class GeeksForGeeks
class GeeksForGeeks {
  // Defining call method which create the class callable
  String call(String a, String b, String c) => 'Welcome to $a$b$c!';

  // Defining another call method for the same class
  String call(String a) => 'Welcome to $a!';
}

// Main Function
void main() {
  // Creating instance of class
  var geek_input = GeeksForGeeks();

  // Calling the class through its instance
  var geek_output = geek_input('Geeks', 'For', 'Geeks');

  // Printing the output
  print(geek_output);
}
```

```
Error compiling to JavaScript:
main.dart:3:10:
Error: 'call' is already declared in this scope.
  String call(String a) => 'Welcome to $a!';
         ^^^^
main.dart:2:10:
Info: Previous declaration of 'call'.
  String call(String a, String b, String c) => 'Welcome to $a$b$c!';
         ^^^^
main.dart:8:31:
Error: Can't use 'call' because it is declared more than once.
  var geek_output = geek_input('Geeks', 'For', 'Geeks');
                    ^^^^
main.dart:8:31:
Error: The method 'call' isn't defined for the class 'GeeksForGeeks'.
 - 'GeeksForGeeks' is from 'main.dart'.
  var geek_output = geek_input('Geeks', 'For', 'Geeks');
                    ^
Error: Compilation failed.
```

Dart: Interface

The interface in the dart provides the user with the blueprint of the class, that any class should follow if it interfaces that class i.e., if a class inherits another it should redefine each function present inside an interfaced class in its way. They are nothing but a set of methods defined for an object. Dart doesn't have any direct way to create inherited class, we have to make use of implements keyword to do so.

Syntax:

```
class Interface_class_name{
  ...
}

class Class_name implements Interface_class_name {
  ...
}
```

Example 1:

```
void main(){
  // Creating Object of the class Gfg
  Gfg geek1= new Gfg();
  // Calling method (After Implementation )
  geek1.printdata();
}
// Class Geek (Interface)
class Geek {
  void printdata() {
    print("Hello Geek !!");
  }
}
// Class Gfg implementing Geek
class Gfg implements Geek {
  void printdata() {
    print("Welcome to GeeksForGeeks");
  }
}
```

```
Welcome to GeeksForGeeks
```

Note: Class should use the implements keyword, instead of extending to be able to use an interface method.

Multiple Inheritance in Dart

In dart, multiple inheritances are achieved by the use of implements. Although practically dart doesn't support multiple inheritances, it supports multiple interfaces.

Syntax:

```
class interface_class1 {
   ...
}
class interface_class2 {
   ...
}
.
.
class interface_classN {
   ...
}

class class_name implements interface_class1, interface_class2, ...., interface_classN {
   ...
}
```

Example 2:

```
// Dart Program to show Multiple Inheritance
void main(){
  // Creating Object of
  // the class Gfg
  Gfg geek1= new Gfg();
  // Calling method (After Implementation)
  geek1.printdata1();
  geek1.printdata2();
  geek1.printdata3();
}
// Class Geek1 (Interface1)
class Geek1 {
  void printdata1() {
    print("Hello Geek1 !!");
  }
}
 // Class Geek2 (Interface2)
class Geek2 {
  void printdata2() {
    print("Hello Geek2 !!");
  }
}
```

```
 // Class Geek3 (Interface3)
class Geek3 {
  void printdata3() {
    print("Hello Geek3 !!");
  }
}
// Class Gfg implementing Geek1, Geek2, Geek3.
class Gfg implements Geek1, Geek2, Geek3 {
  void printdata1() {
    print("Howdy Geek1,\nWelcome to GeeksForGeeks");
  }
 void printdata2() {
    print("Howdy Geek2,\nWelcome to GeeksForGeeks");
  }
 void printdata3() {
    print("Howdy Geek3,\nWelcome to GeeksForGeeks");
  }
}
```

```
Howdy Geek1,
Welcome to GeeksForGeeks
Howdy Geek2,
Welcome to GeeksForGeeks
Howdy Geek3,
Welcome to GeeksForGeeks
```

Importance of Interface:

- Used to achieve **abstraction** in Dart.
- It is a way to achieve **multiple inheritances** in Dart.

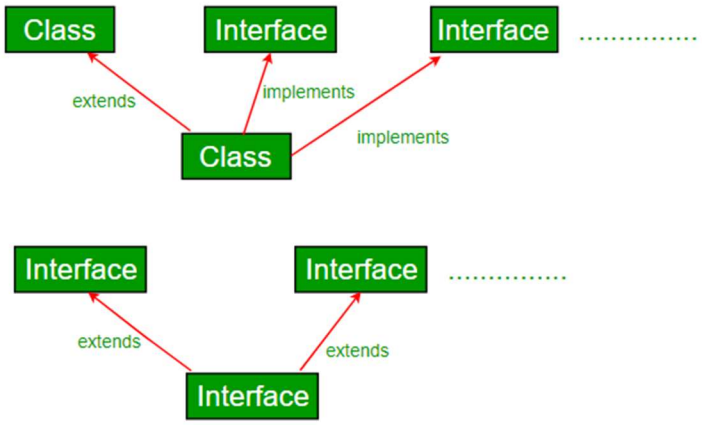
Important Points:

1. If a class has been implemented then all of its method and instance variable must be overridden during the interface.
2. In dart, there are no direct (**does not** have any special syntax to declare interface) means to declare an interface, so a declaration of a class is itself considered as a declaration on the interface.
3. A class can extend only one class but can implement as many as you want.
4. An interface in Dart is a normal class.
5. An interface is used when you need concrete implementation of all of its functions within its sub class. It is mandatory to override all methods in the implementing class.
6. You can **implement** multiple classes but you **cannot extend** multiple classes during Interface.

Dart: extends vs with vs implements

All developers working with dart for application development using the Flutter framework regularly encounters different usage of the implements, extends and with keywords. In Dart, one class can inherit another class i.e., dart can create a new class from an existing class. We make use of keywords to do so. In this article, we will look into 3 of the keywords used for the same purpose and compare them, namely:

- extends
- with
- implements



Extends

In Dart, the extends keyword is typically used to alter the behavior of a class using Inheritance. The capability of a class to derive properties and characteristics from another class is called **Inheritance**. It is ability of a program to create new class from an existing class. In simpler words, we can say that we use extends to create a subclass, and super to refer to the superclass. The class whose properties are inherited by child class is called **Parent Class**. Parent class is also known as base class or **super class**. The class that inherits properties from another class is called **child class**. Child class is also known as derived class, heir class, or subclass. The extends keyword is the typical OOP class inheritance. If class Second extends class First all properties, variables, methods implemented in class First are also available in Second class. Additionally, you can override methods.

You use extend if you want to create a more specific version of a class. For instance if Apple extends from Fruit class, it means all the properties, variables and functions defined in the Fruit class will be available in the Apple class.

Example:

Below we can see an example of implementation of the extends keyword. We are not required to override the definition of the inherited class and can use the existing definition in the child class.

```
// Class with name First
class First {
  static int num = 1;
  void firstFunc(){
    print('hello');
  }
}

// inherits from First class
class Second extends First{
//   No need to override
}

void main(){

  // instance of First Class
  var one = First();

  // calling firstFunc()
  one.firstFunc();

  // printing static variable of class
  print(First.num);

  // instance of Second Class
  var second = Second();

  // calling firstFunc() that
  // has been inherited
  second.firstFunc();
}
```

```
hello
1
hello
```

Implements

Interfaces define a set of methods available on an object. Dart does not have a syntax for declaring interfaces. Class declarations are themselves interfaces in Dart. An interface is something that enforces the deriving class to implement a set list of public fields and methods.

The implement keyword is used to implement an interface by forcing the redefinition of the functions.

Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B's API without inheriting B's implementation, class A should implement the B interface. A class implements one or more interfaces by declaring them in an implements clause and then providing the APIs required by the interfaces.

Example:

```
// Class with name First
class First {
  // function to print "hello"
  void firstFunc(){
    print('hello');
  }
}
// We inherit the propertied of implemented class
class Second implements First{
  // by overriding the functions in implemented class
  @override
  void firstFunc(){
    print('We had to declare the methods of implemented class');
  }
}
void main(){
  // instance of First Class
  var one = First();
  // calling firstFunc()
  one.firstFunc();
  // instance of Second Class
  var second = Second();
  // calling firstFunc() that has been inherited
  second.firstFunc();
}
```

```
hello
We had to declare the methods of implemented class
```

With

Mixins are a way of reusing a class's methods in multiple class hierarchies. Mixins can be understood as **abstract classes** used for reusing the methods in various classes that have similar functions/attribute. Mixins are a way to abstract and reuse a family of operations and state. It is similar to the reuse you get from extending a class, but is not multiple inheritances. There still only exists one superclass.

With is used to include Mixins. A mixin is a different type of structure, which can only be used with the keyword with.

Mixin is a different type of structure, which can only be used with the keyword **with**. In Dart, a class can play the role of mixin if the class is does not have a constructor. It is also important to note that a mixin doesn't force a type restriction nor it imposes usage restrictions on the class methods.

Example:

```
// mixin with name First
mixin First {
  void firstFunc(){
    print('hello');
  }
}

// mixin with name temp
mixin temp {
  void number(){
    print(10);
  }
}

// mixin type used with keyword
class Second with First, temp{
  @override
  void firstFunc(){
    print('can override if needed');
  }
}

void main(){
  var second = Second();
  second.firstFunc();
  second.number();
}
```

```
hello
can override if needed
```

Dart Programming: Dart Utilities

Dart: Date and Time

A **DateTime** object is a point in time. The time zone is either UTC or the local time zone. Accurate date-time handling is required in almost every data context. Dart has the marvelous built-in classes DateTime and Duration in **dart:core**.

Some of its uses are:

- Compare and calculate with date times
- Get every part of a date-time
- Work with different time zones
- Measure time spans

Example 1:

```
void main(){
// Get the current date and time.
var now = DateTime.now();
print(now);
// Create a new DateTime with the local time zone.
var y2k = DateTime(2000); // January 1, 2000
print(y2k);
// Specify the month and day.
y2k = DateTime(2000, 1, 2); // January 2, 2000
print(y2k);
// Specify the date as a UTC time.
y2k = DateTime.utc(2000); // 1/1/2000, UTC
print(y2k);
// Specify a date and time in ms since the Unix epoch.
y2k = DateTime.fromMillisecondsSinceEpoch(946684800000,
    isUtc: true);
print(y2k);
// Parse an ISO 8601 date.
y2k = DateTime.parse('2000-01-01T00:00:00Z');
print(y2k);
}
```

```
2020-08-25 11:58:56.257
2000-01-01 00:00:00.000
2000-01-02 00:00:00.000
2000-01-01 00:00:00.000Z
2000-01-01 00:00:00.000Z
2000-01-01 00:00:00.000Z
```

The **millisecondsSinceEpoch** property of a date returns the number of milliseconds since the "Unix epoch"—January 1, 1970, UTC.

Example 2:

```
void main(){

// 1/1/2000, UTC
var y2k = DateTime.utc(2000);
print(y2k.millisecondsSinceEpoch == 946684800000);

// 1/1/1970, UTC
var unixEpoch = DateTime.utc(1970);
print(unixEpoch.millisecondsSinceEpoch == 0);
}
```

```
true
true
```

The Duration class can be used to calculate the difference between two dates and to move date forward or backward.

Example 3:

```
void main(){
var y2k = DateTime.utc(2000);
// Add one year.
var y2001 = y2k.add(Duration(days: 366));
print(y2001.year == 2001);

// Subtract 30 days.
var december2000 = y2001.subtract(Duration(days: 30));
assert(december2000.year == 2000);
print(december2000.month == 12);

// Calculate the difference between two dates.
// Returns a Duration object.
var duration = y2001.difference(y2k);
print(duration.inDays == 366); // y2k was a leap year.

}
```

```
true
true
true
```

Using await async in Dart

Async functions:

Functions form the base of asynchronous programming. These async functions have async modifiers in their body. Here is an example of a general async function below:

When an async function is called, a Future is immediately returned and the body of the function is executed later. As the body of the async function is executed, the Future returned by the function call will be completed along with its result.

Any functions you want to run asynchronously need to have the async modifier added to it. This modifier comes right after the function signature, like this:

```
void hello() async {
  print('something exciting is going to happen here...');
}
```

Typically, the function you want to run asynchronously would have some expensive operation in it like file I/O (an API call to a RESTful service).

Await expressions:

Await expressions makes you write the asynchronous code almost as if it were synchronous. In general, an await expression has the form as given below:

```
void main() async {
  await hello();
  print('all done');
}
```

Typically, it is an asynchronous computation and is expected to evaluate to a Future. The await expressions evaluate the main function, and then suspends the currently running function until the result is ready–that is, until the Future has completed. The result of the await expression is the completion of the Future.

- There are **two** important things to grasp concerning the block of code above. First off, we use the async modifier on the main method because we are going to run the hello() function asynchronously.
- Secondly, we place the await modifier directly in front of our asynchronous function. Hence, this is frequently referred to as the async/await pattern.
- Just remember, if you are going to use await, make sure that both the caller function and any functions you call within that function all use the **async** modifier.

Futures:

Dart is a single-threaded programming language. Future<T> object represents the result of an asynchronous operation which produces a result of type T. If the result is not usable value, then the future's type is Future<void>. A Future represents a single value either a data or an error asynchronously

There are 2 ways to handle Futures:

- Using the Future API
- Using the async and await operation

Now, Let's write a program and see the output.

Example:

```
Future delayedPrint(int seconds, String msg) {
  final duration = Duration(seconds: seconds);
  return Future.delayed(duration).then((value) => msg);
}
main() async {
  print('Life');
  await delayedPrint(2, "Is").then((status){
    print(status);
  });
  print('Good');
}
```

```
Life
Is
Good
```

In this case, we used async and **await** keywords. Await basically holds the control flow, until the operation completes. To use await within a function, we have to mark the function by **async** keyword. Which means, this function is an asynchronous function.

Combined with the Futures class, the async / await pattern in Dart is a powerful and expressive way of doing asynchronous programming.

Dart: Type System

The Dart programming language is considered type safe, meaning it ensures that the variable's value always matches the variable's static type through a combination of static type checking and runtime checking. It is also known as Sound Typing. It comes in handy while debugging the code at compile time.

All forms of static errors can be resolved by adding type annotations to generic classes. Some frequently used generic collection classes are listed below:

- List
- Map

Example 1: The below code will throw an error on list when a call to the printInts(list) is made.

```
void printInts(List<int> x) => print(x);

void main() {
  var list = [];
  list.add(1000);
  list.add(2000);
  printInts(list);
}
}
```

```
error - The argument type 'List' can't be assigned to the parameter type 'List' at
lib/strong_analysis.dart:27:17 - (argument_type_not_assignable)
```

The above error occurred due to an unsound implicit cast from a dynamic type List to an integer type, meaning declaring var List = [] doesn't provide sufficient information to the dart analyzer about the typing of the list items. To resolve this issue, we pass the type annotation to the list variable as shown below:

```
// Using num data type in Dart
void printInts(List<int> a) => print(a);

void main() {
  var list = <int>[];
  list.add(1000);
  list.add(2000);
  printInts(list);
}
}
```

```
[1000, 2000]
```

Concept of Soundness:

Soundness is the process of making sure that the code doesn't run into an invalid state. For instance, if a variable's static type is Boolean, it is not possible to run into a state where the variable evaluates to a non-string value during runtime.

Benefits of having Soundness:

- Debug type related bugs at compile time.
- Makes code easier to read and debug.
- Get alters when changing code pieces that breaks other dependent part of the code.
- One of its key features is **Ahead of Time** (AOT) compilation, which significantly reduces the compile time of the code and increases efficiency.

Dart: Generators

Generators in Dart allows the user to produce a sequence of value easily. One can generate a sequence of values in dart with the help of two generator functions:

1. **Synchronous** Generator: Returns an **Iterable** object.
2. **Asynchronous** Generator: Returns a **Stream** object.

Synchronous Generator

The synchronous generator returns an Iterable object i.e., it returns the collection of values, or "elements", that can be accessed sequentially. To implement synchronous generator function, mark the function body as sync*, and use yield statements to deliver value(s).

Example: Implementing a synchronous generator in Dart.

```
// sync* functions return an Iterable
Iterable geeksForGeeks(int number) sync* {
 int geek = number;
 while (geek >= 0) {

   // Checking for even number
   if (geek % 2 == 0) {

     // 'yield' suspends
     // the function
     yield geek;

   }

   // Decreasing the
   // variable geek
   geek--;
 }
}
```

```
// Main Function
void main()
{
  print("------- Geeks For Geeks --------");

  print("Dart Synchronous Generator Example for Printing Even Numbers From 10 In
       Reverse Order:");

  // Printing positive even numbers
  // from 10 in reverse order
  geeksForGeeks(10).forEach(print);
}
```

```
---------- Geeks For Geeks ----------
Dart Synchronous Generator Example for Printing Even Numbers From 10 In Reverse Order:
10
8
6
4
2
0
```

Asynchronous Generator

The asynchronous generator returns a stream object. A Stream provides a way to receive a sequence of events. Each event is either a data event, also called an **element** of the stream, or an error event, which is a notification that something has failed. To implement an **asynchronous** generator function, mark the function body as **async***, and use **yield statements** to deliver value(s).

Example: Implementing an asynchronous generator in Dart.

```
// async* function(s) return an stream
Stream geeksForGeeks(int number) async* {
  int geek = 0;

  // Checking for every
  // geek less than number
  while (geek <= number) yield geek++;
  // Incrementing geek
  // after printing it
}

// Main Function
void main()
{
  print("-------- Geeks For Geeks ----------");
```

```
  print("Dart Asynchronous Generator Example for Printing Numbers Less Than 10:");

  // Printing numbers less
  // than or equal to 10
  geeksForGeeks(10).forEach(print);
}
```

```
---------- Geeks For Geeks ----------
Dart Asynchronous Generator Example for Printing Numbers Less Than 10:
0
1
2
3
4
5
6
7
8
9
10
```

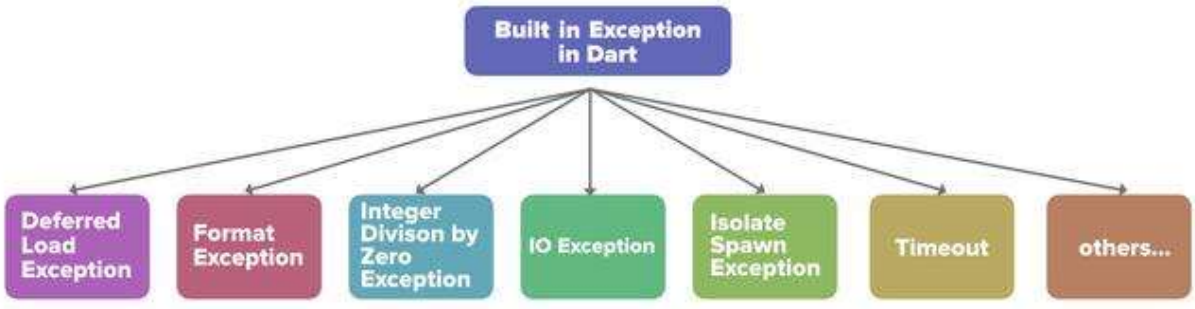
Dart Programming: Advanced Concepts

Dart: Exception Handling

When normal flow of program is disrupted and application crashes, then it is called Exception Handling.

An exception is an error that takes place inside the program. When an exception occurs inside a program the normal flow of the program is disrupted and it terminates abnormally, displaying the error and exception stack as output. So, an exception must be taken care to prevent the application from termination.

Built-in Exceptions in Dart:



Sr.No	Exceptions & Description
1	**DeferredLoadException** Thrown when a deferred library fails to load.
2	**FormatException** Exception thrown when a string or some other data does not have an expected format and cannot be parsed or processed.
3	**IntegerDivisionByZeroException** Thrown when a number is divided by zero.
4	**IOException** Base class for all Inupt-Output related exceptions.
5	**IsolateSpawnException** Thrown when an isolate cannot be created.
6	**Timeout** Thrown when a scheduled timeout happens while waiting for an async result.

Every built-in exception in Drat comes under a pre-defined class named **Exception**. To prevent the program from exception we make use of **try/on/catch blocks** in Dart.

```
try {
  // program that might throw an exception
}
on Exception1 {
  // code for handling exception 1
}
catch Exception2 {
  // code for handling exception 2
}
```

Example 1: using a try-on block in the dart.

```
void main() {
  String geek = "GeeksForGeeks";
  try{
    var geek2 = geek ~/ 0;
    print(geek2);
  }
  on FormatException{
    print("Error!! \nCan't act as input is not an integer.");
  }
}
```

```
Error!!
Can't act as input is not an integer.
```

Example 2: Using a try-catch block in the dart.

```
void main() {
  String geek = "GeeksForGeeks";
  try{
    var geek2 = geek ~/ 0;
    print(geek2);
  }
  catch(e){
    print(e);
  }
}
```

```
Class 'String' has no instance method '~/'.
NoSuchMethodError: method not found: '~/'
Receiver: "GeeksForGeeks"
Arguments: [0]
```

Final block:

The final block in dart is used to include specific code that must be executed irrespective of error in the code. Although it is optional to include finally block if you include it then it should be after try and catch block are over.

```
finally {
  ...
}
```

Example:

```
void main() {
  int geek = 10;
  try{
    var geek2 = geek ~/ 0;
    print(geek2);
  }
  catch(e){
    print(e);
  }
  finally {
    print("Code is at end, Geek");
  }
}
```

```
Unsupported operation: Result of truncating division is Infinity: 10 ~/ 0
Code is at end, Geek
```

Unlike other languages, in Dart to one can create a custom exception. To do, so we make use of throw new keyword in the dart.

Example: Creating custom exceptions in the dart.

```
// extending Class Age
// with Exception class
class Age implements Exception {
  String error() => 'Geek, your age is less than 18 :(';
}

void main() {
  int geek_age1 = 20;
  int geek_age2 = 10;

  try{
    // Checking Age and
```

```
    // calling if the
    // exception occur
    check(geek_age1);
    check(geek_age2);
  }
  catch(e){
    // Printing error
    print(e.error());
  }
}

// Checking Age
void check(int age){
  if(age < 18){
    throw new Age();
  }
  else
  {
    print("You are eligible to visit GeeksForGeeks :)");
  }
}
```

```
You are eligible to visit GeeksForGeeks :)
Geek, your age is less than 18 :(
```

Dart: Assert Statements

As a programmer, it is very necessary to make an errorless code is very necessary and to find the error is very difficult in a big program. Dart provides the programmer with **assert statements** to check for the error. The assert statement is a useful tool to debug the code and it uses Boolean condition for testing. If the boolean expression in assert statement is true then the code continues **to execute**, but if it returns **false** then the **code ends with Assertion Error**.

```
Syntax: assert(condition);
```

It must be noted that if you want to use assert then you have to enable it while execution as it can only be used in the development mode and not in productive mode. If it is not enabled then it will be simply be ignored while execution.

Enable the assert while executing a dart file via cmd as:

```
dart --enable-asserts file_name.dart
```

Example 1: Using assert in a dart program.

```
void main()
{
  String geek = "Geeks For Geeks";
  assert(geek != "Geeks For Geeks");
  print("You Can See This Line Geek as a Output");
}
```

```
Unhandled exception:
'file:///C:/Users/msaur/Desktop/GeeksForGeeks.dart': Failed assertion: line 4 pos 10: 'geek != "Geeks For
Geeks"': is not true.
#0      _AssertionError._doThrowNew (dart:core-patch/errors_patch.dart:42:39)
#1     _AssertionError._throwNew (dart:core-patch/errors_patch.dart:38:5)
#2     main (file:///C:/Users/msaur/Desktop/GeeksForGeeks.dart:4:10)
#3     _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:301:19)
#4     _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:168:12)
```

Output when asserts are not enabled:

```
You Can See This Line Geek as a Output
```

Apart from that, you can also send a message for the case if the assert returns false as:

```
assert(condition, "message");
```

It is very useful when you are trying to debug various errors and want to know which assert returned the error in the code.

Example: Using assert to give the message in a dart program

```
void main()
{
  String geek = "Geeks For Geeks";
  assert(geek != "Geeks For Geeks", "Strings are equal So this message is been displayed!!");
  print("You Can See This Line Geek as a Output if assert returns true");
}
```

```
C:\Users\msaur\Desktop>dart --enable-asserts GeeksForGeeks.dart
Unhandled exception:
'file:///C:/Users/msaur/Desktop/GeeksForGeeks.dart': Failed assertion: line 4 pos 10: 'geek !=
"Geeks For Geeks"': Strings are equal So this message is been displayed!!
#0      _AssertionError._doThrowNew (dart:core-patch/errors_patch.dart:42:39)
#1     _AssertionError._throwNew (dart:core-patch/errors_patch.dart:38:5)
#2     main (file:///C:/Users/msaur/Desktop/GeeksForGeeks.dart:4:10)
#3     _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:301:19)
#4     _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:168:12)
```

Dart: Fallthrough Condition

Fall through is a type of error that occurs in various programming languages like C, C++, Java, Dart ...etc. It occurs in switch-case statements where when we forget to add break statement and in that case flow of control jumps to the next line.

"If no break appears, the flow of control will fall through all the cases following true case until the break is reached or end of the switch is reached."

So, it is clear that the most basic way of creating the situation of fall through is skipping the break statements in Dart, but in the dart, it will give a compilation error.

Example 1: Skipping break statements

```
// This code will display error
void main() {
  int gfg = 1;
  switch ( gfg ){
    case 1:{
      print("GeeksforGeeks number 1");
    }
    case 2:{
      print("GeeksforGeeks number 2");
    }
    case 3:{
      print("GeeksforGeeks number 3");
    }
    default :{
      print("This is default case");
    }
  }
}
```

```
ERROS:
Error compiling to JavaScript:
main.dart:4:5:
Error: Switch case may fall through to the next case.
    case 1:{
    ^
main.dart:7:5:
Error: Switch case may fall through to the next case.
    case 2:{
    ^
main.dart:10:5:
Error: Switch case may fall through to the next case.
    case 3:{
    ^
Error: Compilation failed.
```

However, it allows skipping of break statement in the case when there is only one case statement defined.

NOTE: It must be noted that Dart allows empty cases.

Example 2: Providing an empty class.

```
void main() {

  // Declaring value
  // of the variable
  int gfg = 2;
  switch ( gfg ){
    case 1:{
      print("GeeksforGeeks number 1");
    } break;

    // Empty case causes fall through
    case 2:
    case 3:{
      print("GeeksforGeeks number 3");
    } break;
    default :{
      print("This is default case");
    } break;
  }
}
```

```
GeeksforGeeks number 3
```

In the above code if we initialize the value of gfg = 3 than the output will not change. Another way to achieve fall through is via using continue instead of break statement in switch-case.

Example 3: Using continue instead of break

```
void main() {
  int gfg = 1;
  switch ( gfg ){
    case 1:{
      print("GeeksforGeeks number 1");
    } continue next;
    next:case 2:{
      print("GeeksforGeeks number 2");
    } break;
    case 3:{
      print("GeeksforGeeks number 3");
    } break;
    default :{
```

```
    print("This is default case");
   } break;
  }
}
```

```
GeeksforGeeks number 1
GeeksforGeeks number 2
```

NOTE: It must be noted that if we will not declare label with continue in the above code then the code will display error.

Dart: Concept of Isolates

Dart was traditionally designed to create single-page applications. And we also know that most computers, even mobile platforms, have multi-core CPUs. To take advantage of all those cores, developers traditionally use **shared-memory threads** running concurrently. However, shared-state concurrency is error-prone and can lead to complicated code. Instead of **threads**, all Dart code runs inside of **isolates**. Each isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate.

The **isolates** and **threads** are different than each other as in threads memory are shared whereas in isolates it is not. Moreover, isolates talk to each other via passing messages.

To use **isolates**, you have to add import '**dart:isolate**'; statement in your program code.

Creating an Isolate in Dart

To create an isolate we make use of **.spawn ()** method in Dart.

```
Syntax: Isolate isolate_name = await Isolate.spawn( parameter );
```

This parameter represents the port that will receive the message back.

Destroying an isolate in Dart

To destroy an isolate we make use of **.kill ()** method in Dart.

```
Syntax: isolate_name.kill( parameters );
```

We generally use **spawn** () and **kill** () together in a single program.

Dart: Typedef

Typedef in Dart is used to create a user-defined identity (**alias**) for a function, and we can use that identity in place of the function in the program code. When we use typedef we can define the parameters of the function.

```
Syntax: typedef function_name ( parameters );
```

With the help of typedef, we can also assign a variable to a function.

```
Syntax:typedef variable_name = function_name;
```

After assigning the variable, if we have to invoke it then we go as:

```
Syntax: variable_name( parameters );
```

By this we will be able to use a single function in different ways:

Example 1: Using typedef in Dart

```
// Dart program to show the usage of typedef

// Defining alias name
typedef GeeksForGeeks(int a, int b);

// Defining Geek1 function
Geek1(int a, int b) {
  print("This is Geek1");
  print("$a and $b are lucky geek numbers !!");
}

// Defining Geek2 function
Geek2(int a, int b) {
  print("This is Geek2");
  print("$a + $b is equal to ${a + b}.");
}

// Main Function
void main()
{
  // Using alias name to define
  // number with Geek1 function
    GeeksForGeeks number = Geek1;
  // Calling number
  number(1,2);
```

```
  // Redefining number
  // with Geek2 function
  number = Geek2;
  // Calling number
  number(3,4);
}
```

```
This is Geek1
1 and 2 are lucky geek numbers !!
This is Geek2
3 + 4 is equal to 7.
```

NOTE: Apart from this, typedef can also act as parameters of a function.

Example 2: Using typedef as a parameter of a function.

```
// Dart program to show the usage of typedef

// Defining alias name
typedef GeeksForGeeks(int a, int b);

// Defining Geek1 function
Geek1(int a, int b) {
  print("This is Geek1");
  print("$a and $b are lucky geek numbers !!");
}

// Defining a function with a typedef variable
number(int a, int b, GeeksForGeeks geek) {
  print("Welcome to GeeksForGeeks");
  geek(a, b);
}

// Main Function
void main()
{
  // Calling number function
  number(21,23, Geek1);
}
```

```
Welcome to GeeksForGeeks
This is Geek1
21 and 23 are lucky geek numbers!!
```

Dart: URIs

The Uri class supports functions to encode and decode strings for use in URIs (which you might know as URLs). These functions control characters that are unique for URIs, such as & and =. This class also parses and exposes the components of a URI—host, port, scheme, and so on.

Encoding and decoding fully qualified URIs

The **encodeFull**() and **decodeFull**() methods are used to encode and decode characters except those with special meaning in a URI (such as /, :, &, #).

Example:

```
void main(){

var uri = 'https://example.org/api?foo=some message';

var encoded = Uri.encodeFull(uri);
assert(encoded ==
    'https://example.org/api?foo=some%20message');

var decoded = Uri.decodeFull(encoded);
print(uri == decoded);
}
```

```
true
```

Encoding and decoding URI components

To encode and decode all of a string's characters that have special meaning in a URI, including (but not limited to) /, &, and :, use the **encodeComponent**() and **decodeComponent**() methods.

Example:

```
void main(){

var uri = 'https://example.org/api?foo=some message';

var encoded = Uri.encodeComponent(uri);
assert(encoded ==
    'https%3A%2F%2Fexample.org%2Fapi%3Ffoo%3Dsome%20message');

var decoded = Uri.decodeComponent(encoded);
print(uri == decoded);
}
```

```
true
```

Parsing URIs

If you have a URI object or a URI string, you can get its parts using URI fields such as path. To create a URI from a string, use the **parse() static method**.

Example:

```
void main(){

var uri =
    Uri.parse('https://example.org:8080/foo/bar#frag');

assert(uri.scheme == 'https');
assert(uri.host == 'example.org');
assert(uri.path == '/foo/bar');
assert(uri.fragment == 'frag');
print(uri.origin == 'https://example.org:8080');
}
```

```
true
```

Building URIs

You can using the Uri() constructor is used to build up a URI from individual parts

Example:

```
void main(){

var uri = Uri(
    scheme: 'https',
    host: 'example.org',
    path: '/foo/bar',
    fragment: 'frag');
print(
    uri.toString() == 'https://example.org/foo/bar#frag');
}
```

```
true
```

Dart: Packages

The **package** is a set of dart program organized in an independent, reusable unit. It contains a set of functions and classes for a specific purpose or utility along with the compiled code and sample data. Dart comes with a rich set of default packages, loaded automatically when Dart console is started. Any other package other than the default needs to be installed and loaded explicitly first in order to use it. Once a packages Is loaded, it can be used throughout the Dart environment.

Dart Package Manager

Dart comes with an inbuilt Package Manager known as **pub package manager**. It is used to Install, organize, and manage third-party libraries, tools, and dependencies. Every Dart application has a pubspec.yaml file that keeps track of all the third-party libraries and application dependencies along with the metadata of applications like application name, author, version, and description, Most of the Dart IDEs offer in-built support for using a pub that includes creating, downloading, updating, and publishing packages. Otherwise, pub ca be used on the command line.

Below is a list of the important pub commands:

	Description
pub get	Gets all package your application is dependent upon
pub upgrade	Upgrades all your application dependencies to latest version
pub build	Used to create a bit of your web application and it will create a build folder with all related scripts in it
pub help	This command is used to know about all different pub commands

Installing a Package

Step 1: Add the package name in the dependencies section of your project's pubspec.yaml file. Then, the following command can be run from the application directory to get the package installed in the project.

```
pub get
```

This will download the packages under the packages folder in the application directory.

Example:

An application needs to parse XML. Dart XML is a lightweight library that is open source and stable for parsing, traversing, querying, and building XML documents.

```
name: GFGapp
version: 0.0.1
description: A simple core application.
#dependencies:
#  foo_bar: '>=1.0.0 <2.0.0'
dependencies: https://mail.google.com/mail/u/0/images/cleardot.gif
xml:
```

To refer to the dart XML in the project. The syntax is as follows:

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

Read XML String

XML string can read and verify the input, Dart XML uses a parse() method.

```
Syntax: xml.parse(String input):
```

Example:

Parsing XML String Input. The following example shows how to parse XML string input.

```
// Dart program to illustrate
// parsing XML in Dart
import 'package:xml/xml.dart' as xml;
void main(){
  print("GFG-XML");
  var bookshelfXml = '''
```

```
GFG-XML
<?xml version = "1.0"?><bookshelf>
 <book>
   <title lang = "english">Growing a Language</title>
   <price>29.99</price>
 </book>

 <book>
   <title lang = "english">Learning XML</title>
   <price>39.95</price>
 </book>
 <price>132.00</price>
</bookshelf>
```

Dart: String codeUnits Property

The string property codeunits in Dart programming language returned the list of UTF-16 code of the characters of a string. It is very useful string properties to directly find UTF-16 code units.

```
Syntax: String.codeUnits
Returns: a list of UTF-16 code units
```

Example 1:

```
// main function start
void main() {

  // initialize string st
  String st = "Geeksforgeeks";

  // print the UTF-16 code
  print(st.codeUnits);
}
```

```
[71, 101, 101, 107, 115, 102, 111, 114, 103, 101, 101, 107, 115]
```

Example 2:

```
// main function start
void main() {

   // initialize string st
  String st = "Computer";

  // print the UTF-16 code
  print(st.codeUnits);
}
```

```
[67, 111, 109, 112, 117, 116, 101, 114]
```

Dart: Closure

A **closure** is a bundle of a function and variables from the outer scope that the function depends on.

For **example**, if a function depends on the variable from the scope of the parent function which has already returned, a closure holds the variables of the returned function.

Closure is a special function.

Within a closure you can **mutate** (modify) the values of variables present in the parent scope.

In Java 8, You are not allowed to modify parent scope variables.

Example:

```
// Closures
void main() {
  // Definition 1:
  // A closure is a function that has access to the parent scope, even after the scope has closed.
  String message = "Dar is good";

  Function showMessage = () {
    message = "Dart is awesome";
    print(message);
  };
  showMessage();
  // Definition 2:
  // A closure is a function object that has access to variables in its lexical scope,
  // even when the function is used outside of its original scope.
  Function talk = () {
    String msg = "Hi";

    Function say = () {
      msg = "Hello";
      print(msg);
    };

    return say;
  };

  Function speak = talk();
  speak(); // talk()      // say()      //  print(msg)     // "Hello"
}
```

```
Dart is awesome
Hello
```